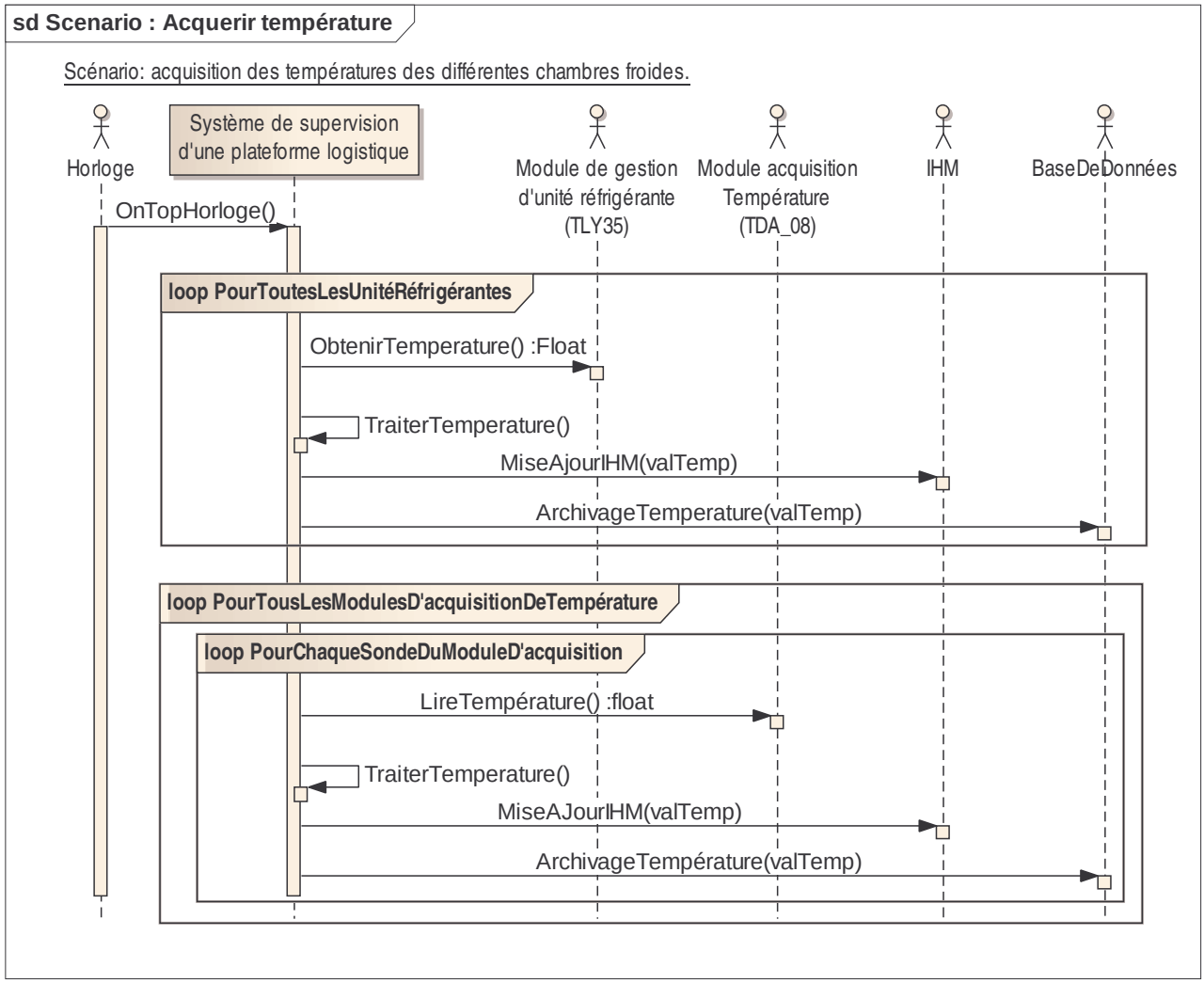
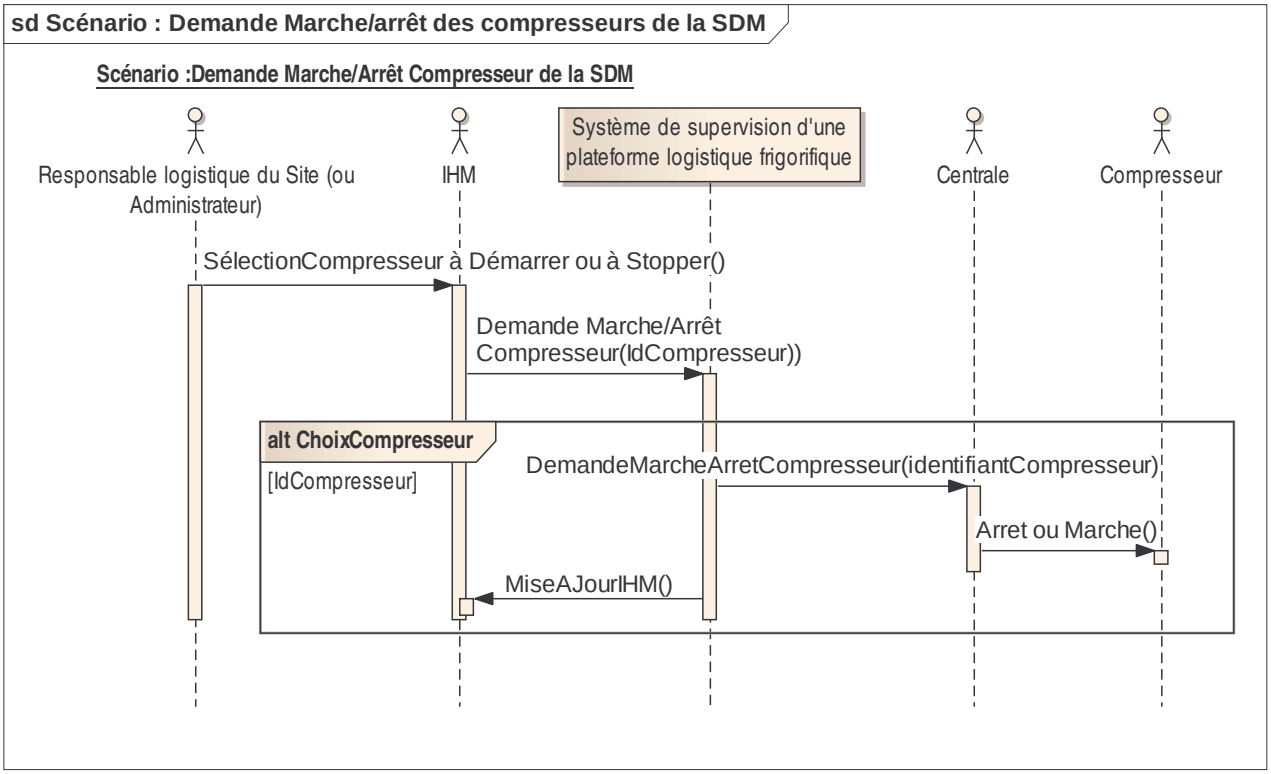
Annexe 2 : Éléments d'analyse

