


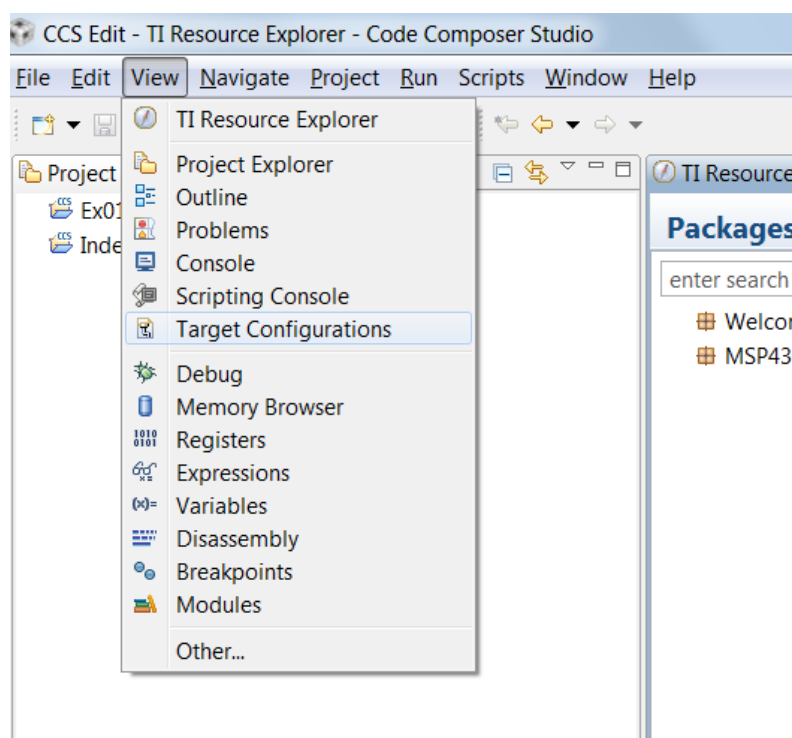
Laboratoire 2 : Introduction à la carte DSP et à sa programmation


Le but de ce laboratoire est de se familiariser avec l'utilisation du logiciel Code Composer Studio et à l'utilisation de la carte TMS320C6713. Il fait suite au cours et permet de rentrer concrètement dans la programmation en langage C et assembleur. Le logiciel a été installé sur http://processors.wiki.ti.com/index.php/Download_CCS et il s'agit de la version 5.2.1.00018.

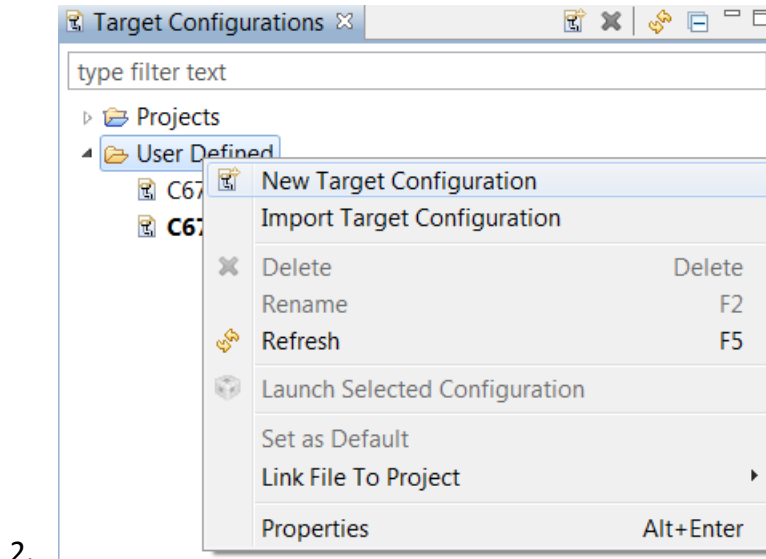
Nous commencerons par voir une brève introduction à la carte DSP su CCS, ensuite nous verrons la programmation en elle-même pour enfin étudier les performances des programmes.



I. Introduction à la carte DSK TMS320C6713

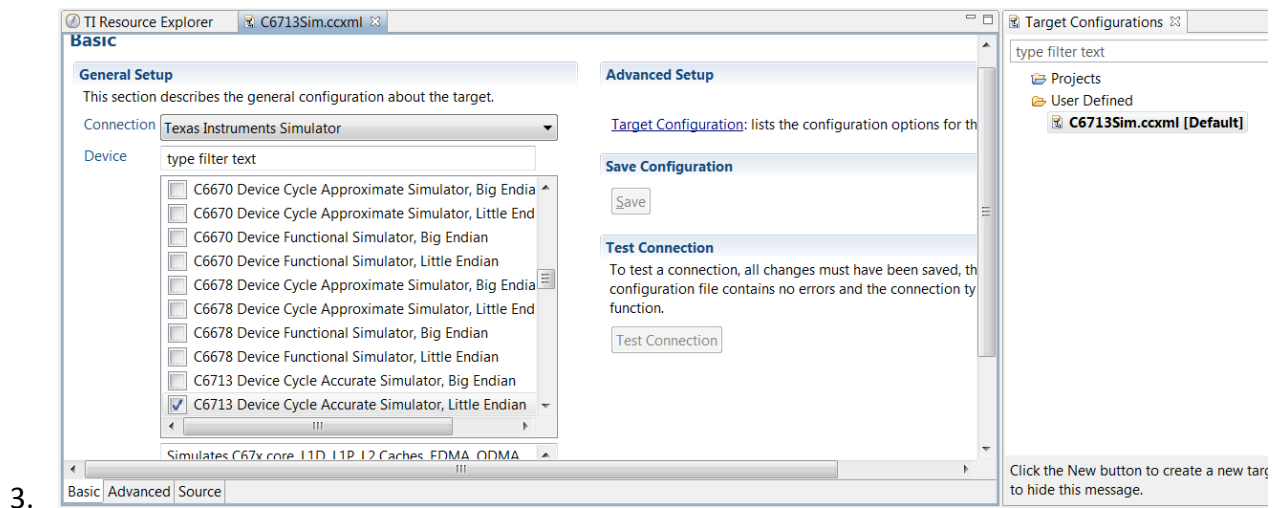
Avant de pouvoir se lancer dans la programmation, il est nécessaire de passer par toute sorte de configuration pour rendre la carte DSP compatible avec le logiciel. On commence donc par définir la carte utilisée dans  **Target Configurations** après avoir bien sûr lancer le logiciel...



