

Offset	R/W	NOM	FONCTION	Type d'accès	Section
\$0D	RW	RCTRL (en lecture)	<u>En lecture :</u> - Bits de contrôle du multiplexeur AMUX0 et AMUX1 - Bits de contrôle d'anti-rebond ANT0 et ANT1 - Bits du niveau d'interruption NIT1 à NIT3 - Bit d'état du convertisseur STS	Octet	6.4
		WMUX (en écriture)	<u>En écriture :</u> - Bits de contrôle du multiplexeur AMUX0,AMUX1		4.1.3 3.5.1 6.7.3 6.4
\$0E	W	ITLEV	Programmation du niveau d'interruption VME	Octet ou mot	3.5.1
\$0F	W	DEB	Programmation de la durée de filtrage de l'anti-rebond	Octet	4.1.3
\$10			Non utilisé	Mot	
\$11	RW	9513C	Registre de contrôle du timer: pointeur de données, registre de commande, registre d'état	Octet	5.2
\$12	RW	9513DH	Poids forts des registres de données du timer	Mot	5.2
\$13	RW	9513DL	Poids faibles des registres de données du timer	Octet	5.2
\$14			Non utilisé	Mot	
\$15	RW	8259M0	Registres du 82C59A2 maître avec A0 = 0	Octet	3.5.7
\$16			Non utilisé	Mot	

Offset	R/W	NOM	FONCTION	Type d'accès	Section
\$17	RW	8259M1	Registres du 82C59A2 maître avec A0 =1	Octet	3.5.7
\$18			Non utilisé	Mot	
\$19	RW	8259S0	Registres du 82C59A2 esclave avec A0 =0	Octet	3.5.7
\$1A			Non utilisé	Mot	
\$1B	RW	8259S1	Registres du 82C59A2 esclave avec A0 =1	Octet	3.5.7
\$1C	R	AD1674H	Poids forts du CAN	Mot	6.7
\$1D	R	AD1674L	Poids faibles du CAN	Octet	6.7
	W	ADCTRIG	Lancement de la CAN		6.6
\$1E	W	RSTTIM	Reset des interruptions du timer	Mot	3.5.4
\$1F	W	RSTITIN	Reset des interruptions des entrées E16 à E23	Octet	3.5.3

Dans la colonne R/W : R signifie mode lecture
W signifie mode écriture

Offset	R/W	NOM	FONCTION	Type d'accès	Section
\$0D	RW	RCTRL (en lecture) WMUX (en écriture)	<u>En lecture :</u> - Bits de contrôle du multiplexeur AMUX0 et AMUX1 - Bits de contrôle d'anti-rebond ANT0 et ANT1 - Bits du niveau d'interruption NIT1 à NIT3 - Bit d'état du convertisseur STS <u>En écriture :</u> - Bits de contrôle du multiplexeur AMUX0,AMUX1	Octet	6.4 4.1.3 3.5.1 6.7.3 6.4
\$0E	W	ITLEV	Programmation du niveau d'interruption VME	Octet ou mot	3.5.1
\$0F	W	DEB	Programmation de la durée de filtrage de l'anti-rebond	Octet	4.1.3
\$10			Non utilisé	Mot	
\$11	RW	9513C	Registre de contrôle du timer: pointeur de données, registre de commande, registre d'état	Octet	5.2
\$12	RW	9513DH	Poids forts des registres de données du timer	Mot	5.2
\$13	RW	9513DL	Poids faibles des registres de données du timer	Octet	5.2
\$14			Non utilisé	Mot	
\$15	RW	8259M0	Registres du 82C59A2 maître avec A0 = 0	Octet	3.5.7
\$16			Non utilisé	Mot	

entrées	E23	E22	E21	E20	E19	E18	E17	E16
bits	7	6	5	4	3	2	1	0

LHR									transition sur front
	LH7	LH6	LH5	LH4	LH3	LH2	LH1	LH0	montant

Adresse:\$09, accès en octet

un bit à 0 = pas d'interruption générée lors d'une transition sur l'entrée correspondante.

un bit à 1 = interruption générée lors d'une transition sur l'entrée correspondante.

Il est à noter qu'une même entrée peut générer une interruption à la fois sur front montant et front descendant.

Au reset, ces 2 registres ont pour valeur 0.

3.5.3 - Reset de l'interruption en cours d'une entrée (E23 à E16) dans la routine d'interruption

Les demandes d'interruptions générées par les entrées E23 à E16 sont gérées localement par des contrôleurs d'interruptions 82C59A2 d'INTEL (voir § 3.5.5).

Ces demandes d'interruptions sont mémorisées (à la valeur 1) dans un registre interne et présentées au contrôleur d'interruptions.