1 Les principaux scenarii du système

1.1 Acquisition des températures



1.2 Demande d'arrêt ou de démarrage d'un compresseur



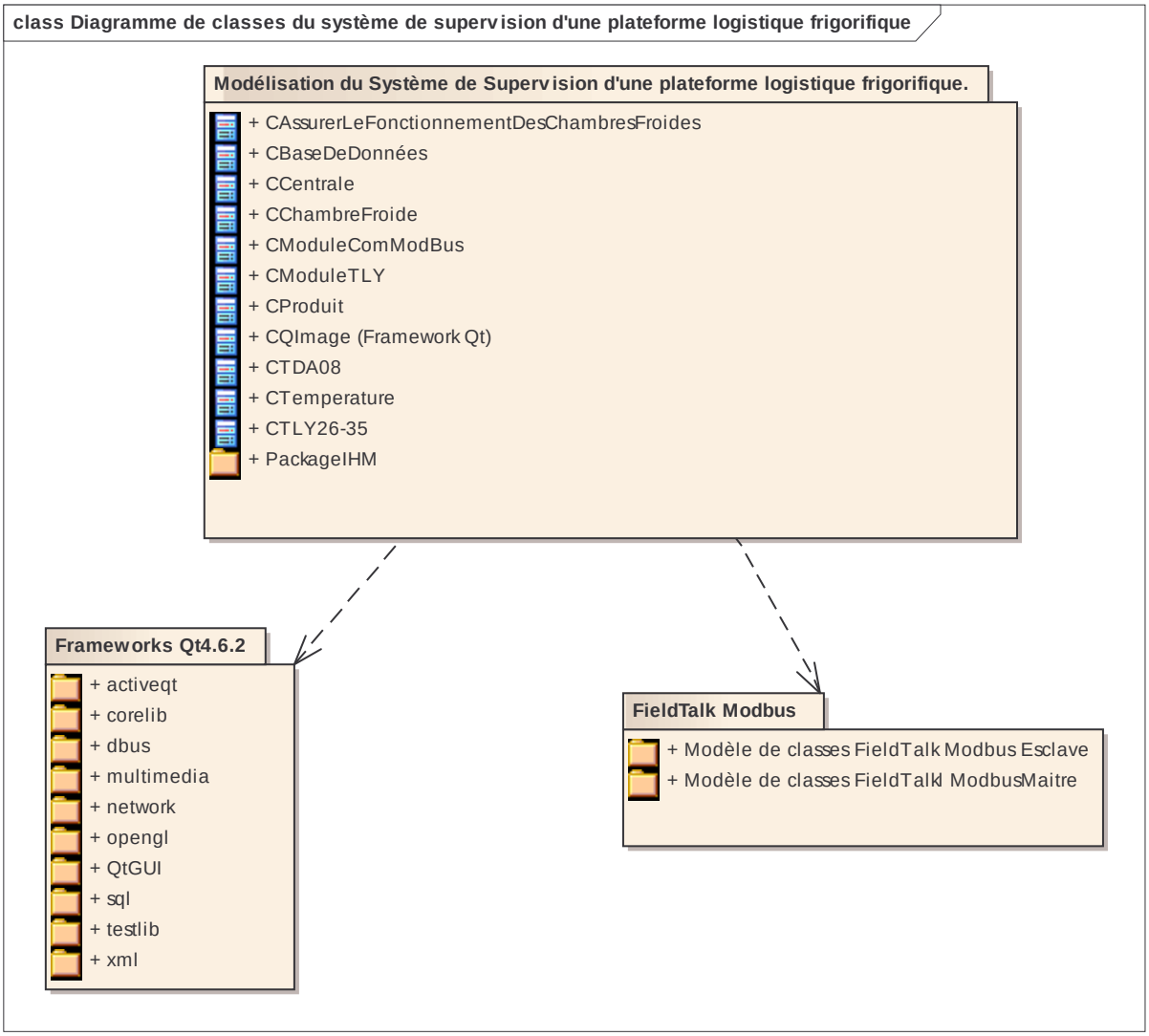
Les différentes centrales assurent la gestion des compresseurs.

Remarque : chaque compresseur possède un compresseur redondant. Chaque zone de réfrigération possède donc deux compresseurs. Le dimensionnement de ces derniers a été calculé afin de garantir le refroidissement de la zone réfrigérée même en cas de défaillance de l'un des deux.
Une répartition de charge d'utilisation des compresseurs permet de garantir que ceux-ci sont toujours opérationnels et sans utilisation intensive. Les centrales assurent le fonctionnement optimal des compresseurs et donc la régulation en température des différentes chambres froides. En cas d'arrêt de la supervision, le fonctionnement en mode dégradé est pris en charge par les centrales.

Ainsi lorsque l'on veut arrêter un compresseur, la supervision n'agit pas directement sur le compresseur lui-même dans la SDM (Salle Des Machines), mais il en fait la demande à la centrale appropriée qui gèrera, elle, l'arrêt du compresseur. Idem pour la remise en marche.

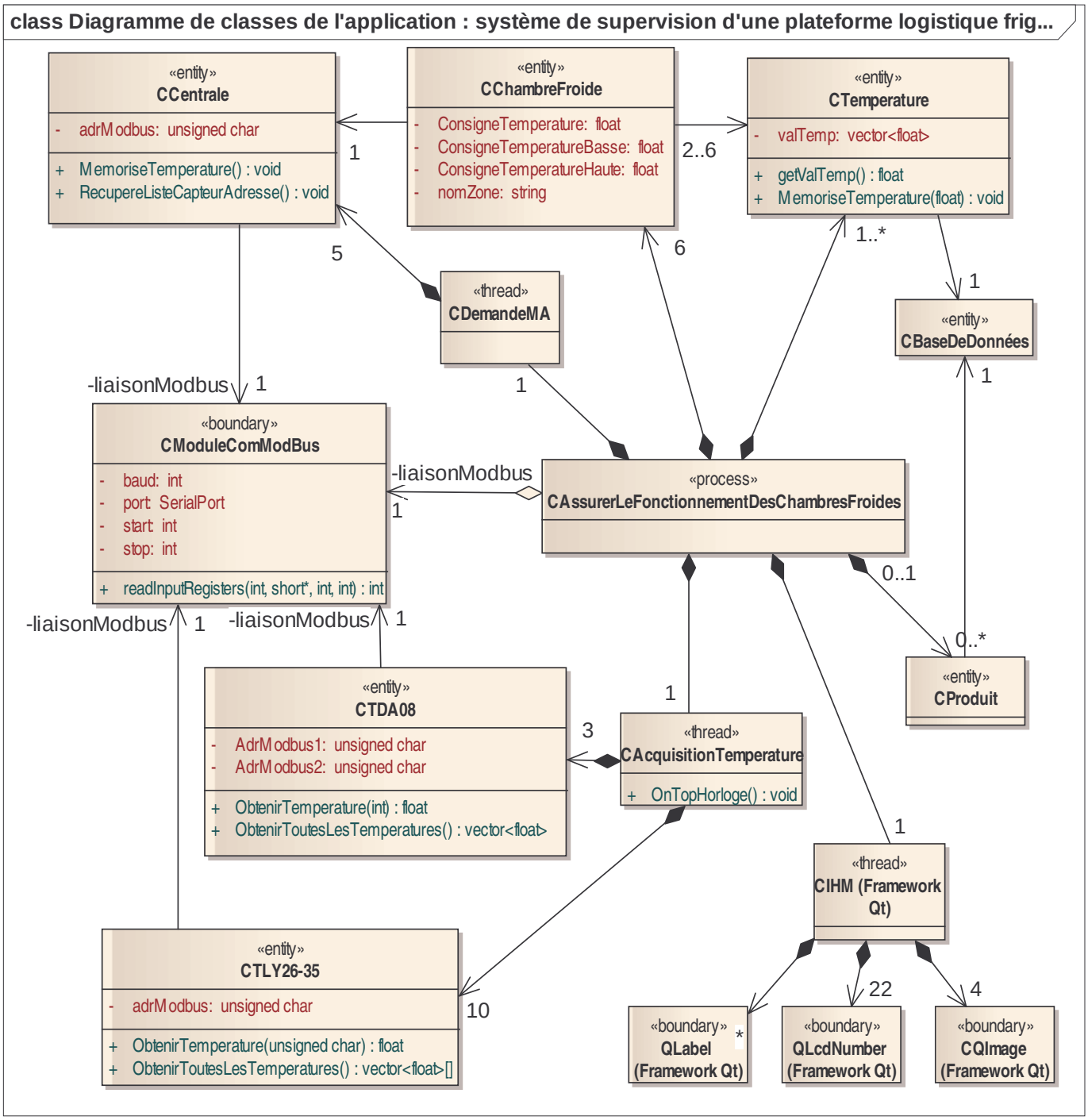
2 Diagramme de classes

2.1 Les différents « packages » utilisés pour l'application



2.2 Le diagramme de classes de la partie supervision

Pour des raisons de lisibilité, les classes associées exclusivement aux cas d'utilisations qui suivent, ne sont pas représentées : « Gérer la traçabilité des produits », « Configuration et Maintenance », « Fixer les températures des chambres froides » et « Gérer le dégivrage des unités réfrigérantes ».



Remarque : le Thread **CAcquisitionTemperature** accède aux différentes températures, via le process principal : **CAssurerLeFonctionnementDesChambresFroides**. Il peut être vu comme une tâche périodique.
De même, le process **CAssurerLeFonctionnementDesChambresFroides** met à jour périodiquement l'IHM.

