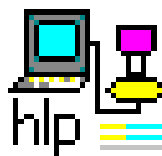


WorldFIP Tools library

FipEngine

version 1.0 for LINUX

User manual



Author

I.Lanteri

Controller



35, Rue Tournefort - 75005 Paris - France

Document History

Index	Date	Page	Description
A	09/16/1998	All	Creation of the document
B	02/16/2000	All	Updated for version 1.4
C	07/12/2000	7 to 8	Modification of the setup program
D	08/30/2005	All	Updated for version 2.1
E	01/21/2010	All	Linux adaptation version 1.0

CONTENTS

1. WORLDVIP TOOLS LIBRARY FIPENGINE OVERVIEW.....	6
1.1. INTRODUCTION.....	6
1.2. SOFTWARE AND MATERIAL REQUIRED.....	8
1.3. FIPENGINE COMPONENTS.....	8
2. GENERAL INFORMATION FOR USING FIPENGINE FUNCTIONS.....	9
2.1. FOR OLDER WINDOWS VERSION USERS.....	9
2.2. SEQUENTIAL AND MULTI-THREADED MODES.....	9
2.3. START AND STOP LIBRARY.....	11
2.4. VARIABLES.....	12
2.4.1. PERIODICAL VARIABLES.....	12
2.4.2. APERIODICAL VARIABLES.....	12
2.4.2.1. <i>How to transmit an aperiodical variable.....</i>	<i>12</i>
2.4.2.2. <i>How to receive an aperiodical variable.....</i>	<i>12</i>
2.5. MESSAGES.....	13
2.5.1. HOW TO TRANSMIT A MESSAGE.....	13
2.5.2. HOW TO RECEIVE MESSAGE.....	14
2.6. HANDLING EVENTS.....	14
2.6.1. PREDEFINED EVENTS.....	15
2.6.2. EVENT DEFINITION.....	16
2.6.3. EVENT READ.....	16
2.7. HANDLING THE BUS ARBITRATOR.....	16
2.7.1. BUS ARBITRATOR STATES.....	17
2.7.2. PROGRAMMING THE BUS ARBITRATOR.....	19
2.7.3. SIZE OF BUS ARBITRATOR INSTRUCTIONS IN BOARD MEMORY.....	19
2.7.4. EXAMPLE OF PROGRAMMING THE BUS ARBITRATOR.....	20

2.7.5. FORMAT OF THE BUS ARBITRATOR PROGRAMMING FILES.....	21
2.8. HANDLING THE COMMUNICATION MEDIUMS.....	21
2.8.1. STATUS BYTE FORMAT.....	21
2.8.2. AVAILABLE COMMANDS.....	22
2.8.3. COMMUNICATION MANAGEMENT.....	22
2.8.3.1. Principle.....	22
2.9. DEFINITIONS, ERROR CODES.....	23
3. FIPENGINE FUNCTION COMPLETE DESCRIPTION.....	23
3.1. START/STOP CONFIGURATION.....	24
3.1.1. START_FIP_ENGINE.....	24
3.1.2. STOP_FIP_ENGINE.....	24
3.1.3. DOWNLOAD_CNF.....	25
3.1.4. START_COMMUNICATION.....	25
3.2. VARIABLES.....	26
3.2.1. FIP_READ_VAR.....	26
3.2.2. FIP_WRITE_VAR.....	27
3.2.3. SEND_APER.....	27
3.3. MESSAGES.....	28
3.3.1. FIP_READ_MSG.....	28
3.3.2. FIP_WRITE_MSG.....	29
3.4. EVENTS.....	31
3.4.1. IS_EVENT.....	31
3.4.2. READ_EVENT.....	31
3.4.3. DEF_VAR_EVENT.....	34
3.5. BUS ARBITRATOR.....	34
3.5.1. BA_PROG.....	34
3.5.2. START_BA.....	36
3.5.3. STOP_BA.....	36
3.5.4. CONTINUE_BA.....	36
3.5.5. BA_REPORT.....	37