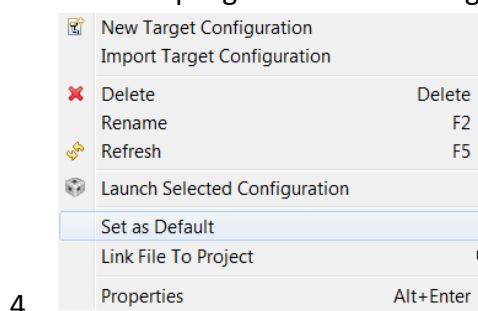
Sur la nouvelle fenêtre, on clique droit sur  **New Target Configuration** et on crée avec le nom voulu.



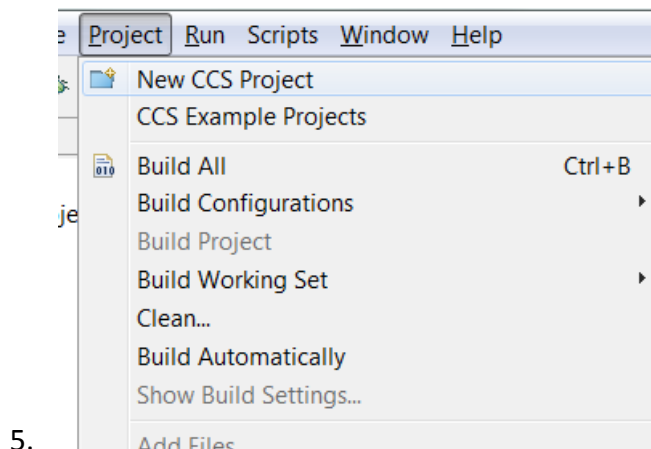
On double clique sur la nouvelle configuration que l'on vient de créer (toujours dans la fenêtre  **Target Configurations**) et on sélectionne la bonne carte parmi la liste disponible. Sans oublier de .



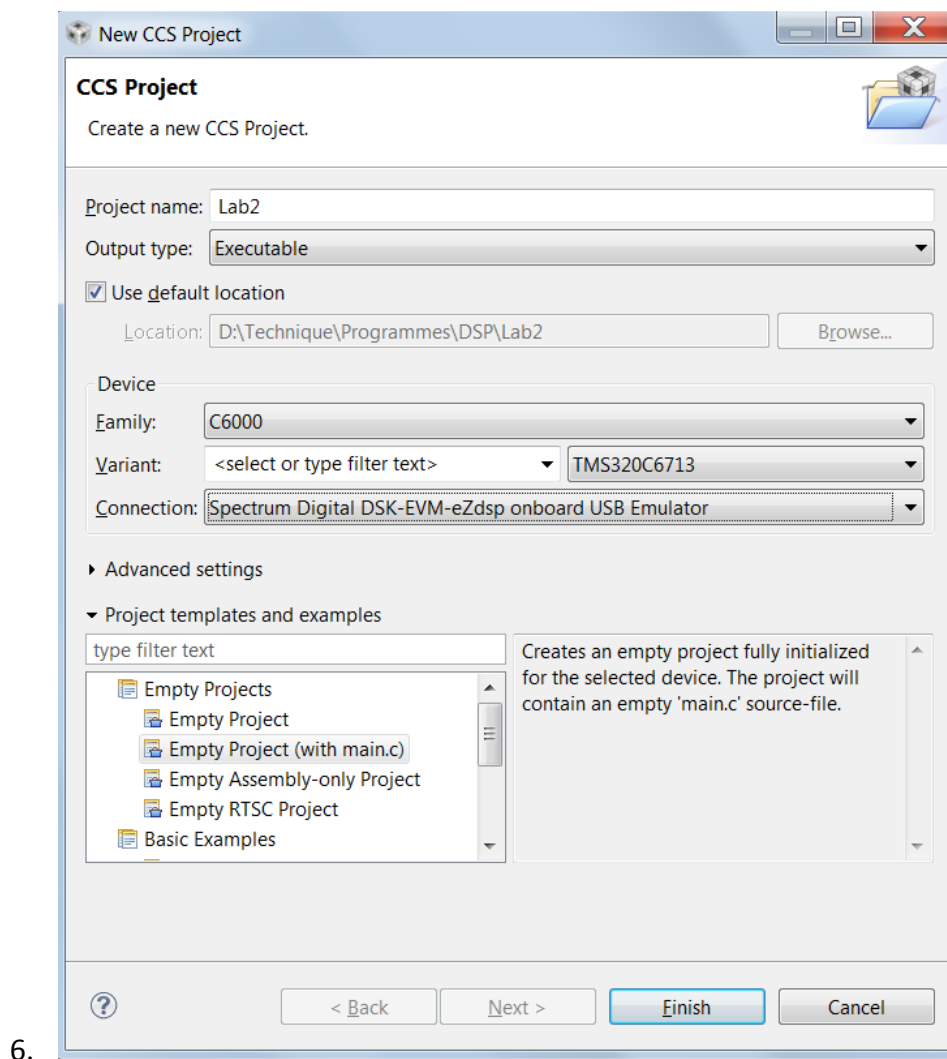
En faisant un clique gauche sur la configuration, on la met par défaut.




