PARTIE 2

Noyau Temps Réel

DSP BIOS

Maîtrise EEA
Maîtrise IUP
UJF Grenoble
Noyau DSP BIOS

PA DEGRYSE

SOMMAIRE

Chapitre 1 : Description générale

A) Caractéristiques et avantages de DSP BIOS

1) Caractéristiques générales du noyau

Le noyau DSP BIOS est spécifique aux processeurs de traitement de signal de Texas Instruments. Il existe des versions pour les familles TMS320C2x, TMS320C5x et TMS320C6x.
Ce noyau fait partie intégrante de «Code Composer Studio ».
Il utilise très peu de mémoire et de ressources sur la cible (200 à 2000 words de code).
C'est un noyau configurable : seules les primitives utilisées sont chargées dans la cible.

- ✓ Tous les objets utilisés sont crées par un outil de configuration qui permet de construire le noyau : on réduit ainsi le code produit et les tables de données au strict minimum.
- ✓ Les outils de mise au point (mode trace, messages etc.) envoient des données sur le PC de développement. C'est lui qui a la charge de l'affichage et non le DSP cible.
- ✓ La librairie système est optimisée pour réduire le nombre de cycles machines utilisés. La plupart des fonctions sont écrites en assembleur.
- ✓ La communication entre le PC et la carte cible DSP se fait quand ce dernier est dans une boucle de mise en sommeil, située en arrière plan : **IDLE LOOP.** Cela permet de s'assurer que les tâches de communication avec le PC ne perturbent pas le temps réel sur la cible.

2) Caractéristiques générales des primitives

Il y a de nombreuses possibilités pour créer une application.

- ✓ Une application peut posséder des objets, (tâches, sémaphores, pipe etc.), crées soit de manière statique avec l'outil de configuration, soit de manière dynamique lors de l'exécution du code programme.
- ✓ Les tâches (threads) peuvent être de plusieurs natures :
  - **HWI : interruption matérielle.**
  - **SWI : interruption logicielle.**
  - **TSK : tâches soumises à l'algorithme de préemption.**
  - **IDL : tâches de fond en mode sommeil**.

  A l'intérieur de chaque groupe on peut contrôler la priorité de chacune des tâches.
- ✓ Il existe des mécanismes de synchronisation entre les tâches comme les sémaphores, les boites à lettres (mailbox), les pipes, et les rendez-vous (resource lock).
- ✓ Deux modèles d'entrées sorties existent :
  - Les pipes pour la communication entre les tâches.
  - Les flux de données (streams) pour les entrées sorties complexes et les drivers de périphériques.
- ✓ Il existe des primitives de bas niveau pour gérer les erreurs, les structures de données communes et la mémoire.

3) Standardisation des API

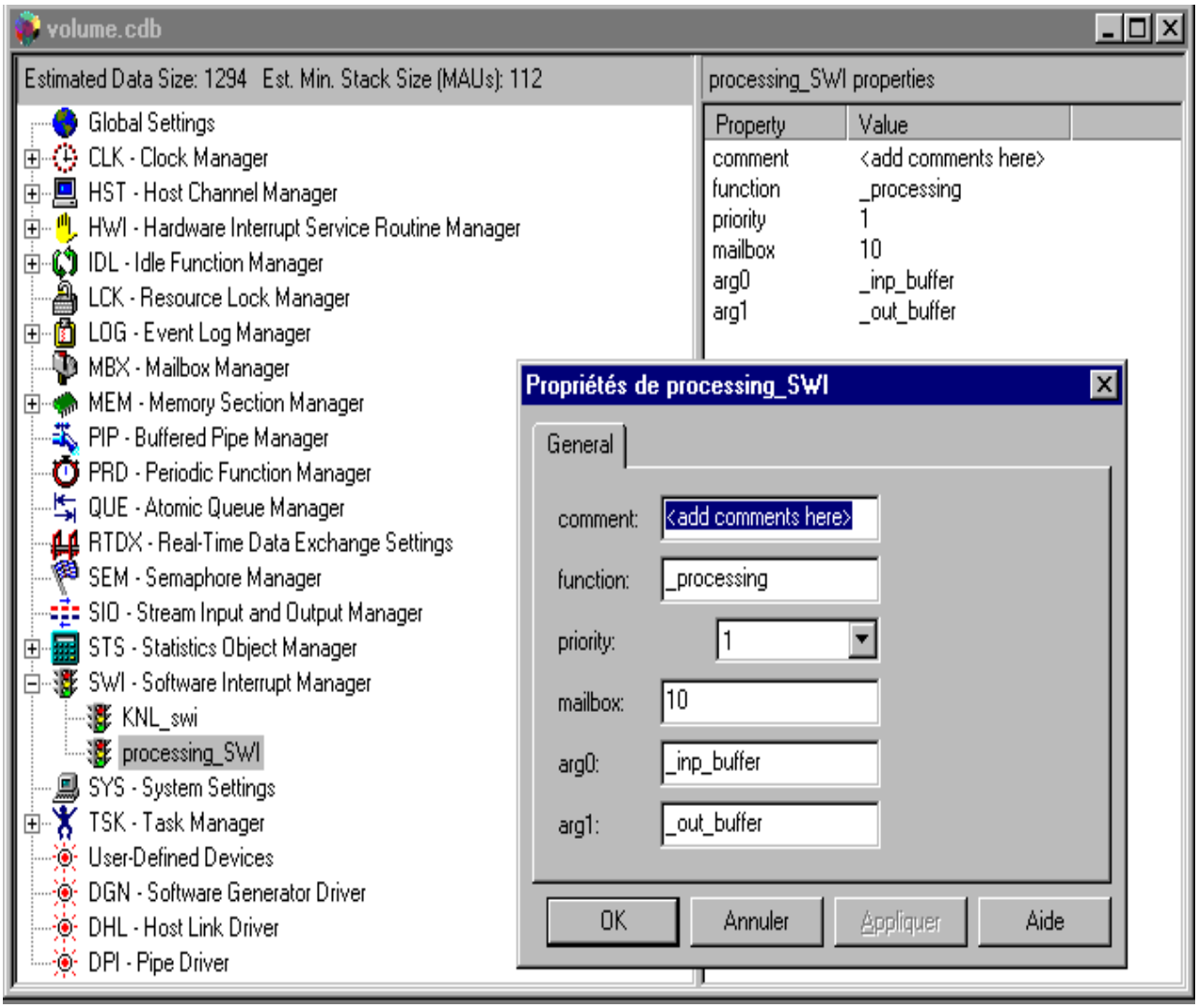
Une primitive système se dit en anglais **API** (**A**pplication **P**rogram **I**nterface). Ces API sont standardisées pour tous les composants de type TMS320. Les appels sont les mêmes pour le C2x, le C5x et le C6x.

- ✓ L'outil de configuration, inclus dans Code Composer Studio, est le même pour tous les DSP et permet de créer tous les objets de l'application.
- ✓ L'outil de configuration détecte les erreurs avant la génération du code.
- ✓ L'outil de configuration permet de charger dans la cible des modules de trace par message et d'analyse de fonctionnement de l'application en temps réel qui ne perturbent pas l'application.
- ✓ Les modules de DSP BIOS permettent l'analyse du fonctionnement temps réel de la cible.
- ✓ Les appels aux API sont standardisés : cela permet le développement de code portable sur d'autres cibles matérielles.

B) Les composants logiciels de DSP BIOS

4) Code composer studio et DSP BIOS

L'ordinateur PC permet d'éditer les programmes sources (en « c » ou en « assembleur »).
L'outil de configuration de DSP BIOS permet de choisir les API implantées dans le programme d'application.



5) Les familles de primitives temps réel et les API de DSP BIOS

La librairie temps réel de DSP BIOS permet de produire des programmes embarqués qui s'exécutent sur la cible. Cela comprend la gestion des tâches, les entrées sorties, la capture et le traitement d'informations pour la mise au point du programme.

Les API de DSP BIOS sont divisées en modules. L'outil de configuration permet de choisir les modules qui seront embarqués dans la cible. Toutes les fonctions d'une même famille commence par les **mêmes codes de lettres XXX**.

Les primitives de DSP BIOS peuvent être regroupées en six familles

a) *System service*
On trouve dans cette famille toutes les fonctions système de gestion du noyau et de la mémoire.

b) *Instrumentation*
On trouve dans cette famille toutes les fonctions qui permettent le dialogue avec le PC sans interrompre le temps réel de l'application

c) *Sheduling*
On trouve dans cette famille toutes les fonctions de gestion du temps et des tâches.

d) *Synchronisation*
On trouve dans cette famille toutes les fonctions de synchronisation des tâches

e) *Input/Output*
On trouve dans cette famille toutes les fonctions de gestion des entrées sorties

f) *Chip Support library*
On trouve dans cette famille toutes les fonctions de gestion des circuits périphériques implantés dans la puce du DSP. Ces fonctions n'existent qu'à partir de la version 2.0 de BIOS DSP.

			Object Creations Language Support		
Description Module	**Name**	**Static**	**Dynamic**	**C**	**ASM**
System Services					
Global Settings		X			
Static Memory Segment Manager	**MEM**	X			
Dynamic Memory Segment Manager	**MEM**		X	X	
System Services Manager	**SYS**	N/A	N/A	X	
Atomic Functions (optimized non-preemptive)	**ATM**				
Instrumentation		X		X	X
Message Log Manager	**LOG**	X		X	X
Statistics Accumulator Manager	**STS**	X		X	X
Trace Manager	**TRC**	X			X
Scheduling					
System Clock Manager	**CLK**	X		X	X
Periodic Function Manager	**PRD**	X		X	X
Hardware Interrupt Manager	**HWI**	X			X
Software Interrupt Manager	**SWI**	X	X	X	
Multitasking Manager	**TSK**	X	X	X	
Idle Function and Processing Loop Manager	**IDL**	X		X	
Synchronization					
Semaphore Manager	**SEM**	X	X	X	
Resource Lock Manager	**LCK**	X	X	X	
Mailbox Manager MB	**MBX**	X	X	X	
Queue Manager	**QUE**	X	X	X	
Input/Output					
Target-to-Host Communication Manager	**RTDX**	X		X	X
Host I/O Manager	**HST**	X		X	X
Data Pipe Manager	**PIP**	X		X	X
Stream I/O and Device Driver Manager	**SIO/DEV**	X	X	X	
Chip Support Library (CSL)					
Direct Memory Access	**DMA**	X	X	X	
Enhanced Direct Memory Access C6x only	**EDMA**	X	X	X	
External Memory Interface C6x, C55x	**EMIF**	X	X	X	
Multichannel Buffered Serial Port	**McBSP**	X	X	X	
Timer Device	**TIMER**	X	X	X	
Expansion Bus (TMS320C6x only)	**XBUS**	X	X	X	
Instruction Cache (TMS320C55x only)	**CACHE**	X	X	X	
General Purpose Input/Output C5x only	**GPIO**	X	X	X	
Clock Generator C5x only	**PLL**	X	X	X	
Watchdog Timer Device C54x only	**WDTIMER**	X	X	X	

- ✓ Le mot clef **« static »** veut que ces fonctions soient crées par l'outil de configuration et quelles sont permanentes en mémoire
- ✓ Le mot clef **« dynamic »** veut dire quelles peuvent être lancées lors de l'exécution du programme.
- ✓ Dans chaque famille on trouve des sous fonctions qui commencent toutes par le même mot clef :

  Par exemple : LOG_printf(..), SEM_pend(..), SEM_post(..), QUE_get(..), QUE_put(..) etc.

- ✓ Le détail de chaque fonction est décrit dans le manuel technique donné en annexe.

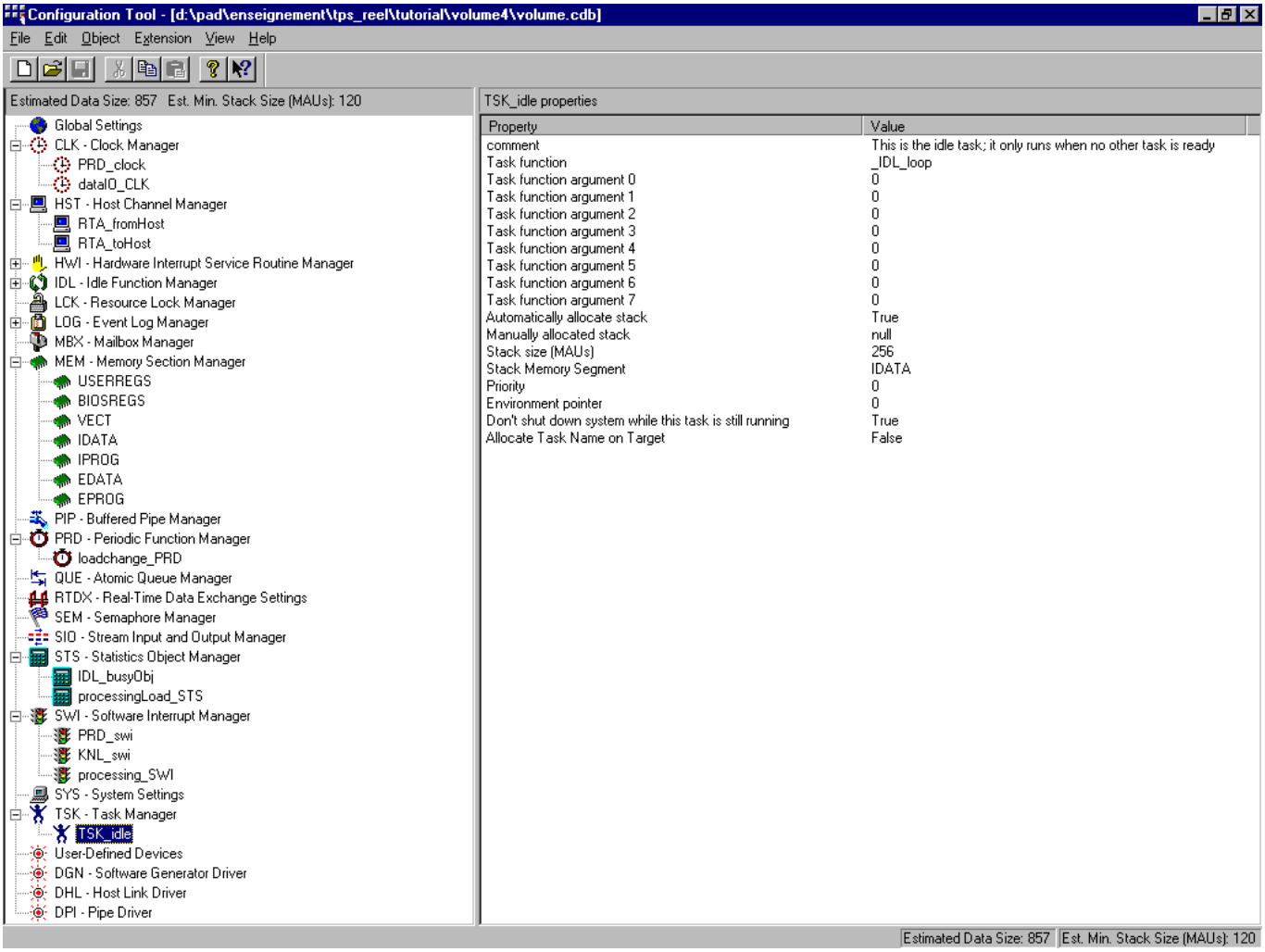
6) L'outil de configuration de DSP BIOP

L'outil de configuration de DSP BIOS se présente comme l'explorateur de Windows et possède trois fonctions :

- ✓ Il permet de fixer les paramètres de chaque primitive API du noyau embarqué.
- ✓ Il sert de d'éditeur graphique des objets implantés dans la cible.
- ✓ Il prépare l'image mémoire du noyau qui sera utilisée par le compilateur.

REMARQUE :

Les objets qui sont crées par l'outil de configuration, sont statiques et présents dans la cible au BOOT. Il est possible aussi de créer dynamiquement des objets lors de l'exécution du programme.
L'utilisation de cet outil réduit donc la taille du code cible.



7) Les outils d'analyse et d'instrumentation temps réel

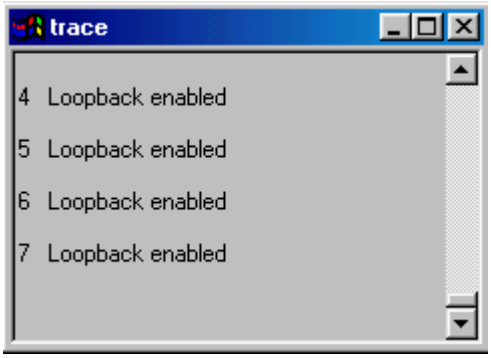
Pour la mise au point du programme on dispose de plusieurs outils. Ces outils envoient au PC les informations sur le comportement de l'application quand le **DSP est en mode IDLE** : cela ne perturbe donc pas le temps réel sur la cible.
C'est le PC qui est chargé de l'affichage des données sur l'écran et non le DSP
L'ensemble de ces modules constitue le **RTA** : **« Real Time Analysis »**.

DSP BIOS peut donc jouer le rôle d'analyseur logique Matériel et Logiciel

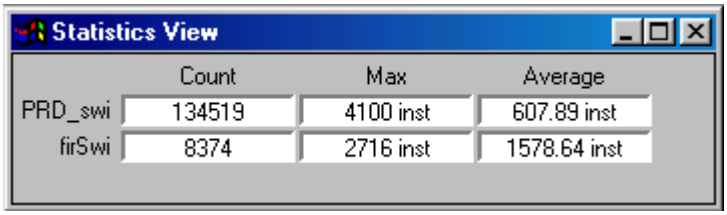
a) Principe de la communication avec le PC.
La communication avec le PC se fait par l'intermédiaire du bus **JTAG** : « **J**oint **T**est **G**roup **A**ction ». Ce bus d'échange de données est normalisé pour tous les constructeurs et son protocole sert de support de base de communication entre une carte cible DSP et le PC.

Le protocole spécifique défini par Texas Instruments est **RTDX** : **R**eal **T**ime **D**ata **E**xchange.

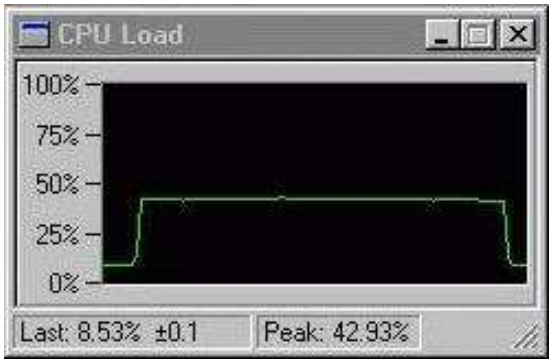
b) Envoie de messages.
Chaque tâche peut envoyer des messages au PC en utilisant la fonction « LOG_printf »



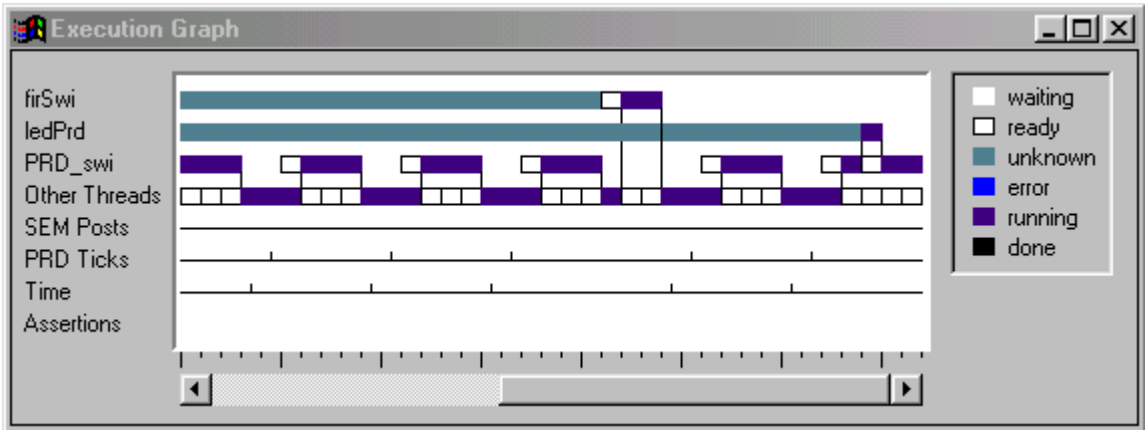
c) Statistiques sur les temps de cycle.
Pour chaque tâche il est possible de visualiser graphiquement des statistiques sur le nombre de cycles machines utilisés



d) Graphique de charge du CPU.
La charge de travail du CPU est aussi visible graphiquement.



e) Graphique d'exécution des tâches.
L'affectation du CPU dans le temps aux différentes tâches est aussi visible graphiquement.

