

# Rapport de Travaux Pratiques

Réalisé par :

SANTOS Wylisse

Supervisé par :

Mr BELHADJ Nidahmeddine

Années scolaire :

2020-2021

## Table des matières

TP1 : Découverte de l'environnement.....	3
1. Installation du logiciel.....	3
2. Création et Configuration de base des projets.....	4
a. Ajout d'une Target Configuration File.....	8
b. Ajout d'une Dsp/bios V5.x Configuration File .....	9
c. Ajout d'un fichier source.....	13
3. Compilation et exécution d'un projet.....	13
TP2 : Calcul du nombre de cycles d'un simple traitement.....	17
1. Inclusion de fichiers : .....	17
2. Calcul du nombre de cycles.....	20
TP3 : Calcul du nombre de cycles d'un traitement d'image .....	22
1. Implémentation du code nécessaire.....	22
2. Calcul du nombre de cycles pour le chargement et la copie d'une image.....	25
TP4 : Optimisation .....	25
1. Génération du code assembleur.....	25
2. Optimisation .....	26
3. Optimisation de structures itératives .....	29
a. Instruction <b>UNROLL()</b> .....	29

## TP1 : Découverte de l'environnement

L'environnement utilisé pendant nos séances de travaux pratiques est celui de Code Composer Studio v4 et le langage de programmation de base exploité est le C.

### 1. Installation du logiciel

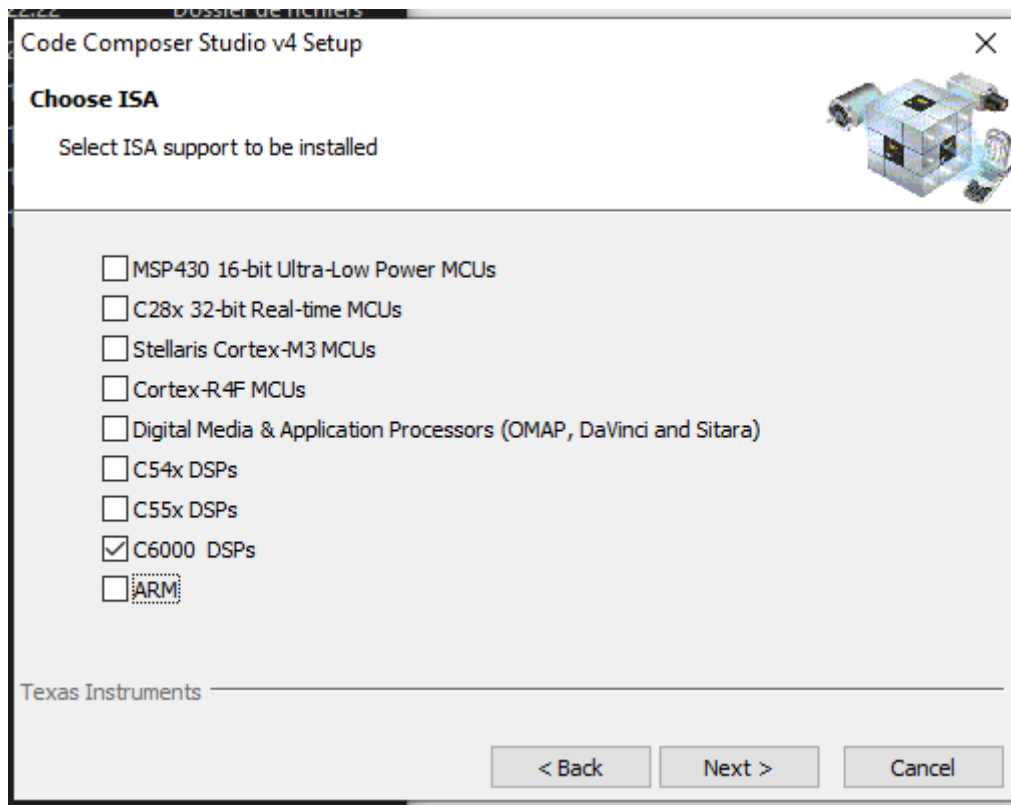
L'installation de Code Composer v4 n'est pas compliquée. Il suffit de suivre les instructions demandées à chaque étape au cours de l'installation.

Mais, il y a deux points que nous avons eu à noter durant nos travaux pratiques :

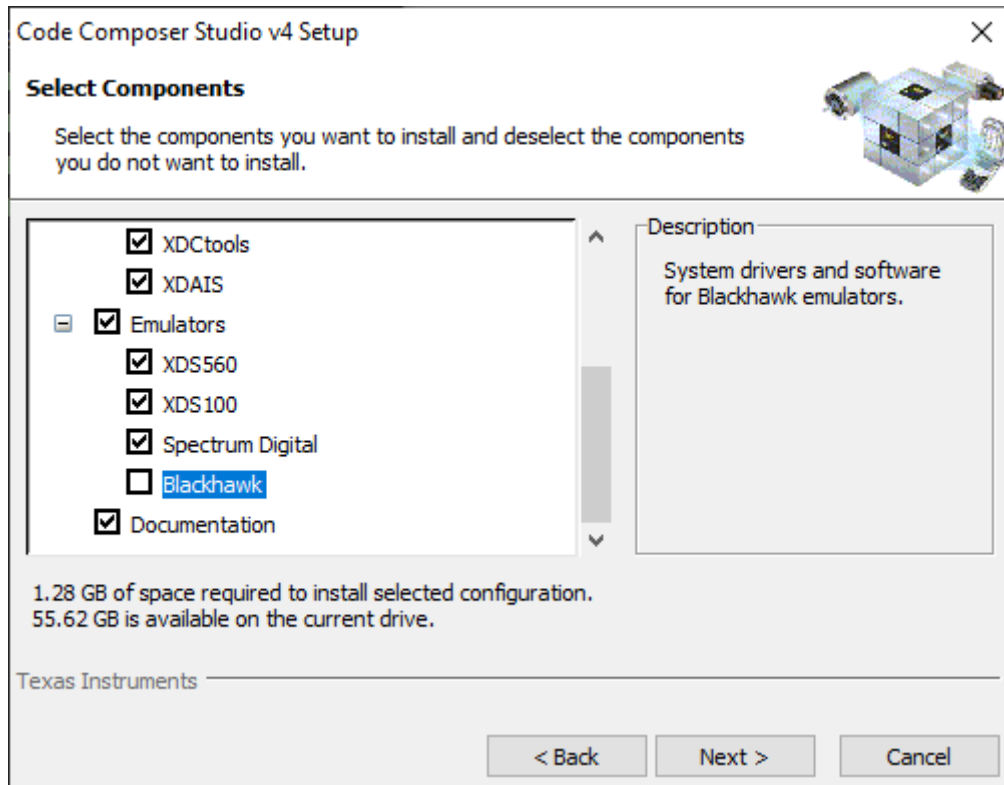
- Le premier point concerne la plateforme à laquelle nous ferons référence dans nos projets.

Ainsi, il vous sera demandé au cours de l'installation de choisir les plateformes concernées. Nos TP tourneront autour de la plateforme DSP C6000.

Il faudra donc décocher toutes les autres options sauf **C600 DSPs** comme illustré ci-dessus, afin d'éviter l'installation de fichiers inutiles.



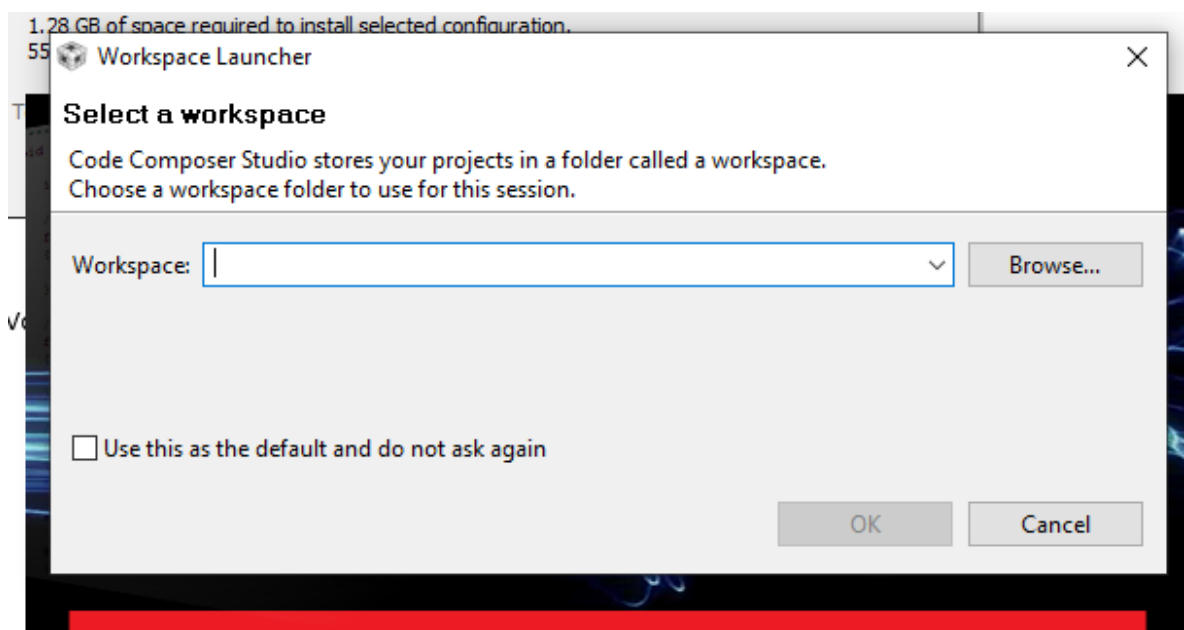
Le second point important concerne les composants à utiliser. Ici, nous retirerons l'émulateur **Blackhawk** de la liste comme illustré ci-dessus :



Vous n'avez plus qu'à attendre la fin de l'installation.

## 2. Création et Configuration de base des projets

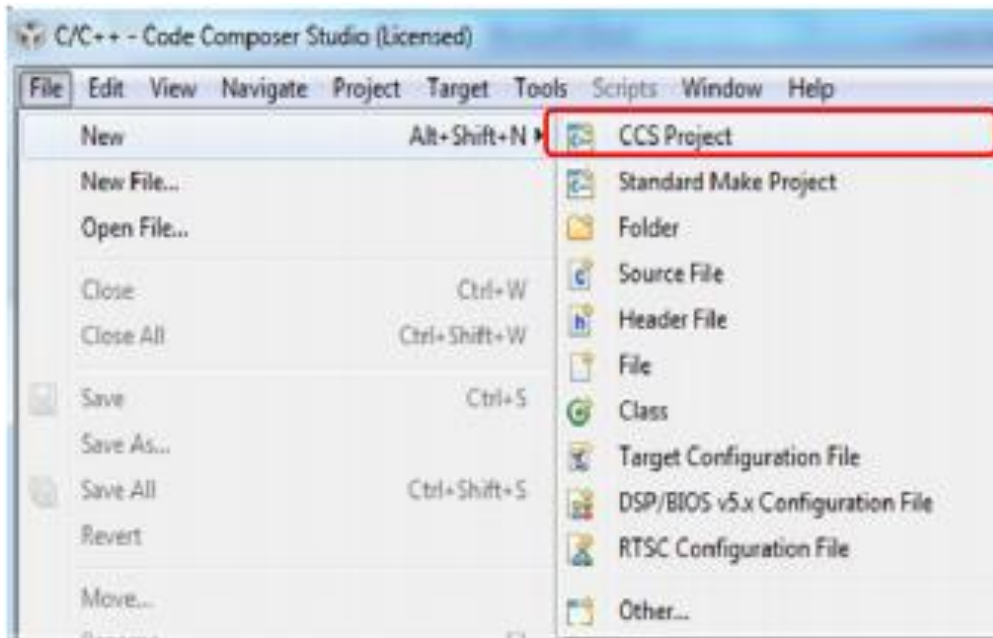
A chacune des ouvertures de Code Composer V4, il vous sera demandé le **répertoire de travail** ou **workspace**. C'est le répertoire dans lequel seront sauvegardés tous nos projets.