Après avoir défini la configuration, on peut créer un nouveau projet

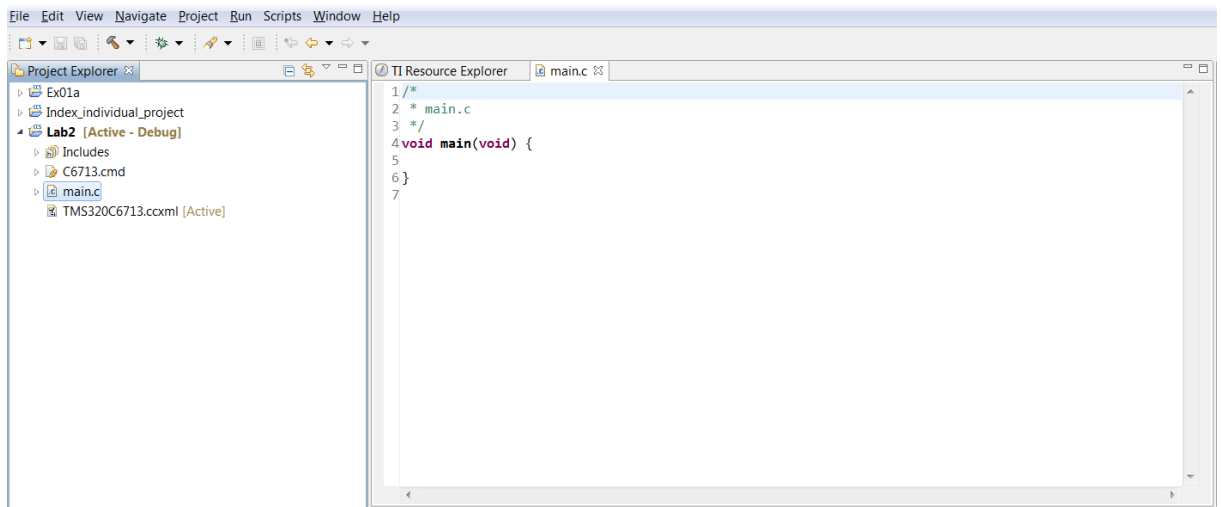


En rentrant les informations voulues, en laissant bien **Executable** et en choisissant un projet vide avec main.c



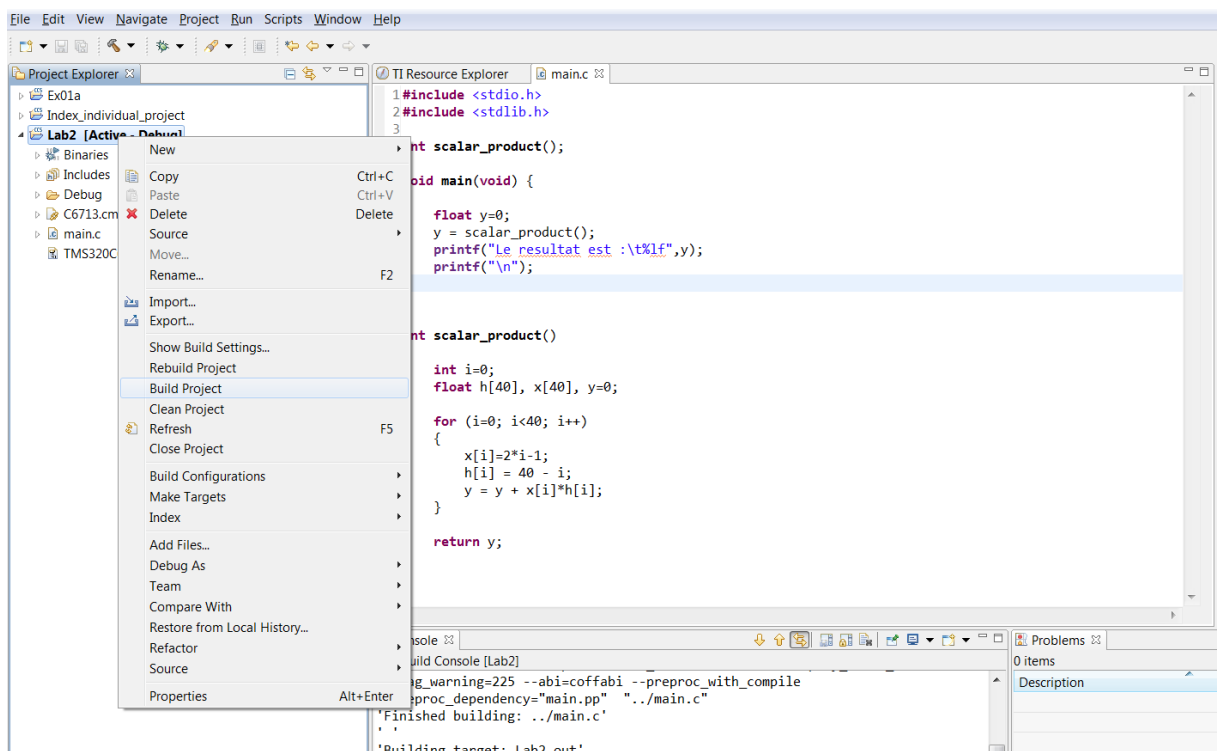
Le projet s'affiche alors dans la fenêtre  Project Explorer

7.

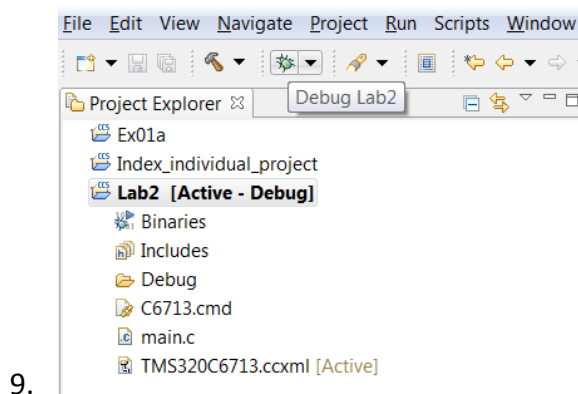


Il « suffit » ensuite de coder, et une fois fini on « build ».

8.

