Quelques éléments de programmation en C du MSP430

Type de données de base

Type de données	Taille	Plage	Alignement
bool	8 bits	0 … 1	1
char	8 bits	0 … 255	1
signed char	8 bits	-128 …127	1
unsigned char	8 bits	0 … 255	1
signal short	16 bits	-32768 … 32767	2
unsigned short	16 bits	0 … 65535	2
signed int	16 bits	-32768 … 32767	2
unsigned int	16 bits	0 … 65535	2
signed long	32 bits	-2^{31} … $2^{31}-1$	2
unsigned long	32 bits	0 … $2^{32}-1$	2
signed long long	64 bits	-2^{63} … $2^{63}-1$	2
unsigned long long	64 bits	0 … $2^{64}-1$	2

Accès aux registres des périphériques

Dans les microcontrôleurs, l'accès direct à un registre en C est souvent facilité par des déclarations simples définissant qu'un certain nom est un synonyme pour le contenu d'une adresse matérielle.

Ces définitions se trouvent dans les fichiers d'entête de type **`io430.h`** ou **`msp430.h`**.

Le fichier **`io430.h`** peut être intégré à un fichier du projet par la directive

```
#include "io430.h"
```

Ce fichier d'entête contient une définition de l'ensemble des registres contenus dans le microcontrôleur utilisé (dans le cas du eZ430-F2012 le MSP430F2012).

Des noms symboliques sont attribués aux registres, en particulier ceux adressant les périphériques, qui sont aussi présentés de manière structurée aux moyen de types **`struct`** et **`union`** en C.

Grâce à ce fichier il va être possible de rendre **plus lisible** la lecture du programme C.

Exemple de fichier **io430xx.h**

```
/*****************************************************************
 * Standard register and bit definitions for the Texas Instruments MSP430 microcontroller.
 * This file supports assembler and C/EC++ development for MSP430x20x2 devices.
                    ...
#define BIT0            (0x0001)
#define BIT1            (0x0002)
#define BIT2            (0x0004)
                    ...
__no_init volatile union
{
  unsigned __READ char P1IN;  /* Port 1 Input */

  struct
  {
    unsigned __READ char P0IN_0  : 1;
    unsigned __READ char P0IN_1  : 1;
    unsigned __READ char P0IN_2  : 1;
    unsigned __READ char P0IN_3  : 1;
    unsigned __READ char P0IN_4  : 1;
    unsigned __READ char P0IN_5  : 1;
    unsigned __READ char P0IN_6  : 1;
    unsigned __READ char P0IN_7  : 1;
  } P1IN_bit;
} @ 0x0020;
                    ...
```

Structure

La structure C permet de grouper sous un même nom des variables ayant un point commun

Déclaration

```
struct myStruct
{
        int var1;
        long var2;
};
```

Utilisation

```
void main(void)
{
        struct myStruct name;
        name.var1 = 19;
        name.var2 = 314;
        ...
}
```

Déclaration de la variable name de type struct myStruct

Union

L'union en C permet d'accéder à une même variable en l'interprétant selon différents types.

Déclaration

```
union myUnion
{
        int varInt;
        char varChar;
};
```

Utilisation

```
void main(void)
{
        union myUnion name;
        name.varInt = 234;
        name.varChar = 31;
        ...
}
```

Déclaration de la variable name de type union myUnion

union et *struct*

Dans une *struct*, les champs mémoires sont contigus et il existe une zone mémoire par membre de la struct.

Dans une *union*, on peut avoir autant de membres qu’on veut, mais une seule zone mémoire qui sera de la taille du plus grand membre.

Pour être plus clair, une *struct* peut contenir autant de données qu’on veut, mais une *union* ne peut avoir qu'une seule donnée à la fois, même si on en déclare plusieurs.

Union

Une utilisation des *union*s est fréquente dans les cas où il est pratique d'accéder à des mêmes espaces contenant les mêmes types de données, mais dans des subdivisions différentes.