Annexe 3 : Principe de production du froid par une machine frigorifique

Le système frigorifique, qui prélève de la chaleur à la source froide grâce à un circuit de captage, dispose de quatre organes principaux (voir figure 1 ci-dessous) :

1. L'**évaporateur** (c'est la source froide) : la chaleur est prélevée au fluide secondaire (eau, air) pour vaporiser le fluide frigorigène. Soit **E+** l'énergie prise dans la chambre froide (le frigo) par la machine frigorifique.
2. Le **compresseur** : actionné par un moteur électrique, il élève la pression et la température du fluide frigorigène gazeux en le comprimant. Soit **W** l'énergie consommée par le compresseur (travail, consommation électrique). Autrement dit **W** est l'énergie prise au réseau électrique par la machine frigorifique.
3. Le **condenseur** (c'est la source chaude) : le fluide frigorigène libère sa chaleur au fluide secondaire (eau, air...) en passant de l'état gazeux à l'état liquide. Soit **E-** l'énergie rejetée à l'extérieur.
4. Le **détendeur** : il réduit la pression du fluide frigorigène en phase liquide. L'apport du détendeur est neutre dans le bilan énergétique.

Vocabulaire métier : Un **fluide frigorigène** (ou **réfrigérant**) est un fluide pur ou un mélange de fluides purs présents en phase liquide, gazeuse ou les deux à la fois en fonction de la température et de la pression de celui-ci. La principale propriété des fluides frigorigènes est de s'évaporer à une faible température sous pression atmosphérique. Les fluides frigorigènes sont utilisés dans les systèmes de production de froid (climatisation, congélateur, réfrigérateur, etc.)

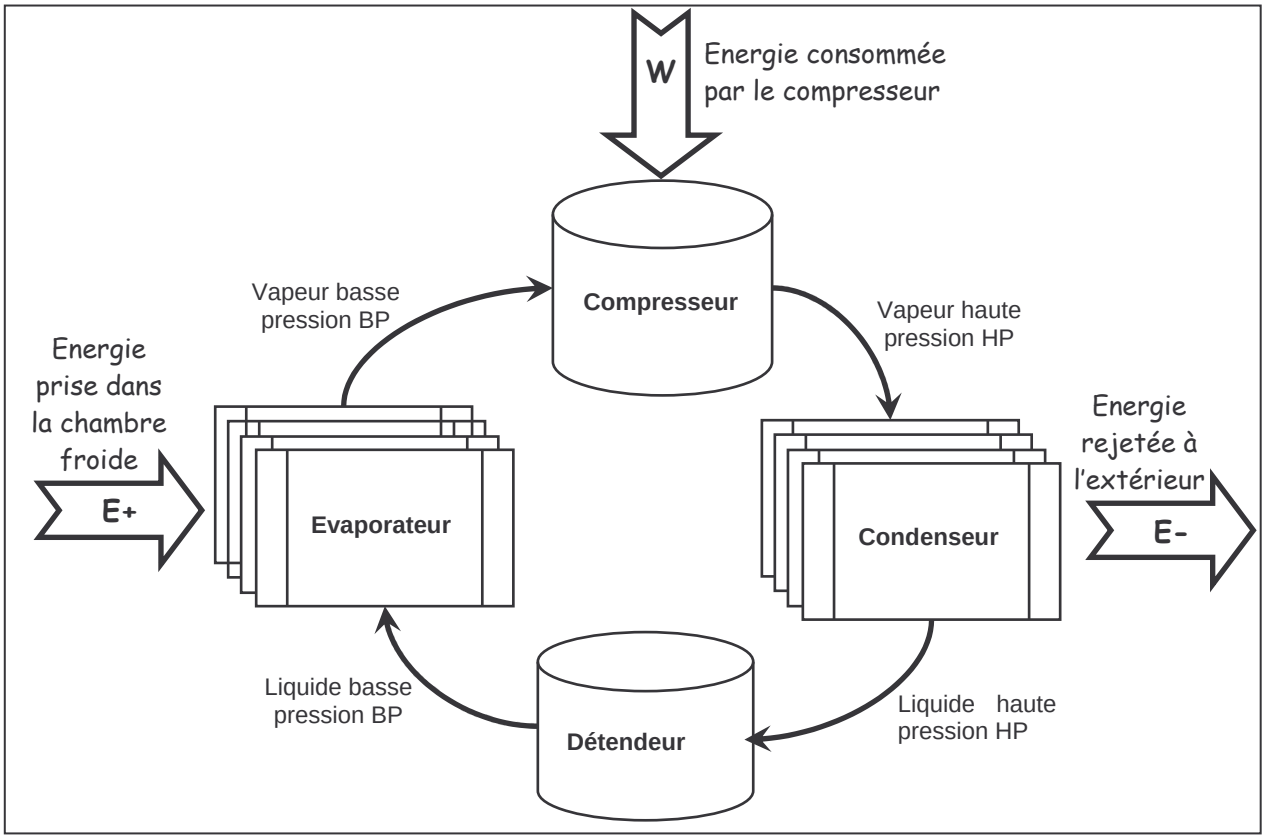


Figure 1 : Principe de fonctionnement du cycle thermodynamique d'une machine frigorifique

Le fluide circulant dans une machine frigorifique subit un cycle de transformation composé de quatre étapes :

- Dans **l'évaporateur**, le fluide à l'état liquide récupère de l'énergie sous forme de chaleur en s'évaporant. La pression reste constante et le fluide passe à l'état vapeur.
- À la sortie de **l'évaporateur**, le fluide est surchauffé et à faible pression.
- Dans le **compresseur**, la vapeur est comprimée et passe donc d'une basse pression à une pression plus élevée grâce à l'énergie mécanique fournie par le compresseur. Conséquence, sa température s'élève aussi.
- À la sortie du **compresseur**, le fluide est à l'état vapeur, à haute pression et sa température est élevée.
- Dans le **condenseur**, le fluide passe à l'état liquide et cède de l'énergie qui est transférée vers l'extérieur (circuit de chauffage) sous forme de chaleur. Conséquence, à la sortie du **condenseur**, le fluide (en phase liquide) voit sa température fortement diminuer.
- Le fluide rentre dans le **détendeur** à l'état liquide et passe de la haute à la basse pression, sans échange d'énergie. La température du fluide baisse et le cycle recommence.

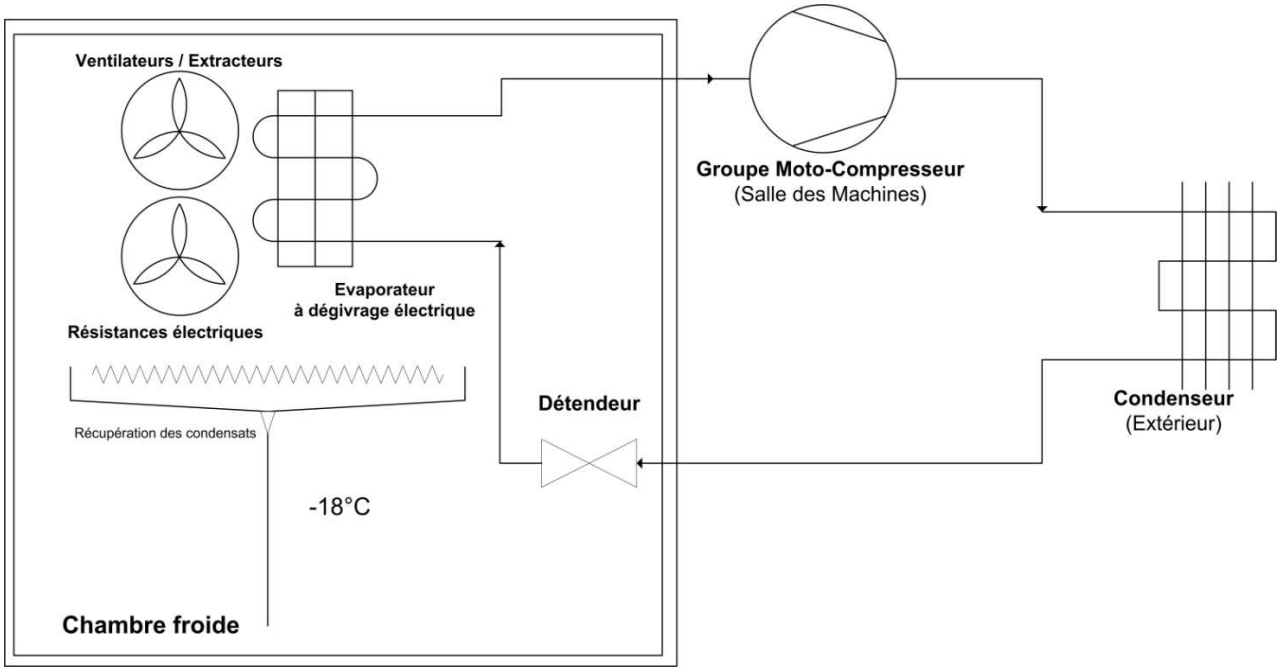


Figure 2 : Synoptique du cycle thermodynamique d'une machine frigorifique

Les condensats sont constitués de l'eau de dégivrage des évaporateurs. L'humidité résiduelle, captée dans l'air ambiant des chambres froides, se condense sur l'évaporateur et forme du givre. Il faut donc périodiquement réchauffer ces derniers à l'aide de résistances électriques, afin de faire fondre la glace et conserver un maximum d'efficacité pour la production de froid.