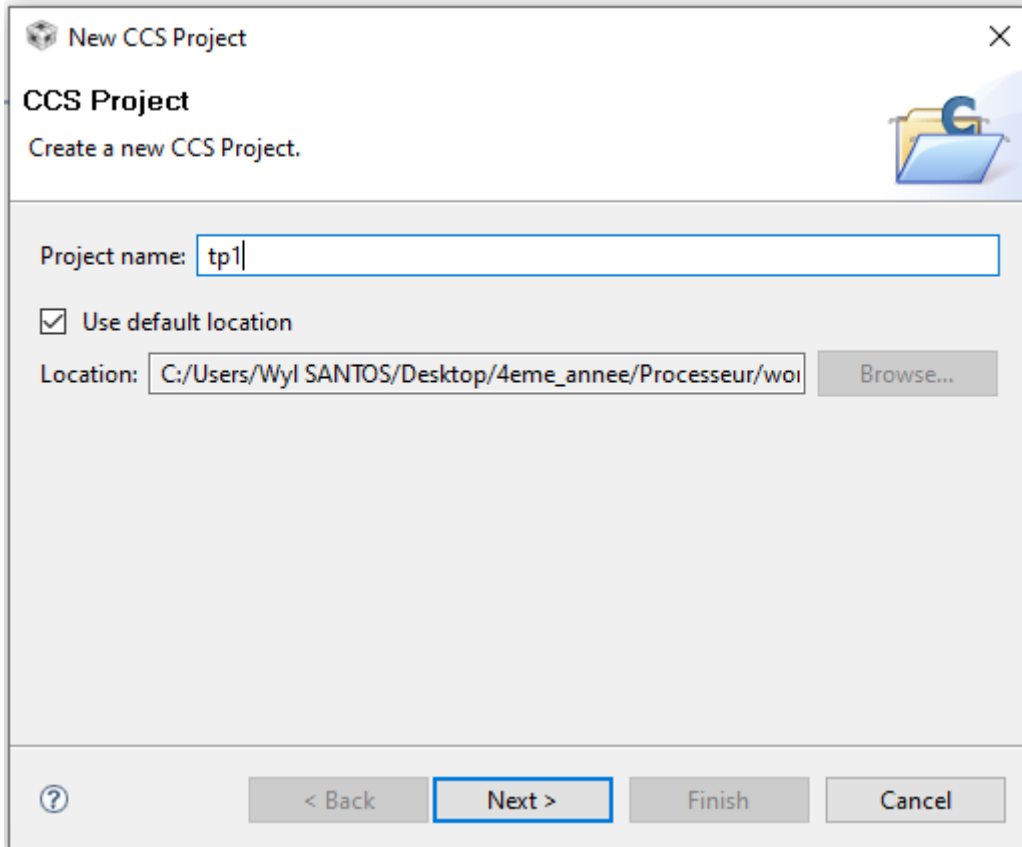
Pour ce premier projet, nous avons effectué la configuration de base de nos projets et réalisé un projet qui affiche un simple message dans la console.

Créons un nouveau projet. Il suffit de faire comme suit :

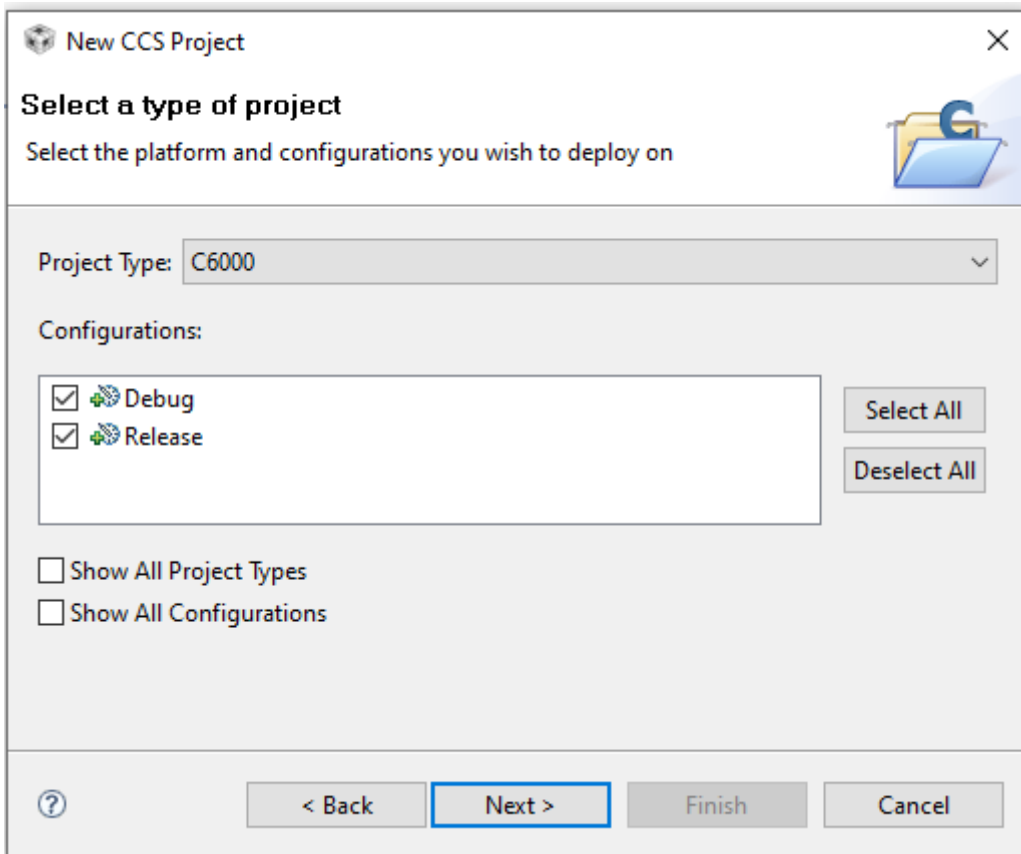
**File -> New -> CCS Project**



La première fenêtre qui s'affiche vous demandera le nom du nouveau projet :



Puis, il suffira de mettre les mêmes options que sur les images suivantes :



**New CCS Project**

**Select a type of project**

Select the platform and configurations you wish to deploy on

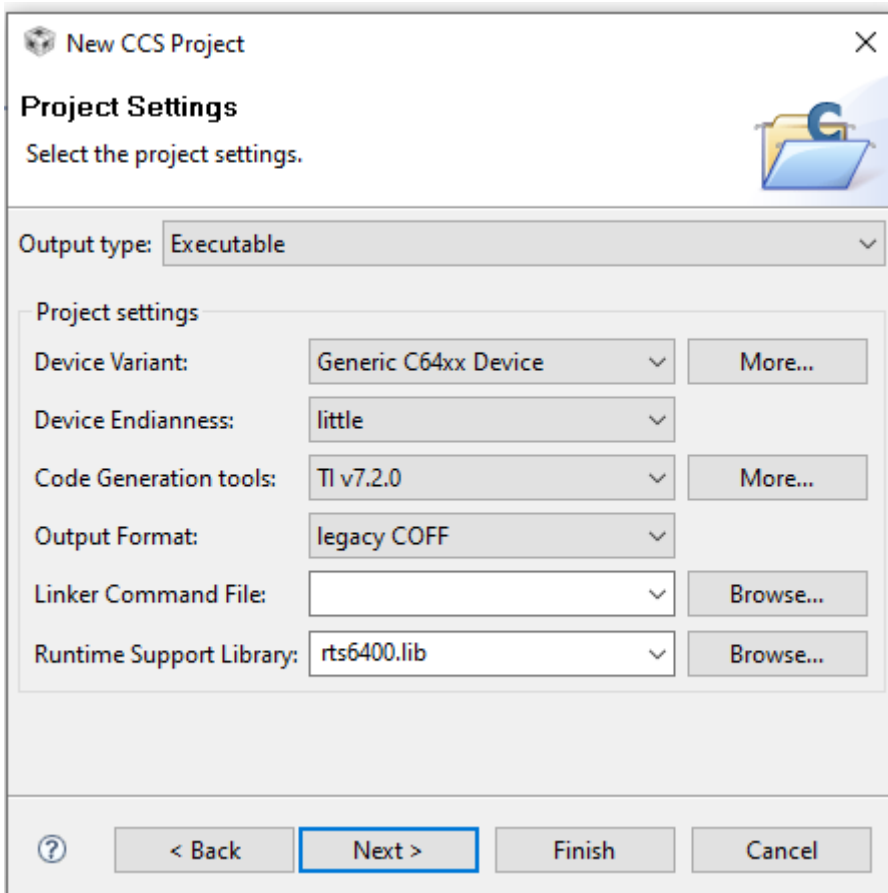
Project Type: C6000

Configurations:

- ☒ Debug
- ☒ Release

☐ Show All Project Types

☐ Show All Configurations



**New CCS Project**

**Project Settings**

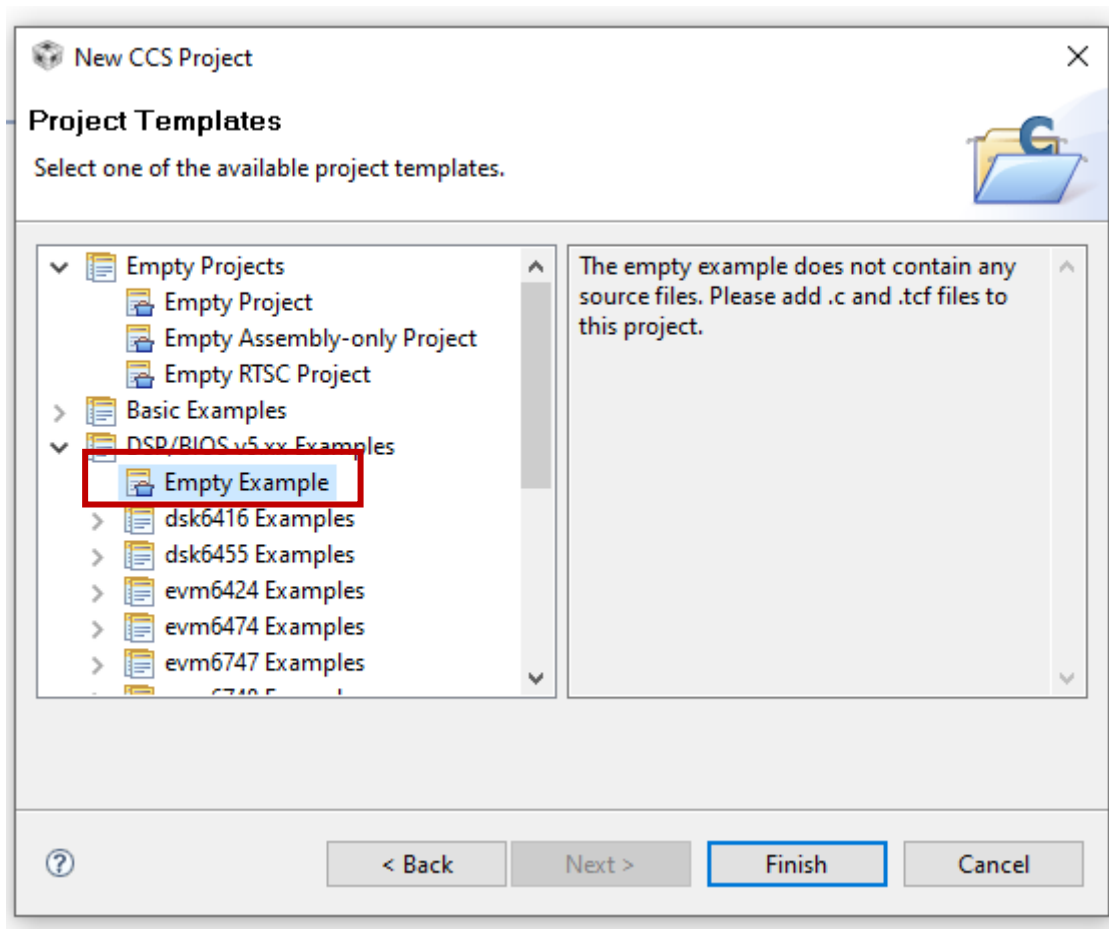
Select the project settings.