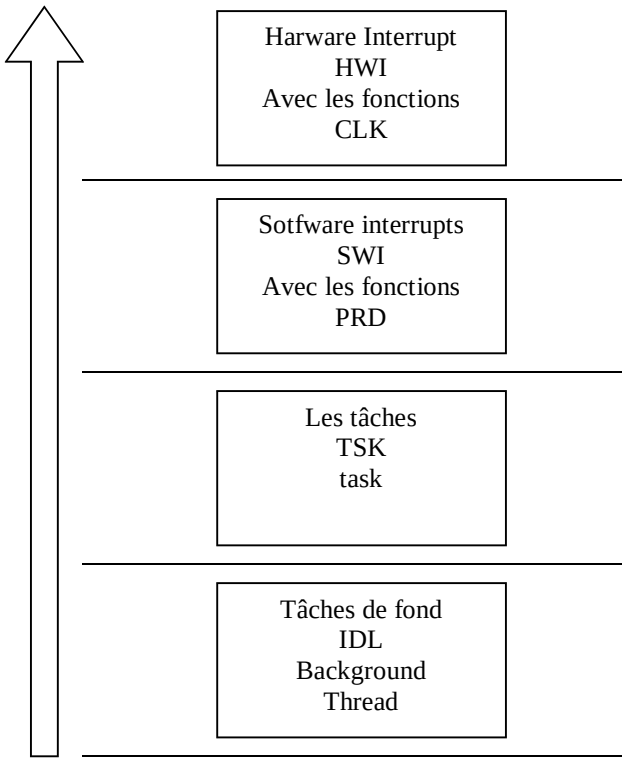


C) Principe de fonctionnement de DSP BIOS

8) Principe général

a) Les principales familles de tâches.

DSP BIOS possède quatre familles de tâches : **HWI – SWI – TSK – IDL.** Ces familles de tâches sont hiérarchisées suivant le schéma suivant :



- ✓ Les tâches **HWI**
  Se sont les tâches d'interruption matérielles encore appelée **ISR I**nterrupt **S**ervice **R**outine. Elles ont la plus haute priorité et sont dues à l'arrivée d'un signal sur un dispositif d'entrées sorties, McBSP, Timer, DMA ou autre.
  Elles sont classées par priorité, la plus haute étant celle du **R**eset « HWI_RS », puis celle de l'**I**nterruption **non-Masquable** HWI_NMI.
  Elles doivent exécuter une tâche critique et ont la plus haute priorité pour DSP BIOS.
  Leur fréquence de fonctionnement peut approcher les 200 kHz et doivent être très courtes de durée comprise entre 2 et 100 micros secondes.
  Elles doivent s'exécuter complètement jusqu'à une instruction de retour. Elles ne peuvent pas se mettre en sommeil

- ✓ Les tâches **SW**I
  Les interruptions logicielles sont de niveau inférieur aux précédentes. Elles possèdent elles-mêmes 15 niveaux relatifs. Elles sont exécutées en général par une instruction « SWI » depuis une routine « HWI ».
  Elles sont utilisées pour traiter des événements de durée 100 micros secondes ou plus. Comme les HWI, **elles doivent s'exécuter complètement jusqu'à une instruction de retour** et ne peuvent pas se mettre en sommeil.

- ✓ Les tâches **TSK**
  Les tâches ont une priorité inférieure aux SWI. **Elles peuvent être suspendues** et l'on peut utiliser les mécanismes de synchronisation et de communication inter-tâches tels que les événements, les boîtes à lettre, les sémaphores et les pipes. Elles sont soumises à l'algorithme de préemption.

Elles peuvent être lancées, détruites ou suspendues par les autres types de tâches.

- ✓ Les tâches **IDL**
  La boucle de sommeil est la tâche de plus basse priorité dans DSP BIOS. Le programme principal « main » doit exécuter sa séquence d'initialisation puis l'instruction « return » pour lancer DSP BIOS.
  La boucle IDL est une boucle infinie qui appelle des fonctions utilisant les objets de type IDL. Seule les fonctions, qui n'ont de temps critique pour s'exécuter, doivent être placées dans cette boucle.
  On sort par l'appel à une fonction d'un niveau supérieur HWI , SWI ou TSK.

b) *Les autres types de tâches*

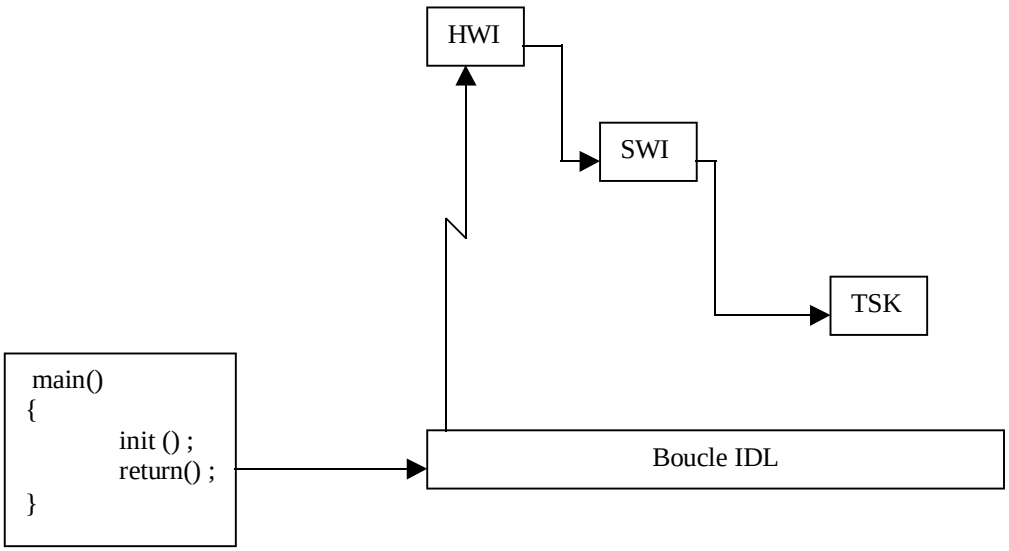
D'autres types de tâches existent. Elles sont exécutées dans le contexte des autres tâches.

- ✓ Les fonctions horloges **CLK** (clock). Elles sont lancées périodiquement par une interruption Timer et utilisent par défaut l'interruption HWI_TINT.

- ✓ Les fonctions périodiques **PRD.** Se sont des fonctions de type SWI lancée sur un temps multiple de l'interruption timer ou de tout autre signal.

- ✓ Les fonctions de données : **Data Notification Functions**. Elles sont exécutées lors des transferts de données de type **PIPE** ou de communication avec le PC **HST** (host channels). Elles appartiennent aux fonctions qui font appel aux primitives de type PIP_alloc, PIP_get, PIP_free, ou PIP_put.

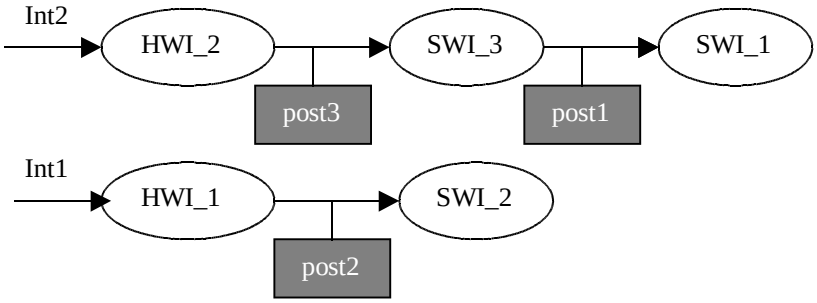
c) Principe de fonctionnement de DSP BIOS
Toute application est construite sur le principe des interruptions. Après une séquence d'initialisation le programme principal « main » donne la main à la boucle infinie IDL et on attend une interruption.
Pendant la boucle IDL, le noyau formate dans une petite zone mémoire les données à envoyer au PC.

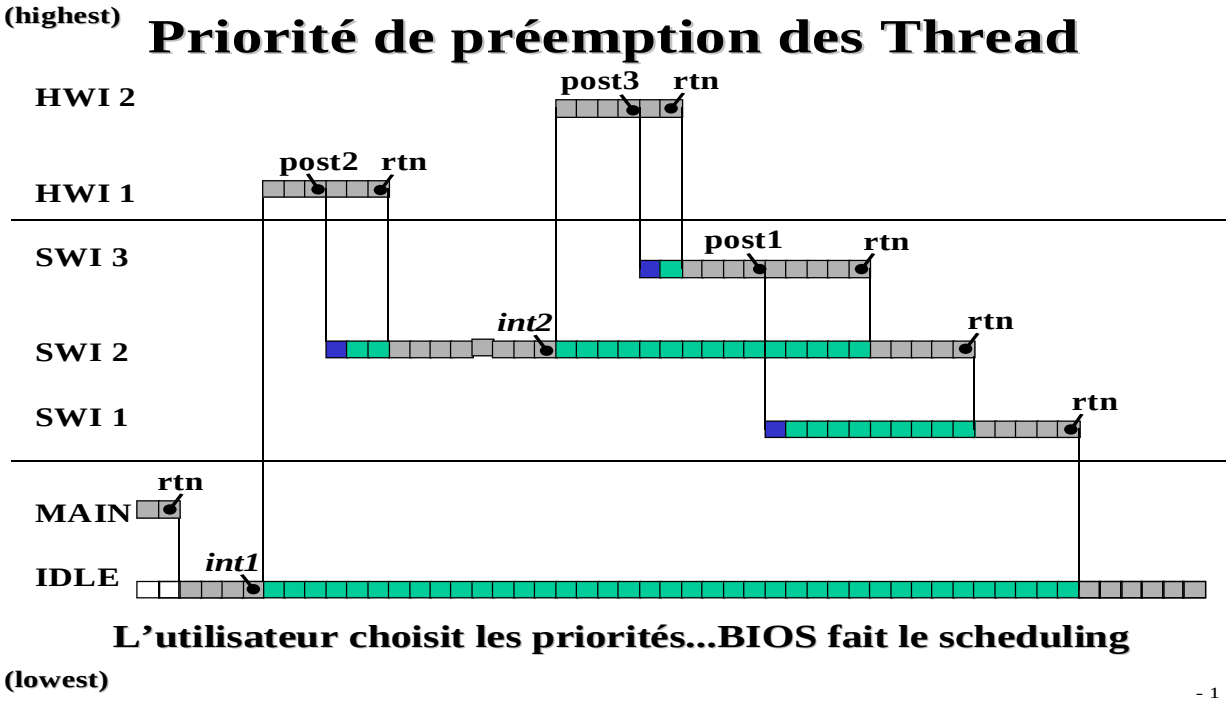


9) Exemple

Soit une application composée de :
- ❑ Deux tâches HWI 1_et HWI_2
- ❑ Trois tâches SWI_1, SWI_2 et SWI_3

Liaison entre les tâches





- 1

CONCLUSION

Pour bien définir une application

- **Etudier les digramme des relation entre les tâches.**
- **Etudier les relation temporelles du systèmes.**

D) Les conventions d'écriture

10) Les conventions générales

Chaque API ou module de DSP BIOS possède un nom unique de 3 ou 4 lettres majuscules utilisé comme préfixe dans les fichiers en-tête, suivies par son nom de fonction ou de module :

XXX_*

Les identificateurs commençant le caractère souligné (underscore) **« _ »** sont aussi réservés pour les fonctions internes écrites en assembleur.

11) Les fichiers d'en-tête

Chaque module DSP BIOS utilise deux fichiers d'en-tête pour toutes les déclaration de constantes, de types etc.

- ✓ **XXX.h**. C'est le fichier d'en-tête pour le prototype des fonctions « C ». Il faut inclure le fichier « std.h » dans toute application DSP BIOS.

- ✓ **XXX.h54**. C'est le fichier d'en-tête pour les programmes écrits en assembleur. Il contient des définitions spécifique au composant utilisé : par exemple le TMS320C5402 pour le kit DSK. Les structures de données communes à tous les composants sont dans le fichier « module.hti », inclus dans le fichier XXX.h54.
- ✓ Le programme principal doit comporter tous les fichiers d'en-tête correspondant aux modules embarqués dans l'application. Le premier doit être « std.h ».

```
#include <std.h>
#include <tsk.h>
#include
<sem.h>
#include
<prd.h>
#include
<swi.h>
```

12) Le nom des objets

Les objets inclus dans la configuration du noyau respectent la même syntaxe. La configuration par défaut contient un objet LOG de nom **LOG_system**.
Les objets crées par l'utilisateur ont un nom libre. Il recommandé d'utiliser une convention analogue : par exemple **Tsk_xxx** pour un objet de type **TSK**. Cela rend le programme plus lisible.

13) Le nom des primitives API

Le format du nom d'une API de DSP BIOS est de la forme **API_action** ou API représente le nom de la famille de primitives système. Par exemple **SWI_post()** lance une interruption logicielle.

14) Les primitives communes accessibles à toutes les fonctions

Il y a aussi des fonctions internes utilisées et communes à toutes les familles IDL-TSK-SWI-HWI.

a) **CLK_F_isr**
Objet de bas niveau utilisé par les fonctions HWI_TINT et HWI pour régler et utiliser les tranches de temps en interruption « tick ».

b) **PRD_F_tick**
Module horloge utilisé par la primitive **PRD_clock** pour régler les tranches de temps.

c) **PRD_F_swi**
Lancée par le programme **SWI** de plus haut niveau, **PRD_swi,** pour lancer les fonctions périodiques **PRD**.

d) **_KNL_run**
Lancée par le programme **SWI** de plus bas niveau, **KNL_swi,** pour lancer le gestionnaire de tâches **TSK**. C'est un nom précédé du caractère **« _ »** car cette fonction est appelée depuis un programme écrit en assembleur.

e) **_IDL_loop**
Lancée par l'objet **TSK** de plus bas niveau **TSK_idle** pour exécuter les fonction de type **IDL**.

f) **IDL_F_busy**
Lancée par la fonction **IDL_cpuload** chargée de calculer la charge CPU.

g) **RTA_F_dispatch**
Lancée par la fonction **RTA_dispatcher** pour faire l'analyse temps réel de l'application.

h) **LNK_F_dataPump**
Lancée par la fonction **LNK_datapump** de type **IDL** pour transférer l'analyse temps réelle vers le canal de communication de données **HST** avec le PC.

i) **HWI_unused**

Ce n'est pas une fonction, cela indique que l'interruption matérielle correspondante n'est pas utilisée.

REMARQUE

Ne jamais utiliser dans un programme un nom de la forme **MOD_** *et* **MOD_F_** *car ils sont utilisés en interne par le noyau et spécifiés par l'outil de configuration.*

15) Le nom des types de données

Compte tenu de la structure interne des DSP, DSP BIOS n'utilise par les types standard du « C » ANSI. Pour assurer la compatibilité entre processeur, on utilise des noms analogues mais dont la première lettre est une majuscule.

Types	Description
Arg	Type capable de définir soit un entier soit un pointeur
Bool	Variable Booléenne 0/1
Char	Valeur de caractère
Int	Entier signé
Lgint	Entier long signé
Types	**Description**
LgUns	Entier long non signé
Ptr	Pointeur générique
String	Chaîne de caractères terminée par un zéro (\0)
Uns	Entier non signé
Void	Type vide

Tous ces type sont définis dans le fichier « std.h », et peuvent être utilisés par les API de DSP BIOS.
De manière classique la constante NULL (0) (pour les pointeurs) et les Booléens TRUE (1) / FALSE (0) sont définis.

Les objets et les structures prédéfinis par les API de DSP BIOS on un nom de la forme **MOD_Obj**. Si on utilise ce type d'objet il doit être défini en externe dans le programme

```
extern LOG_Obj trace ;
```

16) Le nom des segments mémoire et leur utilisation par DSP BIOS

a) Le nom des segments mémoire

Segment Mémoire	Description
IDATA	Mémoire interne (on chip memory)
EDATA	Premier block de mémoire externe
EDATA1	Second block de mémoire externe
IPROG	Mémoire programme interne (on chip memory)
EPROG	Premier block de mémoire programme externe
EPROG1	Second block de mémoire programme externe
USERREGS	Page 0 de la mémoire utilisateur (28 mots/words)
BIOSREGS	Page 0 réservée pour les registres (4 mots/words)
VECT	Segment des vecteurs d'interruption

b) Utilisation des segments mémoire

Segment Mémoire	Utilisation
IDATA	Pile de la mémoire d'application (stack)
EDATA	Mémoire des arguments de l'application
EDATA	Mémoire de constantes de l'application
IPROG	Mémoire programme de DSP BIOS
EDATA	Mémoire de données de DSP BIOS

IDATA	Mémoire de travail (heap) de DSP BIOS
EPROG	Mémoire du code de BOOT de DSP BIOS

Chapitre 2 : Génération de programme

A) Cycle de développement

PHASE 1 : Analyse de l'application :
Faire une analyse logique et temps réelle des programmes de l'application.

PHASE 2 : Ecriture du code de programme
Ecrire à l'aide de Code Composer Studio les programmes « C » ou « ASM » correspondant.

PHASE 3 : Les objets de DSP BIOS
A partir de l'étude ci-dessus utiliser l'outil de configuration de DSP BIOS pour créer le noyau temps réel. Sauvegarder la configuration

PHASE 4 : Compilation
A l'aide de Code Composer Studio faire la compilation et l'édition de tous le modules du programme.

PHASE 5 : Mise au point
Mettre au point l'application à l'aide des outils inclus dans DSP BIOS : « log » « trace » « statistique » « timing » « SWI » etc.

PHASE 6 : Répéter les opérations
Recommencer les phases de 2) à 5) jusqu'à la mise au point finale.