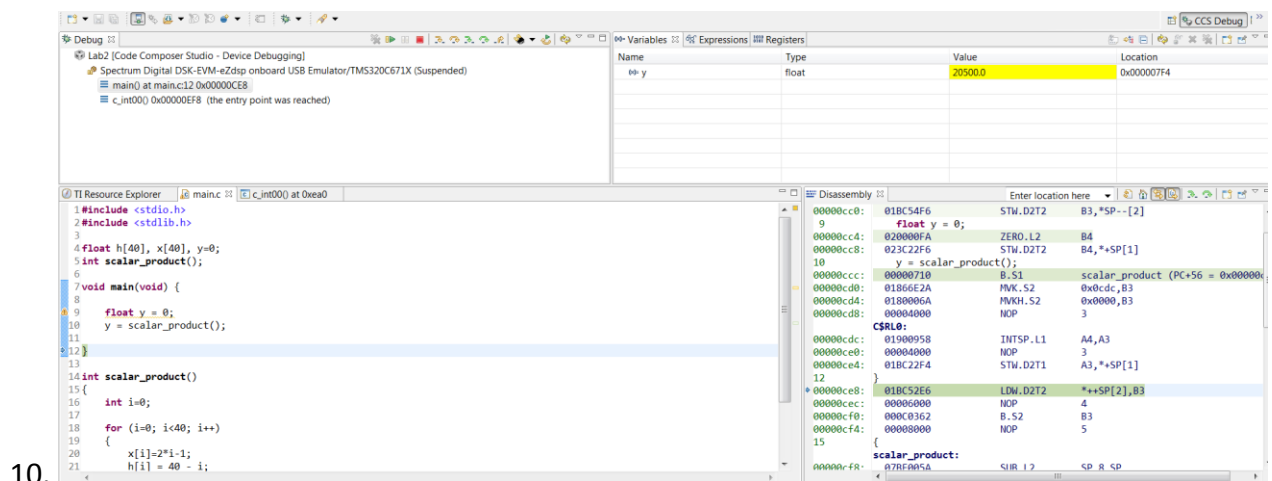


Lorsqu'il n'y a plus d'erreur, on peut alors debugger notre code en cliquant sur l'icône correspondante,



En déroulant le code on vérifie le résultat de retour dans la fenêtre

Variables

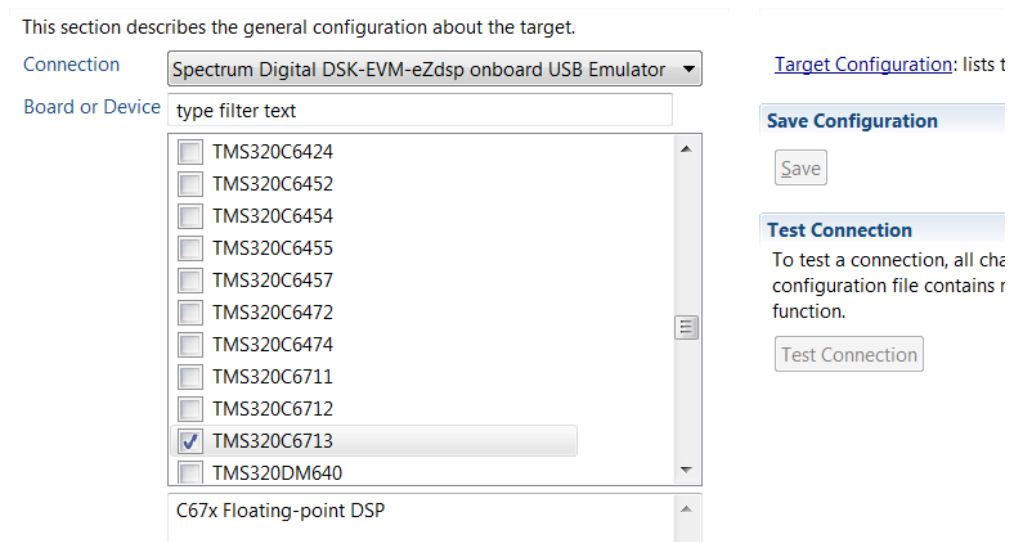


II. Mesure de performance

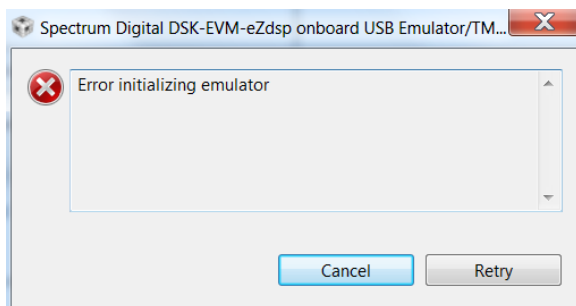
Après avoir déboguer son programme, il convient de mesurer ses performances pour ensuite optimiser son code. Pour analyser les performances, CCS propose un profiler assez puissant, seulement il ne marche pas avec toutes les cartes (dont TMS320C6713). C'est pourquoi on créer un nouveau **Target Configurations** qui va permettre de simuler une carte.

(Répéter I-2 à I-4)

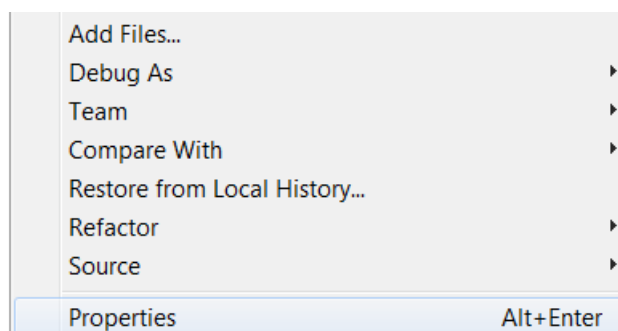
En utilisant la configuration voulue.



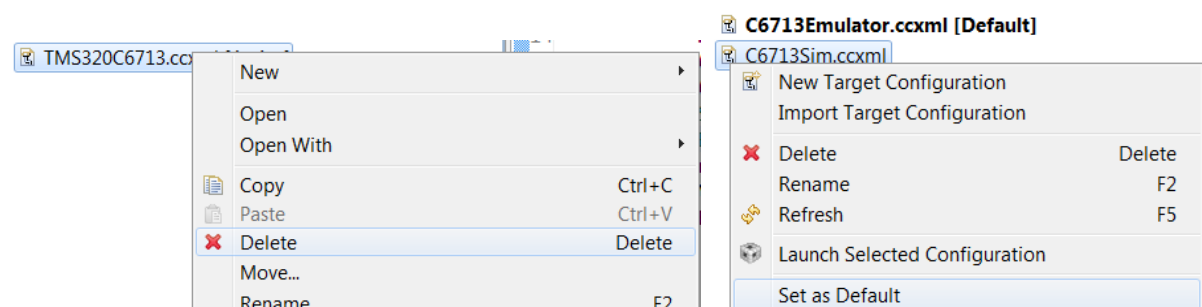
Si cela ne marche pas toujours pas lorsque l'on se met en mode debug



Alors il est nécessaire de configurer manuellement en cliquant gauche sur notre projet et en cliquant sur **Properties**