Output type: Executable

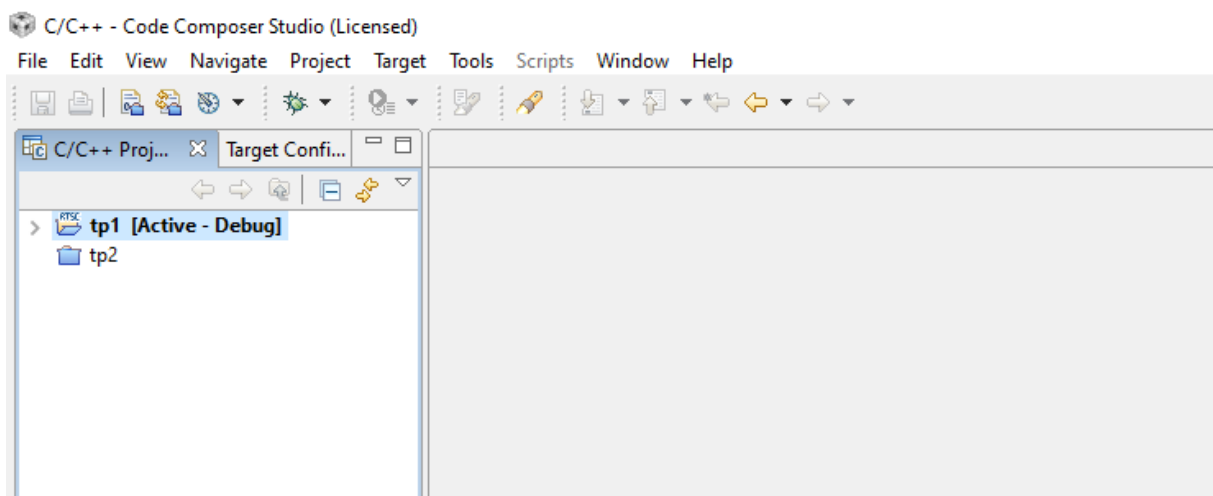
Project settings

Device Variant:	Generic C64xx Device	<input data-bbox="874 1462 1054 1507" type="button" value="More..."/>
Device Endianness:	little	
Code Generation tools:	TI v7.2.0	<input data-bbox="874 1581 1054 1626" type="button" value="More..."/>
Output Format:	legacy COFF	
Linker Command File:		<input data-bbox="874 1700 1054 1744" type="button" value="Browse..."/>
Runtime Support Library:	rts6400.lib	<input data-bbox="874 1818 1054 1863" type="button" value="Browse..."/>

Sur la fenêtre ci-dessus, ne cliquez pas sur le bouton **Finish**. Cliquez plutôt sur le bouton **Next** afin d'obtenir la fenêtre ci-dessous où vous aurez à mentionner le mode de base pour ce nouveau projet



Le nouveau projet étant créé, il est affiché dans la barre verticale à gauche :



Après création d'un nouveau projet, nous devons effectuer les configurations requises.

Pour cela nous devons ajouter trois fichiers :

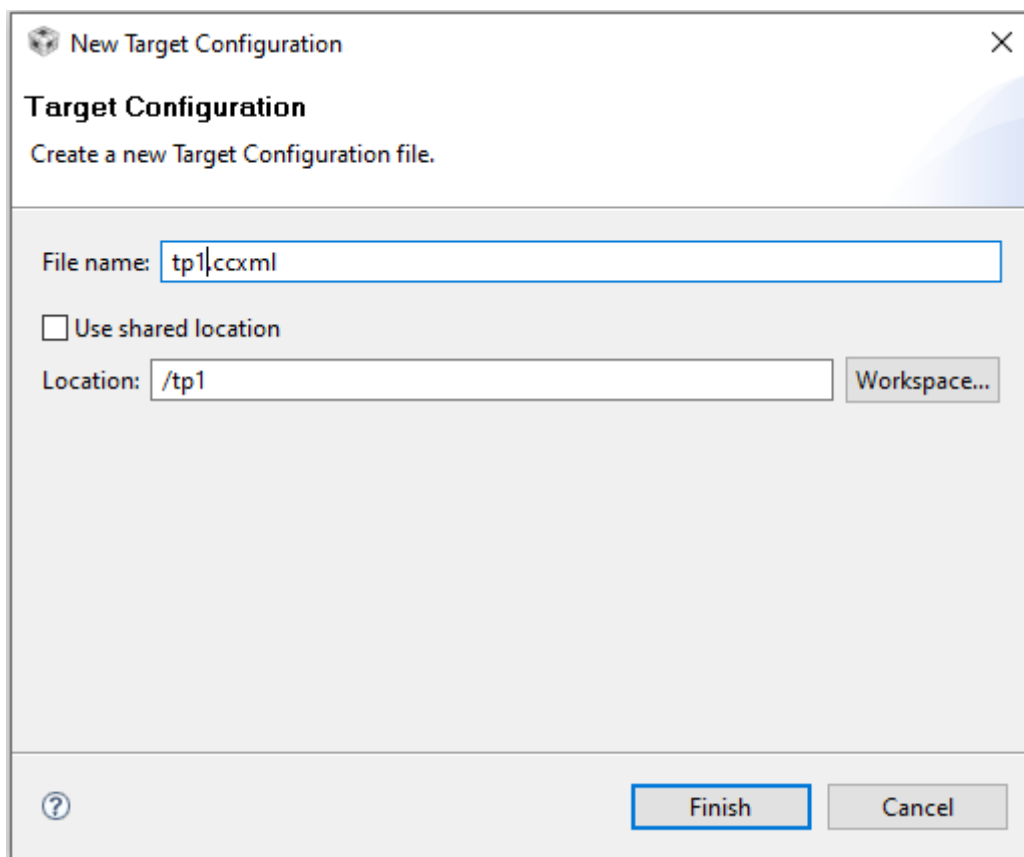
- Une **Target Configuration File**
- Une **Dsp/bios V5.x Configuration File**
- Et bien sûr, les fichiers sources d'extension **.c** et les headers **.h**

a. Ajout d'une Target Configuration File

Pour ajouter une **Target Configuration File**, il suffit de :

- faire un clic droit sur le projet
- sélectionner **New ->Target Configuration File**

Nous obtenons alors la fenêtre suivante où nous sont demandés le nom du fichier (ici **tp1.ccxml**) et son répertoire :



Suite à la création de notre **Target Configuration File**, la fenêtre suivante apparaît dans l'interface de notre environnement :



## Basic

**General Setup**  
This section describes the general configuration about the target.

**Connection:** Texas Instruments Simulator

**Device:** 6416

☐ C6416 Device Cycle Accurate Simulator, Big Endian

☒ C6416 Device Cycle Accurate Simulator, Little Endian

**Advanced Setup**

**Target Configuration:** lists the configuration options for the target.

**Save Configuration**

Save

Comme l'illustre l'image ci-dessus, nous utiliserons comme device, le **6416 Device Cycle Accurate Simulator, Little Endian**. Pour l'obtenir dans la liste, il suffit de saisir **6416**.

Après avoir choisi, il faudra sauvegarder le fichier.

### b. Ajout d'une Dsp/bios V5.x Configuration File

Pour ajouter ce type de fichier, nous avons effectué un clic droit sur notre projet, puis **New-> Dsp/bios V5.x Configuration File**.

Nous avons ainsi obtenu la fenêtre suivante :

New DSP/BIOS Configuration

**DSP/BIOS Configuration**

Specify the TCF file name and the project it should be stored in

Filename: tp1.tcf

Project: tp1 Browse...

< Back Next > Finish Cancel

La fenêtre qui suit nous demande la plateforme à utiliser. Nous utiliserons la **Sim64xx (600)Mhz using 6416 device**. Il suffit de saisir **6416** pour l'obtenir dans la liste de recherche.

New DSP/BIOS Configuration

DSP/BIOS Configuration

Specify a platform

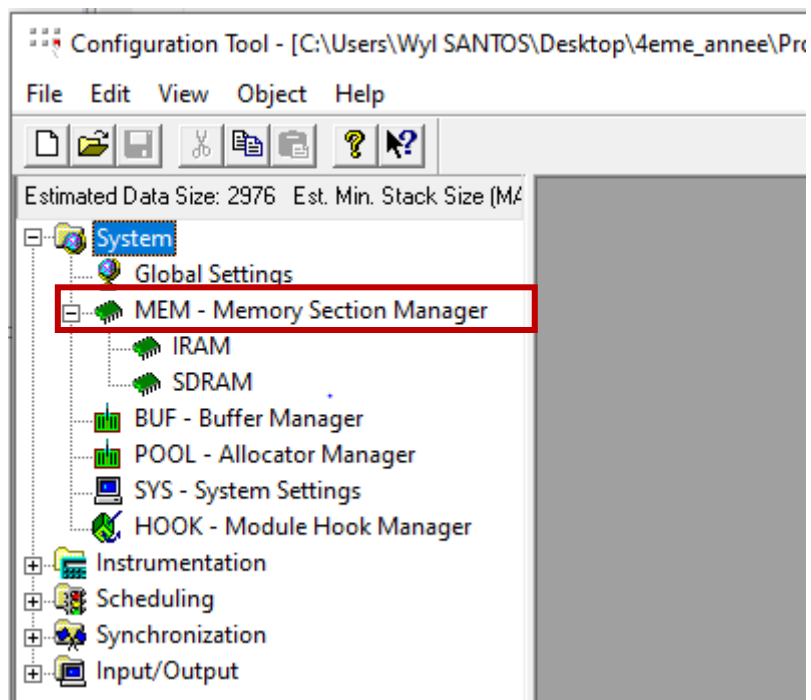
Platform Location: C:/Program Files (x86)/Texas Instruments/xdctools\_3\_20\_08\_88/packages/ti/platforms Modify...

Filter Platforms: 6416 Clear

Name	Description
ti.platforms.sim64xx	Sim64xx (600 Mhz) using 6416 device
ti.platforms.dsk6416	Dsk6416 (720 Mhz) with 16 Mbyte SDRAM

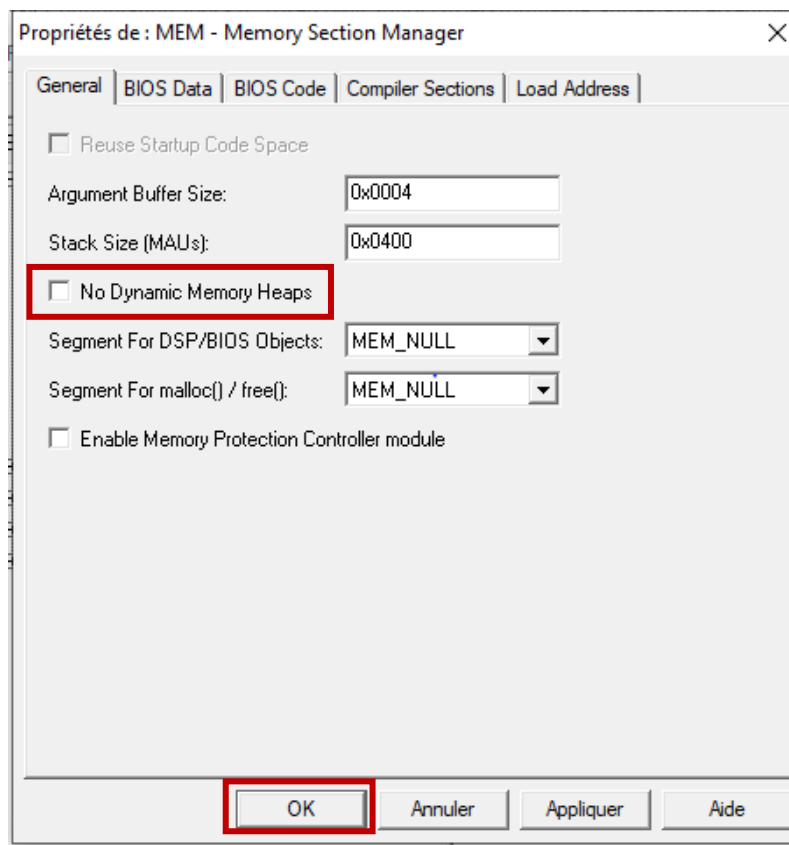
? < Back Next > Finish Cancel

Après création de la **Dsp/bios V5.x Configuration File**, nous devons la configurer. La fenêtre ci-dessus apparaît immédiatement après création du fichier.