Il est indispensable de remettre à zéro cette mémorisation dans la routine d'interruption correspondante, sous peine de ne pas voir la demande d'interruption suivante sur le même niveau ou d'avoir une "fausse" interruption si le contrôleur est configuré en mode interruption sur front.

entrées	E23	E22	E21	E20	E19	E18	E17	E16
bits	7	6	5	4	3	2	1	0

Registre								
RSTITIN	IT7	IT6	IT5	IT4	IT3	IT2	IT1	IT0

Adresse : \$1F

- une écriture d'un 0 entraîne le reset de la demande d'interruption
- une écriture d'un 1 n'a pas d'effet.

Ce registre est à écriture seule et accès en octet.

Dans la pratique, dès que le logiciel entrera dans la routine d'interruption, on fera un "reset" de l'interruption en cours.

(Le vecteur d'interruption donné par le 82C59A2 permet de déterminer quelle est l'interruption locale qui a généré une interruption sur le bus VME, voir § 3.5.7).

ATTENTION :

Il est recommandé de ne mettre qu'un seul bit à 0 à la fois pour ne pas faire un Reset d'autres demandes d'interruptions en attente.

Par exemple, on fait le reset de la demande d'interruption IT4 de la manière suivante : MOVE.B #\$EF,@+\$1F.

3.5.4 - Reset de l'interruption en cours d'une sortie du Timer (COUT1 à COUT5)

L'utilisation est semblable à celle décrite au § 3.5.3

bits	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
Registre																
RSTTIM	NU	NU	COUT5	COUT4	COUT3	COUT2	COUT1	NU	IT7	IT6	IT5	IT4	IT3	IT2	IT1	IT0

Adresse : \$1E

- une écriture d'un 0 entraîne le reset de la demande d'interruption
- une écriture d'un 1 n'a pas d'effet.

NU : Non utilisé

Ce registre est à écriture seule et accès en mot.

Il faut veiller à mettre à 1 tous les bits de D0 à D7 pour ne pas faire le reset de demandes d'interruption des entrées logiques.

Par exemple, on fait le reset de la demande d'interruption de COUT3 de la manière suivante : MOVE.W #\$F7FF,@+\$1E.

3.5.5 - Gestion des interruptions par le maître

Les interruptions sont gérées sur la carte TSVME410 par deux contrôleurs d'interruption 82C59A2 en cascade. L'un est programmé en maître (adresses \$15 et \$17) et gère les ITs du timer et l'IT de fin de conversion A/N, l'autre est programmé en esclave (adresses \$19 et \$1B) et gère les ITs des entrées logiques.

La sortie "fin de conversion" du AD1674 est reliée à la broche IR0 du 82C59A2 maître

La sortie COUT5 du timer est reliée à la broche IR5 du 82C59A2 maître

La sortie COUT4 du timer est reliée à la broche IR4 du 82C59A2 maître

La sortie COUT3 du timer est reliée à la broche IR3 du 82C59A2 maître

La sortie COUT2 du timer est reliée à la broche IR2 du 82C59A2 maître

La sortie COUT1 du timer est reliée à la broche IR1 du 82C59A2 maître

La broche INT de l'esclave est reliée à la broche IR6 du maître

Important :

Dans le cas de la TSVME410-1, il faut masquer l'interruption "fin de conversion A/N" sur IR0 du 82C59A2 maître en mettant à 1 le bit D0 du registre OCW1.

3.5.6 - Interruptions gérées par l'esclave

L'IT de l'entrée E23 correspond à la broche IR7 du 82C59A2 esclave

L'IT de l'entrée E22 correspond à la broche IR6 du 82C59A2 esclave

L'IT de l'entrée E21 correspond à la broche IR5 du 82C59A2 esclave

L'IT de l'entrée E20 correspond à la broche IR4 du 82C59A2 esclave

L'IT de l'entrée E19 correspond à la broche IR3 du 82C59A2 esclave

L'IT de l'entrée E18 correspond à la broche IR2 du 82C59A2 esclave

L'IT de l'entrée E17 correspond à la broche IR1 du 82C59A2 esclave

L'IT de l'entrée E16 correspond à la broche IR0 du 82C59A2 esclave

3.5.7 - Commandes d'initialisation des 82C59A2

Le signal nommé A0 dans la documentation du 82C59A2 correspond à 2 adresses différentes sur la carte TSVME410.

Pour le maître:

A0=0 correspond à l'adresse \$15 (Registre 8259M0)

A0=1 correspond à l'adresse \$17 (Registre 8259M1)

Pour l'esclave:

A0=0 correspond à l'adresse \$19 (Registre 8259S0)

A0=1 correspond à l'adresse \$1B (Registre 8259S1)

Le 82C59A2 possède deux types de mots de commandes :

- les mots de commandes d'initialisation (ICWx) correspondant à l'initialisation du 82C59A2.
- les mots de commandes d'opération (OCWx) permettant de changer de mode de fonctionnement ou de masquer des interruptions à tout moment après l'initialisation.