Exemple

Accès à une variable de 32 bits par 4 octets ou par un seul mot de 32 bits

```
union registre32
{
	long registre;
	struct _octet
	{
		unsigned char octet4;
		unsigned char octet3;
		unsigned char octet2;
		unsigned char octet1;
	} octet;
};
```

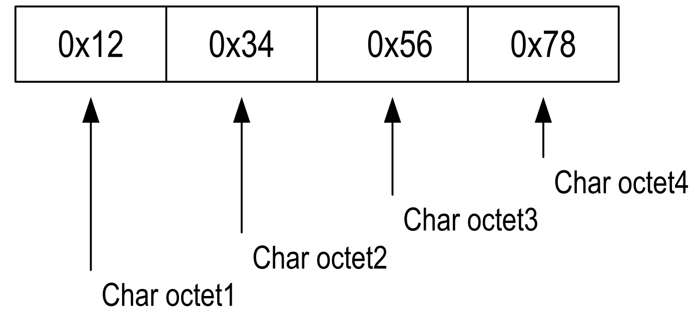
Déclaration de la variable registre

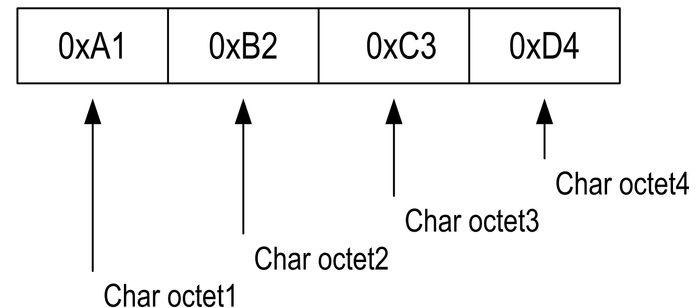
Décomposition de la variable de 32 bits en 4 octets (8 bits)

Union

```
main()
{
        union registre32 reg_name;
        ...
        reg_name.octet.octet1 = 0x12;
        reg_name.octet.octet2 = 0x34;
        reg_name.octet.octet3 = 0x56;
        reg_name.octet.octet4 = 0x78;
        ...


        ...
        toto.registre = 0xa1b2c3d4;
        ...
}
```



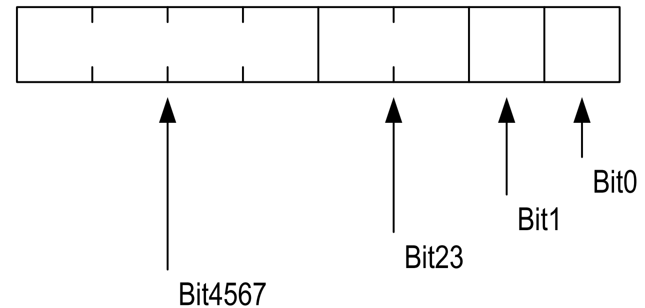


Champ de bits

Le champs de bits en C permet d'accéder aux bits internes d'un registre

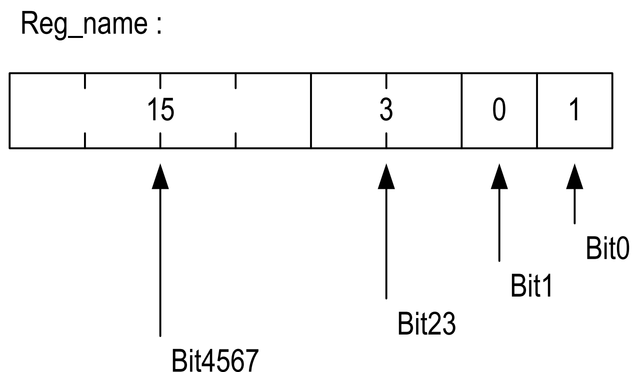
Déclaration

```
struct registre8
{
        unsigned char bit0      : 1;
        unsigned char bit1      : 1;
        unsigned char bit23     : 2;
        unsigned char bit4567 : 4;
};
```



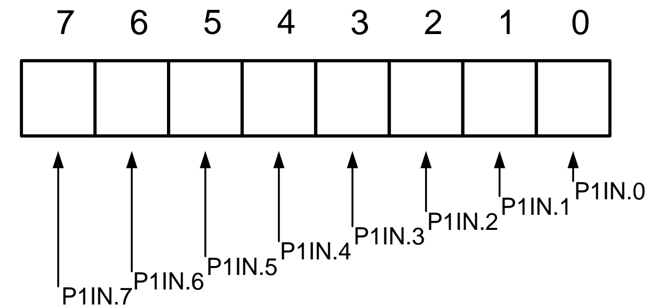
Utilisation

```
void main(void)
{
        struct registre8 reg_name;
        reg_name.bit0 = 1;
        reg_name.bit1 = 0;
        reg_name.bit23 = 3;
        reg_name.bit4567 = 15;
}
```



Registre d'entrée : P1IN, lecture des entrées en C

P1IN : (0x0020)