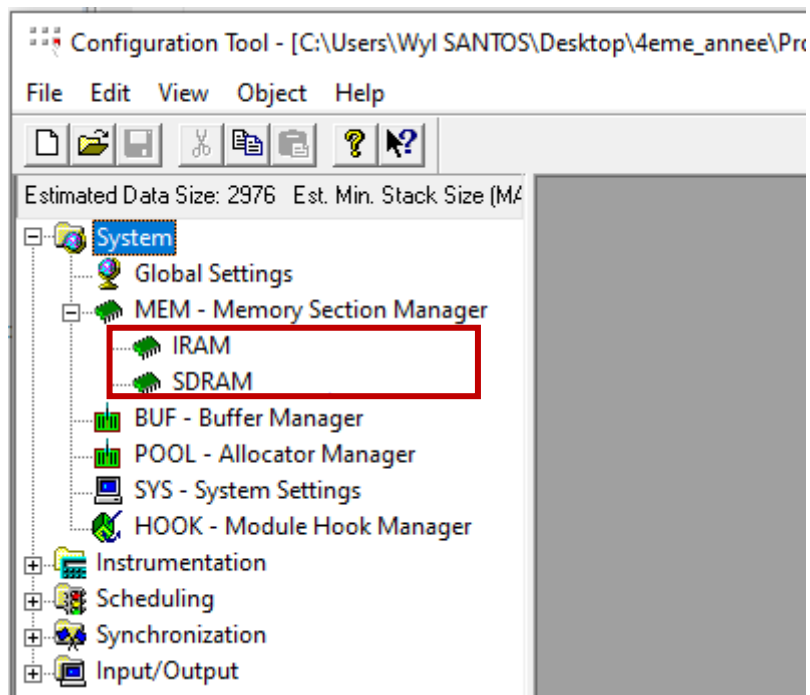


Pour configurer notre **Dsp/bios V5.x Configuration File**, nous devons, dans cet ordre :

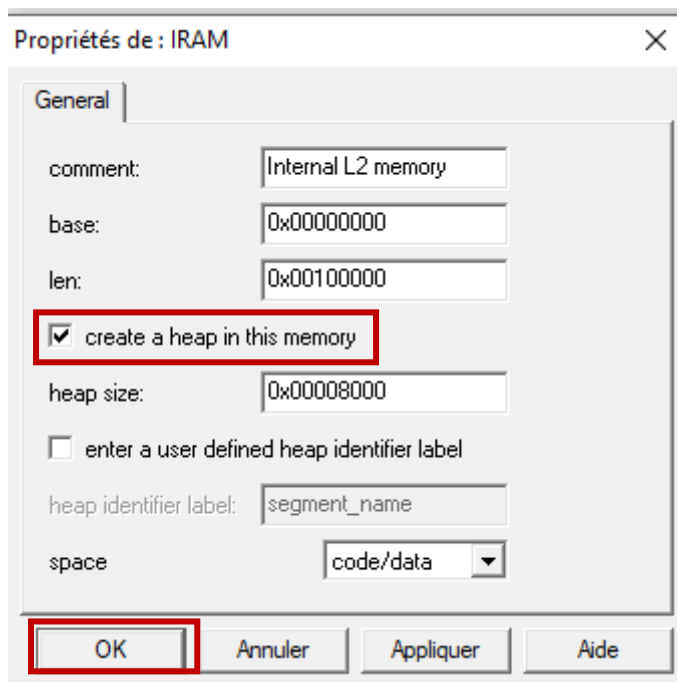
- Faire un clic droit sur **MEM – Memory Section Manager** ->**Propriétés** pour obtenir l’affichage ci-dessous. Par défaut, l’option **No Dynamics Memory Heaps** est activé. Il faut la désactiver puis sauvegarder



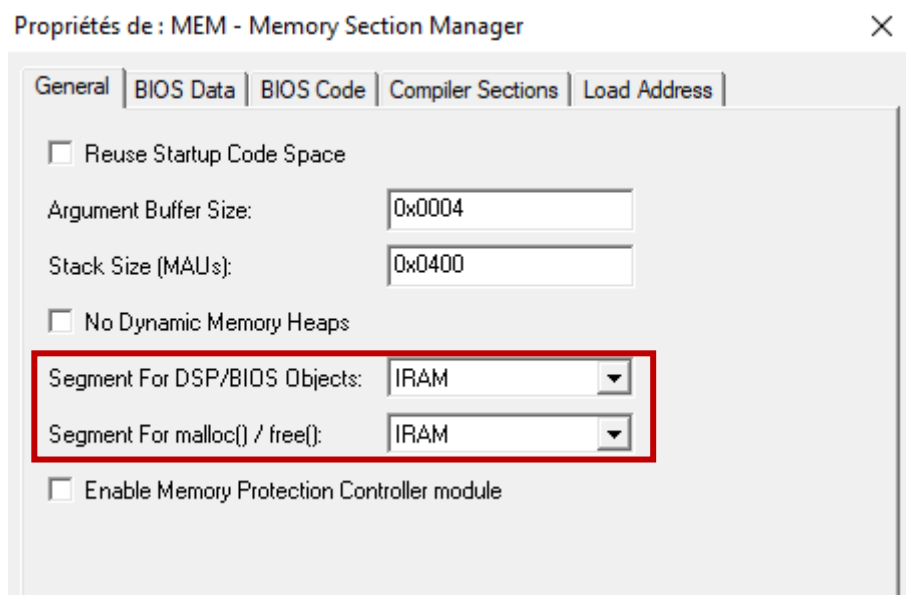
Ensuite, nous effectuons également un clic droit sur **IRAM** et **SDRAM**



Puis, pour chacun d'eux, nous activons l'option **create a heap in this memory** et nous sauvegardons.

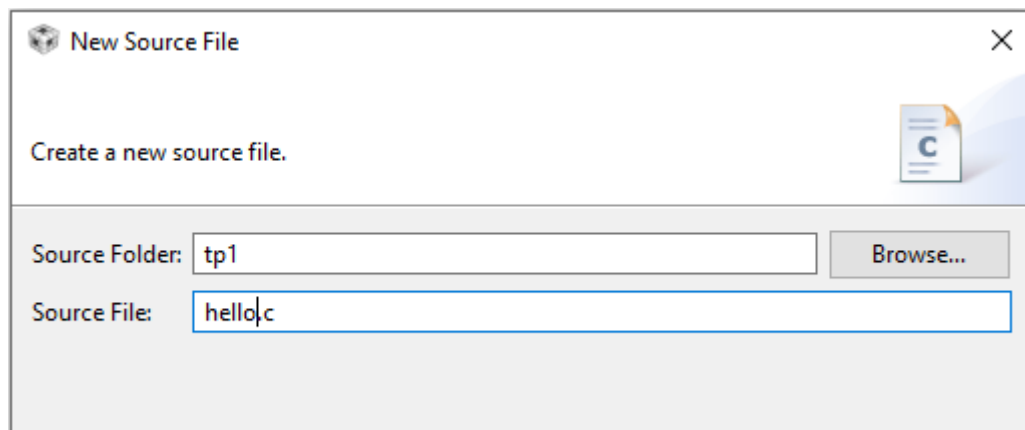


De plus, il faudra retourner dans **Propriétés** de **MEM – Memory Section Manager** pour attribuer **IRAM** pour **Segment for DSP/BIOS Objects** et **Segment for malloc() / free()**.



### c. Ajout d'un fichier source

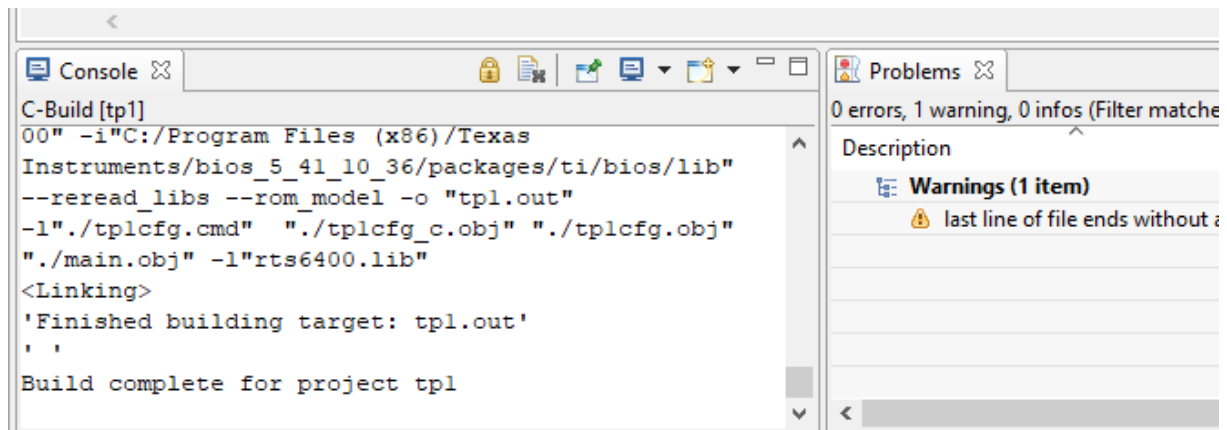
Pour ajouter nos fichiers sources, nous avons eu à effectuer un clic droit sur notre projet, puis **New** -> **Source File**.



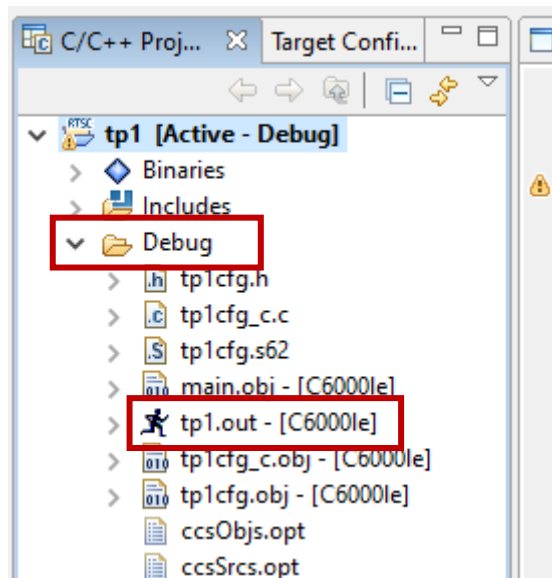
## 3. Compilation et exécution d'un projet

Au cours du TP1, en dehors, des configurations de base, nous avons également effectué la compilation de projet.