Pour accéder aux mots de commandes d'initialisation (ICWx), il faut faire un premier accès avec A0 = 0 donc à l'adresse \$15 pour le maître ou \$19 pour l'esclave, et avoir D4 = 1. Ceci permet d'accéder au registre ICW1.

Il faut faire ensuite des accès avec A0 = 1 donc à l'adresse \$17 pour le maître ou \$1B pour l'esclave pour accéder de façon successive aux registres d'initialisation ICW2, ICW3 et ICW4.

ATTENTION :

Le fonctionnement matériel de la carte impose un fonctionnement du 82C59A2 en mode 80C86. Il faut donc mettre le bit D0 de ICW4 à 1.

Le vecteur d'interruption fourni par le 82C59A2 sera de cette forme :

Vecteur d'interruption:

T7	T6	T5	T4	T3	X3	X2	X1
D7	D6	D5	D4	D3	D2	D1	D0

T7, T6, T5, T4, T3 correspondent aux bits programmés dans ICW2, ils représentent un vecteur de base pour une zone mémoire réservée aux vecteurs d'interruptions.

X3, X2, X1 correspondent au niveau d'interruption locale demandée au 82C59A2 (IR0 à IR7).

Exemple 1 : initialisation possible du maître

@ : adresse de base de la carte.

a) MOVE.B #\$11,@+\$15 -> ICW1 = \$11

Mode cascade, interruption sur front, utilisation de ICW4
Les bits D2, D5, D6 et D7 pouvant être quelconques.

b) MOVE.B #\$A0,@+\$17 -> ICW2 = \$A0

Le vecteur de base d'interruption du maître vaut \$A0.
Les bits D0, D1 et D2 pouvant être quelconques.

c) MOVE.B #\$40,@+\$17 -> ICW3 = \$40

La sortie INT du 82C59A2 esclave est reliée à la broche IR6 du 82C59A2 maître.

d) MOVE.B #\$01,@+\$17 -> ICW4 = \$01

Mode "priorité fixe" validé (fully nested mode), mode non bufferisé, EOI (End of Interrupt) normal, mode 80C86.

Le mode 80C86 est obligatoire sur la TSVME410 pour des raisons matérielles.

Exemple 2 : initialisation possible de l'esclave

a) MOVE.B #\$11,@+\$19 -> ICW1 = \$11

Mode cascade, interruption sur front, utilisation de ICW4
Les bits D2, D5, D6 et D7 pouvant être quelconques.

b) MOVE.B #\$80,@+\$1B -> ICW2 = \$80

Le vecteur de base d'interruption de l'esclave vaut \$80.
Les bits D0, D1 et D2 pouvant être quelconques.

c) MOVE.B #\$06,@+\$1B -> ICW3 = \$06

Indique que l'esclave est relié à la broche IR6 du maître
Les bits D3, D4, D5, D6 et D7 doivent être mis à 0.

d) MOVE.B #\$01,@+\$1B -> ICW4 = \$01

Mode "priorité fixe" validé (fully nested mode), mode non bufferisé, EOI (End of Interrupt) normal, mode 80C86.

Le mode 80C86 est obligatoire sur la TSVME410 pour des raisons matérielles.

3.5.8 - Commandes d'opération des 82C59A2

Pour accéder aux mots de commande d'opération (OCWx), il faut faire un premier accès avec A0 = 1, donc à l'adresse \$17 pour le maître ou \$1B pour l'esclave, ceci permet d'accéder au registre OCW1.

Il faut faire ensuite un accès avec A0 = 0, D4 = 0 et D3 = 0 pour accéder à OCW2, ensuite faire un accès avec A0 = 0, D7 = 0 D4 = 0, D3 = 1 pour accéder à OCW3.

Exemple de gestion des interruptions provenant des entrées logiques :

Après avoir initialisé les contrôleurs d'interruption maître et esclave, comme au § 3.5.7, on écrit les mots de commande d'opération de la façon suivante :

Pour le maître

a) MOVE.B #\$BF,@+\$17 -> OCW1 = \$BF

On autorise les interruptions seulement sur l'entrée IR6 du 82C59A2 maître (qui est connectée à la sortie INT de l'esclave).

b) MOVE.B #\$40,@+\$15 -> OCW2 = \$40

Utilisation du mode de priorité fixe (fully nested mode).

c) MOVE.B #\$48,@+\$15 -> OCW3 = \$48

Reset du masque spécial, mode POLL invalidé (fonctionnement sous interruption).