Voir ficher d'entête io430.h → io430x20x2.h

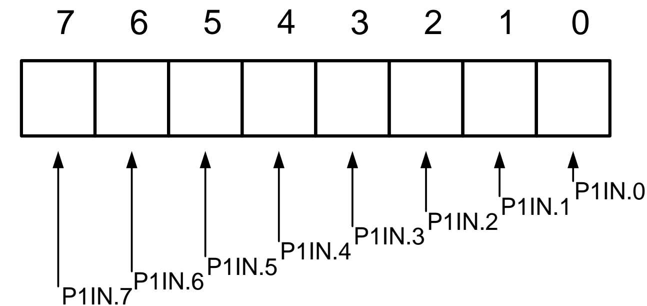
```
#define __READ  const

__no_init volatile union
{
  unsigned __READ char P1IN;  /* Port 1
Input */

  struct
  {
    unsigned __READ char P0IN_0  : 1;
    unsigned __READ char P0IN_1  : 1;
    unsigned __READ char P0IN_2  : 1;
    unsigned __READ char P0IN_3  : 1;
    unsigned __READ char P0IN_4  : 1;
    unsigned __READ char P0IN_5  : 1;
    unsigned __READ char P0IN_6  : 1;
    unsigned __READ char P0IN_7  : 1;
  } P1IN_bit;
} @ 0x0020;
```

`__no_init` est un mot clé étendu (ne fait pas partie du C ANSI), il signifie que toute initialisation est supprimée

Registre d'entrée : P1IN, lecture des entrées en C



Voir ficher d'entête io430.h → io430x20x2.h

```
/*----------------------------
 *   Standard Bits
 *----------------------------*/

#define BIT0                (0x0001)
#define BIT1                (0x0002)
#define BIT2                (0x0004)
#define BIT3                (0x0008)
#define BIT4                (0x0010)
#define BIT5                (0x0020)
#define BIT6                (0x0040)
#define BIT7                (0x0080)
```

Par exemple, lecture de l'entrée P1.5 et P1.2

```
A = P1IN & (BIT5 | BIT2);
```

ou

```
A = P1IN_bit.P0IN_2 | P1IN_bit.P0IN_5;
```

Typedef

Le mot clé *Typedef* permet de renommer un type donné. On peut grâce à lui s'éviter de longues déclarations de variables, ou donner un nom plus parlant à un type.

Exemple : définition d'un type entier

```
typedef int TypeEntier; // signifie que TypeEntier est équivalent à int

void main(void)
{
    TypeEntier  toto;       // toto est en fait un int!
    toto = 12;
    ...
}
```

Typedef

Exemple : définition d’un type structure

```
struct myStruct
{
     int var1;
     float var2;
};

typedef struct myStruct mS;   // signifie que mS est équivalent à struct myStruct

void main(void)
{
     mS variable;    // plus court à écrire que "struct myStruct Variable"
     variable.var1 = 12;
     ...
};
```

Typedef

On peut aussi combiner la déclaration d'un type avec le typedef

Exemple : définition d'un type structure

```
typedef struct myStruct
{
     int var1;
     float var2;
} mS;                    // même utilisation que ci-dessus

void main(void)
{
     mS variable;   // plus court à écrire que "struct myStruct Variable"
     variable.var1 = 12;
     ...
};
```

Volatile

- Le mot clé volatile est un modificateur de type qui permet d'indiquer au compilateur que la variable déclarée est susceptible d'être modifiée à tout moment par autre chose que le code dans lequel elle est déclarée.
- Le compilateur doit donc **éviter toute optimalisation** portant sur cette variable sous peine de risquer de modifier le programme .

Les principaux cas de figure dans lesquels peut se présenter cette situation sont :

- ⇨ variables mappées sur un registre matériel
- ⇨ variables globales modifiées par une routine d'interruption
- ⇨ variables globales dans un environnement *multithreadé*

Le mot clé volatile peut résoudre des erreurs si:

- ⇨ Un code fonctionne bien tant que la compilation n'est pas optimisée, mais fonctionne mal (ou plus du tout) après optimisation;
- ⇨ Un code fonctionne bien tant que les interruptions sont désactivées, mais fonctionne mal (ou plus du tout) après leur activation;
- ⇨ Un code qui accède à du matériel fonctionne parfois, mais parfois pas;
- ⇨ Un code fonctionne bien tant qu'aucune autre tâche ne fonctionne en parallèle,

Volatile : exemple d'optimalisation non souhaitée

```
static int foo;
void bar(void)
{
    foo = 0;
    while (foo != 255)
    continue;
}
```