3.5.6. CHANGE_BA.....	37
3.6. TOOLS FUNCTIONS.....	39
3.6.1. PURGE.....	39
3.6.2. FIND_MSG_BY_ID.....	39
3.6.3. FIND_MSG_BY_NAME.....	41
3.6.4. FIND_VAR_BY_ID.....	41
3.6.5. FIND_VAR_BY_NAME.....	43
3.6.6. GET_BA_PROGRAM_ADDR.....	45
3.6.7. GET_TRANSMISSION_QUEUE.....	45
3.6.8. MEDIUM_CONTROL.....	46
3.6.9. ERROR_MESSAGE.....	48
4. APPENDIX.....	49
4.1. DETAILED FORMAT OF "CNF" FILES FROM FIPDESIGNER VERSION 3.....	49
4.2. THE MULTI THREAD POSIX LIBRARY.....	53
4.2.1. CREATION/DESTRUCTION.....	54
4.2.2. UTILITIES.....	55
4.2.3. THREAD SPECIFIC DATA.....	56
4.2.4. ATTRIBUTES.....	57
4.2.5. CONDITION VARIABLES.....	60
4.2.6. SEMAPHORES.....	62
4.2.7. SPINLOCKS.....	63
4.2.8. MUTEX.....	64
4.2.9. READ-WRITE LOCKS.....	68
4.2.10. BARRIERS.....	71
4.3. THE PROGRAMS OF EXAMPLE.....	72

1. WORLDVIP TOOLS LIBRARY FIPENGINE OVERVIEW

1.1. INTRODUCTION

The WorldVIP tools library FipEngine from HLP Technologies consists of a set of functions designed to help you to write a WorldVIP application based upon a PC/WorldVIP adapter on PCI bus (card named "PCI/FIP"). This library can be used with Windows2000, XP. And now with Linux This document is about the adapting of this library to Linux.

Functionalities at the FIP network level are the same under Linux and Windows but the implementation is different and the Linux library can be used with two different modes:

- Sequential mode: for serial applications: one instruction follows the previous.
- Multi-threaded mode: threads are running simultaneously.

This version supports the following functionalities:

- Configures your PC/WorldVIP adapter:
 - WorldVIP objects (variables and messages);
 - General communication parameters. This software is capable of working in particular at the standard transmission rate of 31,25 kb/s, 1 Mb/s or 2,5 Mb/s.
- Enables you to program a bus arbitrator.
- Reads/writes periodical or aperiodical WorldVIP variables.
- Transmits/receives periodical or aperiodical WorldVIP messages.
- Handles events from the network.
- Operates dynamically the bus arbitrator.
- Handles communication medium (state and redundancy).

What's new with regard to the previous version:

- Configuration of the PC/WorldVIP adapter.

This library enables WorldVIP objects (variables, messages, events) to be configured in the adapter's memory. These objects will be handled by your application when the adapter is in the running state. The configuration is downloaded from a specific file with "cnf" extension. This file can be created in different ways:

- The user makes his own configuration file. The "cnf" configuration file format is given in Appendix.
- The FipDesigner version 3 makes a "cnf" file under Windows 3.11 or 95.
- The FipDesigner version 4 makes a "cnf" file under Windows 95/98 or NT4/2000.
- The FipDesigner version 4.21 and later under Windows 2000/XP.

The library supports the use of any FipDesigner's version "cnf" file. In the previous version of the library, you should use another software to download the configuration in the adapter's memory.

This configuration is also called "*local configuration*" because it is **local** to the used PC, by opposition to the distant configurations of the other WorldVIP stations on the network.

- Search by name or by identifier.

The user handles variables or messages by their name or identifier indifferently, once the configuration is downloaded in the PC/WorldFIP adapter. Some tool functions are responsible for finding an object by its name or identifier in the downloaded database.

- Bus arbitrator configuration and control.

The user can describe the bus arbitrator program using an ASCII file, which format is given in 2.7.5 chapter, or with a “cnf” configuration file. In this second case, two bus arbitrator programs can be described, and you can toggle from one to the other during communication.

This program is downloaded and you could operate it with some library function.

- Communication medium management.

The user can know the physical state of the communication at anytime. The library indicates if there are reception or transmission errors, if the communication channels are enable and the user can toggle from one medium to the other.

The use of this manual implies that the user has already a good working knowledge of WorldFIP standards, WorldFIP frame sequences and WorldFIP time notions (promptness, refreshment, silence time, turn-around time).

1.2. SOFTWARE AND MATERIAL REQUIRED

Software:

- Linux Red Hat .Enterprise 5.3 (preferably its derivative Scientific Linux 5.3 from CERN and FermiLab)
- Kernel Version 2.6.8.128.1.1.el5 (32 bits).
- PCIFIP driver for Linux from HLP Technologies (version 1,0)

Hardware

- **PCI/FIP** adapter card from **HLP Technologies** (versions 2.3 and further).