B) Utilisation de l'outil de configuration

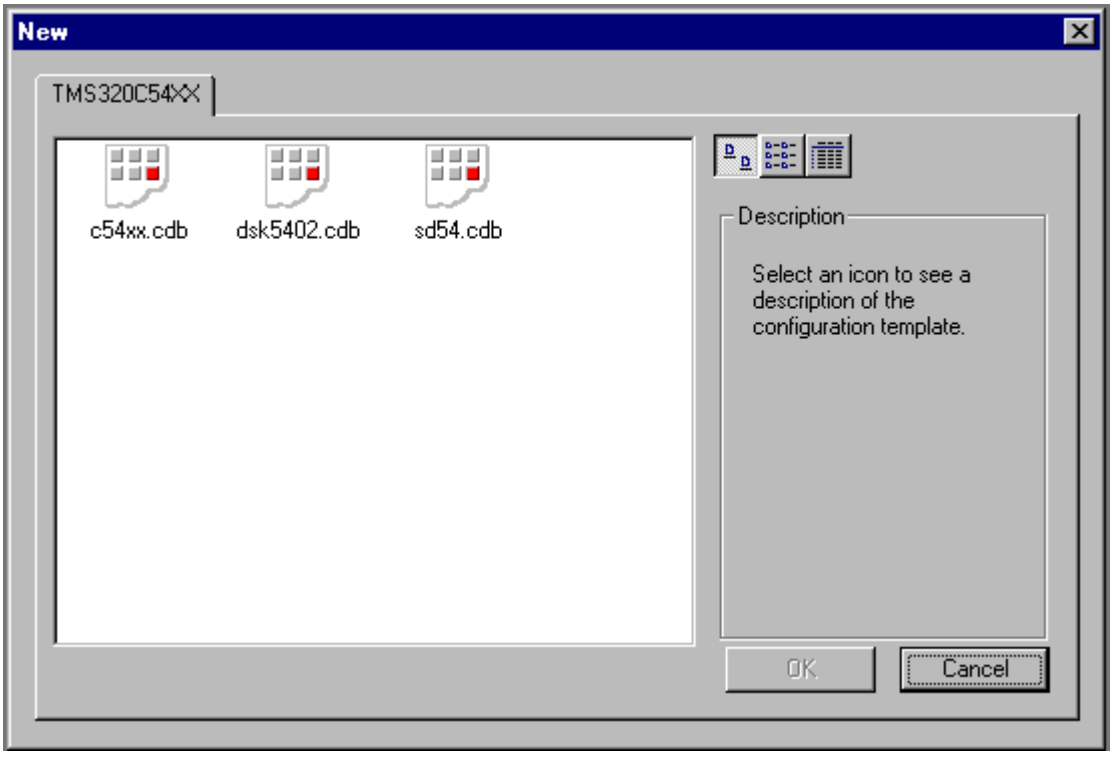
17) Création d'un module de configuration

La création d'un module de configuration peut se faire soir à partir du menu démarrer soit à partir de Code Composer Studio.

a) Menu démarrer
Exécuter la suite de commande :DEMARRER->PROGRAMMES->C5402 DSK Development Tools->Configuration Tool->New

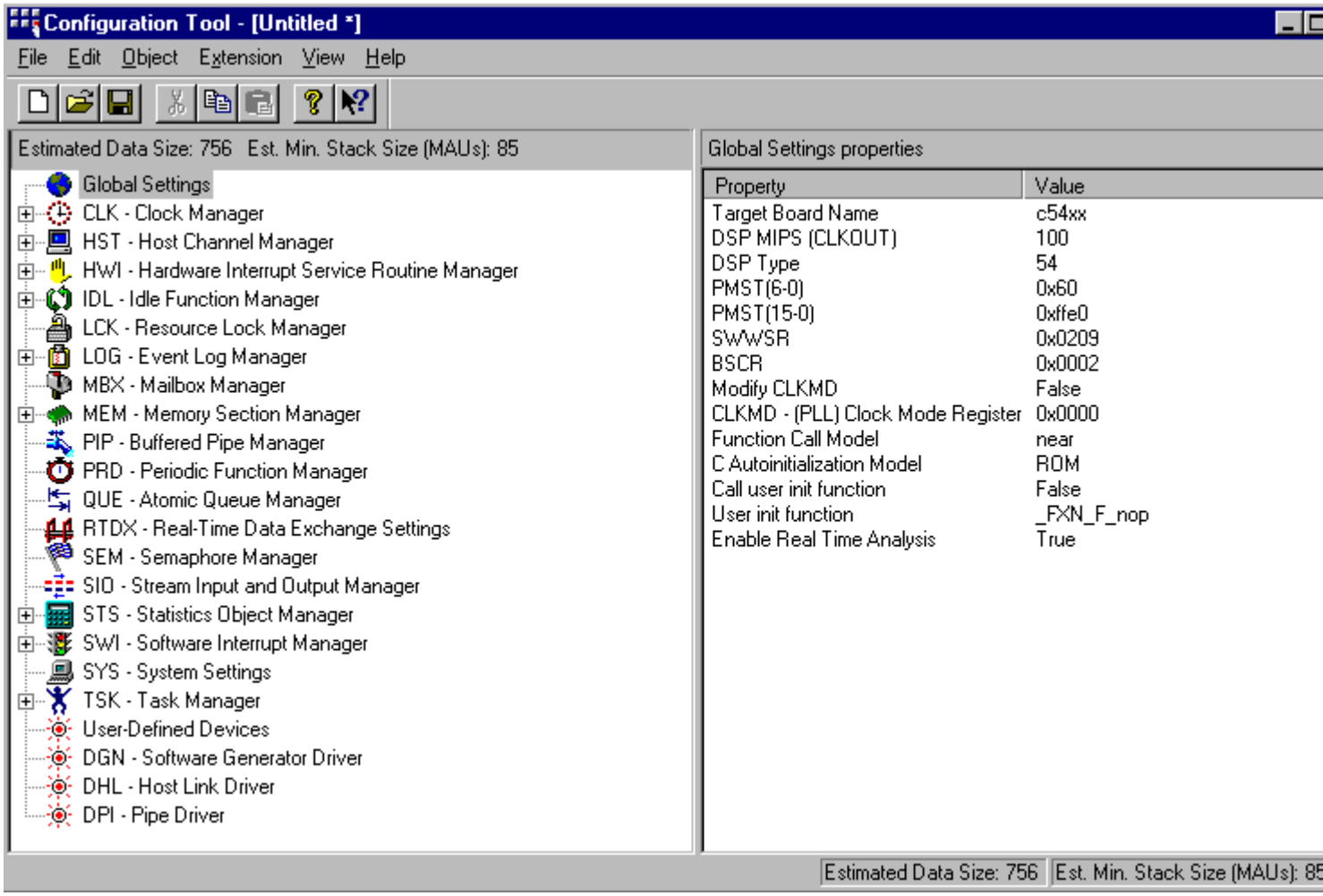
b) Code Composer studio
Exécuter la suite de commande : File->New->DSP BIOS Config

On se retrouve avec la figure ci-dessous : chaque fichier proposé correspond à une configuration matérielle donnée. Il est possible de choisir son type de carte ou de créer sa propre configuration.



18) Les propriétés globales d'une application DSP BIOS

Les propriété globales d'un module, **Global Settings,** définissent le type de carte CPU, le type de DSP, la fréquence d'horloge, les adresses des registres, le type d'initialisation etc.



19) Création d'objets statiques

Un objet statique est crée par l'outil de configuration de BIOS DSP. Ces objets peuvent être utilisés pendant toute la vie du programme et sont présents dans le code binaire du noyau.

a) Pour créer un objet de ce type il faut faire la séquence suivante :

- ✓ Click droit sur la famille l'objet : **TSK, LOG, SWI etc.**
- ✓ Sélectionner : **insert MOD**
- ✓ Click droit sur l'objet : **Rename**
- ✓ Click droit sur l'objet : **Properties**
- ✓ Changer les propriétés : **FIN par OK**

b) Avantages de ce type d'objet

- ✓ Présence permanente en mémoire : analyse dynamique des statistiques
- ✓ Taille de code binaire réduite : 50 % pour XXX_create ou XXX_delete
- ✓ Rapidité du code : pas de création dynamique

c) Limitation de ce type d'objet

- ✓ Ne pas créer d'objets inutiles : utiliser alors la création dynamique
- ✓ Impossible de les détruire : XXX_delete est inopérante

20) Création d'objets dynamique

La plupart des objets peuvent être crées de manière dynamique mais pas tous. Certains ne peuvent l'être que dans l'outil de configuration.

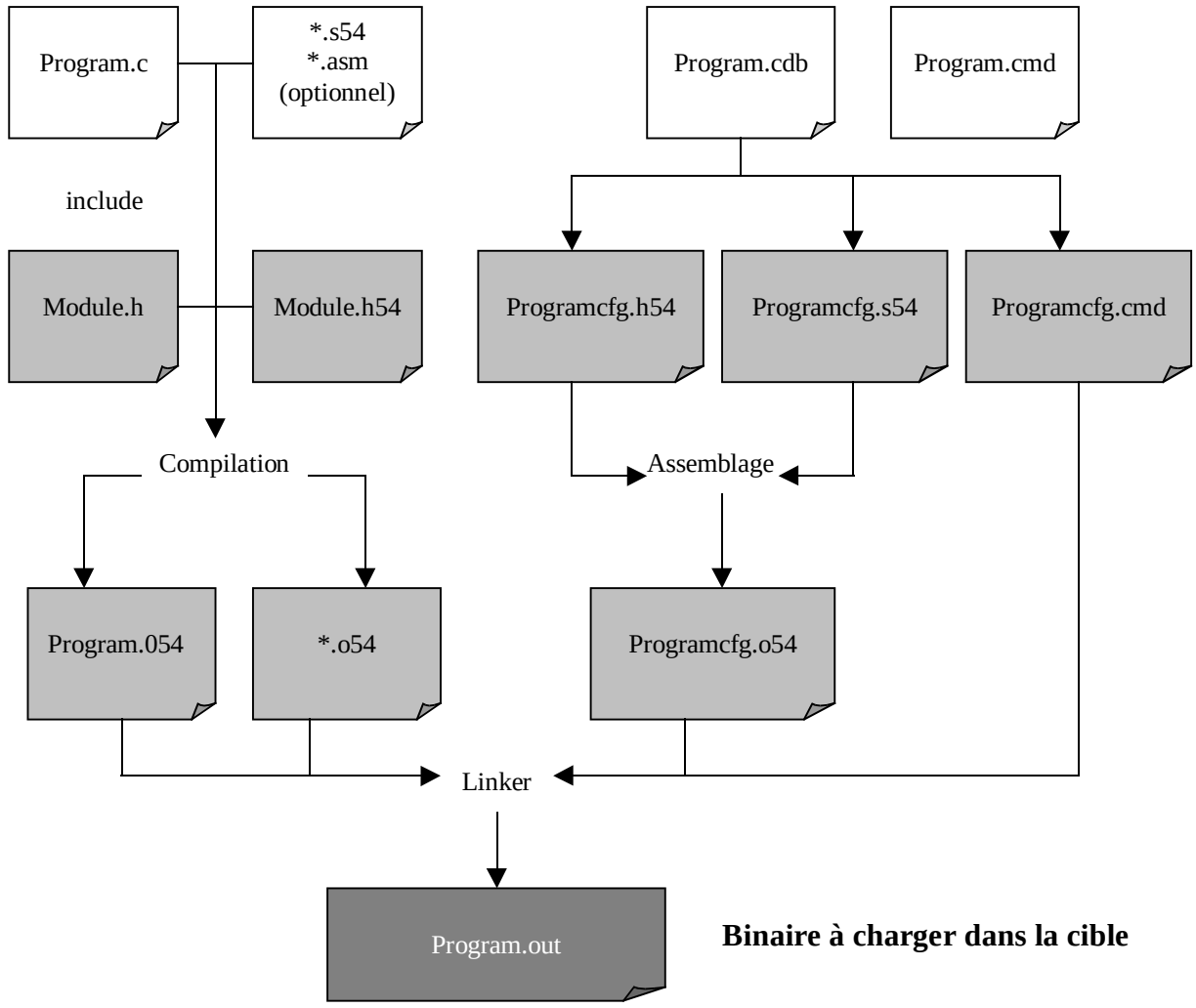
a) Création
- ✓ On utilise alors les fonctions XXX_create
- ✓ Paramètres d'appel : XXX_Attrs pointeur vers une structure de données
- ✓ Paramètre de retour : Handle (pointeur) vers l'objet

A la création d'un objet le noyau affecte une mémoire de travail et initialise la structure de données à une valeur constante par défaut XXX_ATTRS.

b) Destruction d'un objet
- ✓ On utilise la fonction XXX_delete.

21) Les fichiers de l'outil de configuration

La figure ci-dessous représente tous les fichiers de l'outil de configuration.
En fond blanc on trouve les fichiers modifiables par l'utilisateur.



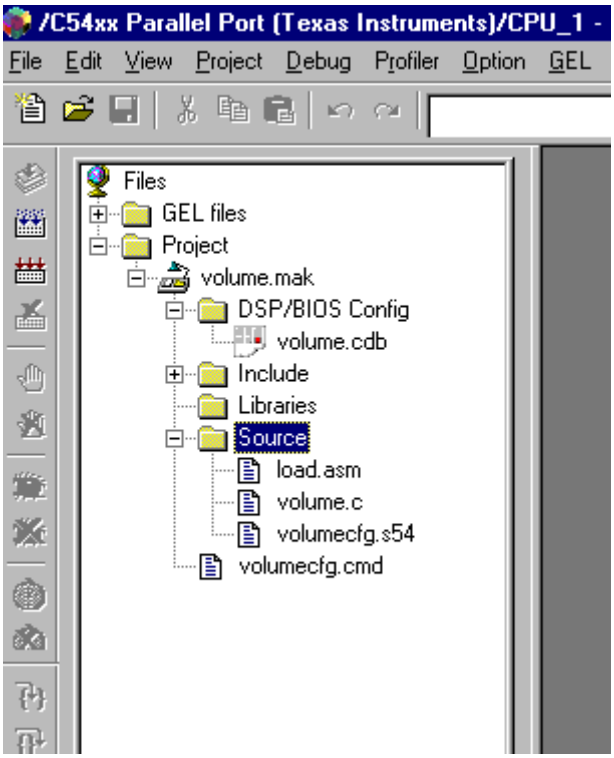
Le signe **54** indique que l'on génère du code binaire pour tous les composants de la famille TMS320C54xx.

Les extensions des fichiers ont pour significations :

- ✓ Program.c : fichier source contenant la fonction « main »
- ✓ *.asm : programme source assembleur
- ✓ Module.h : fichier entête
- ✓ Module.h54 : fichier entête des DSP BIOS API pour les programmes « C »
- ✓ Program.o54 : binaire relogeable issu d'une compilation ou d'un assemblage
- ✓ Program.cdb : fichier crée par l'outil de configuration
- ✓ Programcfg.h54 : fichier d'en-tête crée par l'outil de configuration.
- ✓ Programcfg.s54 : programme assembleur issu de l'outil de configuration
- ✓ Programcfg.cmd : fichier de commande de l'éditeur de liens de DSP BIOS.
- ✓ Programcfg.o54 : module relogeable crée par l'outil de configuration
- ✓ Program.cmd : fichier de commande de l'éditeur de liens (optionnel)
- ✓ Program.out : fichier binaire exécutable à charger dans la cible

22) Compilation et édition de liens

La compilation et l'édition de liens se font à l'aide de la notion de projet sous Code Composer Studio.



23) Séquence de démarrage de DSP BIOS

La séquence de démarrage de DSP BIOS est la suivante :

- ✓ Initialisation du DSP sur le vecteur RESET **C_int0** et du pointeur de pile **STACK**.
- ✓ Appel de BIOS_init et de sa macro MOD_init
  - HWI_init efface les interruption matérielles
  - HST_init lance la communication avec le PC
  - IDL_init calibre la boucle sans fin avec le timer
- ✓ Appel du programme « main(argc,argv,envp) » de l'utilisateur. Les paramètres d'appel sont donnés par DSP BIOS init (les interruptions sont encore bloquées). Il est possible de faire toutes les initialisations matérielles de la carte
- ✓ Appel de BIOS_start. Ce module est lancé lors de l'exécution de l'instruction « return » du programme principal. Ce programme valide tous les modules de DSP BIOS par appel de la procédure MOD_startup
  - CLK_startup initialise le timer pour le CLK manager et le masque d'interruption IMR
  - PIP_startup, SWI_startup, TSK_startup et HWI_startup pour tous les modules correspondant
- ✓ Lance la boucle de sommeil par appel de la fonction IDL_loop

Chapitre 3 : Instrumentation

A) principes généraux de l'instrumentation

24) objectifs et performances

a) Analyse temps réel

L'analyse temps réel permet de savoir si l'application répond aux critères du cahier des charges. Les méthodes classiques consistent à mettre de point d'arrêt et d'observer l'état du système.

Ces méthodes ne sont plus valables dans un système temps réel : l'application est arrêtée et on perd le temps réel.

Une autre méthode consiste à utiliser des analyseurs logiques et des émulateurs. A la fréquence de fonctionnement actuelle des processeurs cela devient aussi impossible.

DSP BIOS contient des fonctions logicielles qui permettent cette analyse de performances et transmettent les données quand le DSP est dans la boucle IDL : cela ne perturbe que peu l'application temps réel.

b) Instrumentation logicielle et matérielle

Une instrumentation matérielle utilise des oscilloscopes, des analyseurs logiques des émulateurs : elle est lourde à manipuler très onéreuse et limitée en fréquence.

L'instrumentation logicielle a besoin de code implanté dans la cible. Elle est flexible, simple, mais elle peut affecter les performances du système : il faut donc trouver un équilibre entre le fonctionnement de l'application et les perturbations engendrées par l'instrumentation.

c) Instrumentation explicite et implicite

- ✓ Une instrumentation implicite est toujours faite par DSP BIOS sans que l'utilisateur ne le demande. Par exemple l'exécution des interruptions logicielles est toujours faite par un objet **LOG** de nom **« LOG_system »**. Elles sont validées par un menu de Code Composer Studio.
- ✓ Une instrumentation **explicite** est faite par l'exécution d'une **primitive système écrite dans le programme utilisateur : exemple LOG_printf.**

d) Les performances obtenues

- ✓ L'instrumentation logicielle de DSP BIOS charge de 1% le CPU
- ✓ Elle communique avec le PC pendant la boucle IDL de plus faible priorité
- ✓ On peut contrôler à partir du PC le rythme d'analyse et arrêter toute fonction inutile.
- ✓ Les données ne sont enregistrées que si elles sont validées à partir du PC.
- ✓ Les données sont mises en forme par le PC et non par la cible.
- ✓ Les modules LOG, STS et TRC sont très rapides :
  - LOG_printf, LOG_event : 30 instructions.
  - STS_add : 30 instructions.
  - STS_delta : 40 instructions.
  - TRC_enable, TRC disable : 4 instructions.
- ✓ Chaque objet STS (statistique) est formé de 8 word à transférer au PC.
- ✓ Un ensemble de statistique est traduit sous forme de variables 32bits.
- ✓ La taille des buffers LOG peut être choisie à volonté.
- ✓ Par défaut l'instrumentation n'est pas validée : sinon chaque interruption d'analyse demande entre 20 et 30 instructions.

25) Les outils d'instrumentation API implicite.

Ces outils de mesure sont toujours présents dans toute application DSP BIOS.

Ces outils d'instrumentation API sont de plusieurs types :

- ✓ Les outils qui font l'acquisition de données LOG (analyse d'objets) STS (statistiques)
- ✓ Les outils de contrôle de l'analyse TRC (Trace Manager)
- ✓ Les outils de transfert vers le PC HST (Host Channel Manager)

a) Le gestionnaire d'événements **LOG**

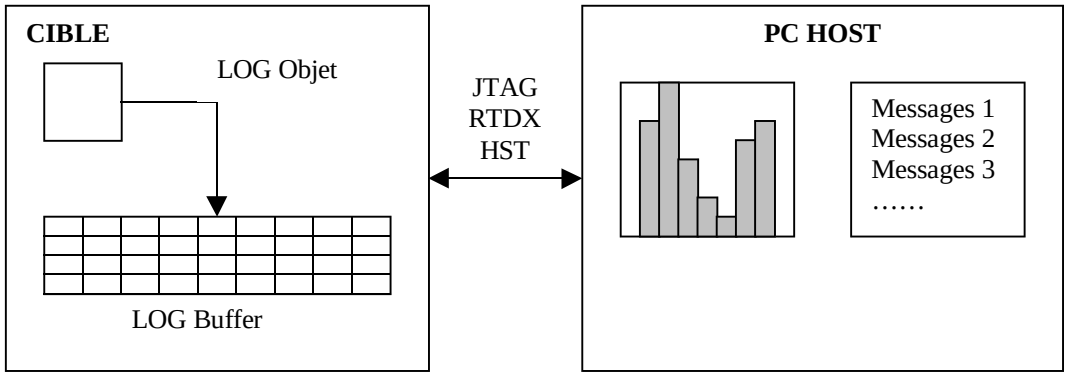
Les objets du module LOG capturent les événements pendant le fonctionnement temps réel de l'application. On peut utiliser directement le graphe d'exécution ou créer des objets avec l'outil de configuration de DSP BIOS.