Annexe 4 : Éléments de thermodynamique

3 Chaleur massique

La **chaleur massique** ou **chaleur spécifique** (symbole **c** ou **s**), qu'il convient d'appeler capacité thermique massique, est déterminée par la quantité d'énergie à apporter par échange thermique pour élever (ou abaisser) d'un degré Kelvin (ou Celsius) la température de l'unité de masse d'une substance.

L'unité du système international est alors le joule par kilogramme-kelvin ($J.kg^{-1}.K^{-1}$). La détermination des valeurs des capacités thermiques des substances relève de la calorimétrie.

4 Chaleur latente

La **chaleur latente** (ou enthalpie de changement d'état) correspond à la quantité de chaleur nécessaire à l'unité de masse (kg) d'une substance pour qu'elle change d'état ; cette transformation ayant lieu à pression constante. Il existe trois états physiques principaux pour toute substance pure : l'état **solide**, l'état **liquide** et l'état **gazeux**.

Par exemple, l'eau bout à 100 °C sous la pression d'1 atmosphère (1atm = 101325 Pascal). La chaleur latente, égale à la quantité de chaleur fournie pour transformer l'eau liquide en vapeur, est de 2257 kJ/kg.

5 Chaleur massique et chaleur latente des aliments

Le tableau ci-après donne la chaleur massique (ou spécifique) des principaux aliments.

Comment lire ce tableau ?

Prenons le cas des laitages :

- Au dessus de 0°Celsius, il faut retirer (ou fournir) 1,05 Watt-heures à 1 kg de laitages pour abaisser (ou élever) la température de ces laitages de 1°Kelvin (ou 1° Celsius) ;
- En dessous de 0°Celsius, il faut retirer (ou fournir) 0,53 Watt-heures à 1 kg de laitages pour abaisser (ou élever) la température de ces laitages de 1°Kelvin (ou 1° Celsius) ;
- Il faut retirer 80 Watt-heures à 1 kg de laitages pour le congeler.

Produits	Chaleur spécifique (moyennes) au-dessus de 0°C (Wh/kgK)	Chaleur spécifique (moyennes) en-dessous de 0°C (Wh/kgK)	Chaleur latente de congélation (moyennes) (Wh/kg)
Viandes	0,87	0,47	64
Poissons	0,93	0,50	67
Fruits et légumes	1,04	0,53	80
Laitages	1,05	0,53	80
Fromage/beurre	0,76	0,41	47
Boissons	1,10	0,56	87
Pain/pâtisserie	0,52	0,52	37

Figure 1 : Chaleur massique et chaleur latente des aliments

Note :

$1\,Wh = 3{,}6\,kJ$ (1 Watt-heure est égal à 3,6 kilo-Joules)

Annexe 5 : Sélection des compresseurs

Type de compresseur	Température de condensation °C	Cooling capacity Q_o [Watt] — Puissance frigorifique — Evaporation temperature °C							Power consumption P_e [kW] — Puissance absorbée — Température d'évaporation °C						
		-15	-20	-25	-30	-35	-40	-45	-15	-20	-25	-30	-35	-40	-45
HSN5343-20	30	58700	49450	41250	34000	27650	22050	17220	22,10	21,50	20,90	20,20	19,46	18,63	17,70
	40	55600	46700	38750	31750	25550	20100	15330	27,90	27,10	26,30	25,30	24,20	23,00	21,50
	50			35200	28400	22350	17020	12330			32,00	30,70	29,20	28,10	26,00
HSN5353-25	30	68500	57700	48200	39800	32350	25850	20200	25,30	24,50	23,80	22,90	22,00	21,10	20,00
	40	64800	54500	45350	37200	30050	23750	18260	31,70	30,80	29,80	28,70	27,50	26,10	24,50
	50			43400	34850	27350	20500	15110			39,70	37,20	34,70	32,10	29,90
HSN5363-30	30	78200	66000	55200	45600	37150	29800	23400	28,60	27,60	26,70	25,70	24,60	23,50	22,30
	40	73900	62400	52100	42950	34900	27850	21700	36,00	34,90	33,70	32,40	31,00	29,50	28,00
	50			47700	39150	31550	24800	18930			41,20	39,60	37,90	37,20	35,10
HSN6451-40	30	94400	79600	66300	54700	44400	35350	27500	32,20	31,10	30,00	28,80	27,70	26,40	25,10
	40	89100	75000	62500	51500	41750	33250	25850	39,90	38,70	37,50	36,10	34,70	33,20	31,50
	50			56900	46700	37700	29800	22900			45,30	44,00	42,50	42,00	39,90
HSN6461-50	30	108700	91600	76500	63200	51600	41400	32600	39,60	38,10	36,50	34,90	33,20	31,50	29,60
	40	101200	85400	71500	59100	48250	38750	30500	48,70	46,80	44,90	42,90	40,90	38,80	36,80
	50			64300	53100	43200	34400	26700			53,50	52,20	51,00	48,70	46,60
HSN7451-60	30	136000	113800	94400	77300	62500	49550	38300	47,00	44,70	43,00	41,70	40,50	39,20	37,50
	40	124600	104400	86700	71100	57500	45600	35300	56,90	55,10	53,60	52,20	50,90	49,40	47,50
	50			76600	62900	50800	40150	30850			64,50	63,80	63,70	61,90	60,20
HSN7461-70	30	151800	127500	106100	87400	71000	56700	44250	50,70	48,70	46,90	45,10	43,30	41,50	39,60
	40	140800	118200	98500	81100	65900	52700	41200	61,70	59,80	57,80	55,80	53,70	51,60	49,30
	50		105800	88000	72400	58700	46650	36150		74,30	72,10	69,90	67,50	64,90	62,00
HSN7471-75	30	160100	134500	112100	92500	75500	60700	47850	57,00	54,60	52,30	50,00	47,60	45,10	42,60
	40	147100	123800	103300	85500	69800	56200	44400	69,30	66,50	63,70	60,90	58,10	55,30	52,60
	50	130300	109800	91700	75800	61800	49450	38600							

Température d'évaporation = Température de la chambre froide + (-7°C).
Température de condensation = Température extérieure max + 15°C (pour la région lyonnaise, la température extérieure max vaut 35°C).

Annexe 6 : Extrait documentation modbus

Cette annexe est un extrait du document « MODBUS over serial line specification and implementation guide V1.02 » disponible sur le site http://www.modbus.org.

2.5 The two serial Transmission Modes

Two different serial transmission modes are defined: The RTU mode and the ASCII mode.

It defines the bit contents of message fields transmitted serially on the line. It determines how information is packed into the message fields and decoded.

The transmission mode (and serial port parameters) must be the same for all devices on a MODBUS Serial Line.

Although the ASCII mode is required in some specific applications, interoperability between MODBUS devices can be reached only if each device has the same transmission mode: **All devices must implement the RTU Mode.** The ASCII transmission mode is an option.

Devices should be set up by the users to the desired transmission mode, RTU or ASCII. Default setup must be the RTU mode.

2.5.1 RTU Transmission Mode

When devices communicate on a MODBUS serial line using the RTU (Remote Terminal Unit) mode, each 8–bit byte in a message contains two 4–bit hexadecimal characters. The main advantage of this mode is that its greater character density allows better data throughput than ASCII mode for the same baud rate. Each message must be transmitted in a continuous stream of characters.

The format (11 bits) for each byte in RTU mode is:

Coding System: 8–bit binary

Bits per Byte: 1 start bit
8 data bits, least significant bit sent first
1 bit for parity completion
1 stop bit

Even parity is required; other modes (odd parity, no parity) may also be used. In order to ensure a maximum compatibility with other products, it is recommended to support also No parity mode. The default parity mode must be even parity.

Remark: the use of no parity requires 2 stop bits.