1.3. FIPENGINE COMPONENTS

The FipEngine library is delivered and installed along with the driver. For more information, refer to the WorldFIP Tools installation guide under Linux.

The libFipEngine.so file is installed into the /usr/lib directory. The following files, located into the install directory, are part of the FipEngine library.

- FipEngin.h: Header file for the FipEngine API functions.
- Sample programs with Makefiles and sources demonstrating the use of the library.
- Documentation files like this file.

2. GENERAL INFORMATION FOR USING FIPENGINE FUNCTIONS

2.1. FOR OLDER WINDOWS VERSION USERS

The functionalities offered by FipEngine are more practical than those supported by the previous versions.

The following tips will make the use of the library easier:

- This version allows you to download the PC/WorldFIP adapter memory with the WorldFIP configuration. The download functionality uses files as source information. The file formats are given in this document. There are two types of configuration files to download:
 - Configuration file with "cnf" extension: it contains the general variable and message WorldFIP configuration of your application. The format is given in paragraph 4, Appendix. At times, we may use the word "database" instead of "configuration".
 - Bus arbitrator program ASCII file: it contains the instructions sequence of the bus arbitrator program. The format is given in paragraph 2.7.5.

The use of configuration files allows you to edit huge configurations more easily. The FipEngine library could handle 4095 variables and 4096 messages, with the 128 kword memory PC/WorldFIP adapter.

- Each variable and message in this version has a name and an identifier. All the functions handling this kind of objects need an "access key" in parameter. You no longer need to calculate the access key of the WorldFIP object, since this version offers some utility functions that find the corresponding access key. One search per object is enough in your application since the access key is not changed when there is no change in the WorldFIP configuration.
- Now, there is only one function for transmitting a message and another for receiving a message. Previously, you needed two functions for each task, one for the message header, one for the message value.

The following paragraphs however describe some essential information for using the library. We advise you to read this part before beginning your application.

2.2. SEQUENTIAL AND MULTI-THREADED MODES

If the user wants to work in sequential mode with one or several cards, he must use the set of FipEngine functions taking the card number in parameter. This card number depends on the logical order in which the PCI bus detects a PCIFIP card and belongs to the range 1-4. One (for the first plugged card) to four (the driver can support up to 4 cards together).

Here is a sample of a sequential program for 2 cards:

```
DWORD result;  
USHORT card1 = 1;
```

```
USHORT card2=2;
//Library Initialisation : mandatory
result = START_FIPENGINE(card1, FALSE);
if(result!= SUCCESS)
{
    printf(ERROR_MESSAGE(result);
    .....
    //Error processing;
}
result = START_FIPENGINE(card2, FALSE);

// Loading configuration from .cnf files..
result = DOWNLOAD_CNF(card1, " ./Station1.cnf ");
.....
result = DOWNLOAD_CNF(card2, " ./Station2.cnf ");

// Network connection.....
result = START_COMMUNICATION(card1);

result = START_COMMUNICATION(card2);

//Disconnecting, freeing ressources and let the system clean,,,,,
result = STOP_FIPENGINE(card1);
.....
result = STOP_FIPENGINE(card2);
```

In a multi-threaded program each card has its own program running in parallel with the program of the others cards (or threads).
Your program must use the **Pthread** library compatible **POSIX** (Portable Operating System for Computer Environment).
Here is a sample program for a function which could be a canvas for the thread of a card of number I given as *arg parameter.

```
#include "FipEngin.h"
//Thread function
void* FipEngine_func( void* arg)
{
    DWORD result;
    USHORT card =*((USHORT)arg);
    UCHAR status;

    //Initialising library with card number and mode
    result = START_FIPENGINE(card, TRUE);
    if(result!= SUCCESS)
    {
        printf(ERROR_MESSAGE(result);

        //Error processing
    }
    char station[15];
    sprintf(" ./Station%i.cnf, card);
    //Downloading the configuration from a .cnf file
    result = DOWNLOAD_CNF_PTH(station);

    ....// Connecting the network
    result = START_COMMUNICATION_PTH();
```

```
//Disconnecting, freeing ressources and let the system clean  
result = STOP_FIPENGINE_PTH();  
}
```

It's to remark that START_FIPENGINE is the only function with the same prototype in the two modes sequential or multi-threading. The first argument specifies the card number and the second is a boolean. If TRUE, it means the mode is multi-threading and if FALSE that the mode is SEQUENTIAL.

All the FipEngine functions for the multi-thread mode are prefixed by _PTH except START_FIPENGINE.