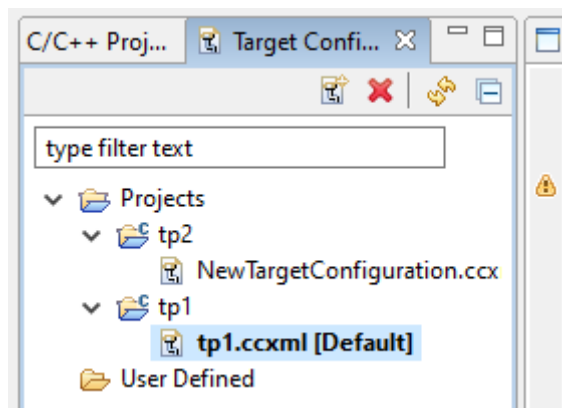
Ainsi, il faut commencer par faire un clic droit sur le projet, puis sélectionner **Build Project**. Si la compilation a été bien effectuée, nous devrions obtenir un affichage tel que celui-ci :



En plus de cet affichage qui signale la fin de la compilation, nous devrions également obtenir un fichier **nomduprojet.out** dans le fichier **Debug** ou **Release** selon le mode choisi. Dans ce TP1, nous sommes en mode **Debug** donc le fichier généré doit se trouver dans un fichier **Debug** :

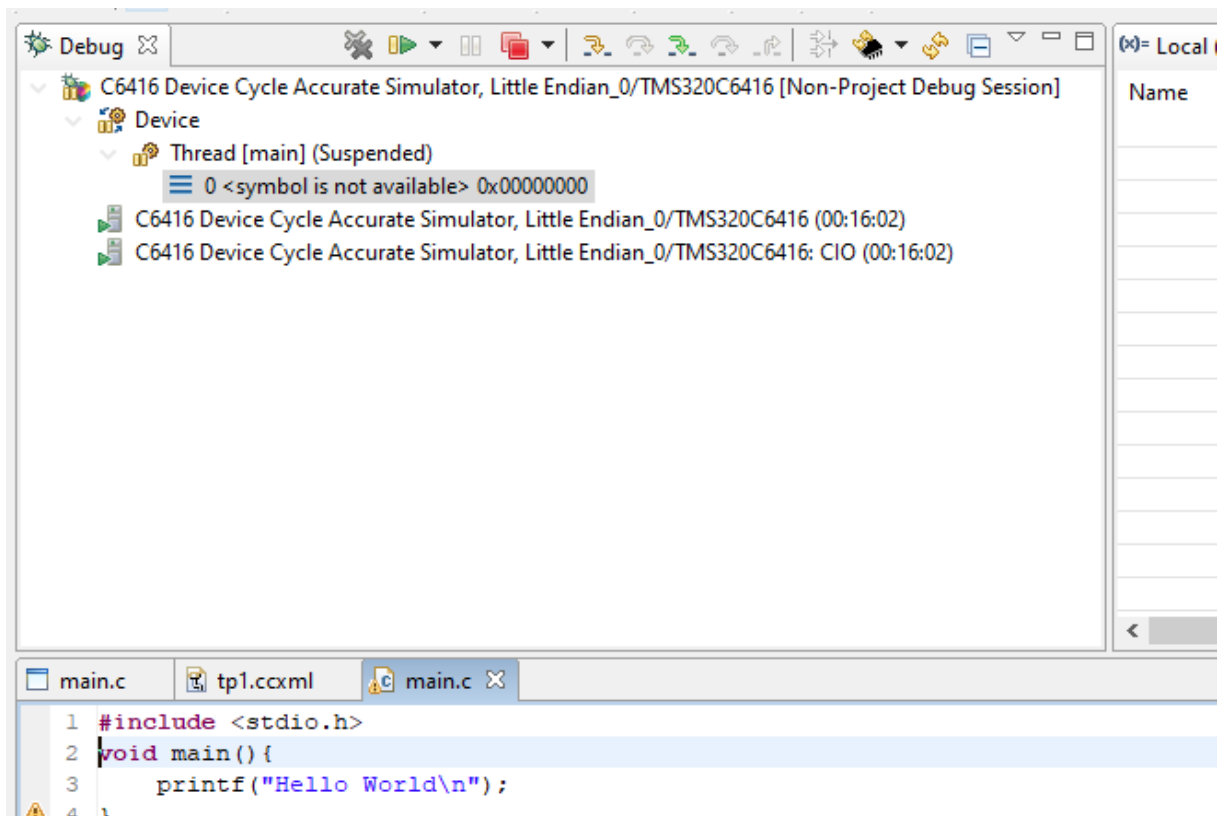


Après cette étape, nous nous rendons dans la **Target Configuration View**.

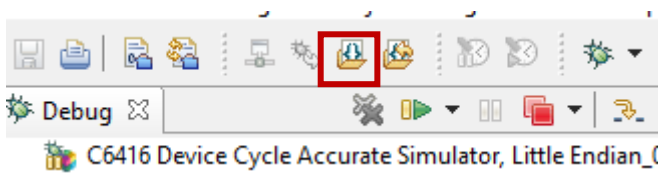


Dans cette interface, nous effectuons un clic droit sur la **Dsp/bios V5.x Configuration File (tp1.ccxml)** de notre projet. Puis, nous cliquons sur **Launch Selected Configuration**.

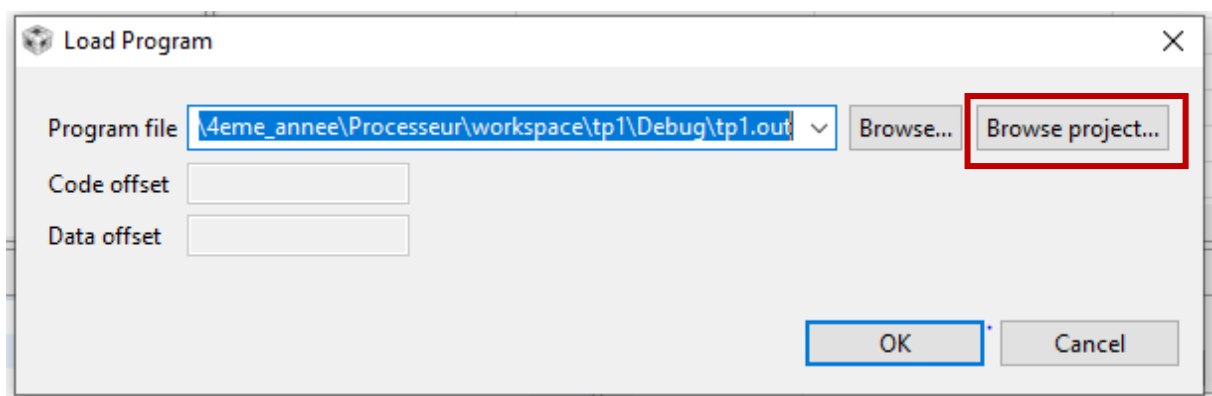
Nous obtenons alors, l'affichage suivant :



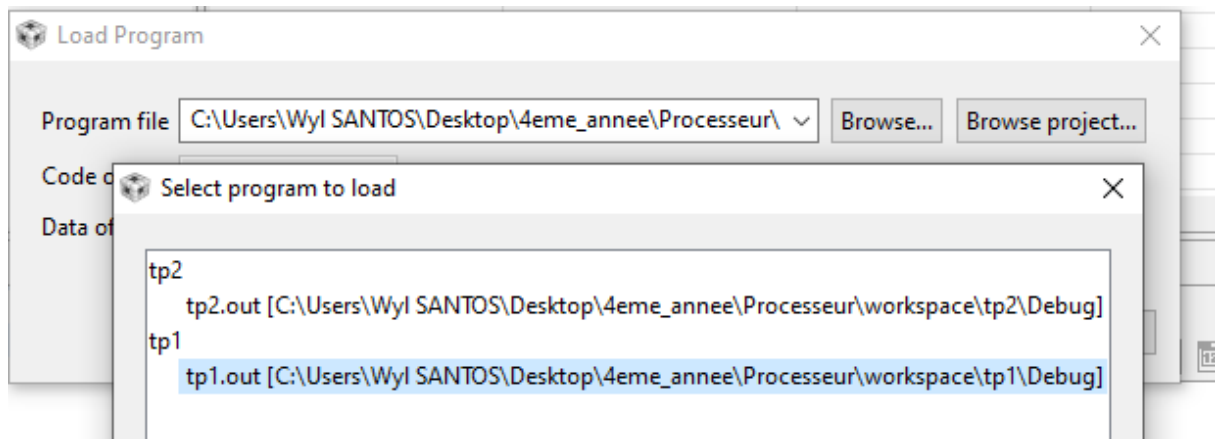
Ici nous devons notre programme dans le **core0**



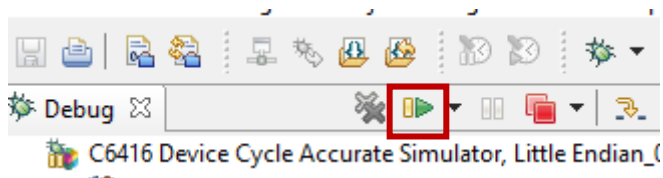
Nous obtenons alors cette boîte de dialogue où nous sommes invités à indiquer le fichier **.out** de notre projet :



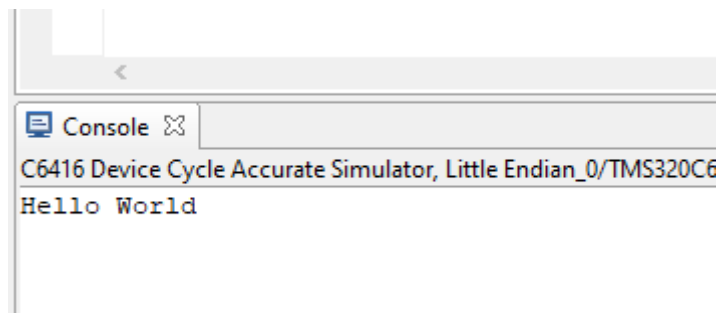
Pour pouvoir le faire facilement, nous nous aidons avec le bouton **browse project**. Et nous choisissons le fichier concerné via la fenêtre qui apparaît ensuite.



Après avoir chargé le fichier **.out**, nous lançons notre programme grâce au bouton **Run** :



Le message '**Hello World**' apparaît alors dans la console pour attester du succès de l'opération.





## TP2 : Calcul du nombre de cycles d'un simple traitement

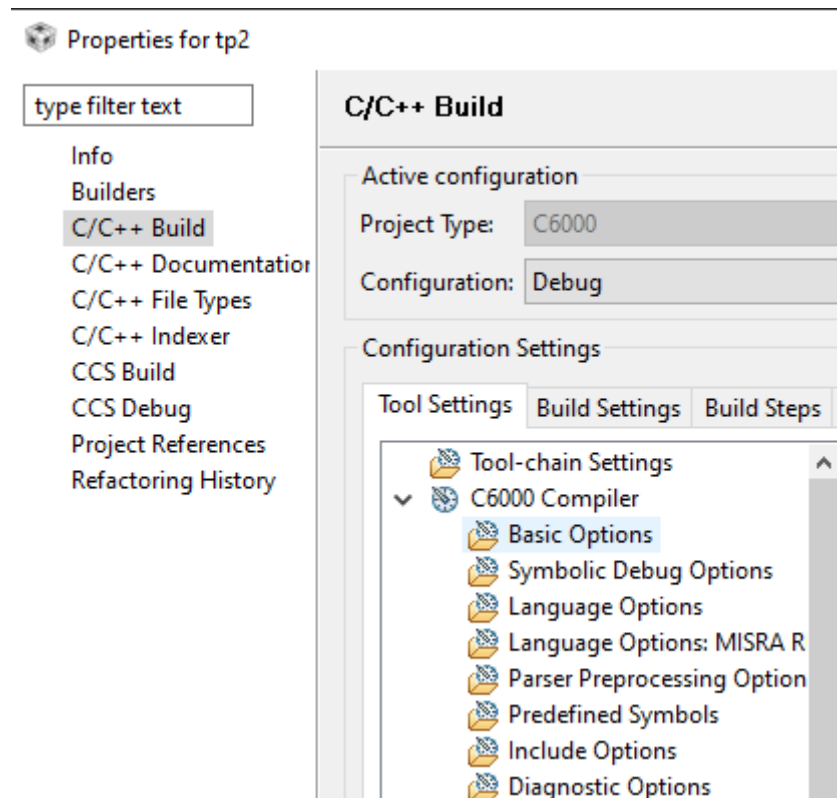
Pour ce TP, nous avons eu besoin de deux fichiers **BIB.h** et **main.c** qui nous ont été fournis par le professeur ainsi que quelques fichiers et répertoires à inclure.

Pour ajouter les fichiers **BIB.h** et **main.c** à notre projet, il nous suffit de faire un **copier-coller** dans le projet.