How Characters are Transmitted Serially:
Each character or byte is sent in this order (left to right):
Least Significant Bit (LSB) . . . Most Significant Bit (MSB)

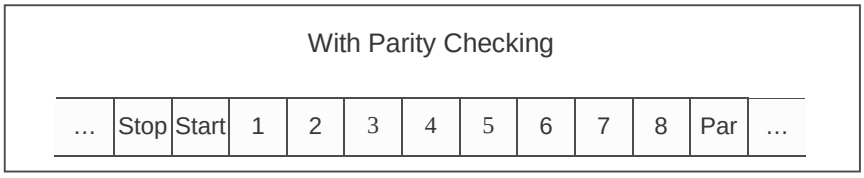


Figure 10: Bit Sequence in RTU mode

Devices may accept by configuration either Even, Odd, or No Parity checking. If No Parity is implemented, an additional stop bit is transmitted to fill out the character frame to a full 11-bit asynchronous character:

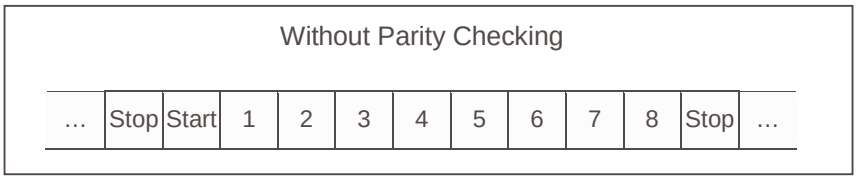


Figure 11: Bit Sequence in RTU mode (specific case of No Parity)

Frame Checking Field: Cyclical Redundancy Checking (CRC)

Frame description:

Slave Address	Function Code	Data	CRC
1 byte	1 byte	0 up to 252 byte(s)	2 bytes CRC Low | CRC Hi

Figure 12: RTU Message Frame

➔ The maximum size of a MQDBUS RTU frame is 256 bytes.

2.5.1.1 MODBUS Message RTU Framing

A MODBUS message is placed by the transmitting device into a frame that has a known beginning and ending point. This allows devices that receive a new frame to begin at the start of the message, and to know when the message is completed. Partial messages must be detected and errors must be set as a result.

In RTU mode, message frames are separated by a silent interval of at least 3.5 character times. In the following sections, this time interval is called t3,5.

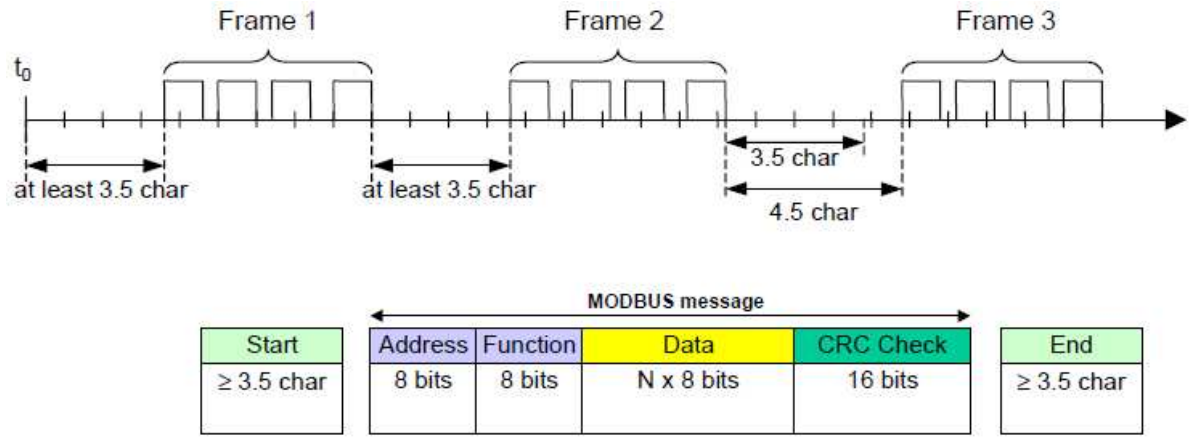
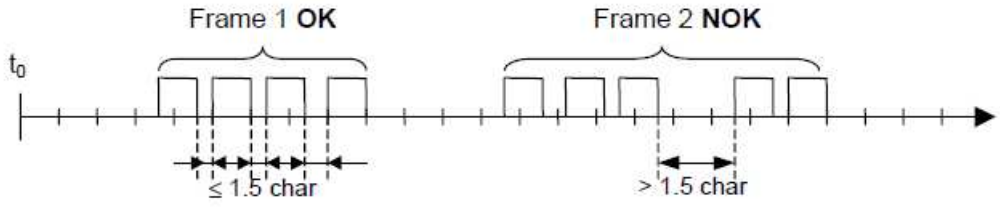


Figure 13: RTU Message Frame

The entire message frame must be transmitted as a continuous stream of characters.

If a silent interval of more than 1.5 character times occurs between two characters, the message frame is declared incomplete and should be discarded by the receiver.



Remark:
The implementation of RTU reception driver may imply the management of a lot of interruptions due to the $t_{1.5}$ and $t_{3.5}$ timers. With high communication baud rates, this leads to a heavy CPU load. Consequently these two timers must be strictly respected when the baud rate is equal or lower than 19200 Bps. For baud rates greater than 19200 Bps, fixed values for the 2 timers should be used: it is recommended to use a value of 750µs for the inter-character time-out ($t_{1.5}$) and a value of 1.750ms for inter-frame delay ($t_{3.5}$).

The following drawing provides a description of the RTU transmission mode state diagram. Both “master“ and “slave“ points of view are expressed in the same drawing:

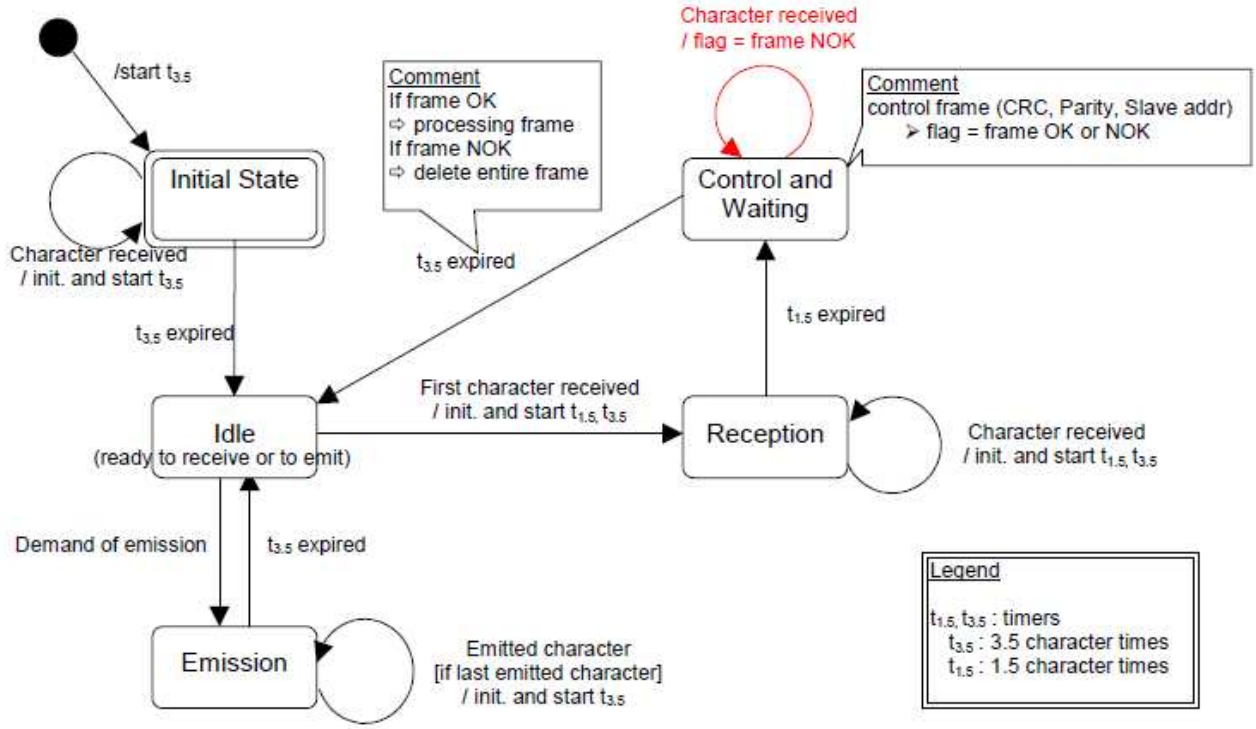


Figure 14: RTU transmission mode state diagram

Some explanations about the above state diagram:

- Transition from “Initial State“ to “Idle“ state needs $t_{3.5}$ time-out expiration: that insures inter-frame delay
- “Idle“ state is the normal state when neither emission nor reception is active.
- In RTU mode, the communication link is declared in “idle“ state when there is no transmission activity after a time interval equal to at least 3,5 characters.
- When the link is in idle state, each transmitted character detected on the link is identified as the **start of a frame**. The link goes to the “active“ state. Then, the **end of frame** is identified when no more character is transmitted on the link after the time interval t3,5.
- After detection of the end of frame, the CRC calculation and checking is completed. Afterwards the address field is analysed to determine if the frame is for the device. If not the frame is discarded. In order to reduce the reception processing time the address field can be analysed as soon as it is received without waiting the end of frame. In this case the CRC will be calculated and checked only if the frame is addressed to the slave (broadcast frame included).