Pour l'esclave

d) MOVE.B #\$0,@+\$1B -> OCW1 = 0

Les interruptions sont valides sur les broches IR0 à IR6 de l'esclave.

e) MOVE.B #\$40,@+\$19 -> OCW2 = \$40

Utilisation du mode de priorité fixe (fully nested mode).

f) MOVE.B #\$48,@+\$19 -> OCW3 = \$48

Reset du masque spécial, mode POLL invalidé (fonctionnement sous interruption).

Par la suite, dans la routine d'interruption, on lance une commande de fin d'interruption non spécifique (non specific EOI command) de la manière suivante :

MOVE.B #\$20,@+\$15 -> OCW2 = \$20 pour le maître

MOVE.B #\$20,@+\$19 -> OCW2 = \$20 pour l'esclave

REQUETES DU MONITEUR MULTITACHES TEMPS REEL

Requêtes de gestion des tâches et de synchronisation directe:

Lancer_tâche(nomTâche)

Cette requête fait passer la tâche concernée de l'état créé à l'état prêt.
Demande d'exécution par le processus "nomTâche".

Désactiver(nomTâche)

Cette requête fait passer la tâche concernée de l'état prêt à l'état créé.

Endormir(temps exprimé en tics)

Cette requête endort la tâche pour un temps déterminé.

Requêtes de synchronisation par événements:

Test_evt(nomEvénement)

Cette requête renvoie vrai (1) ou faux (0) selon que l'événement s'est produit ou non.

Signaler_tempo(délai exprimé en tics, nomEvénement)

Cette requête envoie un événement, lorsque le délai est écoulé, à la tâche qui en a fait la demande.

Signaler_période(périodicité exprimée en tics, nomEvénement)

Cette requête envoie l'événement spécifié de façon cyclique à la tâche qui en a fait la demande. La valeur 0 du premier paramètre met fin à cette primitive.

Attendre_evt()

Cette requête suspend la tâche jusqu'à ce qu'un événement lui soit envoyé.

Signaler_evt(nomTâche, nomEvénement)

Cette requête envoie l'événement spécifié à la tâche désignée lorsqu'il se produit.

Requêtes d'exclusion mutuelle par sémaphore binaire:

Prendre(nomSémaphore)

Demande suspensive d'un sémaphore par son nom.

Vendre(nomSémaphore)

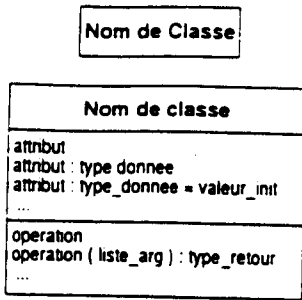
Restitue le sémaphore au système multitâches.

N.B. Dans le système utilisé, le tic est fixé à 10 ms.

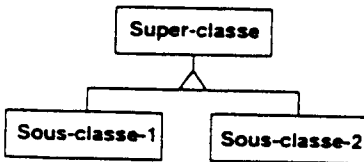
MEMEMTO O.M.T. (Object Modeling Technique)

Notation du modèle objet Concepts de base

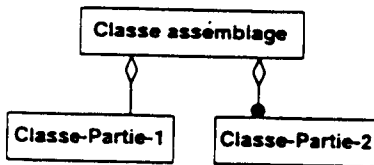
Classe :



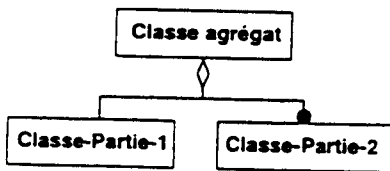
Généralisation (Héritage) :



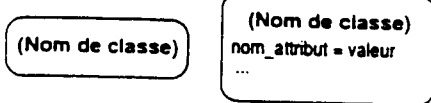
Agrégation :



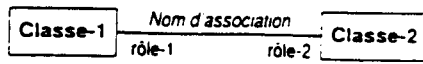
Agrégation (forme alternative) :



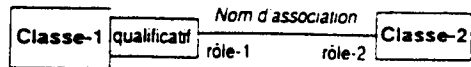
Instances d'objets :



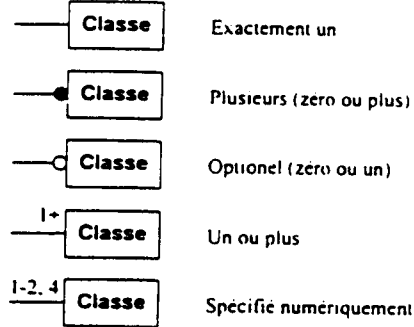
Association :



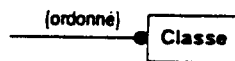
Association qualifiée :



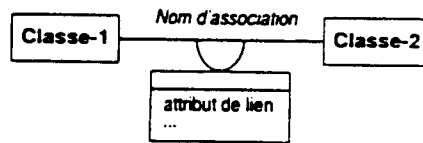
Multiplicité des associations :