La fonction **bar**, après optimalisation par le compilateur, exécutera en fait ceci:

```
void bar_optimized(void)
{
    foo = 0;
    while (TRUE)   // optimisation non souhaitée
    continue;
}
```

Volatile : exemple sans optimisation

Pour éviter cette optimalisation non-souhaitée, il faut que **foo** soit déclarée volatile :

```
static volatile int foo;
void bar(void)
{
    foo = 0;
    while (foo != 255)
    continue;
}
```

Volatile

Exemple

```
volatile int var;           // variable entière volatile
volatile int *ptr_var;      // pointeur sur variable entière volatile
volatile struct myStruct; // structure myStruct volatile
```

Tous les membres d'une struct déclarée volatile sont volatiles.
Pour limiter la volatilité à certains membres, il faut déclarer uniquement les membres concernés en tant que volatile.

```
struct myStruct
{
   volatile int var1;
   float var2;
};
```

Accès aux registres des périphériques

Le fichier **`io430.h`** peut être intégré à un fichier du projet par la directive

```
#include "io430.h"
```

Ce fichier d'entête contient une définition de l'ensemble des registres contenus dans le microcontrôleur utilisé (dans le cas du eZ430-F2012 le MSP430F2012).

Grâce à ce fichier il va être possible de rendre **plus lisible** la lecture du programme C.

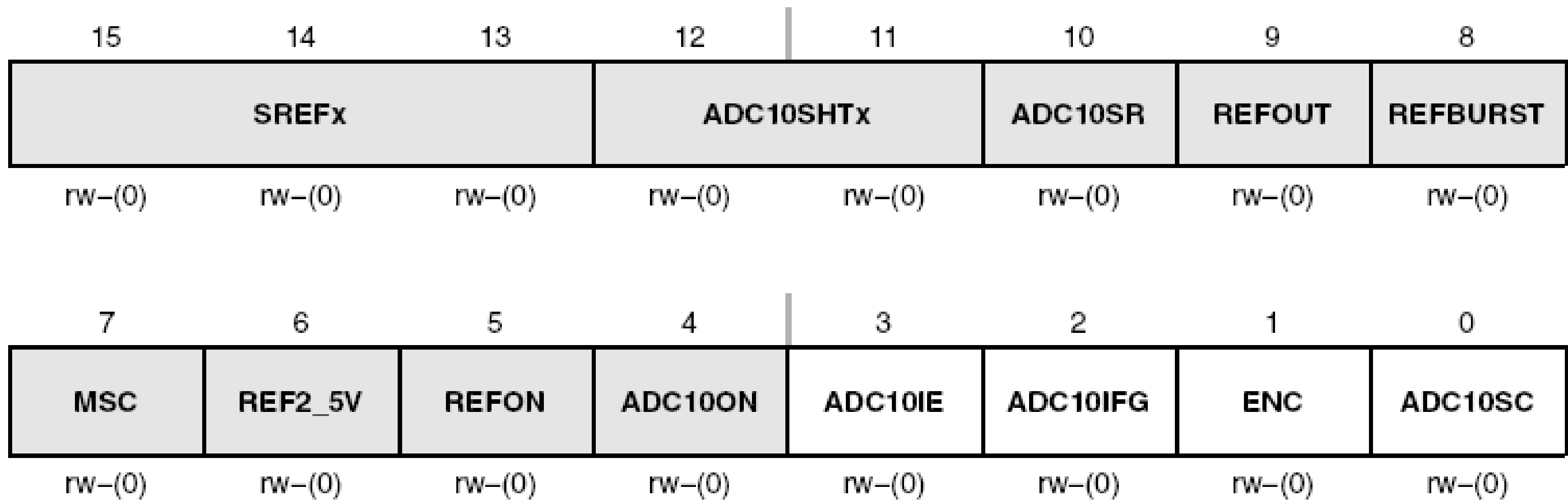
Utilisation du fichier d’entête **io**430.h (**io**430x20x2.h)