Annexe 7 : Modules TDA08/TDA04

TDA08/TDA04

Protocole de communication

Extraits du manuel d'utilisation

SOMMAIRE :
(Note : plusieurs paragraphes qui ne sont pas utiles pour répondre aux questions du sujet ont été retirés)

1 Introduction

Ce document a le but de décrire les capacités de communication de tous les modules d'acquisition TDA qui utilisent le protocole MODBUS et il est surtout adressé aux techniciens, intégrateurs de systèmes et créateurs de logiciel.

Il est subdivisé en quatre parties :

- la première décrit la connexion physique à la ligne ;
- la seconde présente le protocole de communication, qui est un sous-ensemble du MODBUS RTU[1] ;
- la troisième partie décrit les différents types de données qui peuvent être échangées ;
- la quatrième reporte les performances typiques du système.

2 Connexion physique

2.1 Interface

Les modules TDA sont munis d'interface de communication sérielle optoisolée pour éviter l'apparition des problèmes dus aux potentiels de terre.

En position d'attente le module est en condition de réception et passe en transmission après avoir reçu et décodé un message correct qui lui est adressé.

Chaque module est muni d'un switch rotatif à 16 positions qui permet de programmer son adresse modbus. Les positions valables sont 15 (de 1 à 15, l'adresse ZERO est réservée par le MODBUS RTU pour les messages de broadcasting, mais elle n'est pas adoptée pour le TDA vu le manque de fiabilité implicite de ce type de communication).

Le tableau suivant illustre les programmations possibles :

Position switch rotative	Adresse du module	
	TDA08	TDA04
0	Non valable	Non valable
1	2 et 3	1
2	4 et 5	2
3	6 et 7	3
4	8 et 9	4
5	10 et 11	5
6	12 et 13	6
7	14 et 15	7
8	16 et 17	8
9	18 et 19	9
A	20 et 21	10
B	22 et 23	11
C	24 et 25	12
D	26 et 27	13
E	28 et 29	14
F	30 et 31	15

[1] Marque enregistrée par AEG Schneider Automation, Inc

N.B. : Chaque module TDA08 possède 2 adresses pour permettre au data-logger TMS01 d'enregistrer les huit possibles entrées de la sonde. Sur le TMS01, par exemple, il sera possible configurer deux dispositifs pour chaque module TDA08, le premier enregistrera les entrées IN1..IN4, le deuxième les entrées IN5..IN8.

Le baud rate de chaque module a comme programmation d'usine la valeur de 9600 baud. On peut le modifier par le modbus et la nouvelle programmation deviendra active au prochain cycle d'extinction-allumage du module.

3 Protocole de communication

Le protocole adopté par les TDA est un sous-ensemble du protocole largement utilisé MODBUS RTU. Ce choix garantit la facilité de connexion plusieurs PLC et à tous les programmes de supervision commerciaux.
Pour ceux qui veulent développer leur propre logiciel d'application toutes les suggestions et les informations sont disponibles.

Les fonctions du protocole MODBUS RTU implémentées dans les TDA sont :

fonction 3 - lecture de n mots
fonction 6 - écriture d'un mot

Ces fonctions permettent au programme de supervision de lire et modifier toute donne du module. La communication se base sur des messages envoyés par la station master à une station slave (TDA) et le contraire. La station slave qui reconnaît dans le message sa propre adresse, en analyse le contenu et, si elle le trouve formellement et sémantiquement correct, elle engendre un message de réponse pour le master.

Le procédé de communication implique cinq types de message :

du master au slave	du slave au master
fonction 3 : demande de lecture de n mots	fonction 3 : réponse contenant n mots lus
fonction 6 : demande d'écriture d'un mot	fonction 6 : confirmation de l'écriture d'un mot
	Réponse d'exception (en réponse aux deux fonctions, en cas d'anomalie)

Tout message contient quatre zones :

- adresse du slave : sont valables les valeurs comprises entre 1 et 31 (voir tableau a **2.1**); l'adresse 0 (zéro) est réservée par le MODBUS RTU pour les messages de broadcasting, mais il n'est pas adopté pour le TDA vu le manque de fiabilité de ce type de communication ;
- code fonction : contient 3 ou 6 selon la fonction spécifiée ;
- zone d'informations : contient les adresses ou la valeur des mots, selon la demande de la fonction utilisée ;
- mot de contrôle : contient un cyclic redundancy check (CRC) calculé selon les règles prévues pour le CRC16.

Les caractéristiques de la communication asynchrone sont : 8 bits, aucune parité, un bit d'arrêt.

3.1 Fonction 3 - lecture de n mots

Le nombre de mots à lire, doit être inférieur ou égal à quatre.

La demande a la structure suivante :

numéro du slave	3	adresse premier mot MSB	LSB	nombre de mots MSB	LSB	CRC LSB	MSB
byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7

La réponse normale (au contraire d'une réponse d'exception) a la structure suivante :

numéro du slave	3	nombre de bytes lus	valeur du premier mot MSB	LSB	mots suivants	CRC LSB	MSB
byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte	byte

3.2 Fonction 6 - écriture d'un mot

La demande a la structure suivante :

numéro du slave	6	Adresse premier mot MSB	LSB	Valeur à écrire MSB	LSB	CRC LSB	MSB
byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7

La réponse normale (au contraire de la réponse d'exception) est purement un écho du message de demande :

numéro du slave	6	Adresse premier mot MSB	LSB	Valeur à écrire MSB	LSB	CRC LSB	MSB
byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	Byte 6	byte 7

4 Echange des données

Cette section contient les informations concernant les données numériques et non numériques échangées avec les modules TDA et leurs formats et limites.

4.1 Certaines définitions

Toutes les données échangées sont constituées par un mot de 16 bits.

On distingue deux types de données : numériques et symboliques (ou no numériques). Les données numériques représentent la valeur d'une grandeur (par exemple la variable mesurée, etc…).

Les données symboliques représentent une valeur particulière à l'intérieur d'une gamme de choix (par exemple, Unité de mesure peut valoir "°C" ou "°F").

Les deux types sont codifiés avec des numéros entiers : on adopte des numéros entiers avec signe pour les données numériques et les numéros entiers sans signe pour les symboliques. Une donnée numérique doit être associée avec le numéro approprié de chiffres décimaux, de façon à représenter une grandeur avec les mêmes unités d'ingénierie adoptées dans le module TDA.

Les données numériques sont représentées avec une virgule fixe, et peuvent être entières ou avec un chiffre décimal.

4.2 Zones de mémoire

Pour les fonctions adoptées, toutes les données lisibles et que l'on peut écrire apparaissent comme des mots de 16 bits placés dans la mémoire du module.

Le plan de la mémoire a cinq zones :

Paramètres,
variables,
commandes, alarmes,
code d'identification de l'instrument.

Les paragraphes suivants examinent les caractéristiques de chaque zone.

Un appendice approprié énumère tous les détails de chaque zone, de façon à permettre la connexion à un système de supervision.

4.2.1 Zone des paramètres

Les données de configuration ainsi que les données opérationnelles se trouvent dans la zone des paramètres et sont physiquement dans une mémoire non volatile située à l'intérieur des TDA.

4.2.2 Zone des variables

Dans cette zone, on a regroupé les variables principales du TDA qui sont fréquemment calculées et mises à jour.