Les fonctions sont les suivantes :

- **LOG_disable**. Invalide le système LOG
- **LOG_enable**. Valide le système LOG
- **LOG_error**. Ecrit un message d'erreur sur le système LOG
- **LOG_event**. Attache un message non formaté au message LOG
- **LOG_message**. Ecrit un message d'événement utilisateur sur le système LOG
- **LOG_printf**. Envoie un message formaté au message LOG
- **LOG_reset**. Initialise le système LOG

Les objets LOG crées par l'utilisateur utilisent soit des buffers de trace fixes ou soit circulaires.

- ✓ Buffer fixe : le stockage des données s'arrête quand le buffer est plein
- ✓ Buffer circulaire : quand le buffer est plein on écrase les données du début
- ✓ Les caractéristiques du buffer , taille, type, sont à préciser dans l'outil de configuration de DSP BIOS.



Exemple d'un programme **DSP BIOS** utilisant la fonction la primitive **LOG_printf** dans le buffer de nom **« trace »**

```
/************************************************/
/*                                              */
/*    H E L L O . C                             */
/*                                              */
/*    Basic LOG event operation from main.      */
/*                                              */
/************************************************/

/* DSP/BIOS header files*/
#include <std.h>
#include <log.h>

/* Objects created by the Configuration Tool */
extern far LOG_Obj trace;

/*
 *  ======== main ========
 */
Void main()
{
   LOG_printf(&trace, "hello world!");

   /* fall into DSP/BIOS idle loop */
   return;
```

```
}
```

b) Le gestionnaire de statistiques **STS**

Ce module calcule des statistiques sur chaque objet crée par l'outil de configuration. Pour calculer ces statistiques on utilise les fonctions :

Fonctions

- **STS_add**. Initialise une statistique à une valeur donnée
- **STS_delta**. Calcule la différence entre une valeur précédente et le point courant
- **STS_reset**. Initialise un objet STS
- **STS_set**. Sauvegarde une valeur courante

La structure de données d'un objet STS a la forme suivante :

```
struct STS_Obj {
   LgInt    num;    /* count */
   LgInt    acc;    /* total value */
   LgInt    max;    /* maximum value */
}
```

Le PC affiche les données suivantes :

- **Count** : le nombre de valeurs dans une série de données fournies par l'application.
- **Total** : la somme arithmétique des valeurs de la série
- **Maximum** : la plus grande valeur de la série
- **Average** : la valeur moyenne de la série.

c) Le module de Trace **TRC**

Le module TRC valide ou invalide l'acquisition des données précédentes : cela permet de contrôler la charge de travail de la cible due à l'instrumentation.

Les fonctions de ce module sont :

- **TRC_disable**. Invalide la trace d'une classe de fonction.
- **TRC_enable**. Valide la trace d'une classe de fonction.
- **TRC_query**. Test l'état d'une analyse d'une classe de fonctions.

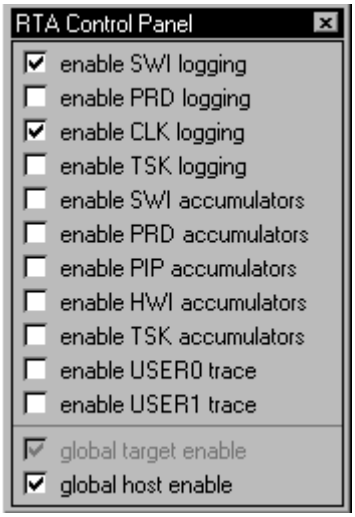
Pour une efficacité plus grande, la cible ne doit pas charger des informations de type LOG ou statistiques avant que la trace ne soit validée.

Les événements et les statistiques suivants explicites peuvent être tracés. Chaque expression correspond à une constante définie dans les fichiers « trc.h » et « trc.h54 » :

Constant	**Tracing Enabled/Disabled**	**Default**
TRC_LOGCLK	Log timer interrupts	off
TRC_LOGPRD	Log periodic ticks and start of periodic functions	off
TRC_LOGSWI	Log events when a software interrupt is posted and completes	off
TRC_LOGTSK	Log events when a task is made ready, starts, becomes blocked, resumes execution, and terminates	off
TRC_STSHWI	Gather statistics on monitored values within HWIs	off
TRC_STSPIP	Count number of frames read from or written to data pipe	off
TRC_STSPRD	Gather statistics on number of ticks elapsed during execution	off
TRC_STSSWI	Gather statistics on length of SWI execution	off
TRC_STSTSK	Gather statistics on length of TSK execution	off
TRC_USER0	Le programme peut directement utiliser ces bits pour valider ou invalider des actions d'instrumentation explicite. On peut utiliser **TRC_query** pour tester l'état d'un bit de validation pour réaliser ou non la fonction d'analyse. DSP/BIOS ne positionne ni n'utilise ces bits.	off
TRC_USER1	même chose que ci-dessus.	off
TRC_GBLHOST	Ce bit doit être mis à « 1 » pour valider toute instrumentation implicite. Cela valide ou arrête tous les types de trace. Ce bit est mis à un lors du « run time » sur le PC en utilisant le panneau de contrôle **RTA**.	off
TRC_GBLTARG	Ce bit doit aussi être mis à « 1 » pour valider toute instrumentation implicite. Il ne peut être modifié **que par le programme cible** et il est validé par défaut.	on
TRC_STSSWI	Assemble les statistiques sur la durée d'exécution des SWI.	off

REMARQUE

Pour pouvoir utiliser toutes ces fonctions implicites, elles doivent être validées par le panneau de contrôle **RTA : Real Time Analysis** de Code Composer Studio.



26) Les outils d'instrumentation explicite

Les outils de mesure peuvent être aussi lancés de manière explicite par appel à des primitives API dans le programme. En voici quelques exemples

a) Validation, invalidation et test de validité des mesures par programme

- valide les statistiques sur les fonctions de type SWI et PRD
  TRC_enable(TRC_STSSWI | TRC_STSPRD);

- invalide la trace des statistiques sur les fonctions de type SWI et PRD
  TRC_disable(TRC_LOGSWI | TRC_LOGPRD);

- teste l'état du bit TRC_LOGSWI.
  result = TRC_query(TRC_LOGSWI)

b) Mesure les temps d'exécution d'une partie de programme

```
STS_set(&sts, CLK_gethltime());
        ……..
        Programme à tester (validé par TRC_enable)
        ……..
STS_delta(&sts, CLK_gethltime());
```

c) Mesure de la charge CPU dans un programme et validation par le bit **USER0 de RTA**

- *Fichier d'en-tête à inclure:*

```
#include <clk.h>
#include <sts.h>
#include <trc.h>
```

- *Déclaration de variable externe crées par l'outil de configuration de DSP BIOS :*

```
extern far STS_Obj processingLoad_STS;
```

- *Lignes à écrire dans la tâche avant l'appel de la fonction à mesurer :*

```
/* enable instrumentation only if TRC_USER0 is set */
if (TRC_query(TRC_USER0) == 0) {
        STS_set(&processingLoad_STS, CLK_gethtime());
}
        ……..
        Programme à tester (validé par TRC_enable)
```

……..

- *Lignes à écrire après l'appel de la fonction à mesurer :*

```
if (TRC_query(TRC_USER0) == 0) {
        STS_delta(&processingLoad_STS, CLK_gethtime());
}
```

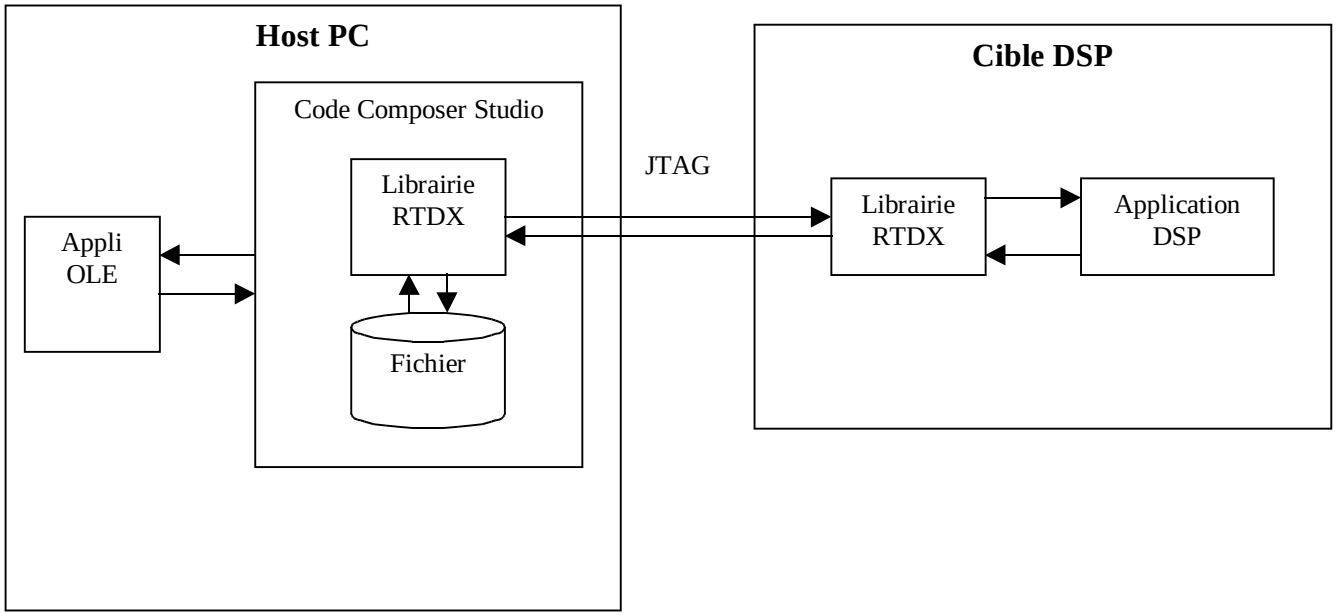
B) Les échanges temps réel avec le PC : RTDX

27) Rôle et domaines d'utilisation

Le système **R**eal **T**ime **D**ata e**X**change permet de transférer des données en temps réel entre un système cible DSP et une application OLE PC sous Windows.
Les données peuvent être transférer soit en mode continu sur l'écran, soit en mode non continu dans un fichier.
Une application de type OLE reçoit les données côté PC. Ces applications peuvent être « LabVIEW », « Excel », « Visual C++ » etc.
L'étude de ce module, indépendant de DSP BIOS, nécessite la connaissance de ces mécanismes sur PC et déborde du cadre de ce cours.
Code Composer Studio utilise cette méthode pour ses échanges temps réel avec le PC.

28) Principes des transferts de données

Le principe des échanges est résumé sur le schéma ci-dessous :



29) Les primitives de API de RTDX

Déclaration Macros :
RTDX_CreateInputChannel
RTDX_CreateOutputChannel

Fonctions:
RTDX_channelBusy
RTDX_disableInput
RTDX_disableOutput
RTDX_enableInput
RTDX_enableOutput
RTDX_read
RTDX_readNB
RTDX_sizeofInput
RTDX_write

Macros:

RTDX_isInputEnabled
RTDX_isOutputEnabled

Variable:

RTDX_writing

Pour tout détail sur ces fonctions consulter la documentation technique des API de Code Composer Studio

Chapitre 4 : Gestion des tâches

A) La préemption entre les familles de tâches

30) Les familles de tâches et leurs niveaux de priorité

a) Les niveaux des familles de tâches

Les niveaux de priorité ont été vus dans un chapitre précédent. Dans l'ordre des priorités croissantes ont a :

IDL -> TST -> SWI -> HWI

Les fonctions périodiques

PRD (SWI) -> CLK (HWI)

b) Choix du type de tâche à utiliser

Pour bien organiser son application il y a quelques règles à respecter dans le choix des objets et des tâches.

- ✓ HWI : se sont les interruptions matérielles dues aux circuits périphériques internes et externes. Pendant une tâche de ce type les interruptions sont masquées et on bloque toutes les autres fonctions. Elles doivent donc être **TRES COURTES** et sont souvent écrites en assembleur. **Leur rôle est de lancer des tâches de type SWI ou TSK** sur l'apparition d'un événement matériel. Elles se terminent toujours par l'instruction « return » et ne peuvent être interrompues que par une HWI de niveau supérieur.

- ✓ SWI : l'utilisation de ce type de tâche est bien adaptée aux fonctions dont l'interdépendance est relativement simple et qui partagent des structures de données simples. Elles se terminent toujours par l'instruction « return » et ne peuvent être interrompues que par une SWI de niveau supérieur ou une HWI. **Lorsqu'une SWI est lancée, il faut que les données soient disponibles**. Il existe cependant des objets SWI de type boîte à lettres (mailbox) pour vérifier si la ressource est disponible

- ✓ TSK : les tâches de ce type peuvent supporter **une organisation plus complexe** : elles peuvent être suspendues mises en sommeil, partager des objets complexes avec un nombre important de primitives.

- ✓ IDL : il faut réserver à ces tâches les fonctions **non critiques** pour le fonctionnement en temps réel de l'application. : communication avec le PC, analyse du fonctionnement du programme etc.

- ✓ CLK : ces fonctions sont déclenchées par les **interruptions du timer** : elles fonctionnement comme les HWI en utilisant la durée des tranches de temps (ticks) comme unité. L'objet correspondant est PRD_clk.

- ✓ PRD : ces fonctions périodiques fonctionnent **comme les SWI.** Elles ont toute la même priorité et utilisent un multiple de la période du timer système.

31) Propriétés et préemption entre les familles de tâches HWI-SWI-TSK-IDL

a) Comparaison des propriétés des familles de tâches

	HWI	**SWI**	**TSK**	**IDL**
Priorité	La plus forte	Second niveau	Troisième niveau	La plus faible
Nombre de niveaux	Fonction du composant	14 + 2(PRD+TSK)	15 + 1 (IDL)	1
Arrêt / Relance	Non (-> return)	Non (-> return)	Oui	Non
Etats	Inactive, prête, en cours	Inactive, prête, en cours	Prête, en cours, bloquée, terminée	Prête, en cours
Invalidée par :	HWI_disable	SWI_disable	TSK_disable	Exit du programme
Lancée ou mise en état prête par	Interruption matérielle	SWI_post, SWI_andn, SWI_dec, SWI_inc, SWI_or	TSK_create	Appel à « return » dans main()

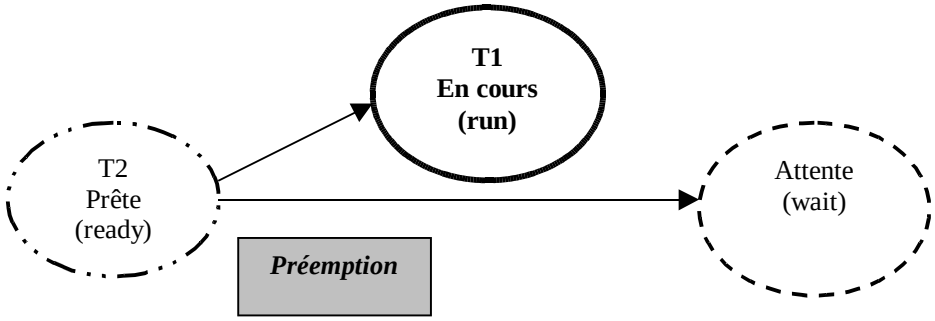
Pile utilisée	Système	Système	Tâches (1 pile/tâche)	Tâches
	HWI	**SWI**	**TSK**	**IDL**
Sauvegarde du contexte	Modifiable	Fixe (qq registres)	Complet dans la pile	Pas de sens
Sauvegarde du contexte lors d'un blocage	Pas de sens	Pas de sens	Contexte sauvé pour les fonctions « C »	Pas de sens
Partage de données avec les taches par	Streams, queues, pipes, variables globales	Streams, queues, pipes, variables globales	Streams, queues, pipes, variables globales, locks, mailbox	Pas de sens
Synchronisation par objets	Pas de sens	SWI-mailbox	Sémaphores et boîte à lettres (mailbox)	Pas de sens
Synchronisation par fonctions	Non	Non	Oui : create, delete, exit, task, switch, ready	Non
Création statique	Oui par défaut	Oui	Oui	Oui
Création dynamique	ISR	Oui	Oui	Non
Changement priorité	Non	Oui	Oui	Non
Lancement implicite	Non	Oui : post et events	Oui : ready, start block, resume, termination, events	Non
Statistiques implicites	Valeurs analysées	Temps d'exécution	Temps d'exécution	Non

b) Préemption entre les familles de tâches HWI-SWI-TSK-IDL

On peut bloquer une famille de tâches :

- ✓ Les HWI sont bloquées si la tâche en cours invalide les interruptions matérielles : HWI_disable ou HWI_enter.
- ✓ Les SWI sont bloquées si la tâche en cours invalide les interruptions logicielles : SWI_disable.
- ✓ Les TSK sont bloquées si la tâche en cours invalide l'algorithme de préemption : TSK_disable.
- ✓ Soit une application qui a une tâche **T1 en cours** de type soit HWI, soit SWI, soit TSK, soit IDL. On considère qu'une **tâche T2 prête à passer en mode « en cours » (run)** arrive. Le tableau ci-dessous indique dans quel état se met la tâche **T2**, **run ou wait**, en fonction de son **état initial** et de celui de la tâche **T1**.

Algorithme de préemption entre les états « prêt » « en cours » et « attente »



P -> préemption du CPU par la tâche T2 : elle devient en cours (run).
W -> mise en attente de la tâche T2 elle devient (wait).
W* -> mise en attente de T2 (wait) jusqu'à la validation des interruptions correspondantes.

Tâche T2 : prête	T1 : HWI (en cours)	T1 : SWI (en cours)	T1 : TSK (en cours)	T1 : IDL (en cours)
HWI supérieure	P	P	P	P
HWI_disable	W*	W*	W*	W*
HWI inférieure	W	Pas de sens	Pas de sens	Pas de sens
SWI supérieure	Pas de sens	P	P	P
SWI_disable	W	W*	W*	W*
SWI inférieure	W	W	Pas de sens	Pas de sens
TST supérieure	Pas de sens	Pas de sens	P	P
TSK_disable	W	W	W*	W*
TSK inférieure	W	W	W	Pas de sens

B) Les tâches de type HWI

32) Les interruptions matérielles sur un DSP C5x

a) Les sources d'interruption

Les interruptions matérielles peuvent être soit d'origine interne soit d'origine externe

✓ Les interruptions externes

Ces interruptions correspondent à des lignes d'entrée sur le DSP. Elles sont actives au niveau bas. On trouve les lignes suivantes :

- RESET : permet d'initialiser le DSP en lançant le programme associé au vecteur d'adresse 0xff80
- NMI : Non Masquable Interrupt. Cette interruption est toujours prise en compte par le DSP
- INT0-INT3 : Ces quatre lignes correspondent à des interruptions masquables par programme

✓ Les interruptions internes

Ces interruptions sont générées par les circuits périphériques implantés sur la puce du DSP

- TINT : l'interruption associée au timer interne
- RINT0-XINT0 : les interruptions en réception et transmission du premier port série McBSP
- RINT1-XINT1 : les interruptions en réception et transmission du second port série McBSP

b) Les registres de contrôle

✓ Le registre **d'état ST0** : ce registre contrôle le fonctionnement du DSP.