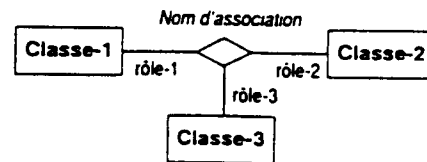
Ordre :



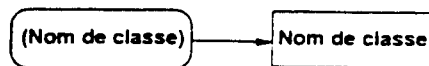
Attribut de lien :



Association ternaire :

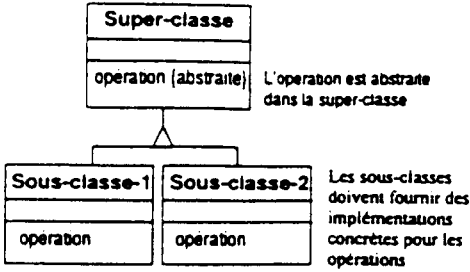


Relation d'instantiation :

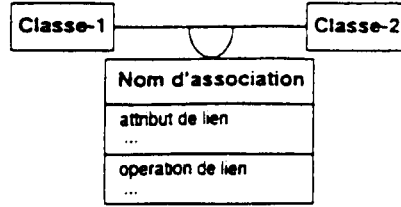


Notation du modèle objet Concepts avancés

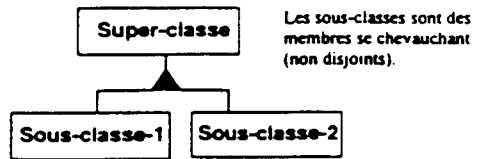
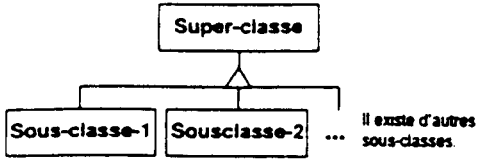
Opération abstraite :



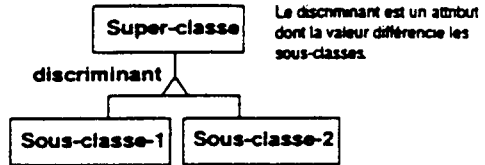
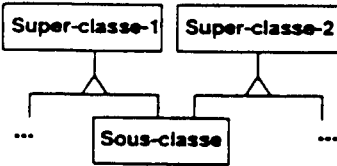
Association en tant que classe :



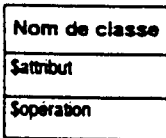
Propriétés de généralisation :



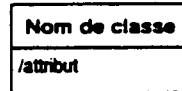
Héritage multiple :



Attributs de classes et attributs d'opérations :



Attribut dérivé :



Propagation des opérations :



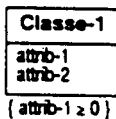
Classe dérivée :



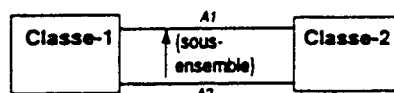
Association dérivée :



Contraintes sur les objets :



Contrainte entre associations :



PROTOTYPES DES CLASSES

```
#ifndef carte_h
#define carte_h
//-----
//
//-----

#include "CarteAn.h"

class Carte {
public:
    Carte();
    void ReturnAdr();
protected:
    long AdrCarte;
    virtual ~Carte()=0;
};
#endif

#ifndef CarteAn_h
#define CarteAn_h
//-----
//
//-----

#include "carte.h"

class CarteAn : public Carte {
public:
    CarteAn(const unsigned long adr, int canaux): Carte(adr);
    virtual ~CarteAn()=0;
    virtual void ConfigAN410() = 0;
    virtual StartConversion(int num_canal) = 0;
    virtual long ReadConversion(int num_canal) = 0;

protected:
    int Nb_Canaux;
};
#endif

#ifndef CarteETOR_h
#define CarteETOR_h
//-----
//
//-----

#include "carte.h"

class CarteETOR : public Carte {
public:
    CarteETOR(const unsigned long adr ,int nb_voies ,int nb_entrees): Carte(adr);
    virtual char ReadInput(int nb_entree , int num_voie) = 0;
    virtual char ReadInputs(int num_voie) = 0;
    virtual ~CarteETOR() = 0;

protected:
    int Nomb_entree;           // Nombre d'entrées par groupe
    int Nomb_voies;           // Nombre de groupes
};
#endif
```

```

#ifndef CarteCompt_h
#define CarteCompt_h
//-----
//
//-----

#include "carte.h"

class CarteCompt : public Carte {
public:
    CarteCompt( const unsigned long adr , int nb_cptr) : Carte(adr) ;
    virtual ConfigCompteur(int num_cptr,short mode) = 0 ;
    virtual StartCompteur(int num_cptr) = 0 ;
    virtual LoadCompteur(int num_cptr) = 0 ;
    virtual unsigned long ReadCompteur(int num_cptr) = 0 ;
    virtual StopCompteur(int num_cptr) = 0 ;
protected:
    int nb_Compt ;
};
#endif

#ifndef Control_IT_h
#define Control_IT_h
//-----
//
//-----

#include "carte.h"

class Control_IT : public Carte {
public:
    Control_IT(const unsigned long adr , int src_it):Carte(adr) ;
    virtual ~Control_IT()=0 ;
    virtual Config_IT(int num_it) = 0 ;
    virtual EnableIT(int num_it) = 0 ;
    virtual DisableIT(int num_it) = 0 ;
    virtual AcquitIT(int num_it) = 0 ;
    virtual SetVector(char vect) = 0 ;
protected:
    int Nb_src_it ;
};
#endif

#ifndef AN410_H
#define AN410_H
//-----
//
//-----

#include "CarteAn.h"

class AN410 : public CarteAn {
public:
    AN410(const unsigned long adr, int canaux): CarteAn(adr, canaux) ;
    ~AN410() ;
    virtual void ConfigAN410() ;
    virtual StartConversion(int num_canal) ;
    virtual long ReadConversion(int num_canal) ;
private:
    short * Rconv_An ;
    char * Wmux ;
};
#endif