On énumère ici les données disponibles :

- v valeur mesurée de la sonde 1,
- v valeur mesurée de la sonde 2,
- v valeur mesurée de la sonde 3,
- v valeur mesurée de la sonde 4,
- v valeur mesurée de la sonde 5,
- v valeur mesurée de la sonde 6,
- v valeur mesurée de la sonde 7,
- v valeur mesurée de la sonde 8,
- v état des entrées digitales,
- v état de la sortie,
- v état des alarmes,
- v état du TDA,

Les conditions d'anomalie des variables de procédé (sonde 1...sonde 8) sont reportées comme des valeurs spéciales de la mesure :

condition d'anomalie	valeur rendue
Underrange ou court-circuit	-10000
Overflow ou sonde ouverte	10000
Variable non disponible	10003

A.2 Appendice - Tableau de la zone des variables

n.	*adresse (hex)*	*nom variable*	*type donnée*	*Étendue de mesure*	*unité*	*chiffres décimaux*	*r/w*
1	0200	Valeur entrée IN1	N	-999 … 9999	(*)	Var 0240	r
2	0201	Valeur entrée IN2	N	-999 … 9999	(*)	Var 0241	r
3	0202	Valeur entrée IN3	N	-999 … 9999	(*)	Var 0242	r
4	0203	Valeur entrée IN4	N	-999 … 9999	(*)	Var 0243	r
(#)5	0204	Valeur entrée IN5	N	-999 … 9999	(* *)	Var 0244	r
(#)6	0205	Valeur entrée IN6	N	-999 … 9999	(* *)	Var 0245	r
(#)7	0206	Valeur entrée IN7	N	-999 … 9999	(* *)	Var 0246	r
(#)8	0207	Valeur entrée IN8	N	-999 … 9999	(* *)	Var 0247	r
9	021F	Lit l'état de la sortie alarm OUT	S	0: OFF 1: ON			r
10	0220	Lit l'état de l'entrée DI01	S	0: ouvert 1: fermé			r
11	0221	Lit l'état de l'entrée DI02	S	0: ouvert 1: fermé			r
12	0222	Lit l'état de l'entrée DI03	S	0: ouvert 1: fermé			r
13	0223	Lit l'état de l'entrée DI04	S	0: ouvert 1: fermé			r
(#)14	0224	Lit l'état de l'entrée DI05 (IN1 quand Endi=YES)	S	0: ouvert 1: fermé			r
(#)15	0225	Lit l'état de l'entrée DI06 (IN2 quand Endi=YES)	S	0: ouvert 1: fermé			r
(#)16	0226	Lit l'état de l'entrée DI07 (IN3 quand Endi=YES)	S	0: ouvert 1: fermé			r
(#)17	0227	Lit l'état de l'entrée DI08 (IN4 quand Endi=YES)	S	0: ouvert 1: fermé			r
(#)18	0228	Lit l'état de l'entrée DI09 (IN5 quand Endi=YES)	S	0: ouvert 1: fermé			r
(#)19	0229	Lit l'état de l'entrée DI10 (IN6 quand Endi=YES)	S	0: ouvert 1: fermé			r
(#)20	022A	Lit l'état de l'entrée DI11 (IN7 quand Endi=YES)	S	0: ouvert 1: fermé			r

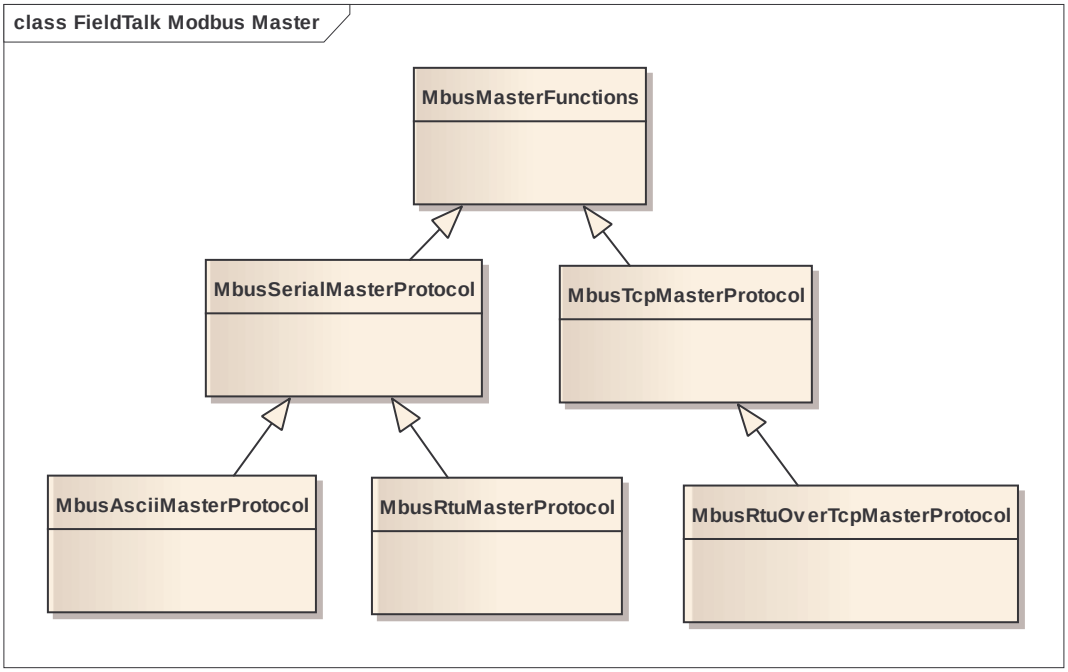
Annexe 8 : FieldTalk Modbus C++ Library

Vous trouverez dans cette annexe des extraits de la documentation FieldTalk.

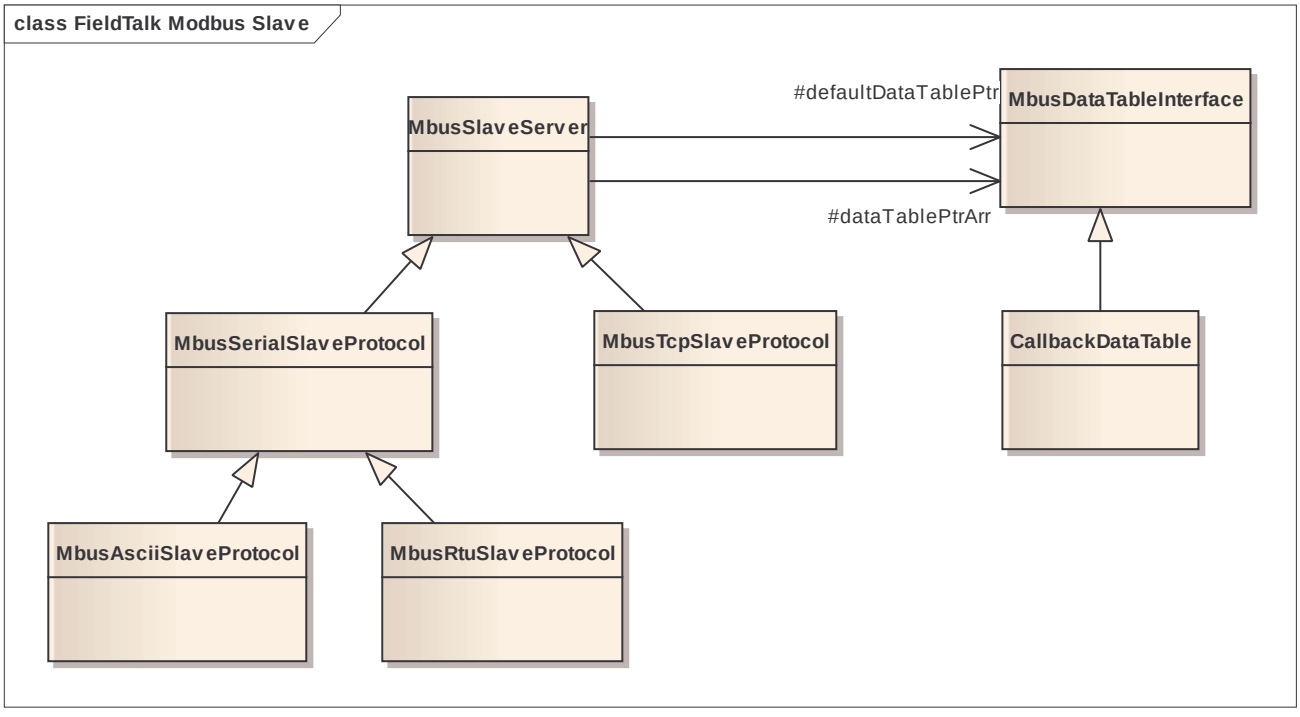
6 Diagramme de classes

Deux catégories de classes sont proposées :

1. The ***FieldTalk***™ Modbus® Master C++ Library provides connectivity to Modbus slave compatible devices and applications.



2. The ***FieldTalk***™ Modbus® Slave C++ Library allows you to incorporate Modbus slave functionality into your device or application.