Bits 15-13	Bit 12	Bit 11	Bit 10	Bit 9	Bits 8-0
ARP (compatibilité avec les C5x)	TC	C	OVB	OVA	Pointeur de page mémoire DP

- Les 3 bits ARP permettent la compatibilité des modes d'adressage entre tous les C5x.
- Le bit TC permet de stocker le résultat de l'ALU
- Le bit C et le bit de retenue
- Les bits OVA et OVB indiquent une saturation des accumulateurs A et B.
- Les bits 8-0 servent de pointeur de page mémoire.

✓ Le registre **d'état ST1** : ce registre contrôle le fonctionnement du DSP. Le bit 11 de ce registre, **INTM**, permet de masquer les interruptions INT0-INT3, TINT, RINT0-XINT0, RINT1-XINT1.

Bit 15	Bit 14	Bit 13	Bit 12	**Bit 11**	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4-0
BRAF	CPL	XF	HM	**INTM**	0	OVM	SXM	C16	FRAC	CMPT	ASM

- INTM=0 : les interruptions sont autorisées
- INTM=1 : les interruptions sont masquées

Ce bit est mis à un lors du service d'une interruption. Pour la signification des autres bits voir la documentation du DSP correspondant.

✓ Le registre des masques des interruptions **IMR**

Ce registre IMR (Interrupt Mask Register) permet de masquer les interruptions : un bit à 1 autorise l'interruption correspondante :

Bits 15-9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Réservé	INT3	XINT1	RINT1	XINT0	RINT0	TINT	INT2	INT1	INT0

c) Les vecteurs d'interruption

Un vecteur d'interruption contient l'adresse du sous programme lancé lors de l'arrivée du signal correspondant. La table des vecteurs est initialement implantée en 0xff80. Elle peut être déplacée sur le début d'une page de 128 mots en modifiant le contenu du registre **PMST** (état des modes du DSP) : pointeur des vecteurs **IPTR** (Interrupt PoinTeR).

Bit 15-7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
IPTR	MC/PC	OVLY	AVIS	DROM	CLKOFF	SIMULT	SSTT

Pour la signification des autres bits voir la documentation du DSP correspondant.

Table des vecteurs d'interruption d'un TMS320C5402

TRAP/INTR Number (K)	Priority	Name	Location (Hex)	Function
0	1	RS/SINTR	0	Reset (hardware and software reset)
1	2	NMI/SINT16	4	Nonmaskable interrupt
2	–	SINT17	8	Software interrupt #17
3	–	SINT18	C	Software interrupt #18
4	–	SINT19	10	Software interrupt #19
5	–	SINT20	14	Software interrupt #20
6	–	SINT21	18	Software interrupt #21
7	–	SINT22	1C	Software interrupt #22
8	–	SINT23	20	Software interrupt #23
9	–	SINT24	24	Software interrupt #24
10	–	SINT25	28	Software interrupt #25
11	–	SINT26	2C	Software interrupt #26
12	–	SINT27	30	Software interrupt #27
13	–	SINT28	34	Software interrupt #28
14	–	SINT29	38	Software interrupt #29
15	–	SINT30	3C	Software interrupt #30
16	3	INT0/SINT0	40	External user interrupt #0
17	4	INT1/SINT1	44	External user interrupt #1
18	5	INT2/SINT2	48	External user interrupt #2
19	6	TINT0/SINT3	4C	Timer0 interrupt
20	7	BRINT0/SINT4	50	McBSP #0 receive interrupt
21	8	BXINT0/SINT5	54	McBSP #0 transmit interrupt
22	9	DMAC0/SINT7	58	DMA channel 0 interrupt
23	10	TINT1/DMAC1/ SINT7	5C	Timer1 interrupt (default) or DMA channel 1 interrupt.
24	11	INT3/SINT8	60	External user interrupt #3
25	12	HPINT/SINT9	64	HPI interrupt
26	13	BRINT1/DMAC2/ SINT10	68	McBSP #1 receive interrupt (default) or DMA channel 2 interrupt
27	14	BXINT1/DMAC3/ SINT11	6C	McBSP #1 transmit interrupt (default) or DMA channel 3 interrupt
28	15	DMAC4/SINT12	70	DMA channel 4 interrupt
29	16	DMAC5/SINT13	74	DMA channel 5 interrupt
120–127	–	Reserved	78–7F	Reserved

33) Génération de la table des vecteurs avec l'outil de configuration

Une interruption matérielle se déroule jusqu'à la fin de son code jusqu'à l'instruction « return » : elle ne doit donc jamais se bloquer et être très courte. Elle utilise la pile système.

a) Configuration

L'outil de configuration permet de :

- ✓ Fixer le nom du sous programme ISR.
- ✓ Fixer le niveau de priorité de ce sous programme
- ✓ Mettre à jour la table des vecteurs
- ✓ Choisir le segment mémoire utilisé pour la table des vecteurs

b) Validation et invalidation

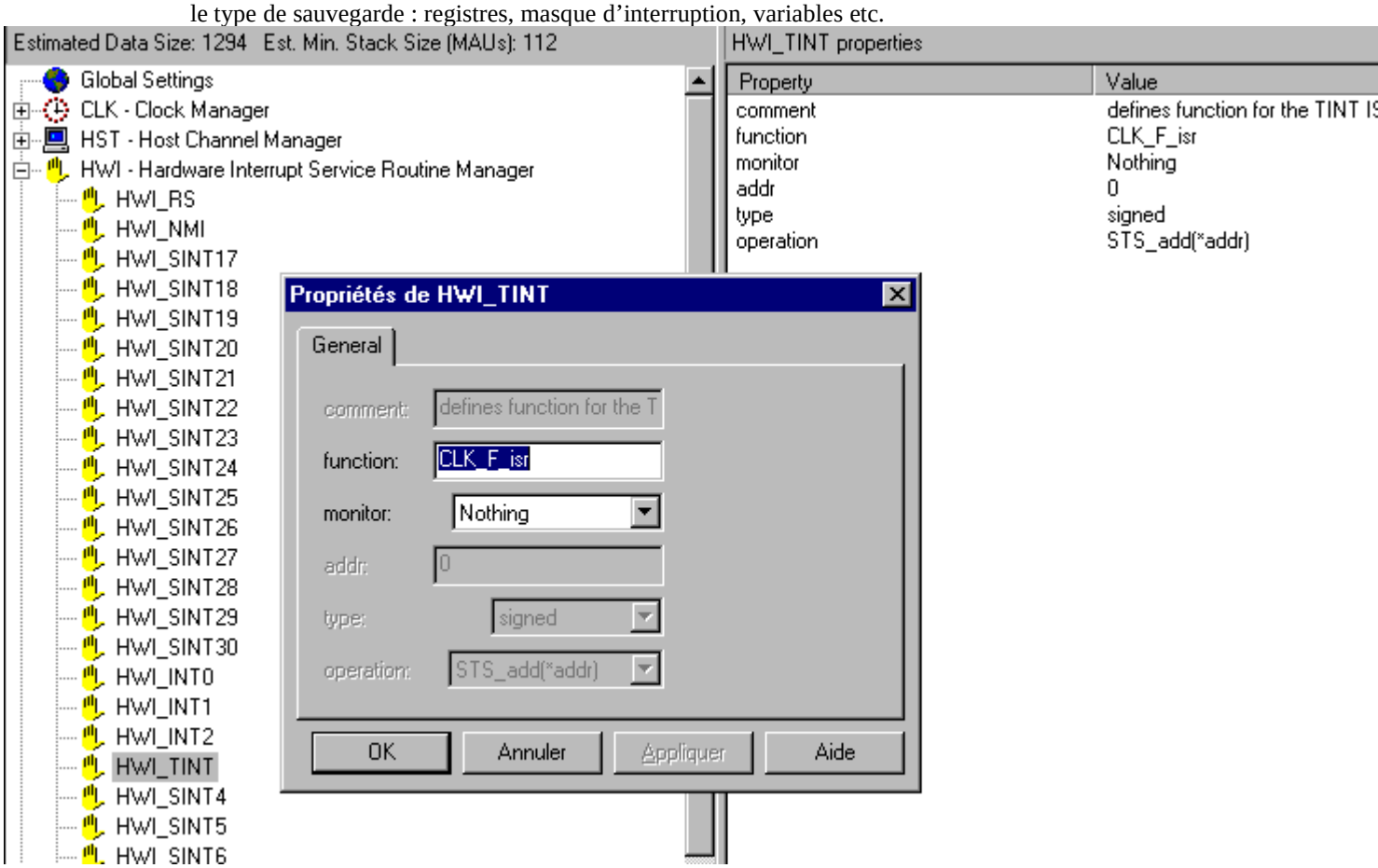
La fonction HWI_disable() permet de masquer les interruptions en positionnant le bit INTM à 1.

Pour valider de nouveau les interruptions on peut utiliser les fonctions :

- HWI_enable() : positionne le bit INTM à 0.
- HWI_restore() : replace le bit INTM à l'état précédent l'instruction HWI_disable

c) Sauvegarde du contexte

Placée au début et à la fin de la routine, les fonctions HWI_enter() et HWI_exit() permettent de sauver et de restaurer le contexte du DSP lors d'une interruption matérielle. Les paramètres passés permettent de choisir le type de sauvegarde : registres, masque d'interruption, variables etc.



C) Les tâches de type SWI

34) Principes généraux et utilisation

a) Fonctionnement d'une tâche SWI

- ✓ Une tâche de type SWI est utilisée dans une application quand les contraintes de temps sont moins sévères que celles des HWI.
- ✓ Pour gérer ces tâches on utilise des objets de type SWI_obj.
- ✓ Comme pour les HWI elle se déroule jusqu'à la fin de son code jusqu'à l'instruction « return » : elle ne doit donc jamais se bloquer et être courte. Elle utilise aussi la pile système. Elle peut être interrompue par une tâche de type HWI ou une tâche de type SWI de priorité supérieure.
- ✓ Chaque tâche SWI possède une variable sur 32 bits qui sert de **« mailbox »**.
- ✓ Les tâches SWI peuvent être validées par la fonction **key=SWI_enable()** et invalidées par la fonction **SWI_disable(key).**
- ✓ Le gestionnaire de tâches TSK est une SWI de niveau le plus bas **TSK_swi.**
- ✓ Il est possible d'avoir 15 niveaux de priorité pour les SWI

b) Création et destruction d'une tâche SWI

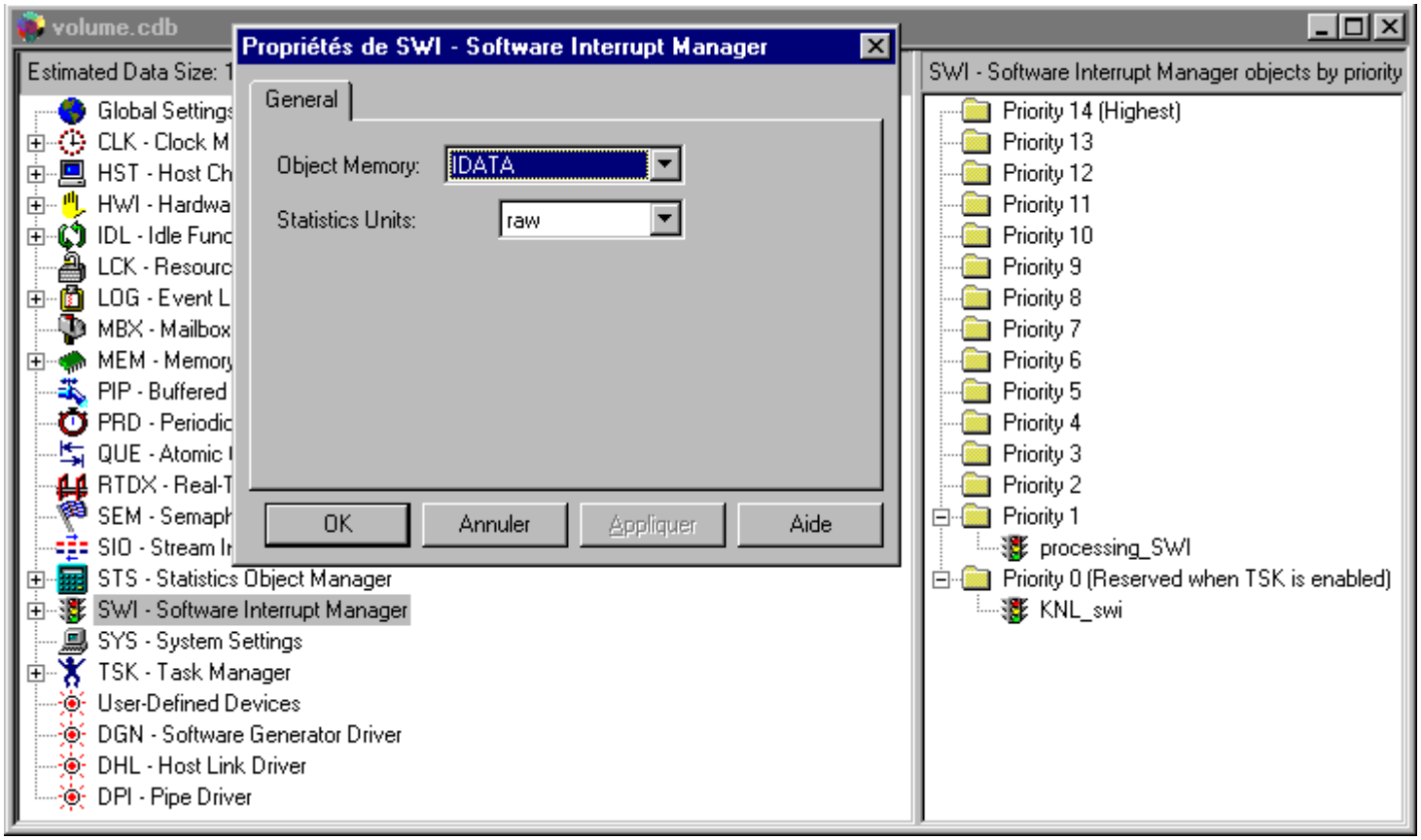
- ✓ Comme tous les objets de DSP BIOS elle peut être crée soit de manière dynamique par l'exécution d'une fonction **swi=SWI_create(attrs),** soit à l'aide de **l'outil de configuration**.
- ✓ Les tâches crées de manière dynamique peuvent être détruites par la fonction **SWI_delete(swi).**
- ✓ Il est possible de connaître les attributs d'une tâche par la fonction **SWI_getattrs(swi, attrs).**
- ✓ Il est possible d'utiliser la fonction **TSK_setpri()** pour changer la priorité d'une tâche SWI. Ceci n'est pas possible pour les tâches HWI.

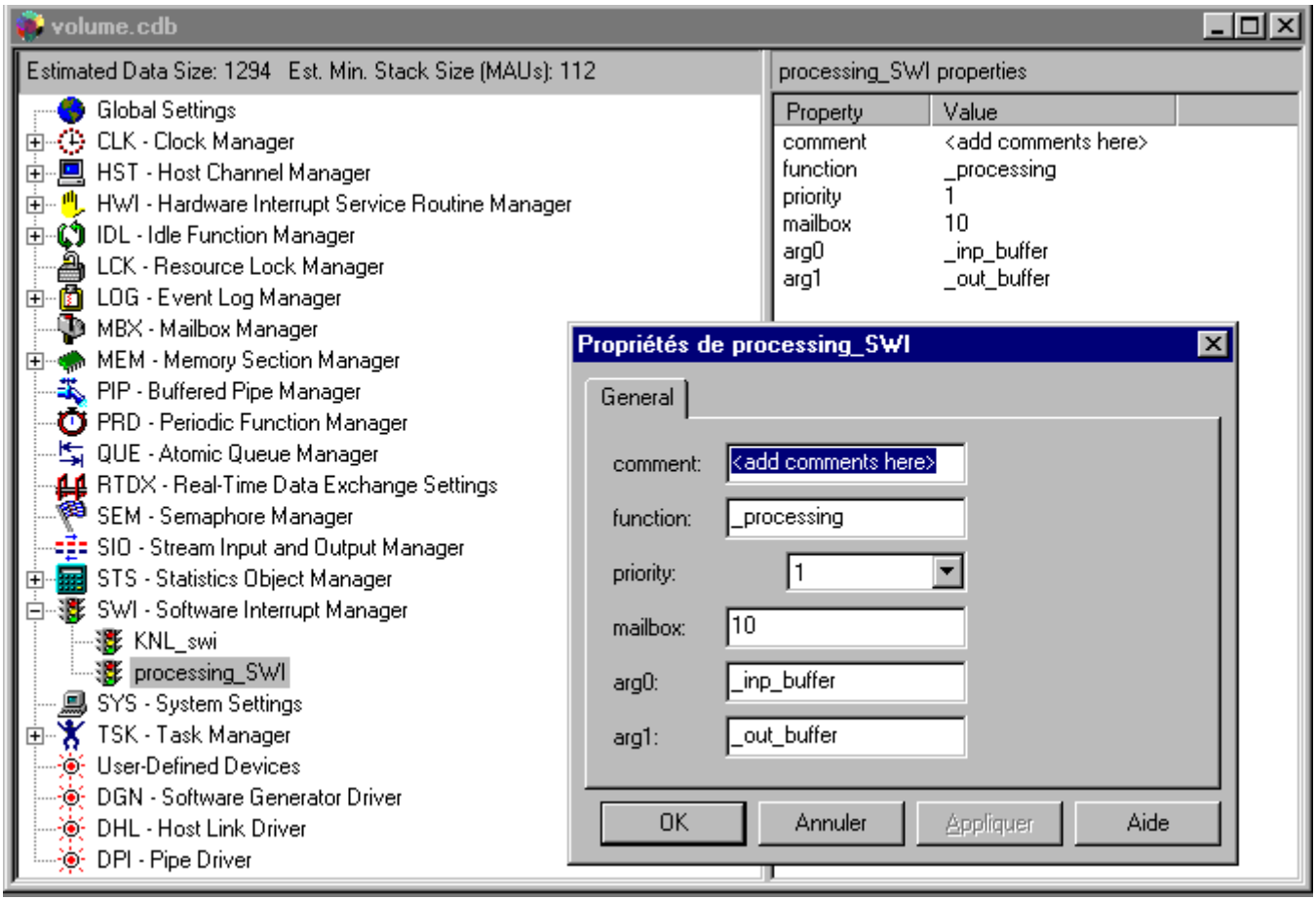
c) Utilisation de la pile système

- ✓ Les interruptions HWI et SWI utilisent la même pile pour sauver l'état des registres du DSP : la pile système.
- ✓ A chaque ajout d'une nouvelle priorité de SWI cette pile augmente.