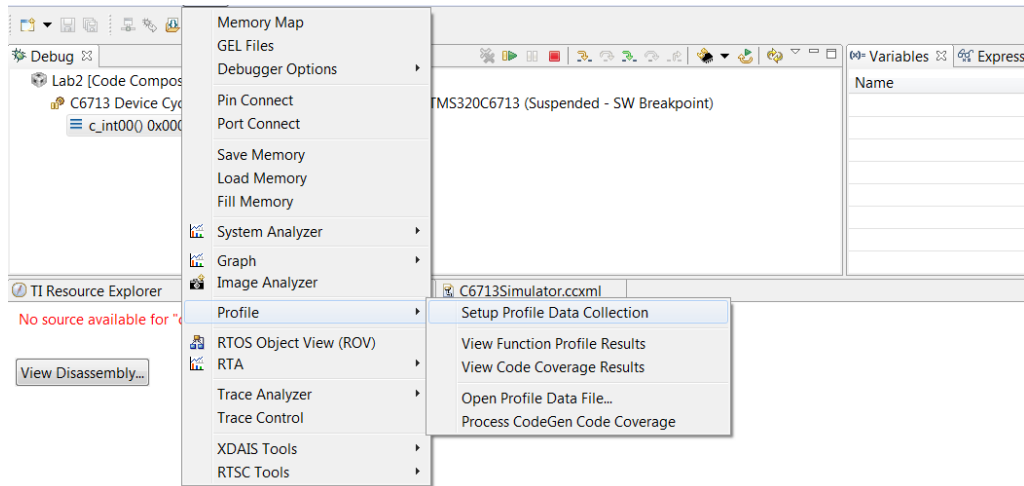
On supprime le fichier suivant et on remet l'ancienne **Target Configurations** par défaut



On peut alors enfin lancer le mode debug.

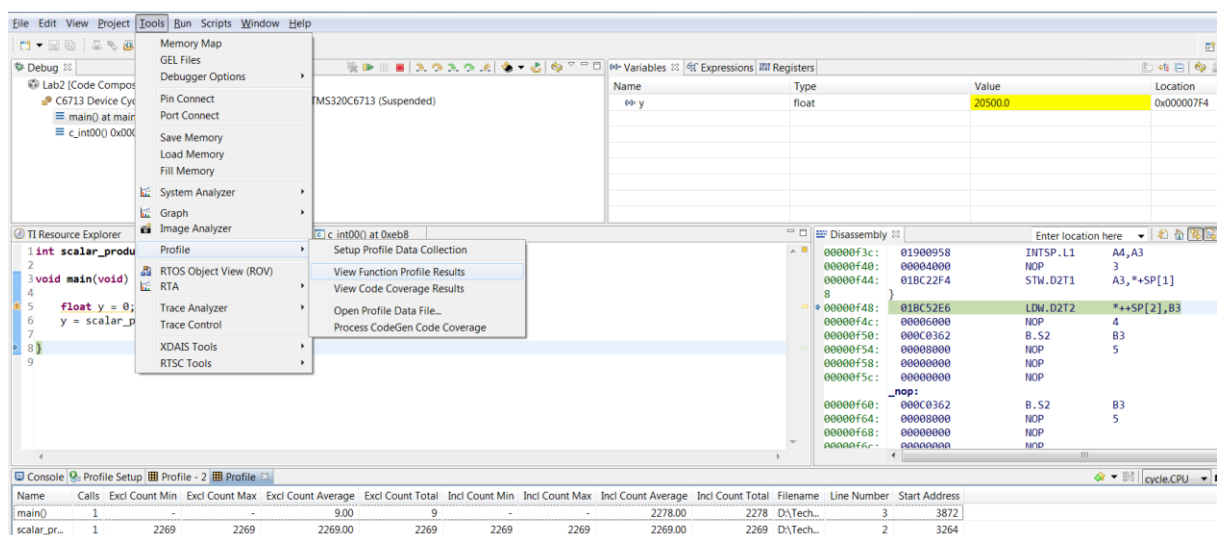
(Répéter I-8 à I-10)

On définit les paramètres de notre profiler sur l'onglet **Tools**, sans oublier de **Save** et **Activate** la configuration.



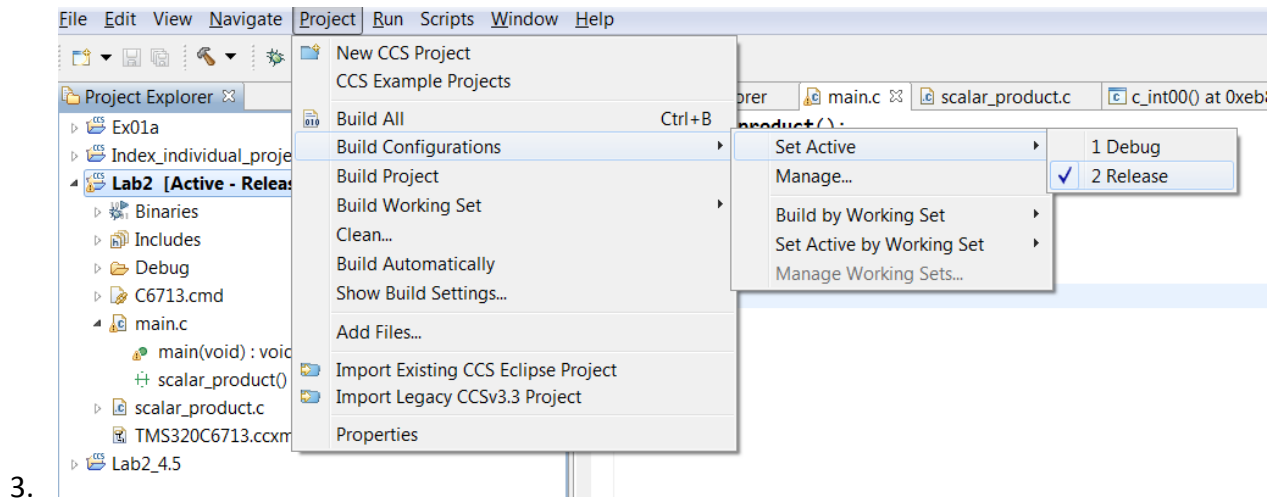
1.

On regarde les résultats du profiler en lançant notre programme 2 fois après avoir cliqué sur **View Function Profile Results**.



2.