If you work with two cards you must create the two threads in the main process in the following way:

```
#include <pthread.h>
```

```
.....
```

```
DWORD result;
```

```
pthread_t thread_ID1, thread_ID2;
```

```
USHORT card1=1;
```

```
USHORT card2 = 2;
```

```
result= pthread_create(&thread_ID1, NULL, FipEngine_func,(void*)&card1);
```

```
result= pthread_create(&thread_ID2, NULL, FipEngine_func,(void*)&card2);
```

If the create call succeeds, each thread begins then its execution.

Multi threaded programming is always tricky and you will find a resume of the POSIX pthreads library functions in appendix.

2.3. START AND STOP LIBRARY

Two functions are needed to allocate and release objects used by the library. The start function START_FIP_ENGINE() must be called before all other library function. When the start function has been called, you need call the stop function STOP_FIP_ENGINE() at the end of your application to release the allocated resource.

After a successful start library function, you must configure your PC/WorldFIP board with the WorldFIP parameters that your application will use. The DOWNLOAD_CNF() function has been designed for this task. It downloads the configuration needed by the PC/WorldFIP board to be a communicating WorldFIP device. This configuration was read from the file whose name is given by the function parameter. The downloaded configuration contains the general WorldFIP parameters, the variable parameters and the message parameters. In the following paragraphs, the downloaded WorldFIP parameters will be indifferently called "configuration", "local configuration" or "database", "local database".

The board is now configured but not started. The user calls the START_COMMUNICATION() function to have a real communicating WorldFIP device.

2.4. VARIABLES

Whatever type the variable is, periodical or not, the mechanism is the same to read or write a variable. When the PC/WorldFIP adapter is configured and running, it uses a memory buffer (call local buffer) for each variable in which it stores the information to be produced on or to be taken up from the bus. The library read and write variable functions access to this local buffer.

The limited duration status, promptness and refreshment, are calculated since a new value has been written to the local buffer. These are status's which inform the consumers about the validity of the variable received. The asynchronous refreshment status is generated by the producer of a variable. It is defined by a maximum refreshment duration T_r . The status is transported on the bus after the data in the RP_DAT_XX frame, so that the consumer is informed of the status of the producer.

The asynchronous promptness status is generated by the consumer of the variable. It is a local qualifier and is defined by the maximum promptness duration T_p .

2.4.1. PERIODICAL VARIABLES

For read and write variables, you only need to call FIP_READ_VAR() and FIP_WRITE_VAR() with the accurate access key. This is given by the utility functions, FIND_VAR_BY_NAME() or FIND_VAR_BY_ID().

Note: the access key does not change if the WorldFIP variable configuration does not change. One call per variable to these utility functions is therefore sufficient for your entire application.

2.4.2. APERIODICAL VARIABLES

2.4.2.1. *How to transmit an aperiodical variable*

Aperiodical variable transmission is carried out in two steps:

- First, the variable value must be written to its local buffer with the FIP_WRITE_VAR() function.
- Second, you must perform an aperiodical variable transmission request. This will be done with the SEND_APER() function. The request will be transmitted to the bus arbitrator, that will operate the variable transmission in his first free aperiodical window.

⇒ Be careful, when using this kind of transmission. You must have a produced variable in your WorldFIP configuration, which is aperiodical variable transmission request authorised. This variable will transmit the transmission request to the bus arbitrator.

⇒ When the request is transmitted onto the bus, an event is generated, and you must read it with the READ_EVENT() function for freeing the event queue.

2.4.2.2. *How to receive an aperiodical variable*

The user may read the local buffer value at anytime with the FIP_READ_VAR() function. In aperiodical receiving, the difficulty is knowing at

which moment the local buffer has been refreshed by the new value. For asynchronous receiving, the user must define a variable receiving event, so that the user can be informed that a new value is available in the local buffer.

The `DEF_VAR_EVENT()` function allows the user to define a variable receiving event. On receiving, an event is pushed onto the event queue. The `IS_EVENT()` function specifies whether an event exists or not. If there is at least one event in the event queue, the user must pop it with `READ_EVENT()`, and verify the event report to find a receiving event. The event number must be the same as those specified with `DEF_VAR_EVENT()`.

The `DEF_VAR_EVENT()` function does not care if the variable is periodical or not. Therefore, if the identifier's variable is enclosed in the bus arbitrator periodical window, the user will receive periodical events, with possibly aperiodical events added.