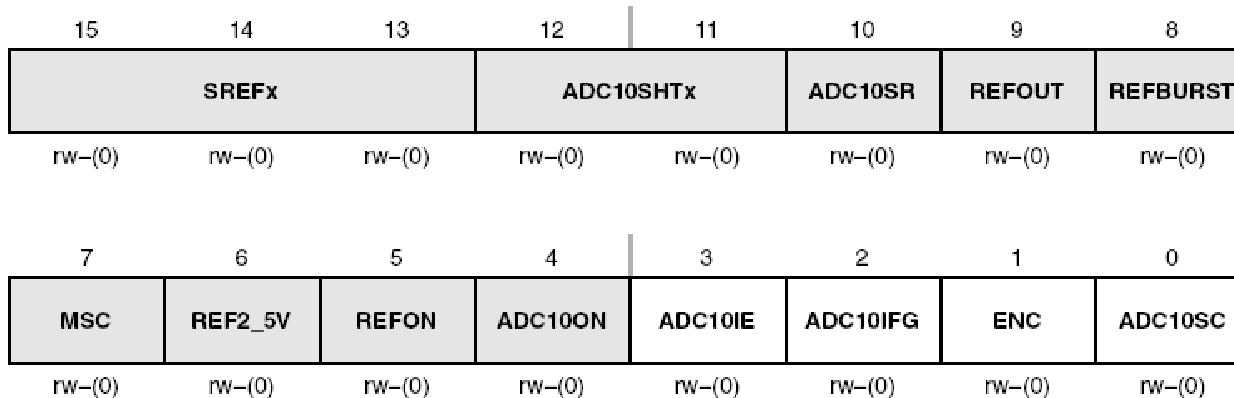
Notez que **`io430.h`** en fait ne fait que **sélectionner et appeler** le fichier d’entête **spécifique au microcontrôleur** utilisé, qui à été défini par le programmeur dans les *Options du Projet*.

```
#if defined (__MSP430C111__) …
…

#elif defined (__MSP430F2002__) || defined (__MSP430F2012__)
#include "io430x20x2.h"
…
…
```

Exemple avec un des registres de contrôle de l'ADC10 : ADC10CTL0






```
__no_init volatile union
{ unsigned short ADC10CTL0;  /* ADC10 Control 0 */
  struct
  { unsigned short ADC10SC        : 1;  /* ADC10 Start Conversion */
    unsigned short ENC            : 1;  /* ADC10 Enable Conversion */
    unsigned short ADC10IFG       : 1;  /* ADC10 Interrupt Flag */
    unsigned short ADC10IE        : 1;  /* ADC10 Interrupt Enable */
    unsigned short ADC10ON        : 1;  /* ADC10 On/Enable */
    unsigned short REFON          : 1;  /* ADC10 Reference on */
    unsigned short REF2_5V        : 1;  /* ADC10 Ref 0:1.5V / 1:2.5V */
    unsigned short MSC            : 1;  /* ADC10 Multiple SampleConversion */
    unsigned short REFBURST       : 1;  /* ADC10 Reference Burst Mode */
    unsigned short REFOUT         : 1;  /* ADC10 Enable output of Ref. */
    unsigned short ADC10SR        : 1;  /* ADC10 Sampling Rate 0:200ksps / 1:50ksps */
    unsigned short ADC10SHT       : 2;  /* ADC10 Sample Hold Select */
    unsigned short ADC10SREF      : 3;  /* ADC10 Reference Select */
  } ADC10CTL0_bit;
} @ 0x01B0;
```

Adresse du registre ADC10CTL0

Initialisation des paramètres du registre de contrôle de l’ADC10 : ADC10CTL0

```
enum {
  ADC10SC               = 0x0001,
  ENC                   = 0x0002,
  ADC10IFG              = 0x0004,
  ADC10IE               = 0x0008,
  ADC10ON               = 0x0010,
  REFON                 = 0x0020,
  REF2_5V               = 0x0040,
  MSC                   = 0x0080,
  REFBURST              = 0x0100,
  REFOUT                = 0x0200,
  ADC10SR               = 0x0400,
  ADC10SHT              = 0x1000,
  ADC10SREF             = 0x8000,
};
```

Alternative: utilisation du fichier d'entête **`MSP430.h`**

Le fichier MSP430.h peut être intégré à un programme par la directive **`#include "MSP430.h"`** qui **sélectionner** et **charger** le fichier d'entête **spécifique** au **microcontrôleur** utilisé, qui à été défini par le programmeur dans les *Options du Projet*.

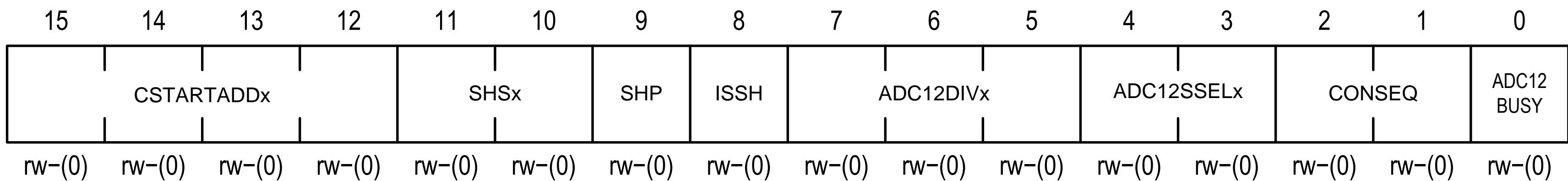
Ce fichier d'entête contient une définition de l'ensemble des registres contenus dans le Microtontrôleur utilisé (par exemple ici le MSP430FG4617 pour la carte IAI).