Lorsque l'on a étudié notre programme en mode « Debug », il convient de le tester de la même manière en mode « Release » cette fois-ci. C'est-à-dire que le compilateur va optimiser le temps de calcul. On commence donc commencer par changer la configuration du build,

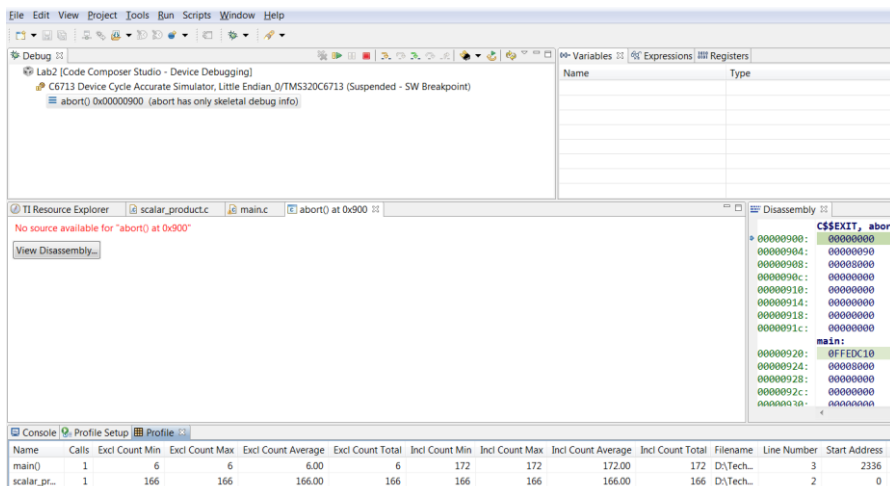


On rebuild et on lance le programme (clique sur debug mais ce n'est pas du debug)

(Répéter I-8 et I-9)

On recommence les étapes pour le profiler

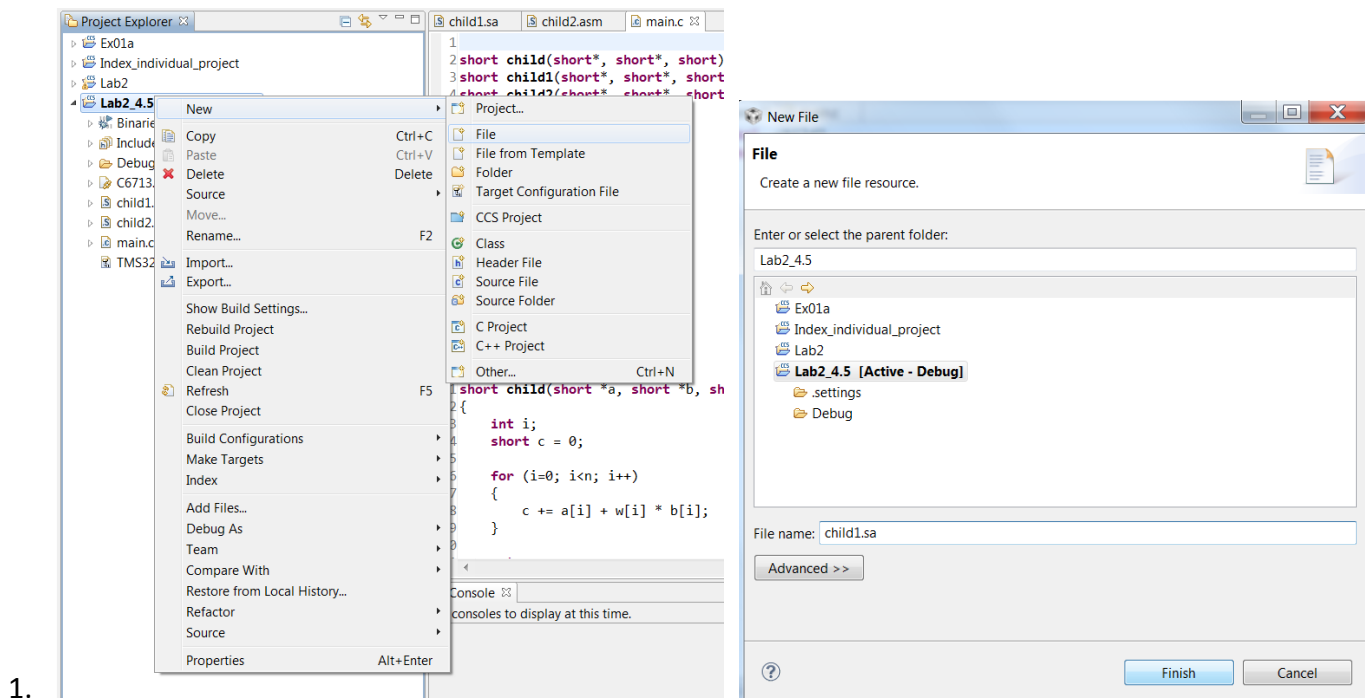
(Répéter II-2)



On voit que l'on a de bien meilleures performances avec le mode release. Par contre en mode release on a plus accès aux informations sur nos variables, c'est pourquoi il convient bien évidemment de débbuger son programme avant de passer en mode « Release ».

III. Programmation en assembleur

La programmation en assembleur nécessite beaucoup plus de rigueur qu'en langage C, le débogage devient encore plus important en assembleur. Après avoir écrit le programme en C, j'ai commencé par faire l'assembleur linéaire. Pour créer un fichier en assembleur linéaire il faut créer un nouveau fichier en précisant l'extension « .sa ».



Contrairement à l'assembleur optimisé, ici on ne s'occupe pas des registres, des unités de calculs, des délais des instructions et de la parallélisation).

```

1      .global _child1
2      .global _w
3
4      _child1: .cproc ptr_a,ptr_b,n
5              .reg    a,b,prod,sum,c,ptr_w,w
6              MVKL    _w,ptr_w ;_w est une adresse, il faut donc utiliser MVKL et MVKH
7              MVKH    _w,ptr_w
8              zero    c
9
10     for:      LDH     *ptr_a++,a
11              LDH     *ptr_b++,b
12              LDH     *ptr_w++,w
13              MPY     w,b,prod
14              ADD     a,prod,sum
15              ADD     c,sum,c
16     [n]      SUB     n,1,n
17     [n]      B       for
18              .return c
19              .endproc
20