### 1. Inclusion de fichiers :

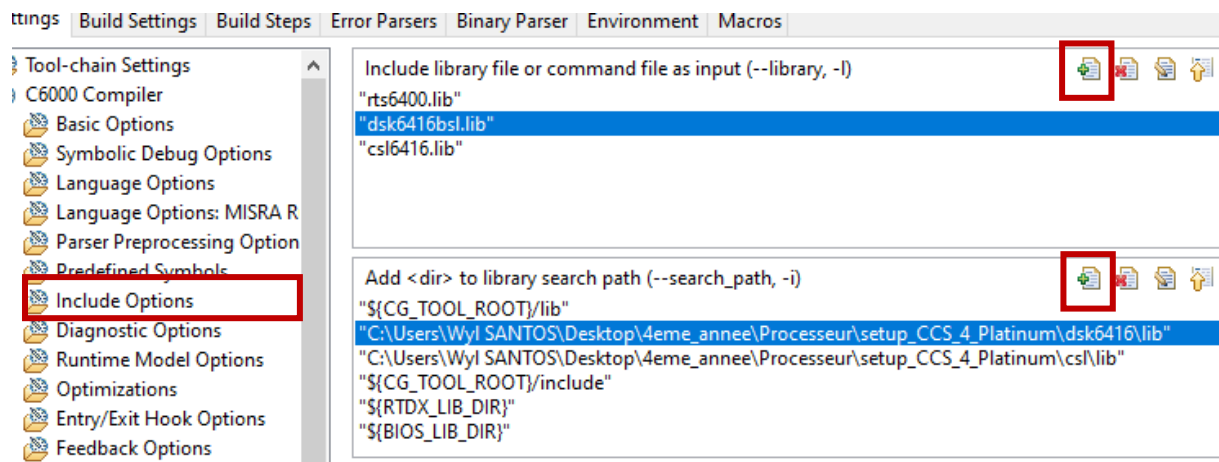
Pour inclure des fichiers ou bibliothèques dans notre projet, nous devons nous rendre dans **Build Properties**. Pour le trouver, nous devons faire un clic droit sur notre projet.

Dans **Build Properties**, nous avons cet affichage :

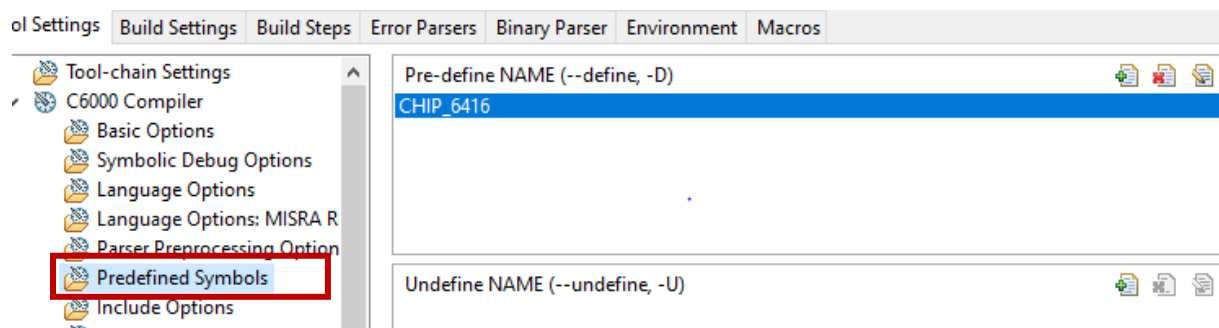


Nous nous intéresserons à **Include Options**, **Predefined Symbols**, **File Search Path**.

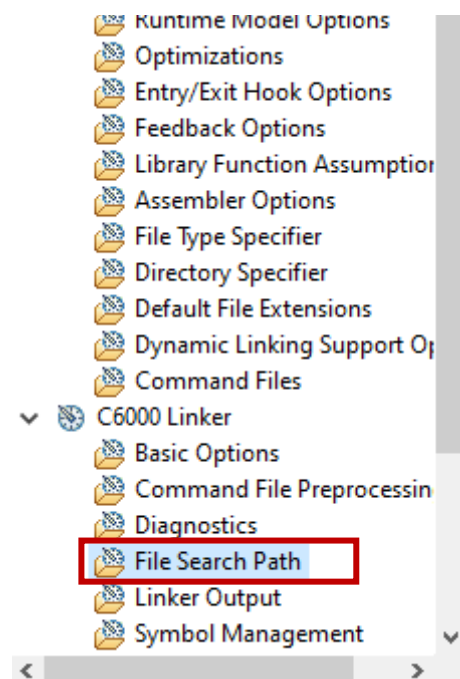
Dans **Include Options**, nous voyons quelques bibliothèques et répertoires comme sur l'image ci-dessous grâce au bouton avec un plus vert à droite

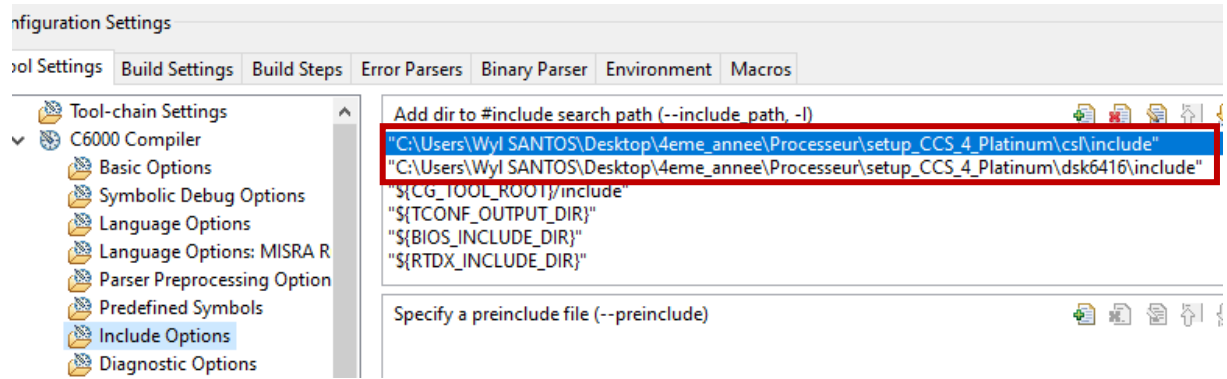


Dans **Predefined Symbols**, nous ajouterons « **CHIP\_6416** »



Et dans **File Search Path**, nous ajouterons des répertoires comme sur la deuxième image ci-dessous





## 2. Calcul du nombre de cycles

Le code que nous avons utilisé est :

```
#include "BIB.h"

void main()
{
    unsigned char src[100];
    unsigned char dst[100];
    int i,N;
    unsigned int start, stop;// timer count store

    hTimer1 = TIMER_open(TIMER_DEV1, NULL);
    DSK6416_init_timer(timer1Ctl,hTimer1);

    // start timer
    TIMER_start( hTimer1);
    start = TIMER_getCount(hTimer1);

    /*****IMPORTANT*****/
    N peut prendre l'une de ces valeurs: 10,20,30,40,50,60,70,80,90,100****/

    printf("donnez N=");
    scanf ("%d",&N);

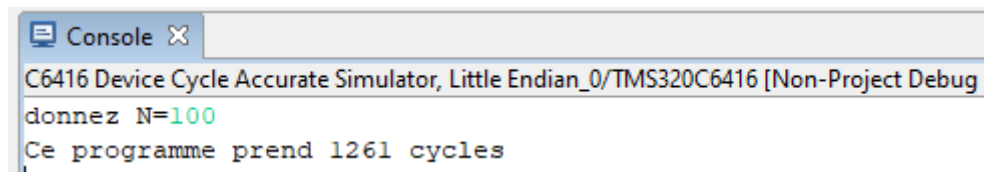
    for(i=0;i<100;i++)
    {
        src[i]=i+20;
    }
    /* for(i=0;i<100;i++)
    {
        src[i]=7*100+i*20;
        printf("%d\n",src[i]);
    }*/

    for(i=0;i<N;i++)
    {
        dst[i]=i;
    }

    stop = TIMER_getCount(hTimer1);
    printf("Ce programme prend %u cycles\n", (stop-start));
}
```

Après avoir effectué toutes les inclusions nécessaires, il suffit de compiler le projet.

Nous obtenons comme résultat :



```
Console X
C6416 Device Cycle Accurate Simulator, Little Endian_0/TMS320C6416 [Non-Project Debug]
donnez N=100
Ce programme prend 1261 cycles
```

The image shows a screenshot of a console window from a TI C6416 simulator. The window title is 'Console X'. The text inside the console shows the simulator's name and version, followed by a prompt 'donnez N=100' where '100' is in green. The final output line is 'Ce programme prend 1261 cycles'.

## TP3 : Calcul du nombre de cycles d'un traitement d'image

### 1. Implémentation du code nécessaire

Ce TP utilisera le contenu du TP2 à savoir les fichiers **BIB.h** et **main.c** ainsi que toutes les inclusions faites.

Ceci est le code initial dans **main.c** c'est-à-dire que le code du TP2 :

```
#include "BIB.h"

void main()
{
    unsigned char src[100];
    unsigned char dst[100];
    int i,N;
    unsigned int start, stop;// timer count store

    hTimer1 = TIMER_open(TIMER_DEVL, NULL);
    DSK6416_init_timer(timer1Ctl,hTimer1);

    // start timer
    TIMER_start( hTimer1);
    start = TIMER_getCount(hTimer1);

    /*****IMPORTANT*****/
    N peut prendre l'une de ces valeurs: 10,20,30,40,50,60,70,80,90,100****/

    printf("donnez N=");
    scanf("%d",&N);

    for(i=0;i<100;i++)
    {
        src[i]=i+20;
    }

    /* for(i=0;i<100;i++)
    {
        src[i]=7*100+i*20;
        printf("%d\n",src[i]);
    }*/

    for(i=0;i<N;i++)
    {
        dst[i]=i;
    }

    stop = TIMER_getCount(hTimer1);
    printf("Ce programme prend %u cycles\n", (stop-start));

}
```

Et voici le code nécessaire à la copie d'une image :

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TAILLE 9485 //Taille de l'image bmp

void main()
{
    int i;
    unsigned char *pixel,*pixel_out, diff=0;

FILE *image_in; //pointeur sur l'image d'entrée
FILE *image_out; //pointeur sur l'image de sortie


pixel=malloc(TAILLE* sizeof(unsigned char)); //Allocation dynamique de la taille de l'image
pixel_out=malloc(TAILLE* sizeof(unsigned char)); //Allocation dynamique de la taille de l'image
memset(pixel,0, TAILLE); //initialisation à zéro de l'espace alloué


image_in = fopen("input1.jpg","rb") ; // ouverture du fichier image d'entrée
image_out= fopen("output.jpg","wb") ; //ouverture du fichier image de sortie


fread(pixel,1,TAILE,image_in); //lecture et stockage de l'image d'entrée dans l'espace mémoire alloué

// copiage de l'entete
for(i=0;i<359;i++)
{
    pixel_out[i]= pixel[i];
}
/*****fin du processus de copiage de l'entete*****/

/****copiage des pixels*****/
for(i=359;i<9485;i++)
{
    pixel_out[i]=pixel[i];
}
/*****fin du processus de copiage des pixels*****/

/*****comparaison entre l'image source et l'image reconstruite*****/
for(i=0;i<9485;i++)
{
    diff+=(pixel_out[i]-pixel[i]);
}

if(diff==0)
    printf("les deux images sont identiques\n");
else
    printf("les deux images sont différentes\n");

/*****fin du processus de comparaison*****/

fwrite(pixel_out,1,TAILE,image_out); //écriture de contenu du pointeur pixel dans le fichier image de sortie

free(pixel); //libérer l'espace mémoire alloué
fclose(image_in);
fclose(image_out);

}

```

Notre objectif étant de déterminer le nombre de cycles nécessaires au chargement et traitement d'une image, nous allons devoir combiner les deux codes :