Grâce à ce fichier il va être possible de rendre plus lisible la lecture du programme C

```
#if defined (__MSP430C111__) ...
...
...

#elif defined (__MSP430FG4616__) || defined (__MSP430FG4617__) ||
defined(__MSP430FG4618__) || defined(__MSP430FG4619__)
#include "MSP430xG46x.h"
```

Exemple de définition dans **msp430.h** du registre de contrôle de l’ADC12 : ADC12CTRL1

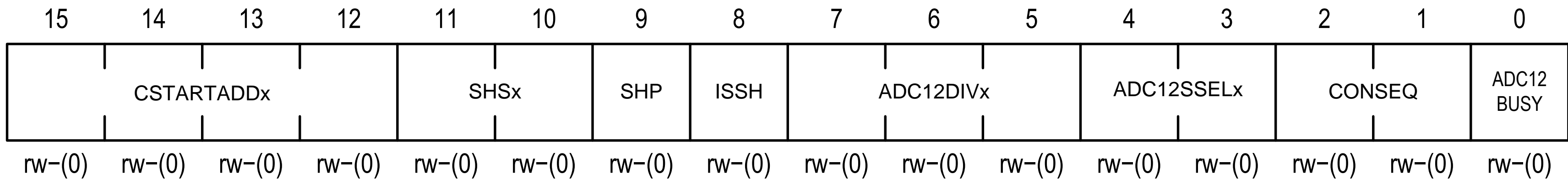



```
#define DEFW(name, address) __no_init volatile unsigned short  name @ address;
```

Adresse du registre ADC12CTRL1

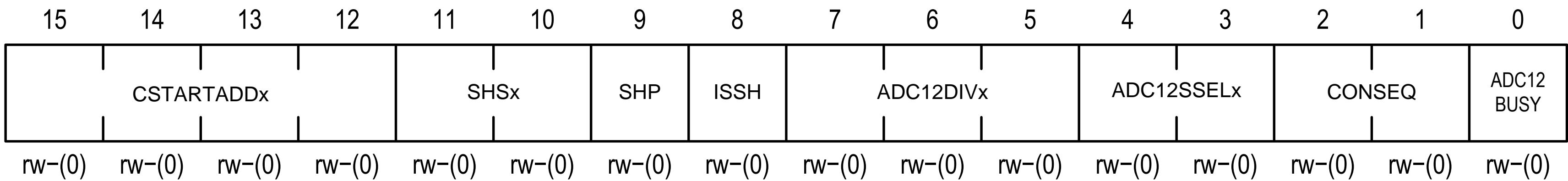
```
#define ADC12CTL1_       (0x01A2)  /* ADC12 Control 1 */
DEFW(   ADC12CTL1  , ADC12CTL1_)
#define ADC12SSEL0            (0x0008)    /* ADC12 Clock Source Select 0 */
#define ADC12SSEL1            (0x0010)    /* ADC12 Clock Source Select 1 */

#define ADC12SSEL_0            (0*8u) /* ADC12 Clock Source ADC12OSC */
#define ADC12SSEL_1            (1*8u) /* ADC12 Clock Source ACLK */
#define ADC12SSEL_2            (2*8u) /* ADC12 Clock Source MCLK */
#define ADC12SSEL_3            (3*8u) /* ADC12 Clock Source SMCLK */
```



Définition de divers mots de contrôle pour le registre ADC12CTRL1

```
…
#define CONSEQ_3           (3*2u)
#define ADC12SSEL_0 (0*8u)
#define ADC12SSEL_1 (1*8u)
#define ADC12SSEL_2 (2*8u)
#define ADC12SSEL_3 (3*8u)
#define ADC12DIV_0              (0*0x20u)
…
```



Exemple de programmation:

Sélection de l'horloge ACLK comme source pour l'ADC12

```
ADC12CTL1_bit.ADC12SSEL = ADC12SSEL_1
```

Utilisation du fichier d'entête io430.h

```
ADC12CTL1 | = ADC12SSEL_1
```

Utilisation du fichier d'entête MSP430.h

Macros

Les macros sont un outil offert aux programmeurs C qui permet de faire du "traitement de texte" dans le code source, avant la compilation.