```

Il ne faut pas oublier de définir la fonction dans le fichier en C.

```

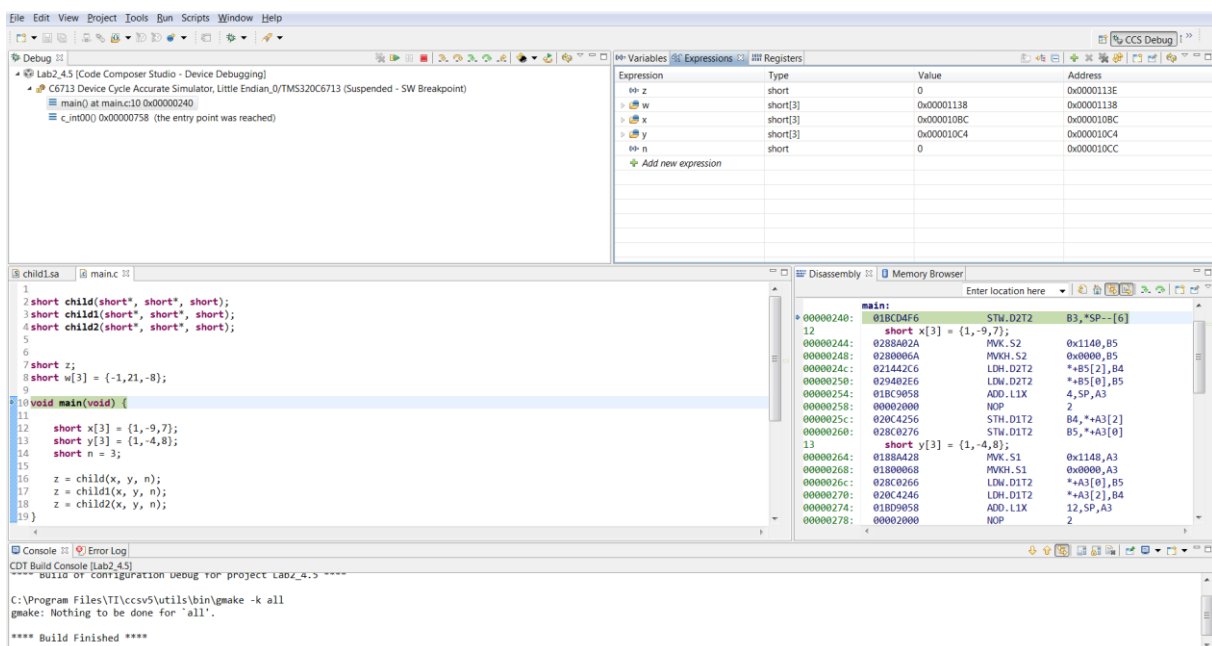
1
2 short child(short*, short*, short);
3 short child1(short*, short*, short);
4 short child2(short*, short*, short);
5
6
7 short z;
8 short w[3] = {-1,21,-8};
9
10 void main(void) {
11
12     short x[3] = {1,-9,7};
13     short y[3] = {1,-4,8};
14     short n = 3;
15
16     z = child(x, y, n);
17     z = child1(x, y, n);
18     z = child2(x, y, n);
19 }
20
21 short child(short *a, short *b, short n)
22 {
23     int i;
24     short c = 0;
25
26     for (i=0; i<n; i++)
27     {
28         c += a[i] + w[i] * b[i];
29     }
30
31     return c;
32 }
33

```

On accède au mode debug,

(Répéter I-8 et I-9)

On a accès à plusieurs informations que l'on peut choisir dans l'onglet Window. Pour lancer le programme pas à pas on peut appuyer sur F5 (et dans de cas on rentre dans les fonctions), en utilisant F6 on se déplace aussi pas à pas mais seulement dans le main (sans rentrer dans les fonctions).



Pour débbugger, il vaut mieux commencer par voir si l'on obtient bien les bonnes valeurs des variables locales et globales, définies dans le C.

Expression	Type	Value	Address
a	int	1	Register A7
b	int	1	Register A6
n	int	3	Register B0
w	int	-1	Register B5

```

4 short child2(short*, short*, short);
5
6
7 short z;
8 short w[3] = {-1, 21, -8};
9
10 void main(void) {
11     short x[3] = {1, -9, 7};
12     short y[3] = {1, -4, 8};
13     short n = 3;
14
15     z = child(x, y, n);
16     z = child1(x, y, n);
17     z = child2(x, y, n);
18 }

```

On peut remarquer que le compilateur choisit de rassembler les variables prod, sum et b par exemple car selon lui il est plus efficace de travailler comme cela

Name	Type	Value	Location
w	int	-8	Register B5
n	int	1	Register B0
a	int	7	Register A7
b	int	-57	Register A6
c	int	-93	Register A3
sum	int	-57	Register A6
ptr_a	int	4266	Register A4
ptr_b	int	4274	Register A5
prod	int	-57	Register A6
ptr_w	int	4414	Register B4

On déroule ensuite le programme pour voir ce que l'on récupère en sortie

Expression	Type	Value	Address
z	short	-150	0x0000113E
w	short[3]	0x00001138	0x00001138

Une fois la simulation terminée on peut l'arrêter

Lab2_4.5 [Code Composer Studio - Device Debugging]

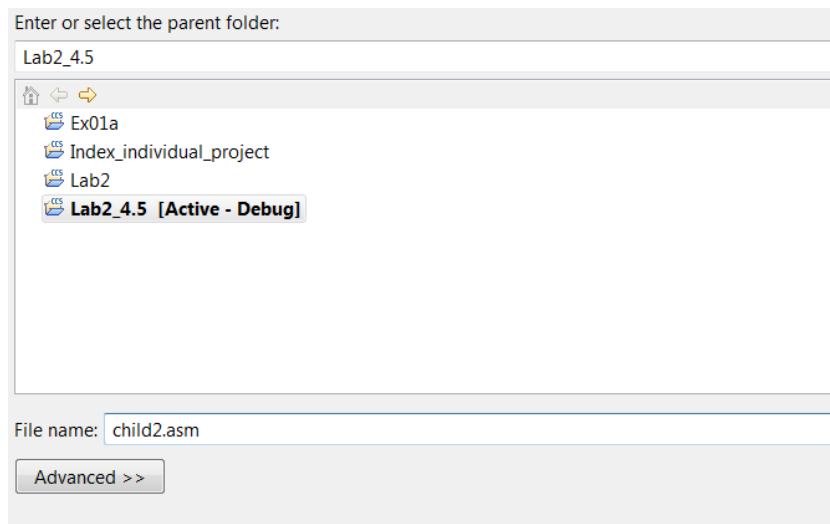
C6713 Device Cycle Accurate Simulator, Little Endian_0/TMS320C6713 (Suspended)

main() at main.c:17 0x000002C0

c_int00() 0x00000758 (the entry point was reached)

Terminate (Ctrl+F2)

Pour l'assembleur normal cette fois-ci, il est nécessaire de créer un autre type de fichier « .asm ».