- ✓ La pile système possède par défaut 256 mots : il faut veiller à ne pas dépasser cette taille. **(MEM module)**
- ✓ La création d'un objet périodique est aussi de type SWI : **PRD_swi.**

35) Création et priorité de tâches SWI statiques





36) Utilisation de la boîte à lettres : Mailbox SWI

Lors de la création d'une tâche de type SWI DSP BIOS lui associe une variable de 32 bits : **« mailbox ».** Cette variable a une valeur initiale et peut être utilisée de différentes façons suivant la primitive qui lance la tâche.

a) Fonctionnement de la « mailbox »
Lorsqu'une tâche SWI est crée la variable « mailbox » est mise à sa valeur initiale. Il en de même quand DSP BIOS retire la tâche de la file des tâches actives.
Certaines API gardent cette valeur, d'autres la modifient lors du lancement de la tâche. Cette variable peut servir de sémaphore à compte pour les fonctions de type SWI.
Il est possible de connaître cette valeur par l'API SWI_getmbox() : la valeur lue est celle obtenue au lancement de la tâche
Les API SWI_andn() et SWI_dec() lancent la tâche uniquement lorsque la « mailbox » vaut zéro.

b) Lancement d'une tâche SWI par la primitive **SWI_post()**
Cette fonction lance simplement la tâche SWI **sans changer** la valeur de la « mailbox ».

c) Lancement d'une tâche SWI par la primitive **SWI_inc()**
Cette fonction lance la tâche SWI en ajoutant un à la valeur de la « mailbox ». La valeur initiale est souvent « 0 ». Si dans une application plusieurs SWI_inc() sont exécutées et que la tâche correspondante n'a pas la priorité suffisante pour être exécutée, la « mailbox » peut alors servir de **compteur de boucle** pour exécuter plusieurs fois une partie du programme :

```
MyswiFunction()
{
        …….
        Repetition=SWI_getmbox() ;
        While(Repetition--){
                ……..loop…..
```

```
        }
        …..
}
```

d) Lancement d'une tâche SWI par la primitive **SWI_or()**
Cette fonction lance la tâche SWI si **un OU entre plusieurs événements** sont arrivés. La valeur initiale correspond alors au nombre de possibilités codées en binaire. Par exemple :
- ✓ Si la « mailbox » initiale vaut « 0….11 » en binaire, la fonction « SWI_or(&myswi , 0x1) lance la fonction avec la valeur binaire « 0….10 ». On peut par un test exécuter une partie du programme.
- ✓ Si la « mailbox » initiale vaut « 0….11 » en binaire, la fonction « SWI_or(&myswi , 0x2) lance la fonction avec la valeur binaire « 0….01 ». On peut par un test exécuter une autre partie du programme.

```
MyswiFunction()
{
        …….
         repetition=SWI_getmbox() ;
         switch repetition :{
                  case 0x2 : …….
                  case 0x1 : …….
        }
        …..
}
```

e) Lancement d'une tâche SWI par la primitive **SWI_andn()**
Cette fonction lance la tâche SWI avec un **ET entre plusieurs événements**. Pour que la fonction soit lancée **il faut que la « mailbox » ait pour valeur « 0 »**. La valeur initiale ne doit donc pas être nulle. Par exemple :
- ✓ si la « mailbox » initiale vaut « 0….11 » en binaire, la fonction « SWI_andn(&myswi , 0x1) place la « mailbox » à la valeur binaire « 0….10 », **mais ne lance pas la tâche.**
- ✓ si par la suite la fonction « SWI_or(&myswi , 0x2) est exécutée la « mailbox » passe à la valeur binaire **« 0…00 » et la tâche est lancée.**

f) Lancement d'une tâche SWI par la primitive **SWI_dec()**
Cette fonction lance la tâche SWI en enlevant un à la valeur de la « mailbox ». La valeur initiale ne doit donc pas être nulle. La fonction lance la tâche quand la **« mailbox » vaut 0**. Elle peut donc servir de **compteur d'événements.**

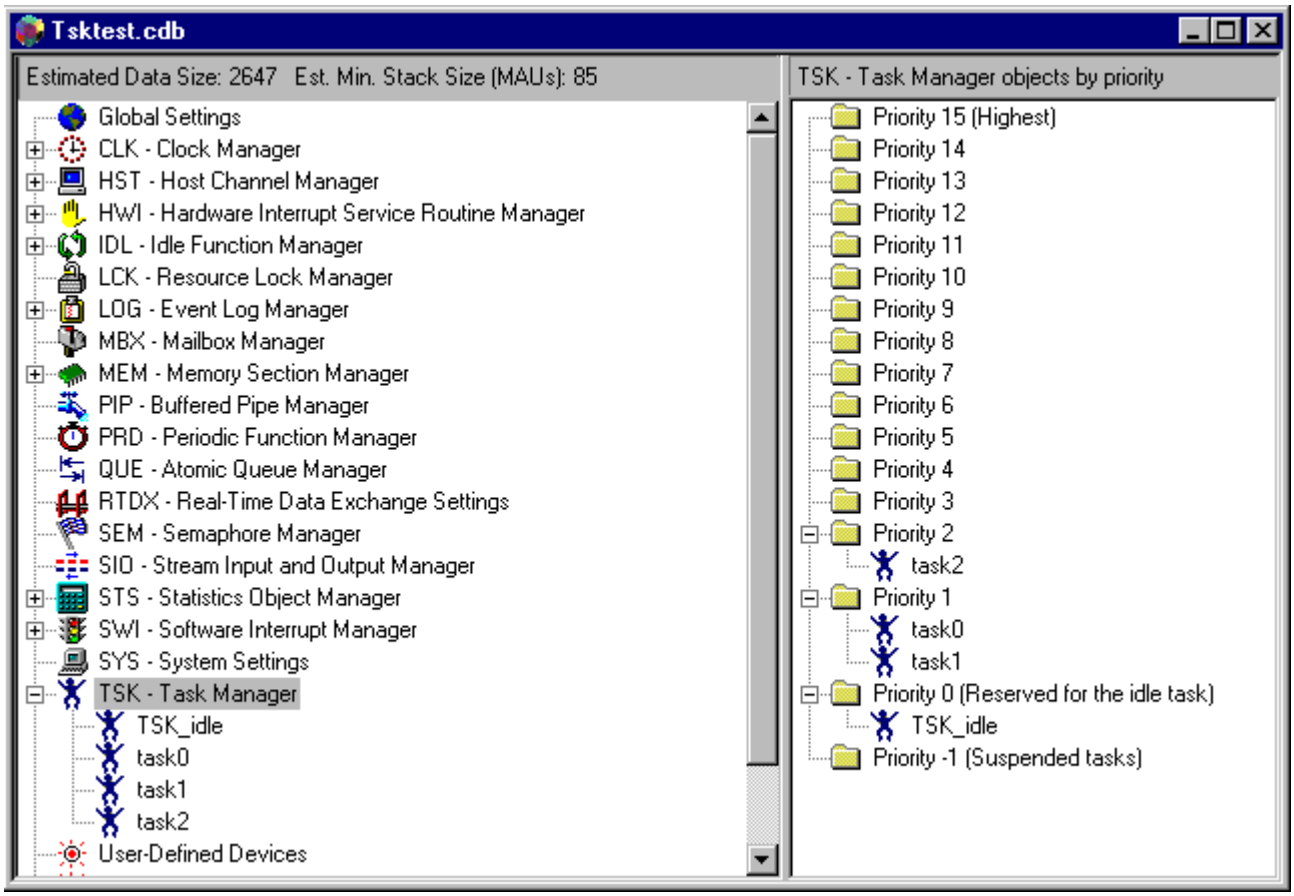
D) Les tâches de type TSK

37) Caractéristiques générales

a) Généralités
Le niveau de priorité de cette famille est inférieure à celle des tâches de type SWI et supérieure à celle des tâches de type IDL
Elles possèdent 15 niveaux de priorité interne. Le niveau le plus bas « 0 » correspond au fonctionnement de la boucle IDL. Une tâche de niveau « -1 » est suspendue.
Le module TSK de gestion de ces tâches produit pour chaque tâche des objets de type TSK. Pour accéder à ces objets on utilise un « handle » de type TSK_handle.
Le noyau sauvegarde une copie des registres du processeur pour chaque tâche de manière automatique. Chaque tâche possède sa propre pile de travail.
Les tâches crées par un programme simple partagent les mêmes variables globales.
Ce type de tâches peut manipuler tous les objets de synchronisation de type PIPE , SEMAPHORE, MAILBOX etc.
Elles peuvent être suspendues et relancées si ces objets ne sont pas disponibles.

b) La création des tâches
Comme pour tous les objets de DSP BIOS les tâches TSK peuvent être crées de manière dynamique ou statique.
Pour la création dynamique on a la primitive : **TSK_Handle TSK_create(tache, attrs, [arg]…) ;**
Pour la destruction dynamique on a la primitive : **Void TSK_delete(TSK_Handle tache) ;**

A chaque appel de ces primitives le noyau réserve et remet à disposition la mémoire système.
Les autres ressources crées par la tâche ne sont détruites



38) Les états et la préemption des tâches TSK

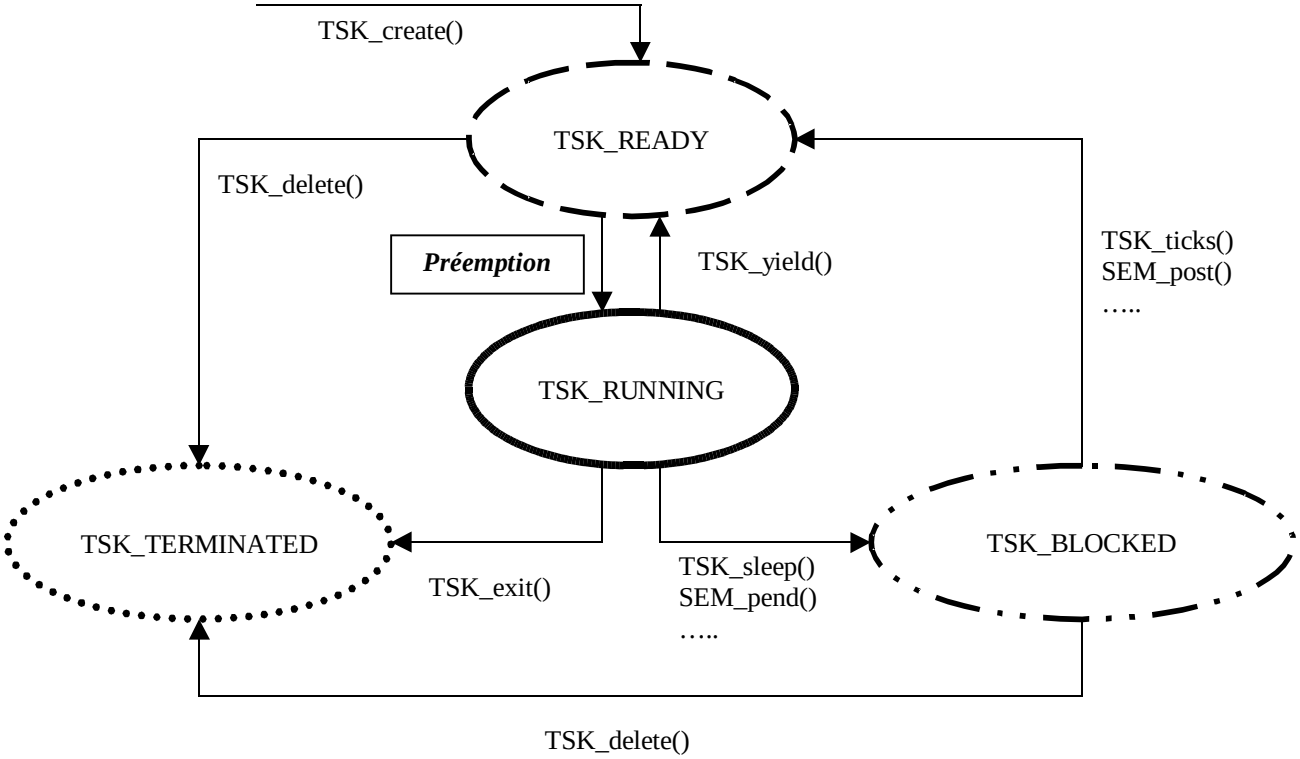
a) Les états des tâches

Ces tâches possèdent 4 états de manière classique :

- ✓ En cours (running)
- ✓ Prête (ready)
- ✓ Suspendue (blocked)
- ✓ Terminée (ternimated)

La préemption des tâches est immédiate dès que la priorité est suffisante.

b) La préemption TSK de DSP BIOS



Si aucune tâche n'est en cours, c'est la boucle infinie TSK_idle qui prend le CPU. En général DSP BIOS utilise cette tâche pour transférer les données de fonctionnement avec le module RTDX sur le bus JTAG vers le PC.

c) Gestion de la pile et commutation active
Si une tâche utilise plus de mémoire que celle qui lui a été attribuée lors de sa création, elle peut écrire sur la pile d'une autre tâche.
On peut utiliser cette séquence de code pour tester l'état de la pile

```
TSK_Stat statbuf ;                                  /* declaration d'un buffer */
….
TSK_stat(TSK_self(),, &statbuf) ;                   /* Lit l'etat de la tâche */
 If (statbuf.used > (statbuf.attrs.stacksize * 9/10 )){
        LOG_printf(&trace, « Plus de 90% de taille de pile utilisée \n ‘ ») ;
}
```

Il existe une fonction qui permet de passer d'une tâche à une autre :

Void (*Swith_function) (TSK_Handle_curtask , TSK_Handle nexTask) ;

E) Les tâches de type IDL

La boucle IDLE est la tâche de fond de DSP BIOS. C'est une boucle infinie qui peut être interrompue par n'importe quelle autre tâche de niveau supérieur.

Toutes les tâches placées dans cette boucle s'exécutent les unes à la suite des autres dans l'ordre défini par l'outil de configuration.
Les tâches de ce type permettent en général la communication avec le PC

- ✓ LNK-dataPump : transfert les statistiques de fonctionnement vers le PC (LOG STS)

- ✓ RTA_dispatcher : lit et met à jour les données de fonctionnement de la cible
- ✓ IDL_cpuload : calcule la charge CPU
- ✓ IDL_rtdxPump : exécute le transfert de données avec le mode RTDX.

F) Les outils de synchronisation des taches

39) Les sémaphores

Les sémaphores de DSP BIOS sont de type à compte. Comme tout objet de DSP BIOS ils peuvent être crées de manière statique ou dynamique. La valeur initiale est en général nulle.

a) Demande d'attente sur un sémaphore **SEM_pend()**
Un accès au sémaphore par la fonction **SEM_pend(sem, timeout)** décrémente la variable associée :
- ✓ Si la variable est supérieure à **« 0 »** le sémaphore est libre et la tâche continue son déroulement.
- ✓ Si la variable est **nulle la tâche se met en attente** pour le temps « timeout » (SYS_FOREVER indique un temps infini)

b) Signal de libération d'un sémaphore **SEM_post()**
Un signal au sémaphore par la fonction **SEM_post(sem)** incrémente la variable associée et réveille toutes les tâches mise en sommeil par un sémaphore nul.

c) Création dynamique
Les fonctions suivantes permettent la création et la destruction dynamique.
SEM_Handle SEM_create(Uns count, SEM_attrs attrs)
Void SEM_delete(SEM_Handle sem)

d) Création statique
Elle se fait de manière classique par l'outil de configuration

40) Les boites à lettres

Une Mailbox, boite à lettres, est une zone mémoire partageable par toutes les tâches. La taille du message, la taille de la boite à lettres et les attributs doivent être fixés lors de la création.
Ces boites à lettres ne sont pas les mêmes que celles des fonctions SWI.

a) Demande d'attente sur une boite à lettres **MBX_pend()**
Un accès à une boite à lettres par la fonction **MBX_pend(mbx, msg, timeout)** lit un message dans une boite à lettres :
- ✓ Si la boite à lettres est pleine la tâche lit le message et continue son déroulement.
- ✓ Si la boite à lettres est vide la tâche se met en attente pour le temps « timeout » (SYS_FOREVER indique un temps infini)

b) Ecriture dans une boite à lettres **MBX_post()**
Une écriture dans une boite à lettres **MBX_post(mbx, msg, timeout)** remplit la zone de message réveille toutes les tâches mise en sommeil par une boite à lettres vide. Si la boite à lettres est pleine la tâche est mise en sommeil pour timeout.

c) Création dynamique
Les fonctions suivantes permettent la création et la destruction dynamique.
MBX_Handle SEM_create(Uns msgsize, Uns mbxlength, MBX_attrs attrs)
Void MBX_delete(MBX_Handle sem)

d) Création statique
Elle se fait de manière classique par l'outil de configuration

41) Les horloges système CLK

a) Les tranches de temps et l'horloge système **CLK**

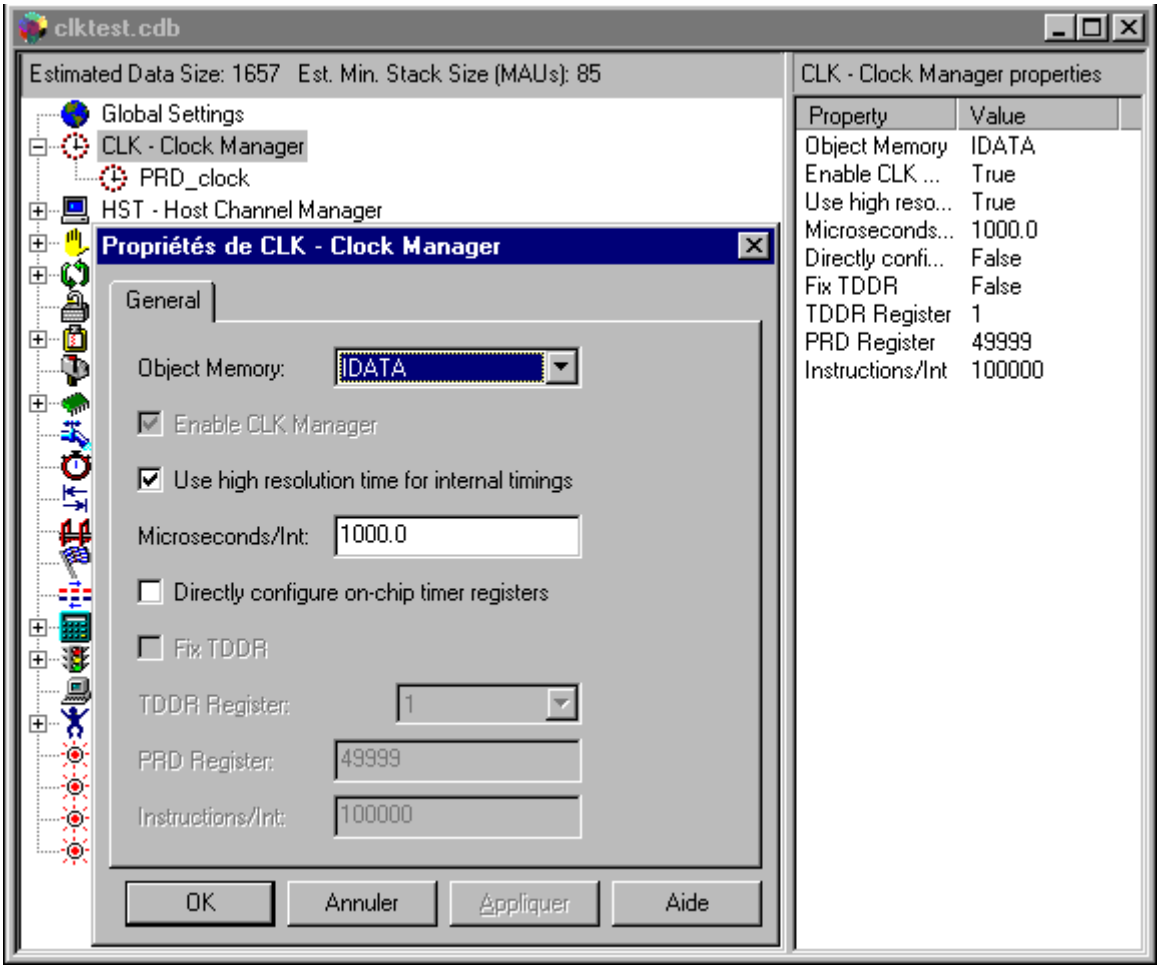
Tout DSP possède un timer interne que l'on peut programmer pour générer les tranches de temps « tick » par génération d'interruption. La résolution possible est proche du cycle d'horloge d'une simple instruction CPU.
L'horloge système est définie par l'outil de configuration et permet de lancer des fonctions à chaque interruption du timer. La fonction HWI CLK_F_isr est la fonction qui génère les tranches de temps.

Il y a deux méthodes de chronogrammes (timing) à *haute ou faible résolution*. Par défaut l'horloge système et la faible résolution sont les mêmes.
Toutes les fonctions de type CLK utilisent cette base de temps. Pour obtenir les valeurs du temps correspondant on peut utiliser les fonctions

- ✓ CLK_getltime : temps d'une tranche de temps
- ✓ CLK_getprd : registre période de l'outil de configuration
- ✓ CLK_countspms : nombre de comptage du timer par millisecondes.