Utilisations principales

- attribution d’un nom aux constantes numériques,
- définition de symboles pour compilation conditionnelle,
- définition de symboles pour options dans la construction de l'exécutable,
- facilité d'écriture pour des opérations répétées.

Macros

Syntaxe:

```
#define MACONSTANTE 3.141592
   #define SYMBOLE
   #define MAMACRO printf("Appel de MAMACRO")
   #define MALONGUEMACRO printf("Longue macro "); \
     printf("appelée sur 2 lignes grâce au signe '\' ")
   #define MACRO_PARAM(x, y) (2 * x + y)
```

On évite de mettre un point virgule (;) à la fin d'une macro, car c'est une source d'erreurs à la compilation.

La coutume veut que les noms de macros soient écrits en majuscules.

Macros

L'outil qui traite les macros s'appelle le préprocesseur. Il analyse le code source et remplace chaque occurrence du nom d'une macro par sa définition. Par exemple :

```
float r, s;
r = 1.0;
s = MACONSTANTE * r * r;
```

sera transformé, avant la compilation, en :

```
float r, s;
r = 1.0;
s = 3.141592 * r * r;
```

Attention: si la définition est vide, le préprocesseur insère un vide (rien) :

```
float r;
r = SYMBOLE;
```

deviendra (erreur à la compilation)

```
float r;
r = ;
```

Macros

Si la macro a des paramètres, ceux-ci seront insérés tels quels dans code de la macro.

Exemple :

```
float n;
n = MACRO_PARAM(3+5, 19);
```

deviendra après traitement:

```
n = (2 * 3+5 + 19);
```

ce qui n'est pas forcément la macro la plus utile qui soit...

NB: une macro récursive (qui s'appelle elle-même) est en principe illégale.

Inline

Le mot-clé **inline** ressemble un peu dans son principe à la macro, mais à la différence près que son travail se situe pendant la compilation, pas avant.
Il sert à indiquer qu'une fonction doit être recopiée dans le code objet à chaque fois qu'elle est appelée.

Le mot clé **inline** permet une l'optimisation du temps d'exécution. En effet, lors d'un appel à une fonction normale, il y a un "saut" (call) dans le code assembleur, suivi d'un retour (RET).
Si la fonction est liée au mot clé inline, le saut et le retours sont supprimés.
En contrepartie, la taille du code exécutable augmente proportionnellement au nombre d'appels de la fonction

Remarque :
le fait d'utilise le mot clé **inline** avec une fonction est en réalité une recommandation pour le compilateur, lui indiquant que la préférence du programmeur.
Toutefois c'est le compilateur qui décide en dernier ressort s'il y a lieu ou pas d'insérer la fonction plutôt que de l'appeler.

Directives #pragma

La directive **`#pragma`** est définie par le standard ISO/ANSI C.

Elle permet de garder une **certaine portabilité** du software en contrôlant de manière particulière des **extensions spécifiques** à un fabricant ou à un modèle de microcontrôleur.

Le compilateur fournit un set de directives pragma prédéfinies, lesquelles peuvent être utilisées pour contrôler le comportement du compilateur, par exemple sur la manière d'allouer de la mémoire, autoriser des mots clés supplémentaires ou générer des message d'avertissement (warning).

La plupart des directives de #pragma sont prétraitées, ce qui signifie que des macros sont substitués à une directive #pragma.

Définition

Syntaxe : **`#pragma location = (adresse|Name)`**

Paramètres : adresse : adresse absolue de la variable globale ou statique pour laquelle la position mémoire absolue est désirée

Name : nom d'un segment prédéfini par l'utilisateur

Description : Cette directive pragma permet de spécifier une adresse absolue (position Mémoire pour la déclaration suivant cette directive. Cette variable doit être déclarée soit **`__no_init`** ou **`const`**.

La directive peut également prendre une chaîne de caractère spécifiant un segment pour placer une variable ou une fonction

Exemple :

```
#pragma location = 0x22E
__no_init volatile char PORT1;

/*PORT1 est un registre du périphérique*/
/*du même nom placé à l'adresse 0x22E*/
```

Insertion d’un vecteur dans la table d’interruption

Syntaxe : **`#pragma vector = vector_address`**

Description : Directive pragma permettant de spécifier l’adresse de la table d’interruption dans laquelle doit se trouver l’adresse de la routine d’interruption déclarée à la ligne suivante.