```

```

#ifndef ETOR410_h
#define ETOR410_h
//-----
//
//-----

#include "CartETOR.h"
#include "Carte410.h"

class ETOR410 : public CarteETOR {
public
    ETOR410(const unsigned long adr ,int nb_voies ,int nb_entree):CarteETOR(adr , nb_voies,nb_entree) ;
    virtual ~ETOR410() ;
    virtual char ReadInput(int nb_entree , int num_voie) ;
    virtual char ReadInputs(int num_voie) ;

private:
    char *In1, *In2, *In3 ;
    char *Deb ;

};
#endif

#ifndef Compt410_h
#define Compt410_h
//-----
//
//-----

#include "CarteCompt.h"

class Compt410 : public CarteCompt {
public
    Compt410( const unsigned long adr ) :CarteCompt(adr ,5 ) ;
    virtual ~CarteCompt() ;
    virtual ConfigCompteur(int num_cptr,short mode) ;
    virtual StartCompteur(int num_cptr) ;
    virtual LoadCompteur(int num_cptr) ;
    virtual unsigned long ReadCompteur(int num_cptr) ;
    virtual StopCompteur(int num_cptr) ;

protected
    char *Crsr ;
    char *DataPort ;
    short mode[5] ;

};
#endif

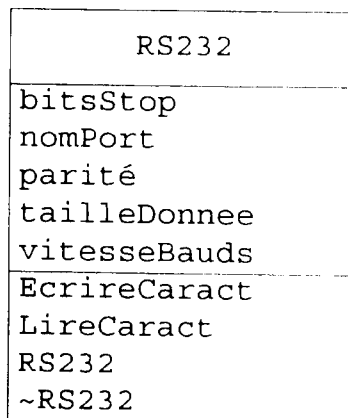
```

CLASSES

Classe RS232

```

////////////////////////////////////
// {{Case Name: RS232
// Code_Filename: c:\selec\system\serie\rs232.hpp
//
class RS232
//
// }}.
{
    // {{Properties...
public :
    virtual virtual void EcrireCaract(char ) = 0 ;
    virtual virtual char LireCaract(void ) = 0 ;
    RS232(char * , long ,char, int, int ) ;
    ~RS232() ;
protected :
    int bitsStop ;
    char nomPort[5] ;
    char parite ;
    int tailleDonne ;
    long vitesseBaud ;
    // }}.
};
// }}.
    
```

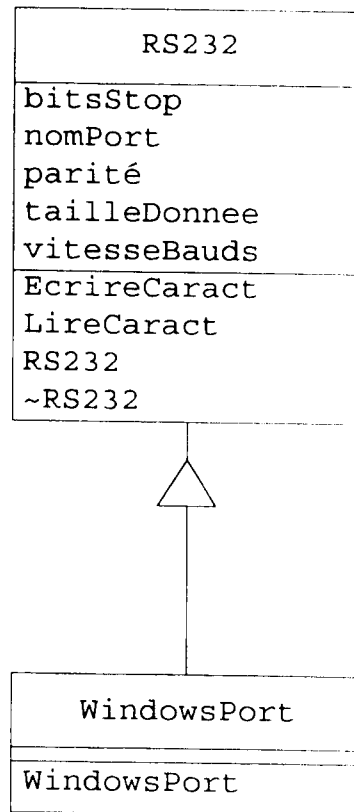


Classe WindowsPort

```

////////////////////////////////////
// {{Case Name: WindowsPort
// Code_Filename: c:\select\system\serie\winport
//
class WindowsPort:
    public RS232
//
// }}.
{
    // {{Properties...
public :
    void WindowsPort(char * nP, long vB, char p , int tD, int bS) ;
    // }}.
};
// }}.

////////////////////////////////////
// {{Case Name: WindowsPort [WindowsPort.]
// Description: Initialise le port série en construisant la variable de
// type DCB.
//
void WindowsPort::WindowsPort(char * nP, long vB, char p , int tD, int bS)
//
// }}.
{
    // Body to be coded here...
}
// }}.
    
```