```
#include "BIB.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TAILLE 9485 //Taille de l'image bmp

void main()
{
    unsigned int start, stop, stop1; // timer count store

    int i;
    unsigned char *pixel,*pixel_out, diff=0;
    FILE *image_in; //pointeur sur l'image d'entrée
    FILE *image_out; //pointeur sur l'image de sortie

    pixel=malloc(TAILLE* sizeof(unsigned char)); //Allocation dynamique de la taille de l'image
    pixel_out=malloc(TAILLE* sizeof(unsigned char)); //Allocation dynamique de la taille de l'image
    memset(pixel,0, TAILLE); //initialisation à zéro de l'espace alloué

    image_in = fopen("input1.jpg","rb") ; // ouverture du fichier image d'entrée
    image_out= fopen("output.jpg","wb") ; //ouverture du fichier image de sortie

    hTimer1 = TIMER_open(TIMER_DEV1, NULL);
    DSK6416_init_timer(timer1Ctl,hTimer1);

    // start timer
    TIMER_start( hTimer1);
    start = TIMER_getCount(hTimer1);

    fread(pixel,1,TAILE,image_in); //lecture et stockage de l'image d'entrée dans l'espace mémoire alloué

    stop1 = TIMER_getCount(hTimer1);
    printf("Le chargement de cette image prend %u cycles\n", (stop1-start));

    // copiage de l'entete
    for(i=0;i<359;i++)
    {
        pixel_out[i]= pixel[i];
    }
    /*****fin du processus de copiage de l'entete*****/

    /*****copiage des pixels*****/
    for(i=359;i<9485;i++)
    {
        pixel_out[i]=pixel[i];
    }
    /*****fin du processus de copiage des pixels*****/

    /*****comparaison entre l'image source et l'image reconstruite*****/
    for(i=0;i<9485;i++)
    {
```



```

    diff+=(pixel_out[i]-pixel[i]);
}

if(diff==0)
    printf("les deux images sont identiques\n");
else
    printf("les deux images sont différentes\n");

/*****fin du processus de comparaison*****/

fwrite(pixel_out,1,TAIILE,image_out); //écriture de contenu du pointeur pixel dans le fichier image de sortie

free(pixel); //libérer l'espace mémoire alloué
fclose(image_in);
fclose(image_out);

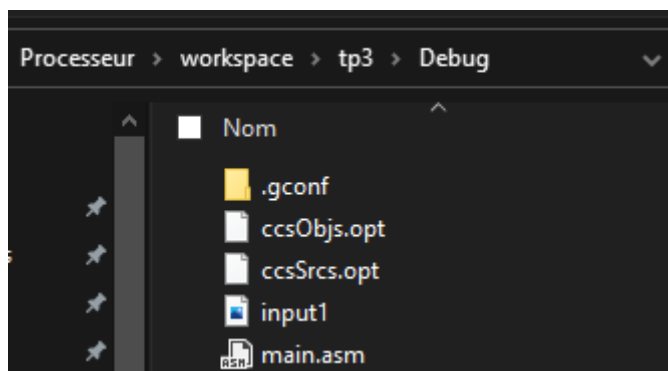
    stop = TIMER_getCount(hTimer1);
    printf("Ce programme prend %u cycles\n", (stop-start));

}

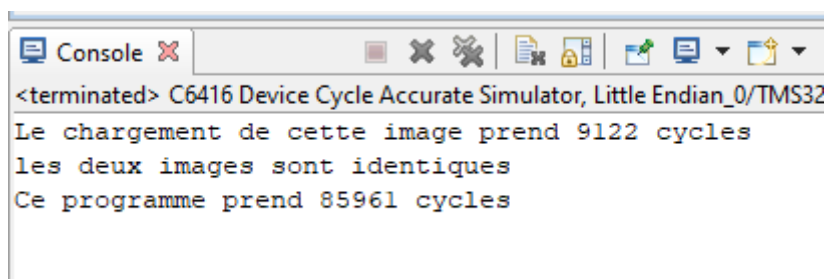
```

## 2. Calcul du nombre de cycles pour le chargement et la copie d'une image

Après compilation du projet, nous obtenons le dossier **Debug**. Incluez-y le fichier **input1.jpg**



Après cela, nous pouvons continuer l'exécution du programme



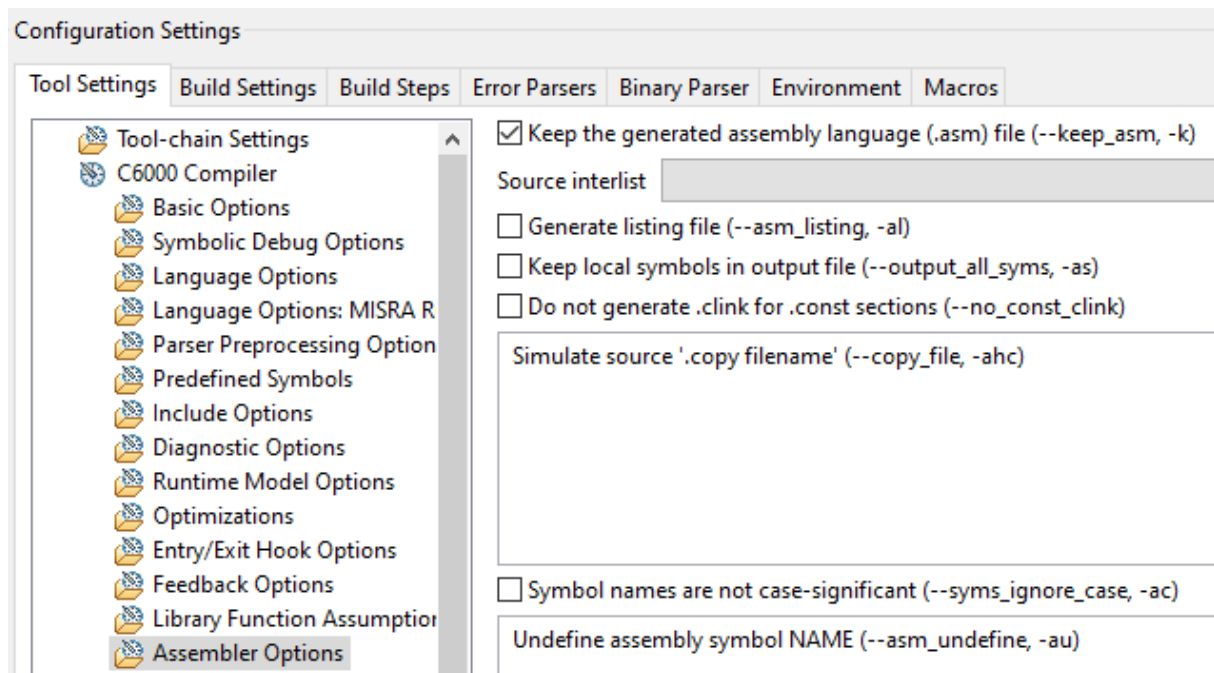
## TP4 : Optimisation

Dans ce TP, nous allons effectuer l'optimisation du programme utilisé dans le TP précédent. En d'autres termes, nous allons réduire le nombre de cycles nécessaires au traitement de ce programme.

### 1. Génération du code assembleur

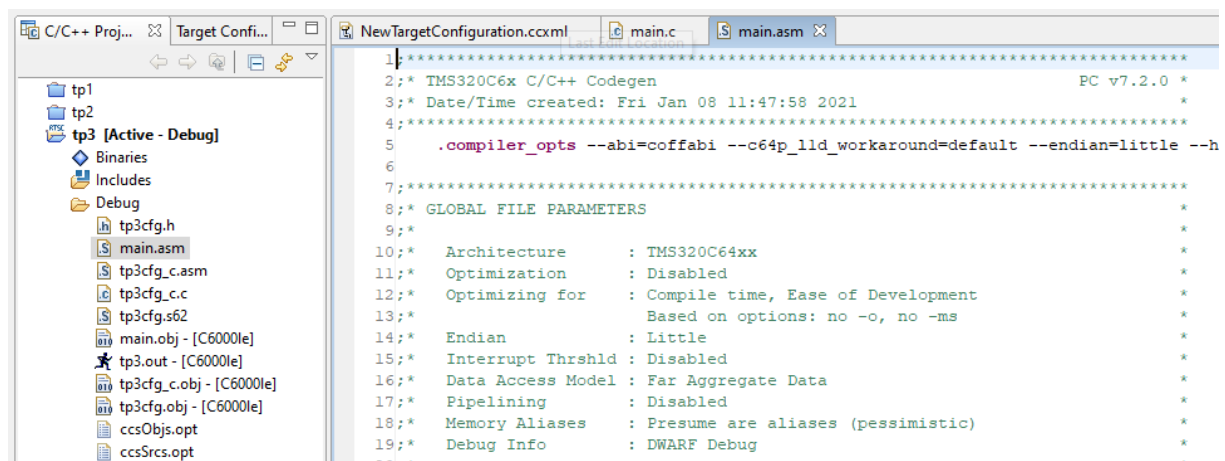
D'abord, vérifions si les options d'optimisation sont activées. Pour cela, rendons nous dans **Build Properties** de notre projet, puis dans la liste des **Tools Settings**, cliquons sur **Assembly Options**.

Dans **Assembly Options**, cochons la case « **Keep the generated assembly language...** ».



Cette option nous permettra de voir le code assembleur généré lors du traitement.

Pour voir ce code, compilons le projet avec **Build Project** et ouvrons le fichier **main.asm** généré dans le dossier **Debug** :

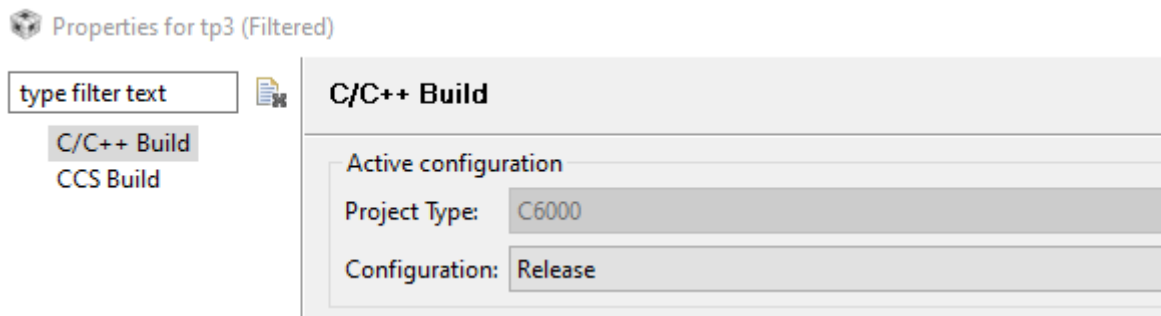


Dans le fichier **main.asm**, nous voyons que l'option **Optimisation** est désactivé(disabled). Nous allons donc l'activer.

## 2. Optimisation

L'optimisation ne s'effectue qu'en mode Release. Nous allons donc passer en mode Release.

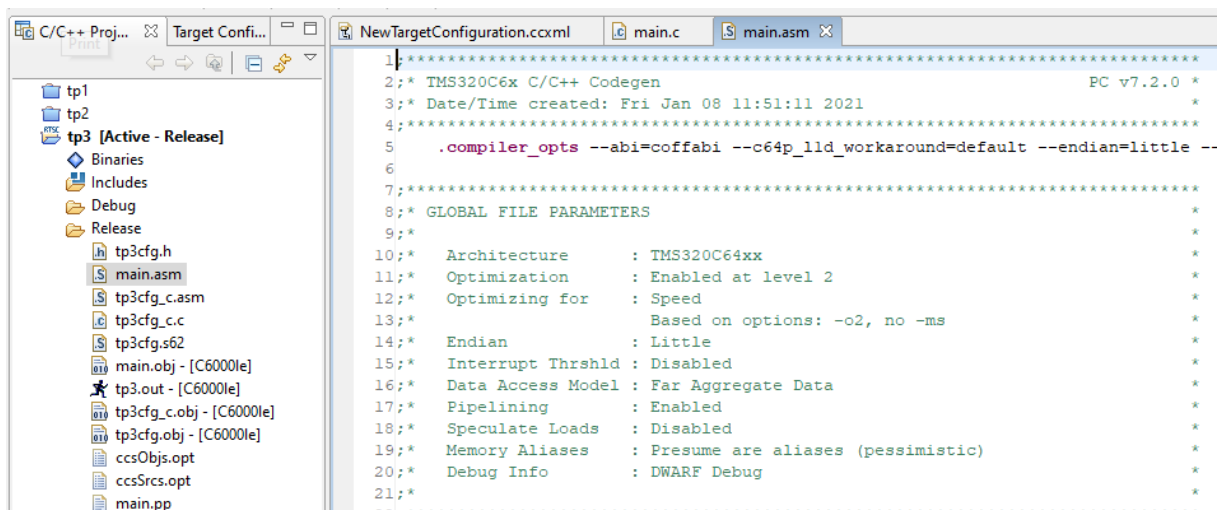
Pour le faire, nous nous rendons le **Build Properties** de notre projet, puis effectuons le changement dans la barre **Configuration** :



Après être passé, il est impératif de refaire les mêmes inclusions qu'au TP précédent. En effet, toutes ces inclusions n'avaient été faites qu'en mode **Debug**, donc elles ne sont pas appliquées au mode **Release**.

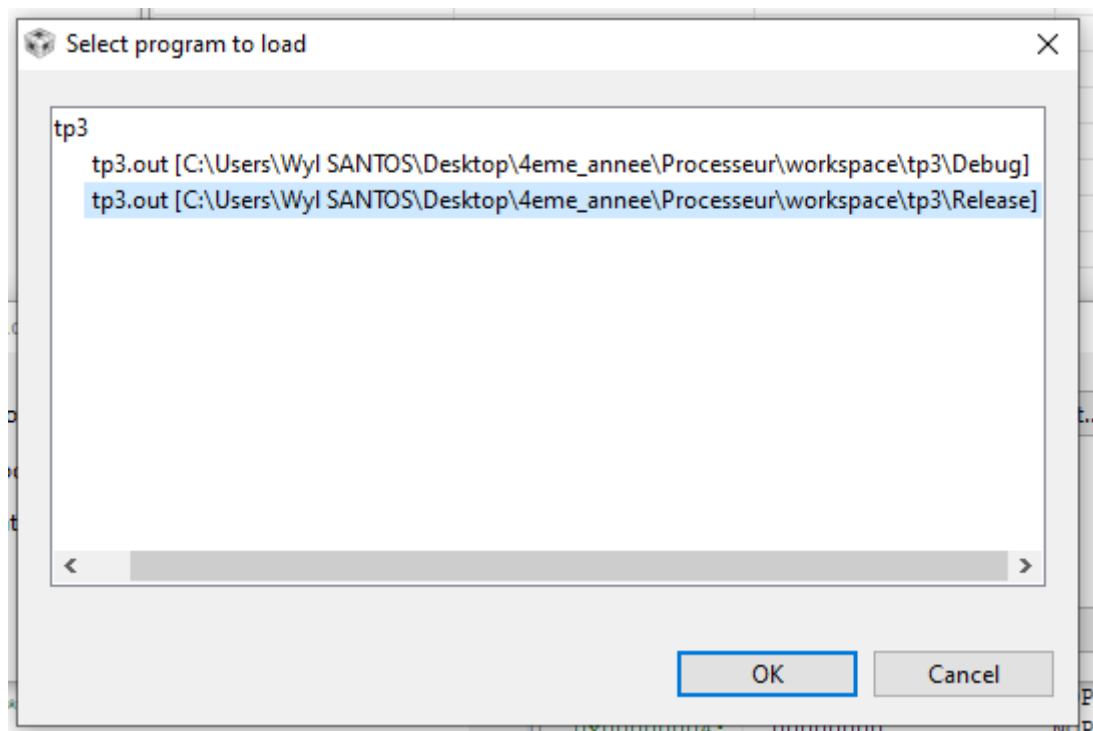
Il est également important de ne pas oublier de cocher la case « **Keep the generated assembly language...** » comme effectués plus haut dans le mode **Debug** sinon le fichier **main.asm**.

Après avoir effectué les différentes inclusions, compilons le projet pour voir l'état des options d'optimisation



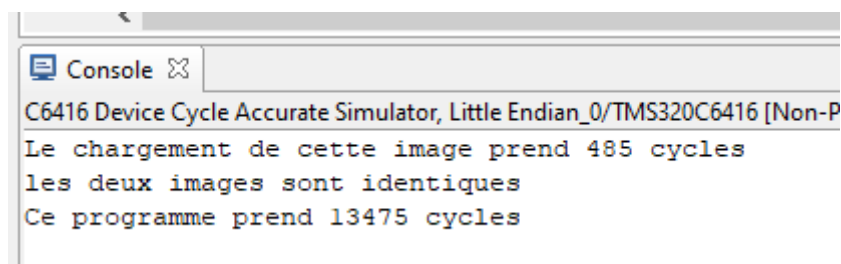
Nous voyons que **Optimization** est passé à l'état « **Enabled at level 2** ». Voyons les conséquences sur notre traitement.

Exécutons notre programme :

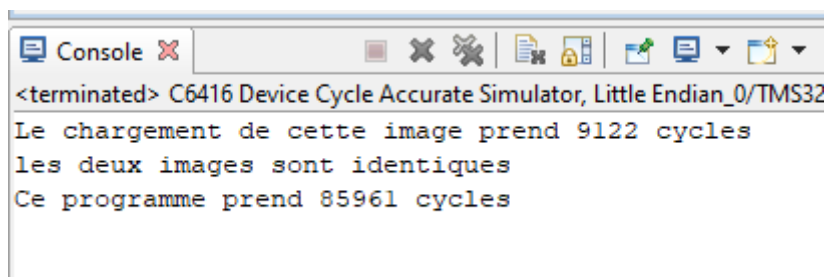


Il est à noter que le fichier à charger cette fois-ci dans le mode **Release** est le fichier **.out** du dossier **Release**.

Voici les nombres de cycles que nous obtenons après optimisation :



Sans optimisation, nous avons :



La différence est flagrante :

- Chargement de l'image : sans optimisation **9122** vs avec optimisation **485**
- Traitement total : sans optimisation **85961** vs avec optimisation **13475**

En terme de pourcentage, nous aurions eu un gain de :

$$\text{Gain} = \frac{\text{Temps initial} - \text{Temps optimisé}}{\text{temps initial}} \times 100$$

$$\text{Gain} = \frac{85961 - 13475}{85961} \times 100$$

**Gain=84,32%**

Ainsi, nous avons eu un gain de près de **84%**.

Mais, il est à noter que cette optimisation est faite automatiquement par le compilateur en mode **Release**.

### 3. Optimisation de structures itératives

Il est possible d'optimiser le traitement de structures itératives ou boucles par déroulage. Le déroulage consiste en la réduction du nombre d'itérations nécessaires pour remplir toutes les conditions de la boucle sans en affecter les résultats.

Pour effectuer ce type d'optimisation, nous utilisons les instructions de type **pragma**. Cette directive du préprocesseur offre deux fonctions capables d'effectuer le déroulage de boucles : **UNROLL()** et **MUST\_ITERATE()**.

#### a. Instruction UNROLL()

Pour ce TP, nous utiliserons ce code :

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include "BIB.h"
5
6  #define TAILLE 9485 //Taille de l'image bmp
7
8  void main()
9  {
10     int i;
11     unsigned char *pixel,*pixel_out, diff=0;
12     unsigned int start, stop;
13
14     FILE *image_in; //pointeur sur l'image d'entrée
15     FILE *image_out; //pointeur sur l'image de sortie
16
17
18     hTimer1 = TIMER_open(TIMER_DEV1, NULL);
19     DSK6416_init_timer(timer1Ctl,hTimer1);
20     TIMER_start( hTimer1);
21
22
23     pixel=malloc(TAILLE* sizeof(unsigned char)); //Allocation dynamique de la taille de l'image
24     pixel_out=malloc(TAILLE* sizeof(unsigned char)); //Allocation dynamique de la taille de l'image
25     memset(pixel,0, TAILLE); //initialisation à zéro de l'espace alloué
26
27
28
29     image_in = fopen("input1.jpg","rb") ; // ouverture du fichier image d'entrée