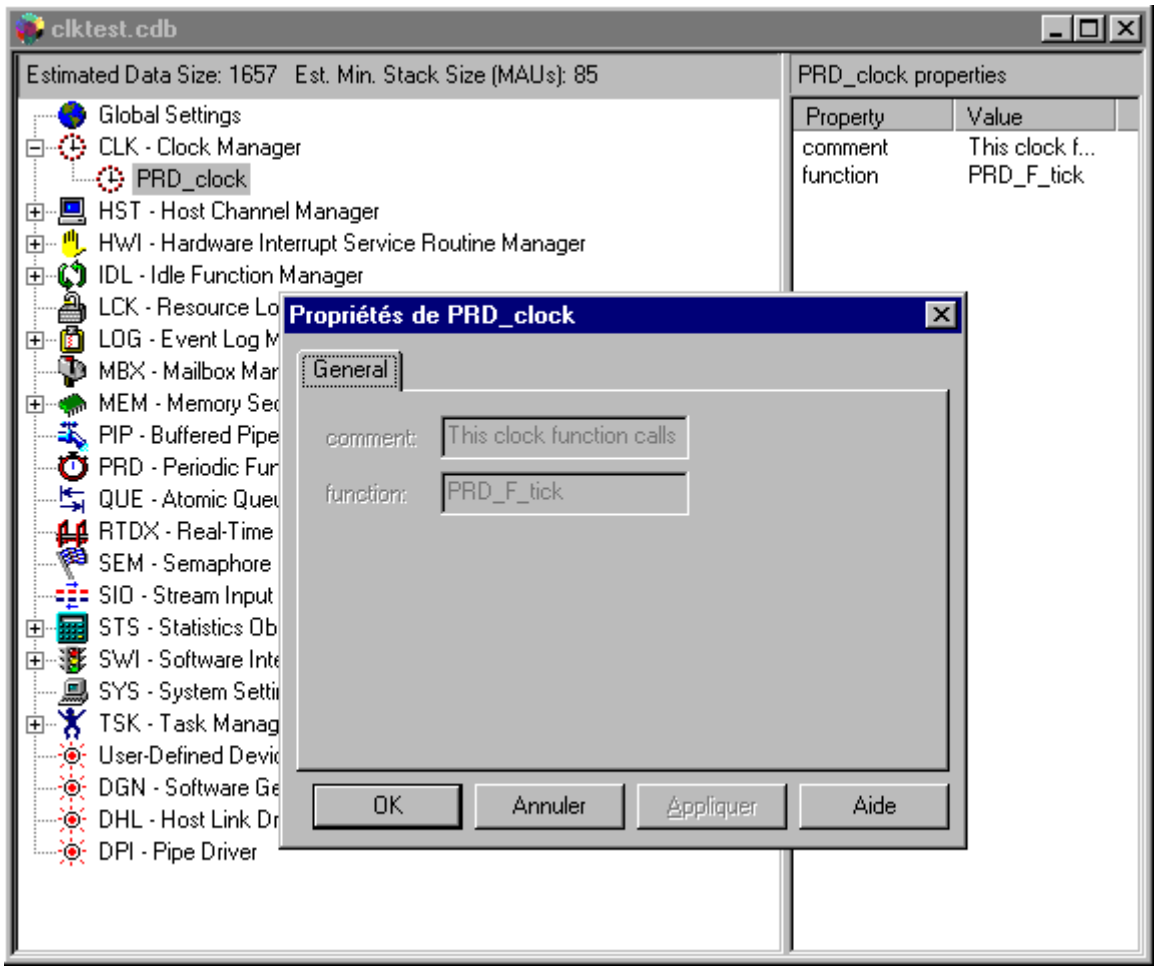
Toutes les fonctions CLK sont très proches des tranches de temps, elles doivent être très courtes et n'utiliser que des primitives utilisables à l'intérieur d'un fonction de type HWI.

b) Base de temps des fonctions
Les fonctions qui utilisent une variable de type « timeout » utilisent comme unité le « tick » TSK_sleep(nb_tick)



42) Les fonctions périodiques PRD

a) Les fonction périodiques **PRD_clock**
DSP BIOS propose une autre base de temps **PRD_tick**. C'est un compteur 32 bits qui utilise aussi le timer pour calculer la base de temps. Il peut avoir plusieurs objets périodiques mais une seule base de temps.
PRD_clock appelle **PRD_tick** qui lui-même appelle deux sous fonctions : **PRD_F_tick et PRD_swi**



b) Les fonctions **PRD_swi**
Ces fonctions utilisent une base de temps issue de l'horloge système, mais ont toutes les propriétés des fonctions de type SWI.
Elles peuvent décrire des objets plus lents que les fonctions de type CLK. Les fonctions utilisables sont les suivantes :

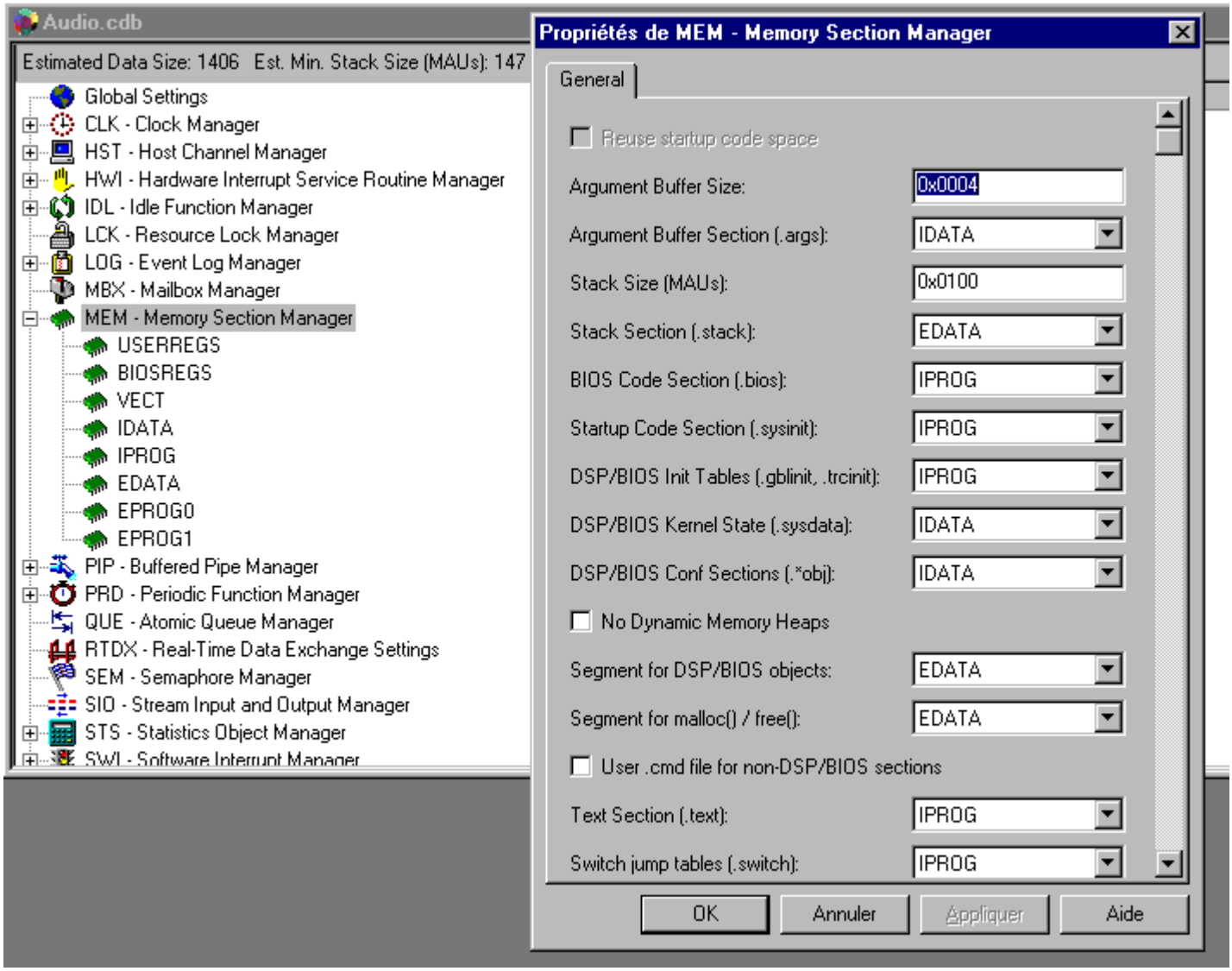
- ✓ PRD_getticks. Get the current tick count
- ✓ PRD_start. Arm a periodic function for one-time execution
- ✓ PRD_stop. Stop a periodic function from continuous execution
- ✓ PRD_tick. Advance tick counter, dispatch periodic functions

Chapitre 5 : Mémoire et fonction de bas niveau

A) la gestion mémoire

43) Les sections mémoire

a) Les fichiers de configuration
C'est l'éditeur de liens « linker » qui affecte les zone de code dans des sections mémoires. Pour cela code Composer Studio utilise des fichiers de configuration de nom « *.cmd ». Le programmeur peut définir son propre fichier, garder celui proposé par BIOS DSP ou choisir une combinaison des deux.
Dans tout système à base de DSP la mémoire interne est la plus rapide : il faut la réserver aux sections de code qui doivent être les plus efficaces.



b) Invalidation de l'allocation dynamique
Il est possible de ne pas autoriser les fonctions d'allocation dynamique mémoire. Pour cela il suffit de cocher la case **« No Dynamic Heaps »**. Il est alors impossible d'utiliser des fonctions qui réservent de la place en mémoire comme la création de tâche (chaque tâche à besoin de sa propre pile)

c) L'unité d'allocation mémoire
Il existe une unité d'allocation mémoire **MAU** : Memory Allocation Unit. Dans l'exemple ci-dessus cette unité vaut 0x0100 soit 256 mots de 16 bits

44) L'allocation dynamique mémoire

a) Allocation dynamique mémoire
Comme en langage « C » on trouve des fonctions qui permettent de réserver des zones mémoires
- ✓ **Ptr MEM_alloc(Int segid, Uns size, Uns align)** segid indique le type de mémoire utilisée, size la taille en MAU et align une adresse d'alignement. cette fonction renvoie l'adresse du début de la zone.
- ✓ Exemple d'allocation d'un tableau de NB_elt structures :

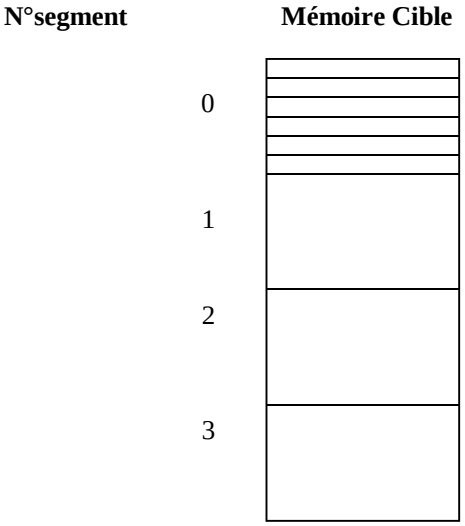
```
typedef struct Obj{
        Int var_1 ;
        Int var_2
        Ptr obj_adr ;
} Obj ;

obj_adr = MEM_alloc(SRAM, siseof(Obj)* NB_elt, 0) ;
```

b) Libération de mémoire
La libération de la zone précédente peut se faire par la fonction **MEM_free()**

```
MEM_free(SRAM, obj_adr, sizeof(Obj)*NB_elt) ;
```

45) La segmentation mémoire

L'affectation et la libération de la mémoire peuvent conduire à une fragmentation des zones réservées. Pour éviter une fragmentation trop importante il est possible de réserver un bloc pour les petites zones et des blocs pour les grandes zones.



B) Les FIFO et les listes chaînées : QUEUE

46) La structures des listes de données

Le module **QUE** permet de gérer des listes chaînées en insérant et enlevant des éléments par gestion de pointeurs. La plupart du temps ces listes sont des FIFO.

a) Structure de base
La structure de base à la forme :

```
typedef struct QUE_Elm{
        struct QUE_elem *next ;
        struct QUE_elem *prev ;
} QUE_elem ;
```

b) Objet d'une liste
QUE_elem est utilisée pour créer les objets des listes chaînées

```
typedef struct Msg_Obj{
        QUE_elem elemt ;
        Char obj ;
} MsgObj ;
```

47) Les fonctions de gestion des listes chaînées

a) Fonctions de création destruction
- ✓ Création d'une QUE : Ptr QUE_create(attrs)
- ✓ Destruction d'une QUE : Void QUE_delete(queue)

b) Fonctions avec blocage des interruptions
- ✓ Ajout d'un élément en fin : Ptr QUE_put(queue, elem)
- ✓ Extraction d'un élément en tête : Ptr QUE_get(queue)

c) Fonctions sans blocage des interruptions
- ✓ Ajout d'un élément en fin : Ptr QUE_enqueue(queue, elem)
- ✓ Extraction d'un élément en tête : Ptr QUE_dequeue(queue)

d) Fonctions d'extraction de pointeur sans modification de la liste
- ✓ Recherche du premier élément : Ptr QUE_head(queue)
- ✓ Recherche de l'élément suivant : Ptr QUE_next(queue)
- ✓ Recherche de l'élément précédent : Ptr QUE_prev(queue)

e) Insertion et extraction d'éléments en un point arbitraire
- ✓ Insertion d'un élément : Void QUE_insert(Ptr qelem, Ptr elem)
- ✓ Extraction d'un élément : Vois QUE_remove(Ptr qelem)

C) Les services système

48) Les fonctions d'arrêt

DSP BIOS propose deux fonctions d'arrêt. Ces fonctions sont des objets du module SYS. Ces fonctions appellent la fonction _halt qui place le CPU dans une boucle infinie en masquant les interruptions

a) Fonction de sortie normale
La fonction **Void SYS_exit(Uns status)** permet de sortir en envoyant au noyau un code d'état.

b) Fonction de sortie lors d'un problème
La fonction **Void SYS_abort(String format, Arg [arg] …)** permet de sortie en envoyant un message formaté.

49) Les messages d'erreur

a) La fonction **SYS_error(Strind s, Uns errno, …)**
Cette fonction gère le fonctionnement des erreurs systèmes de DSP BIOS. Elle lance par défaut une fonction qui est _UTL_doError dont le rôle est d'afficher un message LOG_error. Le chiffre errno est un code d'erreur système inférieur à 256.

b) Fonction d'affichage personnalisée
La fonction standard _UTL_doError peut être remplacée par une fonction personnelle en modifiant le module SYS de l'outil de configuration.

Chapitre 6 : Les buffers tournants et les et Pipes

A) Principes généraux

50) Les flux de données et les buffers

a) Problème général

Un programme d'application doit en général traiter un flux de données important. Les données arrivent de manière asynchrone et non contrôlées par le programme.

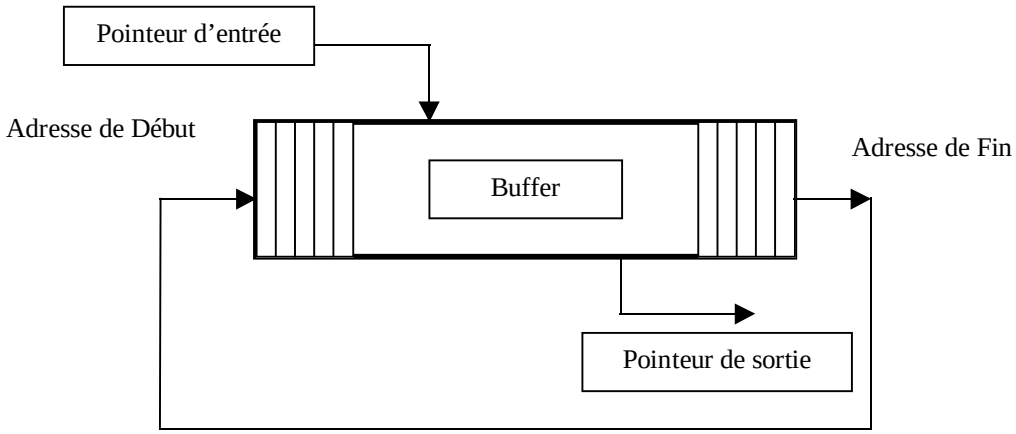
Pour éviter de perde des données en entrées ou en sorties, les données doivent être stockées dans des zones mémoire tampon. Ces buffers doivent être remplis ou vidés de manière indépendante du déroulement du programme. Pour cela on utilise soit le mécanisme des interruptions HWI, soit les DMA.

Pour simplifier la gestion, ces zones mémoires sont gérées par des pointeurs : cela réduit la taille mémoire utilisée et évite la copie de données entre programme.

Les buffers d'entrées sont distincts des buffers de sorties.

b) Les buffers tournants

Un buffer tournant est une zone mémoire continue de taille fixe, gérée par deux pointeurs et un compteur de données.



- ✓ Cette zone mémoire est comprise entre une adresse de début et une adresse de fin.
- ✓ Les données sont écrites dans le buffer au moyen d'un pointeur d'entrée, elles sont sorties au moyen d'un pointeur de sortie. Ces données peuvent être des trames
- ✓ Au début les deux pointeurs sont initilisé à l'adresse de début du buffer.
- ✓ En écriture un drapeau logiciel est positionné lorsque le buffer est plein : ceci permet de supendre la tâche correspondante. A chaque écriture le pointeur est incrémenté.
- ✓ De même en lecture un drapeau logiciel est positionné lorsque le buffer est vide : ceci permet de supendre la tâche correspondante. A chaque lecture le pointeur est incrémenté.
- ✓ Lorsqu'un pointeur arrive sur l'adresse de fin il est rechargé par l'adresse de début de la zone. D'où le nom de buffer tournant.
- ✓ Il existe des buffers d'entrées : les données viennent d'un composant périphérique et vont vers un programme.
- ✓ Il existe des buffers de sorties : les données viennent d'un programme et vont vers un composant périphérique.

c) Utilisation des interruptions

Les pointeurs sont gérés par des tâches en interruption de façon à rendre l'écriture et la lecture des données indépendantes du déroulement du programme.

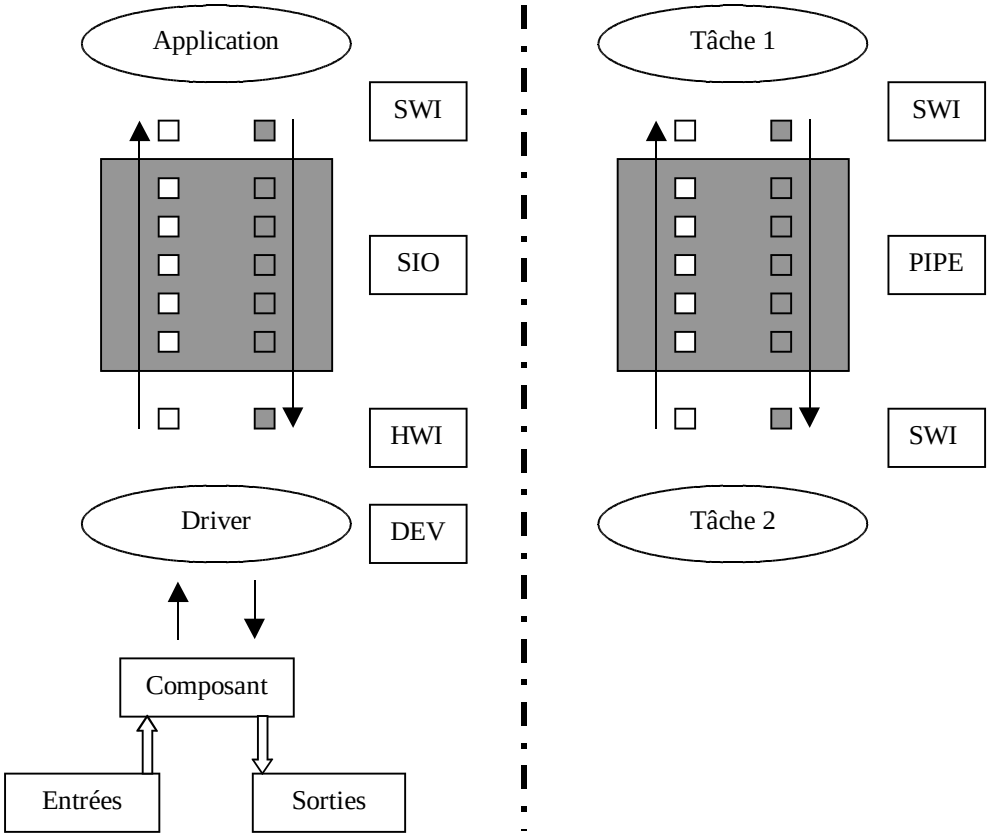
- ✓ Entrées de données : c'est un composant périphérique qui fournit les données et un programme qui les exploite.
  - La tâche d'écriture est de type HWI.
  - La tâche de lecture est de type SWI.