FONCTIONS DE COMMUNICATION WINDOWS

BuildCommDCB (2.x)

```
int BuildCommDCB(lpszDef, lpdcb)
```

```
LPCSTR lpszDef:      /* address of device-control string */
DCB FAR* lpdcb:     /* address of device-control block */
```

The BuildCommDCB function translates a device-definition string into appropriate serial device control block (DCB) codes

Parameter	Description
-----------	-------------

lpszDef	Points to a null-terminated string that specifies device-control information. The string must have the same form as the parameters used in the MS-DOS mode command.
lpdcb	Points to a DCB structure that will receive the translated string. The structure defines the control settings for the serial-communications device.

Returns

The return value is zero if the function is successful. Otherwise, it is -1.

Example

```
err = BuildCommDCB("COM1:9600,n,8,1", &dcb);
if (err < 0) {
    ShowError(err, "BuildCommDCB");
    return 0;
}
```

Comments

The BuildCommDCB function only fills the buffer. To apply the settings to a port, an application should use the SetCommState function.

By default, BuildCommDCB specifies XON/XOFF and hardware flow control as disabled. To enable flow control, an application should set the appropriate members in the DCB structure.

CloseComm (2.x)

```
int CloseComm(idComDev)
```

```
int idComDev: /* device to close */
```

The CloseComm function closes the specified communications device and frees any memory allocated for the device's transmission and receiving queues. All characters in the output queue are sent before the communications device is closed.

Parameter	Description
-----------	-------------

idComDev	Specifies the device to be closed. The OpenComm function returns this value.
----------	--

Returns

The return value is zero if the function is successful. Otherwise, it is less than zero.

GetCommEventMask (2.x)

```
UINT GetCommEventMask(idComDev, fnEvtClear)
```

```
int idComDev: /* communications device identifier */
```

```
int fnEvtClear: /* events to clear in the event word */
```

The GetCommEventMask function retrieves and then clears the event word for a communications device.

Parameter	Description
idComDev	Specifies the communication device to be examined. The OpenComm function returns this value.
fnEvtClear	Specifies which events are to be cleared in the event word. For a list of the event values, see the description of the SetCommEventMask function.

Returns

The return value specifies the current event-word value for the specified communications device if the function is successful. Each bit in the event word specifies whether a given event has occurred; a bit is set (to 1) if the event has occurred.

Comments

Before the GetCommEventMask function can record the occurrence of an event, an application must enable the event by using the SetCommEventMask function.

If the communication device event is a line-status or printer error, the application should call the GetCommError function after calling GetCommEventMask.

OpenComm (2.x)

```
int OpenComm(lpszDevControl, cbInQueue, cbOutQueue)
```

```
LPCSTR lpszDevControl: /* address of device-control information */
```

```
UINT cbInQueue: /* size of receiving queue */
```

```
UINT cbOutQueue: /* size of transmission queue */
```

The OpenComm function opens a communications device.

Parameter	Description
lpszDevControl	Points to a null-terminated string that specifies the device in the form COMn or LPTn, where n is the device number.
cbInQueue	Specifies the size, in bytes, of the receiving queue. This parameter is ignored for LPT devices.
cbOutQueue	Specifies the size, in bytes, of the transmission queue. This parameter is ignored for LPT devices.

Returns

The return value identifies the open device if the function is successful. Otherwise, it is less than zero.

Errors

If the function fails, it may return one of the following error values:

Value	Meaning
IE_BADID	The device identifier is invalid or unsupported.
IE_BAUDRATE	The device's baud rate is unsupported.
IE_BYTESIZE	The specified byte size is invalid.
IE_DEFAULT	The default parameters are in error.

IE_HARDWARE	The hardware is not available (is locked by another device).
IE_MEMORY	The function cannot allocate the queues.
IE_NOPEN	The device is not open.
IE_OPEN	The device is already open.

If this function is called with both queue sizes set to zero, the return value is IE_OPEN if the device is already open or IE_MEMORY if the device is not open.

Comments

Windows allows COM ports 1 through 9 and LPT ports 1 through 3. If the device driver does not support a communications port number, the OpenComm function will fail.

The communications device is initialized to a default configuration.