Exemple :

```
#pragma vector=TIMERA0_VECTOR
__interrupt void TIMERA0ISR (void)
{
      …
      …
}
```

0xFFEC : adresse définie (par exemple) pour le MSP430FG4617

Nom de la routine d’interruption

Fonctions intrinsèques

Les fonctions intrinsèques permettent un accès direct au instruction bas niveau.
Cette possibilité peut être très utile pour des routines critiques en temps.

La compilation des fonctions intrinsèques insère directement le code sous forme d'instruction unique ou sous la forme d'une courte séquence d'instructions.

L'utilisation de fonctions intrinsèques dans une application demande d'inclure le fichier d'entête **`intrinsics.h`**

Il faut noter que le nom réservé au fonctions intrinsèques commence avec un double « underscore » (__)

Exemple :

```
__disable_interrupt
```

Liste de fonctions intrinsèques

```
__bcd_add_type              exécute une addition BCD

__bic_SR_register           mise à « 0 » de bits dans le registre d'état SR
__bic_SR_register_on_exit
                   mise à « 0 » de bits dans le registre d'état SR lors d'un retour
                   de routine d'interruption ou de routine moniteur
__bis_SR_register           mise à « 0 » de bits dans le registre d'état SR
__bis_SR_register_on_exit   mise à « 0 » de bits dans le registre d'état SR lors
                            d'un retour de routine d'interruption ou de routine
                            moniteur

__data16_read_addr          lecture d'une donnée de 16 bits dans un registre SFR
__data16_write_adddr        écriture d'une donnée de 16 bits dans un registre SFR
__data20_read_type          lecture d'une donnée sur une adresse de 20 bits
__data20_write_type         écriture d'une donnée sur une adresse de 20 bits

__delay_cycle               fourni un délai précis

__disable_interrupt         désactivation des interruptions
__enable_interrupt          activation des interruptions
```

Liste de fonctions intrinsèques

`__get_interrupt_state`	retourne l'état des interruptions
`__get_R4_register`	retourne la valeur du registre R4
`__get_R5_register`	retourne la valeur du registre R5
`__get_SP_register`	retourne la valeur du registre de pile SP
`__get_SR_register`	retourne la valeur du registre d'état SR
`__get_SR_register_on_exit`	retourne la valeur du regsitre de status en retour d'interruption
`__low_power_mode_n`	active un mode basse consommation
`__low_power_mode_off_on_exit`	désactive le mode basse consommation en retour d'interruption
`__no_operation`	insert une instruction NOP
`__set_interrupt_state`	restitue l'état des interruptions
`__set_R4_register`	écrit une valeur dans le registre R4
`__set_R5_register`	écrit une valeur dans le registre R5
`__set_SP_register`	écrit une valeur dans le registre de pile SP
`__swap_bytes`	croise les bytes sur une valeur de 16 bits (intruction SWPB)

Fonctions intrinsèques

__bic_SR_register

Syntaxe :	void __bic_SR_register (unsigned short)
Description :	Mise à « 0 » de bits dans le registre d'état SR l'argument est un masque des bits qui doivent être mis à « 0 »

__bis_SR_register

Syntaxe :	void __bis_SR_register (unsigned short)
Description :	Mise à « 1 » de bits dans le registre d'état SR l'argument est un masque des bits qui doivent être mis à « 1 »

__delay_cycles

Syntaxe :	void __delay_cycles (unsigned long *cycles*)
Paramètres :	cycles. Délai en nombre de cycles d'horloge
Description :	insert des instructions assembleur afin de créer un délai en nombre de cycles d'horloge

Fonctions intrinsèques

__disable_interrupt

Syntaxe : void __disable_interrupt (void)
Description : désactivation des interruptions par insertion de l'instruction DI

__enable_interrupt

Syntaxe : void __enable_interrupt (void)
Description : activation des interruptions par insertion de l'instruction EI

__low_power_mode_n

Syntaxe : void __low_power_mode_n (void)
Description : Commande des divers mode de basse consommation (0 – 4)

__no_operation

Syntaxe : void __no_operation (void)
Description : insertion d'une instruction NOP

Travail personnel

- Première exploration des fichiers d'entête IAR pour MSP430:

  io430.h → io430x20x2.h

  msp430.h → msp430x20x2.h

Retrouvez les définitions des registres P1IN, P1OUT, P1SEL, etc., P2IN, P2OUT, etc. ainsi que, par exemple, celles de BIT0, BIT1, etc..