7 Extrait du fichier ***MbusMasterFunctions.h***

```
/**
 * @file MbusMasterFunctions.h
 *
 * @if NOTICE
 *
 * Copyright (c) 2002-2009 proconX Pty Ltd. All rights reserved.
 *
 * THIS IS PROPRIETARY SOFTWARE AND YOU NEED A LICENSE TO USE OR REDISTRIBUTE.
 *
 * THIS SOFTWARE IS PROVIDED BY PROCONX AND CONTRIBUTORS ``AS IS'' AND ANY
 * EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL PROCONX OR CONTRIBUTORS BE
 * LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
 * ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * @endif
 */

#ifndef _MBUSMASTERFUNCTIONS_H_INCLUDED
#define _MBUSMASTERFUNCTIONS_H_INCLUDED

#ifndef __cplusplus
#  error Must use C++ to compile this module!
#endif

// Platform header
#include <stdlib.h>

// Package header
#include "BusProtocolErrors.h"

/*****************************************************************************
 * Forward declaration
 *****************************************************************************/
class HmTimer;




/*****************************************************************************
 * MbusMasterFunctions class declaration
 *****************************************************************************/
/**
 * @brief Base class which implements Modbus data and control functions
 *
 * The functions provided by this base class apply to all protocol flavours
 * via inheritance. For a more detailed description see section @ref
 * mbusmaster.
 *
 * @see mbusmaster
 * @see MbusSerialMasterProtocol, MbusRtuMasterProtocol
 * @see MbusAsciiMasterProtocol, MbusTcpMasterProtocol
 */
```

```
/*****************************************************************************
 * MbusMasterFunctions class declaration
 *****************************************************************************/

class MbusMasterFunctions
{

  protected:

   enum
   {
      PDU_SIZE = 253, // As per MODBUS APPLICATION PROTOCOL SPECIFICATION V1.1a
      MAX_DATA_SIZE = PDU_SIZE - 1, // = PDU minus 1 byte function code

      MAX_FC03_WORDS = (MAX_DATA_SIZE - 1) / 2, // = maximum words per read
request
      //MAX_FC04_WORDS =  MAX_FC03_WORDS,       // same as function code 03
      MAX_FC01_COILS =  MAX_FC03_WORDS * 16,    // = maximum coils per read
request
      //MAX_FC02_COILS =  MAX_FC01_COILS,       // same as function code 1
      MAX_FC16_WORDS = (MAX_DATA_SIZE - 5) / 2, // = maximum words per write
request
      MAX_FC15_COILS =  MAX_FC16_WORDS * 16,    // = maximum coils per write
request
      MAX_FC23_READ  =  MAX_FC03_WORDS,         // = maximum words for read
      MAX_FC23_WRITE = (MAX_DATA_SIZE - 9) / 2  // = maximum words for write
   };

   volatile unsigned long totalCounter;
   volatile unsigned long successCounter;
   int retryCnt;  // Retry counter
   int timeOut;   // Time-out in ms
   int pollDelay; // Delay between two Modbus read/writes in ms
   HmTimer &silenceTimer;

  private:

   char bufferArr [MAX_DATA_SIZE];
   int bigEndianMachine;      // Auto-configured by constructor
   int bigEndianFloatMachine; // Auto-configured by constructor
   int swapInts;              // User flags
   int swapFloats;            // User flags

   //
   // Slave configuration data
   //
   int slaveConfigFlagsArr[256];


  protected:

    MbusMasterFunctions();


  public:

   virtual ~MbusMasterFunctions();
```

```
  /*************************************************************************
   * Modbus function codes
   *************************************************************************/

  public:

   /**
    * @name 16-bit Access
    *  Table 4:00000 (Holding Registers) and Table 3:00000 (Input Registers)
    * @ingroup mbusmaster
    */
   //@{

   int readInputSingleRegister(int slaveAddr,
                               int regAddr,
                               short& regVal);

   int readInputRegisters(int slaveAddr, int startRef,
                          short regArr[], int refCnt);

   int writeSingleRegister(int slaveAddr, int regAddr, short regVal);

   int writeMultipleRegisters(int slaveAddr,
                              int startRef,
                              const short regArr[], int refCnt);

   int maskWriteRegister(int slaveAddr, int regAddr,
                         short andMask, short orMask);

   int readWriteRegisters(int slaveAddr,
                          int readRef, short readArr[], int readCnt,
                          int writeRef,
                          const short writeArr[], int writeCnt);
   //@}
.
.
.
   /*************************************************************************
    * Slave Configuration
    *************************************************************************/

  public:

   /**
    * @name Slave Configuration
    * @ingroup mbusmaster
    */
   //@{

   void configureBigEndianInts();

   void configureLittleEndianInts();
.
.
.
   //@}
```

```
    /*************************************************************************
     * Utility routines
     ************************************************************************/

  public:

    /**
     * Returns whether the protocol is open or not.
     *
     * @retval true = open
     * @retval false = closed
     */
    virtual int isOpen() = 0;

    /**
     * Closes an open protocol including any associated communication
     * resources (com ports or sockets).
     */
    virtual void closeProtocol() = 0;

    static TCHAR *getPackageVersion();


    /*************************************************************************
     * Internal subroutines
     ************************************************************************/

  protected:

    virtual int deliverMessage(int address, int function,
                               char sendDataArr[], int sendDataLen,
                               char rcvDataArr[], int rcvDataLen,
                               int *actualRcvdPtr = NULL) = 0;

  private:

    int readBits(int function, int slaveAddr, int startRef,
                 int bitArr[], int refCnt);

    int readRegisters(int function, int slaveAddr, int startRef,
                      short regArr[], int refArrLen, int regCnt);

    int writeRegisters(int slaveAddr, int startRef,
                       const short regArr[], int refCnt, int regCount);

  private:

    // Disable default operator and copy constructor
    MbusMasterFunctions &operator= (MbusMasterFunctions &);
    MbusMasterFunctions (const MbusMasterFunctions &);

};


#endif // ifdef ..._H_INCLUDED
```

Annexe 9 : Extrait STL Vector

Constructor/Declaration:

Method/operator	Description
vector<T> v;	Vector declaration of data type "T".
vector<T> v(size_type n);	Declaration of vector containing type "T" and of size "n" (quantity).
vector<T> v(size_type n,const T& t);	Declaration of vector containing type "T", of size "n" (quantity) containing value "t". Declaration: `vector(size_type n, const T& t)`
vector<T> v(begin_iterator,end_iterator);	Copy of Vector of data type "T" and range begin_iterator to end_iterator. Declaration: `template vector(InputIterator, InputIterator)`

Size methods/operators:

Method/operator	Description
empty()	Returns bool (true/false). True if empty. Declaration: `bool empty() const`
size()	Number of elements of vector. Declaration: `size_type size() const`
resize(n, t=T())	Adjust by adding or deleting elements of vector so that its size is "n". Declaration: `void resize(n, t = T())`
capacity()	Max number of elements of vector before reallocation. Declaration: `size_type capacity() const`
reserve(size_t n)	Max number of elements of vector set to "n" before reallocation. Declaration: `void reserve(size_t)`
max_size()	Max number of elements of vector possible. Declaration: `size_type max_size() const`

Note: `size_type` is an unsigned integer.

Other methods/operators:

Method/operator	Description
erase() clear()	Erase all elements of vector. Declaration: `void clear()`
at(index) v[index]	Element of vector. Left and Right value assignment: v.at(i)=e; and e=v.at(i); Declaration: `reference operator[](size_type n)`

front() v[0]	First element of vector. (Left and Right value assignment.) Declaration: `reference front()`
back()	Last element of vector. (Left and Right value assignment.) Declaration: `reference back()`
push_back(const T& value)	Add element to end of vector. Declaration: `void push_back(const T&)`
pop_back()	Remove element from end of vector. Declaration: `void pop_back()`
assign(size_type n,const T& t)	Assign first n elements a value "t".
assign(begin_iterator,end_iterator)	Replace data in range defined by iterators. Declaration:
insert(iterator, const T& t)	Insert at element "iterator", element of value "t". Declaration: `iterator insert(iterator pos, const T& x)`
insert(iterator pos, size_type n, const T& x)	Starting before element "pos", insert first n elements of value "x". Declaration: `void insert(iterator pos, size_type n, const T& x)`
insert(iterator pos, begin_iterator,end_iterator)	Starting before element "pos", insert range begin_iterator to end_iterator. Declaration: `void insert(iterator pos, InputIterator f, InputIterator l)`
swap(vector& v2)	Swap contents of two vectors. Declaration: `void swap(vector&)`

Iterator methods/operators:

Method/operator	Description
begin()	Return iterator to first element of vector. Declaration: `const_iterator begin() const`
end()	Return iterator to end of vector (not last element of vector but past last element) Declaration: `const_iterator end() const`
rbegin()	Return iterator to first element of vector (reverse order). Declaration: `const_reverse_iterator rbegin() const`
rend()	Return iterator to end of vector (not last element but past last element) (reverse order). Declaration: `const_reverse_iterator rend() const`
++	Increment iterator.
--	Decrement iterator.