2.

Voici le code du programme en assembleur. Il ne faut pas oublier de prendre en compte les temps de cycles des instructions en rajoutant des nop.

```

1      .global _child2
2      .global _w
3
4 _child2:  MVKL    .S  _w,B0 ; Chargement de l adresse de la variable globale
5          MVKH    .S  _w,B0
6          MV      .S  A6,B2 ; short n  attention pour les cond on ne peut utiliser que A1,A2,B0,B1,B2
7          zero    .L  A1 ;short c = 0
8
9 for:     LDH      .D  *B0++,B1 ; B1 = _w
10         LDH      .D  *B4++,B5 ; short *b
11         LDH      .D  *A4++,A5 ; short *a
12         nop      3
13         MPY      .M  B5,B1,B1 ; B1 = w*b
14         nop
15         ADD      .L  A5,B1,B1 ; B1 = a + w*b
16         ADD      .L  A1,B1,A1 ; A5 = c + a + w*b
17 [B2]     SUB      .L  B2,1,B2
18 [B2]     B        .S  for
19         nop      5
20         MV       .D  A1,A4
21         B        .S2 B3
22         nop      5

```

La procédure de debug est ensuite la même que précédemment.



(Répéter I-8 et I-9)

Pour vérifier les variables, il peut être plus pratique d'utiliser la fenêtre Memory Browser. Ainsi au début du programme, si on veut vérifier le premier argument de la fonction (x dans notre cas) il suffit de taper A4 (registre qui contient l'adresse du 1^{er} paramètre dans le cas d'un pointeur). On trouve ainsi dans l'ordre 0x0001 ; 0xFFF7 et 0x0007 ce qui correspond bien à ce qu'on a défini dans le C pour x . En effet en complément à 2 : 0xFFF7 → 111111111110111 → -9. On peut remarquer que le registre A6 (3^{ème} paramètre) contient directement la valeur de $n = 3$. En effet, on rentre bien directement la valeur comme paramètre et pas par pointeur (adresse).

The screenshot shows the CCS IDE interface. The top panel displays the 'Registers' window with a table of core registers A0 through A8. The bottom panel shows the 'Memory Browser' window for address A4 (0x1004), displaying memory data in hex 32-bit TI style.

Name	Value	Description
A0	0x00000000	Core
A1	0x00000001	Core
A2	0x00000001	Core
A3	0x00000003	Core
A4	0x00001004	Core
A5	0xFFFFFFFF	Core
A6	0x00000003	Core
A7	0x00000000	Core
A8	0x00000000	Core

Address	Value
0x00001004	FFF70001 00000007 FFFC0001 00000008 00000003 00000718
0x0000101C	00000000
0x00001020	cinit, __cinit, __STACK_END
0x00001020	00000004 00001084 00000000 00000000 00000004 00001088
0x00001038	00000000 00000000 00000004 0000108C 00000001 00000000
0x00001050	00000004 00001090 000007E0 00000000 00000004 00001094
0x00001068	000007E0 00000000 00000006 00001098 0015FFFF 0000FFFF
0x00001080	00000000
0x00001084	_cleanup_ptr
0x00001084	00000000
0x00001088	_dtors_ptr
0x00001088	00000000
0x0000108C	__TI_enable_exit_profile_output
0x0000108C	00000001
0x00001090	_lock
0x00001090	000007E0
0x00001094	_unlock
0x00001094	000007E0

Pour être sûr que l'assembleur marche bien indépendamment du reste, il peut être intéressant de désactiver les autres fonctions (comme elles ont modifiés les registres, ça a pu faciliter la tâche à l'assembleur comme pour la variable globale par exemple). On clique sur  pour remettre tout à zero et ensuite cliquer à côté sur .

The screenshot shows the CCS IDE interface in debug mode. The top panel displays the 'Debug' window with a table of expressions and their values. The bottom panel shows the 'Disassembly' window for the main function.

Expression	Type	Value	Address
z	short	-150	0x0000109E
w	short[3]	0x00001098	0x00001098


```

5
6
7 short z;
8 short w[3] = {-1, 21, -8};
9
10 void main(void) {
11     short x[3] = {1, -9, 7};
12     short y[3] = {1, -4, 8};
13     short n = 3;
14
15     // z = child(x, y, n);
16     // z = child1(x, y, n);
17     z = child2(x, y, n);
18 }
19
20
  
```

Voici les résultats du profiler de CCS en mode debug (même procédure que précédemment)

Name	Calls	Excl Count Min	Excl Count Max	Excl Count Average	Excl Count Total	Incl Count Min	Incl Count Max	Incl Count Average	Incl Count Total	Filename	Line Number	Start Address
child(short *, short *, short)	1	137	137	137.00	137	137	137	137.00	137	D:\Tech...	22	752
child1()	1	89	89	89.00	89	89	89	89.00	89	D:\Tech...	4	1920
child2	1	62	62	62.00	62	62	62	62.00	62	D:\Tech...	4	2016
main()	1	-	-	41.00	41	-	-	329.00	329	D:\Tech...	10	576

En mode release, les paramètres d'optimisation sont bien plus puissants que le programme effectué en assembleur. Comme je n'ai pas optimisé l'assembleur, le nombre de cycles est supérieur au langage C et assembleur linéaire. Il ne faut pas oublier qu'en mode Release on a très peu d'informations sur le debug (ce qui est normal car on est censé avoir debuggé avant...)

Name	Calls	Excl Count Min	Excl Count Max	Excl Count Average	Excl Count Total	Incl Count Min	Incl Count Max	Incl Count Average	Incl Count Total	Filename	Line Number	Start Address
child(short *, short *, short)	1	29	29	29.00	29	29	29	29.00	29	D:\Tech...	22	928
child1()	1	23	23	23.00	23	23	23	23.00	23	D:\Tech...	4	1760
child2	1	62	62	62.00	62	62	62	62.00	62	D:\Tech...	4	2048
main()	1	40	40	40.00	40	154	154	154.00	154	D:\Tech...	10	1100