The SetCommState function should be used to initialize the device to alternate values.

The receiving and transmission queues are used by interrupt-driven device drivers. LPT ports are not interrupt driven for these ports, the cbInQueue and cbOutQueue parameters are ignored and the queue size is set to zero.

SetCommEventMask (2.x)

```
UINT FAR* SetCommEventMask(idComDev, fuEvtMask)
```

```
int idComDev;          /* device to enable    */
UINT fuEvtMask;       /* events to enable    */
```

The SetCommEventMask function enables events in the event word of the specified communications device.

Parameter	Description
idComDev	Specifies the communications device to be enabled. The OpenComm function returns this value.
fuEvtMask	Specifies which events are to be enabled. This parameter can be any combination of the following values:

Value Meaning

EV_BREAK	Set when a break is detected on input.
EV_CTS	Set when the CTS (clear-to-send) signal changes state.
EV_CTSS	Set to indicate the current state of the CTS signal.
EV_DSR	Set when the DSR (data-set-ready) signal changes state.
EV_ERR	Set when a line-status error occurs. Line-status errors are CE_FRAME, CE_OVERRUN, and CE_RXPARITY.
EV_PERR	Set when a printer error is detected on a parallel device. Errors are CE_DNS, CE_IOE, CE_LOOP, and CE_PTO.
EV_RING	Set to indicate the state of ring indicator during the last modem interrupt.
EV_RLSD	Set when the RLSD (receive-line-signal-detect) signal changes state.
EV_RLSDS	Set to indicate the current state of the RLSD signal.
EV_RXCHAR	Set when any character is received and placed in the receiving queue.
EV_RXFLAG	Set when the event character is received and placed in the receiving queue. The event character is specified in the device's control block.
EV_TXEMPTY	Set when the last character in the transmission queue is sent.

Returns

The return value is a pointer to the event word for the specified communications device, if the function is successful. Each bit in the event word specifies whether a given event has occurred. A bit is 1 if the event has occurred.

Comments

Only enabled events are recorded. The GetCommEventMask function retrieves and clears the event word.

SetCommState (2.x)

```
int SetCommState(lpdcb)
```

```
const DCB FAR* lpdcb; /* address of device control block */
```

The SetCommState function sets a communications device to the state specified by a device control block.

Parameter	Description
-----------	-------------

lpdcb	Points to a DCB structure that contains the desired communications settings for the device. The Id member of the DCB structure must identify the device.
-------	--

Returns

The return value is zero if the function is successful. Otherwise, it is less than zero.

ReadComm (2.x)

```
int ReadComm(idComDev, lpvBuf, cbRead)
```

```
int idComDev;          /* identifier of device to read from */
void FAR* lpvBuf;      /* address of buffer for read bytes */
int cbRead;            /* number of bytes to read */
```

The ReadComm function reads up to a specified number of bytes from the given communications device.

Parameter	Description
-----------	-------------

idComDev	Specifies the communications device to be read from. The OpenComm function returns this value.
lpvBuf	Points to the buffer for the read bytes.
cbRead	Specifies the number of bytes to be read.

Returns

The return value is the number of bytes read, if the function is successful. Otherwise, it is less than zero and its absolute value is the number of bytes read.

For parallel I/O ports, the return value is always zero.

Comments

When an error occurs, the cause of the error can be determined by using the GetCommError function to retrieve the error value and status. Since errors can occur when no bytes are present, if the return value is zero, the GetCommError function should be used to ensure that no error occurred.

The return value is less than the number specified by the cbRead parameter only if the number of bytes in the receiving queue is less than that specified by cbRead. If the return value is equal to cbRead, additional bytes may be queued for the device. If the return value is zero, no bytes are present.

WriteComm (2.x)

```
int WriteComm(idComDev, lpvBuf, cbWrite)
```

```
int idComDev,          /* identifier of comm. device */  
const void FAR* lpvBuf, /* address of data buffer */  
int cbWrite;          /* number of bytes to write */
```

The WriteComm function writes to the specified communications device.

Parameter	Description
-----------	-------------

idComDev	Specifies the device to receive the bytes. The OpenComm function returns this value.
lpvBuf	Points to the buffer that contains the bytes to be written.
cbWrite	Specifies the number of bytes to be written.

Returns

The return value specifies the number of bytes written, if the function is successful. The return value is less than zero if an error occurs, making the absolute value of the return value the number of bytes written.

Comments

To determine what caused an error, use the GetCommError function to retrieve the error value and status.

For serial ports, the WriteComm function deletes data in the transmission queue if there is not enough room in the queue for the additional bytes. Before calling WriteComm, applications should check the available space in the transmission queue by using the GetCommError function. Also, applications should use the OpenComm function to set the size of the transmission queue to an amount no smaller than the size of the largest expected output string.