- ✓ Sorties de données : c'est un programme qui fournit les données et un composant périphérique qui les exploite.
  - La tâche d'écriture est de type SWI.
  - La tâche de lecture est de type HWI.

- ✓ Communication entre tâches : on utilise le mécanisme des PIPES pour communiquer de données entre les tâches : les deux tâches de lecture et d'écriture sont alors de type SWI.

51) Schéma général

- ✓ Pour le dialogue avec un composant périphérique on utilise le module SIO. Le driver DEV est un module qui traite un programme d'interruption identique pour tous les composants d'une même famille.
- ✓ Pour la communication entre les tâches on utilise les Pipes.



Les mécanismes mis en œuvre pour les SIO et les PIPE sont très voisins.

B) Les PIPES

52) Caractéristiques principales

- ✓ Taille des données : un PIPE est un buffer contenant un nombre fixe de trames. Chaque trame a aussi une longueur donnée
- ✓ Un PIPE possède uniquement une entrée d'écriture (writer) et une sortie de données (reader).
- ✓ Une opération de lecture (écriture) correspond à une trame.
- ✓ **Les PIPES sont non bloquant pour les tâches** : il faut synchroniser les échanges par les fonctions **notifyReader et notifyWriter.**
- ✓ Chaque PIPE possède son propre buffer qui doit être crée dans l'outil de configuration.
- ✓ BIOS_startup initialise tous les PIPE existants
- ✓ La communication HST avec le PC basée sur le principe des PIPE est très utilisée
- ✓ Un PIPE peut être utilisé avec une fonction SWI et une fonction HWI, ou deux fonctions SWI.

53) Ecriture dans un PIPE sans interruption

a) La tâche doit tester si le PIPE est plein et s'il est possible d'écrire une donnée **PIP_getWriterNumFrames**.
b) Si le nombre de trames libres est supérieur à « 0 », on lance la fonction **PIPE_alloc()** récupérer une trame libre.
c) Il faut aussi obtenir l'adresse et la taille de la trame par **PIP_getWriterAddr** et **PIP_getWriterSise**
d) On remplit ensuite la trame de données
e) On écrit dans le PIPE par **PIP_put**. Cela permet de relancer la fonction de lecture par **notifyReader** en lui indiquant qu'une donnée est présente dans le PIPE.

```
extern far PIP_Obj writerPipe; /* pipe object created with the Configuration Tool */
writer()
{
       Uns size;
       Uns newsize;
       Ptr addr;
       if (PIP_getWriterNumFrames(&writerPipe) > 0) {
              PIP_alloc(&writerPipe); /* allocate an empty frame */
       }
       else {
              return; /* There are no available empty frames */
       }
       addr = PIP_getWriterAddr(&writerPipe);
       size = PIP_getWriterSize(&writerPipe);
       ' fill up the frame '
       /* optional */
       newsize = 'number of words written to the frame';
       PIP_setWriterSize(&writerPipe, newsize);
       /* release the full frame back to the pipe */
       PIP_put(&writerPipe);
}
```

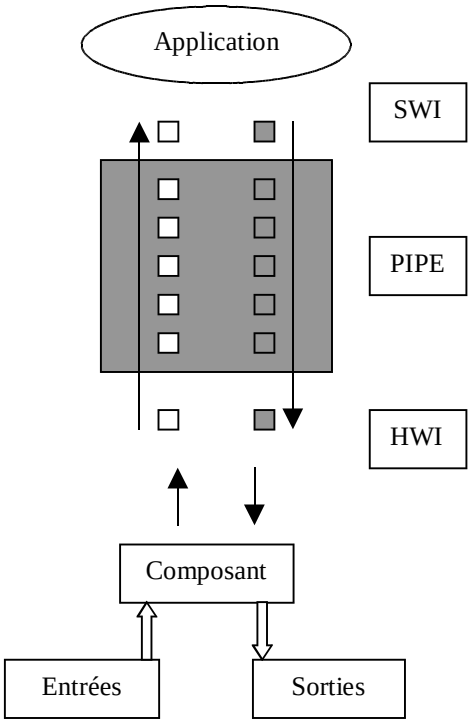
54) Lecture dans un PIPE sans interruption

a) La tâche doit tester si le PIPE est plein et s'il est possible de lire une donnée **PIP_getReaderNumFrames**.
b) Si le nombre de trames libres est supérieur à « 0 », on lance la fonction **PIP_get()** pour récupérer une trame pleine.
c) Il faut obtenir l'adresse et la taille de la trame par **PIP_getReaderAddr** et **PIP_getReaderSise**
d) On récupère la trame de données
e) On libère la trame par **PIP_free**. Cela permet de relancer la fonction d'écriture par **notifyWriter** en lui indiquant qu'une place est libre dans le PIPE.

```
extern far PIP_Obj readerPipe; /* created with the Configuration Tool */
reader()
{
       Uns size;
       Ptr addr;
       if (PIP_getReaderNumFrames(&readerPipe) > 0) {
              PIP_get(&readerPipe); /* get a full frame */
       }
       else {
              return; /* There are no available full frames */
       }
       addr = PIP_getReaderAddr(&readerPipe);
       size = PIP_getReaderSize(&readerPipe);
       ' read the data from the frame '
       /* release the empty frame back to the pipe */
       PIP_free(&readerPipe);
}
```

55) Lecture / écriture dans un PIPE avec interruption

Un composant périphérique est capable de lancer une **HWI** aussi bien en lecture qu'en écriture. Les phases de test de présence de données dans le PIPE peuvent donc être omises : se sont les tâches **HWI_read (HWI_write)** qui peuvent lancer les fonctions **notifyWriter (notifyReader)** de manière automatique.
Il est bon de garder ces tests par sécurité même en interruption.



Chapitre 7 : Les drivers d'entrées sorties

A) Les entrées sorties : principe général

56) Notion d'entrée et de sortie.

On considère comme une entrée toute information allant du milieu extérieur vers le DSP.
On considère comme une sortie toute information allant du DSP vers le milieu extérieur.
Dans le cas d'un DSP on dispose de ports d'entrées sorties intégrés sur la puce et de composants externes sur la carte.
Toutes ces cellules se comportent vis à vis du DSP comme des **mémoires accessibles en lecture pour les entrées et en écriture pour les sorties.**
Ces dispositifs se comportent donc comme une suite de registres dans le plan mémoire I/O. Un registre 16 bits occupe une adresse dans le plan mémoire du DSP.
Plusieurs lignes électriques peuvent être regroupés en PORT d'entrées sorties. En général pour faire un port on regroupe 8 lignes pour travailler directement sur des octets.

57) Les familles de composants

On distingue plusieurs familles de composants d'entrées/sorties:

a) Les entrées sorties parallèles.
Le DSP doit pouvoir lire en parallèle des mots de 8/16 bits provenant du monde extérieur.
Ces mots représentent l'état des capteurs tout ou rien (TOR) lorsqu'il s'agit d'entrées de données.
Ces mots représentent l'état des actionneurs tout ou rien (TOR) lorsqu'il s'agit de sorties de données.
Une imprimante possédant un interface CENTRONICS est gérée par en parallèle par des mots de 8 bits.

b) Les entrées sorties séries.
Pour faire communiquer à distance des systèmes informatiques, on limite le nombre de fils de connexion.
Les données binaire ne peuvent plus être transmises en parallèles mais sous forme série, les bits les uns à la suite des autres sur un fil.
C'est le cas des liaisons séries de type SPI, I2C, RS232C ou des réseaux (Modem), Ethernet.

c) Les entrées sorties analogiques.
De nombreux processus industriels nécessitent des grandeurs de commande analogiques. Ces grandeurs (vitesse, position, température, pression etc.....) sont converties soit en courant soit en tension.
On utilise alors des Convertisseurs Numériques Analogiques (CNA) et des Convertisseurs Analogiques Numériques (CAN).

d) Les entrées sorties de comptage.
Les entrées de comptage permettent de compter des objets ou des impulsions issues des capteurs de position de type incrémentaux.
Les sorties de comptage permettent de générer des signaux carrés de fréquence et de rapport cyclique variables.

58) *Les entrées sorties d'un DSP.*

a) Principe général
Dans le cas d'un DSP les entrées sorties sont regroupées dans des ports ou sur des circuits externes implantés sur la carte. Chaque port est constitué de :
- un ensemble de lignes électriques d'entrées sorties.
- un ensemble de registres mémoires.

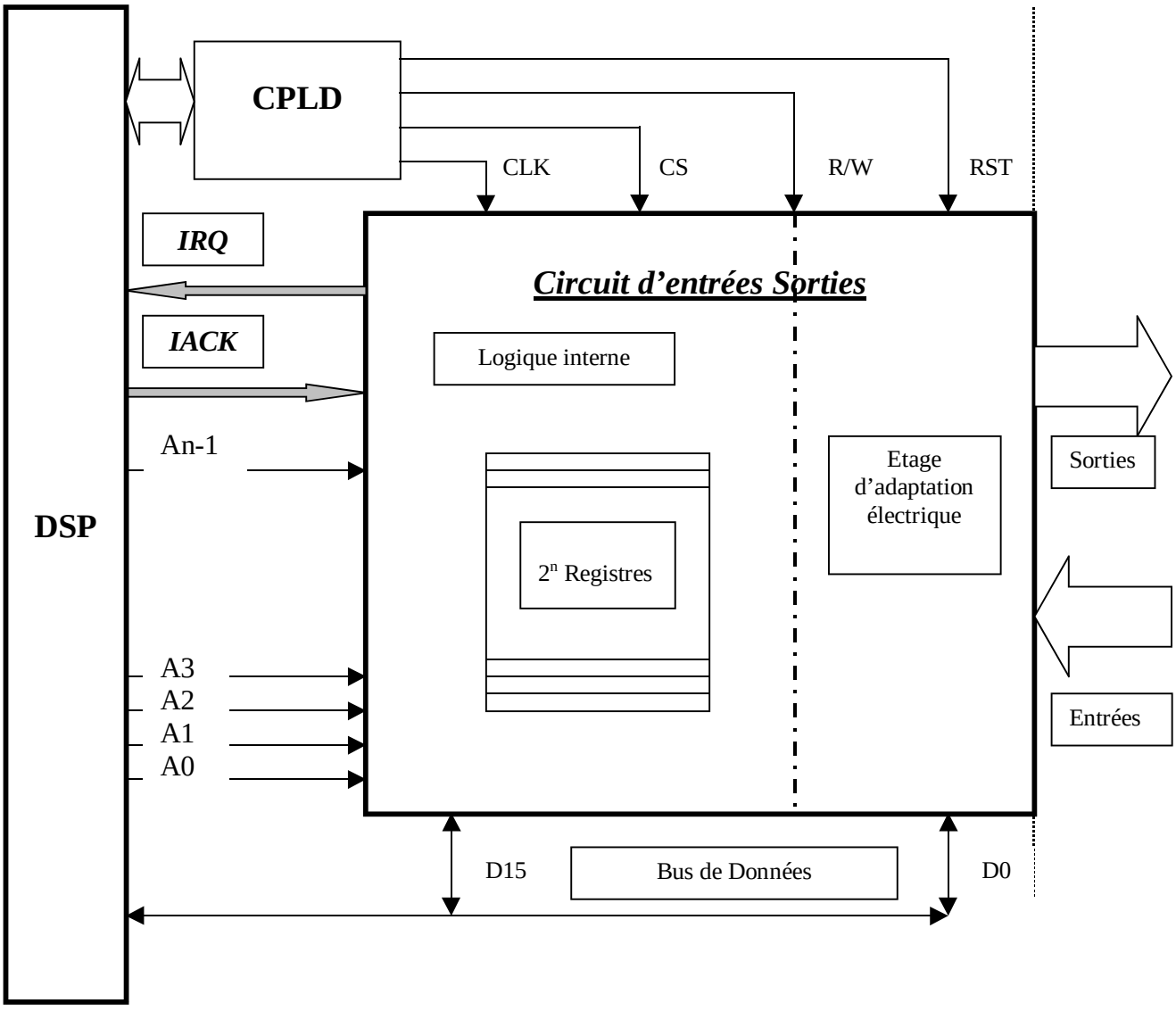
Les registres mémoires permettent de gérer les fonctions des différentes lignes d'entrées sorties. Ils permettent de :

- définir le sens de transfert des données.
- définir le mode de fonctionnement de chaque ligne.
- contrôler l'état du port.
- lire et écrire les données vers l'extérieur.

Dans le cas d'un DSP les adresses de chaque registre des composants internes sont fixes. Pour utiliser un port d'entrée d'entrée sortie il faut connaître:

- sa fonction.
- son interface électrique de sortie.
- son interface électrique d'entrée.
- le nombre des registres internes concernés.
- la fonction de chaque registre.
- la fonction de chaque bit dans chaque registre.
- l'adresse de chaque registre interne dans le plan mémoire du DSP.

b) Structure générale d'un port d'entrées sorties 16 bits**.**



B) Les drivers d'entrées sorties

59) Les types de registres d'un port I/O

a) Les registres de données
Ces registres contiennent les données à lire ou à écrire. Il peuvent avoir aussi la forme de pile FIFO.

b) Les registres de contrôle
Ces registres contiennent de bits de configuration du port. Ces bits permettent de choisir les modes de fonctionnement : avec ou sans interruption, avec ou sans pile, vitesse de transfert, source d'horloge etc.

c) Les registres d'états
Ces registres contiennent des bits qui indique l'état du port : buffer plein/vide, interruption en cours. Plie pleine etc.

d) Programmation d'un port
L'opération de programmation d'un port consiste à choisir un mode de fonctionnement et définir la valeur des bits du registre de contrôle à fixer.
Utiliser un port consiste à lire/écrire les données avec ou sans interruption et tester l'état du port.

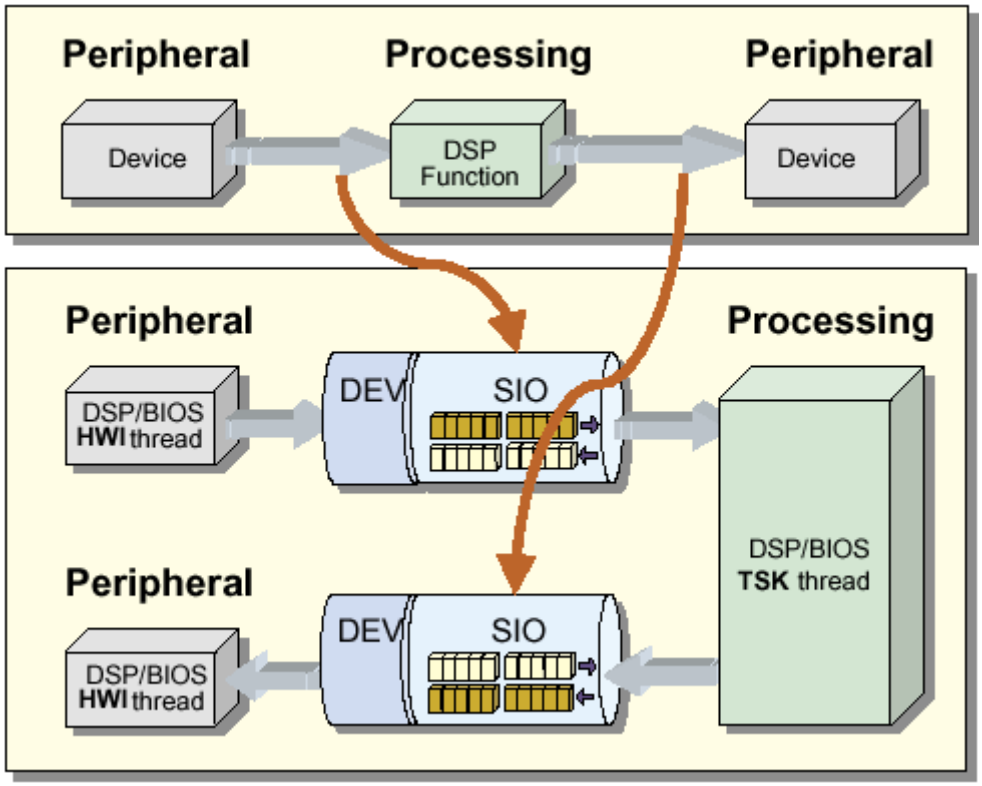
60) Notion de driver

a) Accès aux registres
Un port est considéré par le DSP comme une suite d'adresses mémoires dans le plan mémoire I/O. Il peut donc être défini soit comme un tableau, soit comme une structure de données d'adresses de base fixe.

b) Notion de driver
Un driver est un programme en interruption qui gère ces structures. Il est en général écrit en langage assembleur pour des raisons de rapidité.

61) Architecture des drivers de DSP BIOS

a) Architecture générale
Une application accède aux flux d'entrées sorties à l'aide des fonctions **SIO**. Ces fonctions sont générales et ont la même syntaxe pour tous les dispositifs d'entrées sorties.
A chaque fonction **SIO est associée une fonction DEV** spécifique du circuit périphérique : ce sont elles qui ont accès aux registres des circuits. On voit sur ce schéma que la tâche et le driver peuvent accéder à deux buffers **ping/pong** en même temps : l'un est lus (écrit) par la tâche, l'autre est écrit (lu) par le circuit. Il suffit ensuite d'échanger les pointeurs entre la tâche et le driver.



b) Les fonction SIO et DEV correspondantes

Generic I/O Operation Internal	Driver Operation
SIO_create(name, mode, bufsize, attrs)	Dxx_open(device, name)
SIO_delete(stream)	Dxx_close(device)
SIO_get(stream, &buf)	Dxx_issue(device) and Dxx_reclaim(device)
SIO_put(stream, &buf, nbytes)	Dxx_issue(device and /Dxx_reclaim(device)
SIO_ctrl(stream, cmd, arg)	Dxx_ctrl(device, cmd, arg)
SIO_idle(stream)	Dxx_idle(device, FALSE)
SIO_flush(stream)	Dxx_idle(device, TRUE)
SIO_select(streamtab, n, timeout)	Dxx_ready(device, sem)
SIO_issue(stream, buf, nbytes, arg)	Dxx_issue(device)
SIO_reclaim(stream, &buf, &arg)	Dxx_reclaim(device)
SIO_staticbuf(stream, &buf)	none

62) Les librairies CSl

Tous les DSP possèdent des circuits d'interface de même nature. La version 2 de DSP BIOS possède un ensemble de fonctions librairies pour chaque type de composant : **Chip Support Library**