```

```

image_out= fopen("output.jpg","wb") ;//ouverture du fichier image de sortie

fread(pixel,1,TAILLE,image_in);//lecture et stockage de l'image d'entrée dans l'espace mémoire all

// copiage de l'entete
// start timer

start = TIMER_getCount(hTimer1);
#pragma UNROLL(160);
for(i=0;i<359;i++)
{
    pixel_out[i]= pixel[i];
}
/*****fin du processus de copiage de l'entete*****/

/*****copiage des pixels*****/
/*for(i=359;i<9485;i++)
{
    pixel_out[i]=pixel[i];
}
/*****fin du processus de copiage des pixels*****/

/*****comparaison entre l'image source et l'image reconstruite*****/
/*for(i=0;i<9485;i++)
{
    diff+= (pixel_out[i]-pixel[i]);
}

```


Nous appliquerons l'instruction **pragma** sur cette boucle :

```



/*****copiage des pixels*****/
for(i=359;i<9485;i++)
{
    pixel_out[i]=pixel[i];
}

```

Tout d'abord, passons en mode **Release** :

 **tp4 [Active - Release]**

Maintenant, déterminons le nombre de cycles initial nécessaires au traitement de cette boucle :

 Console 

C6416 Device Cycle Accurate Simulator, Little Endian\_0/TMS

Ce programme prend 9569 cycles

Ainsi, cette boucle, sans optimisation de type **pragma**, nécessite **18884 cycles**.

Maintenant, essayons de l'optimiser avec une instruction **pragma**. L'instruction complète à utiliser est **#pragma UNROLL(nombre de déroulage)** et doit être placé au-dessus de la boucle concerné :

```

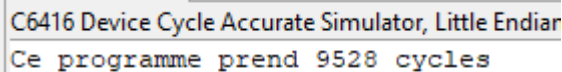
// copiage de l'entete
#pragma UNROLL(nombre de deroulage)
for(i=0;i<359;i++)
{
pixel_out[i]= pixel[i];
}

```

Afin d'effectuer une optimisation optimale, le choix du nombre de déroulage est très important. Il représente le nombre d'itérations de cette boucle nécessaire à l'obtention du temps le plus optimisé sans pour autant en affecter les résultats.

Pour déterminer ce nombre, nous procéderons par tâtonnement :

- 160 :

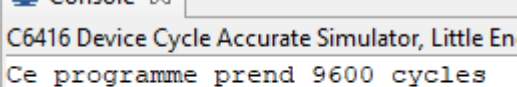


```

C6416 Device Cycle Accurate Simulator, Little Endian
Ce programme prend 9528 cycles

```

- 150 :

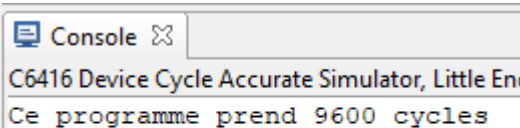


```

C6416 Device Cycle Accurate Simulator, Little Endian
Ce programme prend 9600 cycles

```

- 161 :

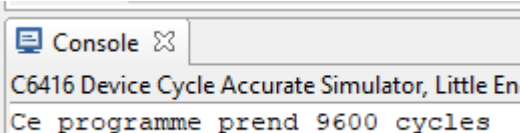


```

C6416 Device Cycle Accurate Simulator, Little Endian
Ce programme prend 9600 cycles

```

- 159 :



```

C6416 Device Cycle Accurate Simulator, Little Endian
Ce programme prend 9600 cycles

```

Ainsi, **160** est le nombre de déroulage pour lequel nous avons le temps de traitement le plus faible (avec pragma 160 : **9528** vs sans pragma : **9569**).

Nous avons pu faire un gain de **0.42%**