If you would only like an asynchronous event, be sure not to take a variable identifier enclosed in a periodical window. (see 2.6 HANDLING EVENTS for more information on events).

2.5. MESSAGES

A message is a data field composed of up to 250 bytes¹. To ensure communication with messaging traffic, you must add some information to a message header. This header includes the following fields:

- Destination address;
- Source address.

The message response frame contains this header and the data field. For WorldFIP, an address is defined with three bytes:

- Two first bytes form an identifier. The source or destination device must have a variable configured with this identifier value in its database.
- The third byte is the segment number of the source or destination device. The WorldFIP network is composed of several segments with 32 devices max. each. If the user has only one WorldFIP segment, the segment number is 0.

2.5.1. HOW TO TRANSMIT A MESSAGE

You only need one function for message transmission, `FIP_WRITE_MSG()`. Several arguments are required for this function:

- The transmission type: it may be acknowledged or not. A non-acknowledged message may be broadcast to several destination devices. These devices must have a variable in their database, authorised to receive messages, with the same identifier as the message destination address field. This broadcast option is strictly forbidden for acknowledged messaging services.
- The channel number: this is the transmission queue number. The messaging service is totally asynchronous so must be managed by a queue. The library offers 9 transmission queues and one receiving queue (this is the event queue), for messaging services. For transmission, you may choose a channel number value from 0 to 8.

¹ 250 bytes max in versions 1.X,
this length will be raised up to 256 bytes in future versions.

- Queue n°0 is only used for the aperiodical messaging service. In this case, the user must have a **produced** variable with **aperiodical message transmission authorised** and the **same channel number** (i.e. 0) in its database. The queue will be created on this variable at the device start up and the transmission request will be send to the bus arbitrator on the variable response frame. (Note: this variable is not necessarily the message source identifier).
- Queues n°1 to 8 are used for periodical messaging service. If one number is chosen, you must have a variable (produced or not) with **periodical message transmission authorised** and the **same queue number**, in the device database. When you choose a number and a variable for the queue, they cannot be reused for the other queues. The queue will be created on the variable. Contrary to the aperiodical case, there is no request to transmit to the bus arbitrator, since the ID_MSG frame is already included in the periodic window.

Note: the channel number will be named "channel number" or "transmission queue number" indifferently in the following paragraphs.

- The event number: an event is generated when the message is sent onto the network. You may choose a number from 1 up to 251. 0 means "no-event".
- The message length: a variable has a fixed length. For a message, the length may grow from 1 to 250 bytes. It becomes a parameter for transmission.

2.5.2. HOW TO RECEIVE MESSAGE

If the message destination identifier is the same as a variable identifier configured with receiving message authorised, an event is generated and is pushed onto the event queue when the message has been received.

You must use the IS_EVENT() function (usually in a loop) to see whether or not an event is present.

When there is at least one event, you must use the READ_EVENT() function to pop up and read the event. The event queue contains up to 32 events, after the events are lost. With READ_EVENT(), you can check the event report to find a receiving message event. When it has been found, you have an access to other information needed for reading the message, particularly the access key of the received message.

When the access key is known, you need only one function to read the message - FIP_READ_MSG().

2.6. HANDLING EVENTS

The events allow access to certain information regarding the running state of the communication. Usually, the event can be activated upon frame transmission or receiving.

Events that the user may meet are the following:

- Transmit a variable or message content.
- Receive a variable or message content.

- Send an aperiodical transmission variable request (urgently or not) to the bus arbitrator.
- Interrupt the bus arbitrator program with the SUSPEND() instruction.

The event are enumerated with a number from 0 up to 255. Among these values, some number are reserved, and the rest are user-defined. A user-defined number can have a value from 1 to 251.

2.6.1. PREDEFINED EVENTS

Some events are predefined with the corresponding number reserved:

- 0, this number means no-event.
- 252, this number means that the user has sent an **aperiodical variable transmission non urgent request** to the bus arbitrator.
- 253, this number means that the user has sent an **aperiodical variable transmission urgent request** to the bus arbitrator.
- 254, this number means **receiving a message**.
- 255, reserved, without meaning.

These predefined events are automatically generated when they occur. Other events are also automatically generated when they occur, but the event number is user-defined:

- **message transmission**, the event number is given when the user calls the FIP_WRITE_MSG() function.
- **Bus arbitrator program interruption**, the event number is given when you use the SUSPEND instruction.

2.6.2. EVENT DEFINITION

There are two kinds of events not automatically generated:

- Variable transmission,
- Variable receiving.

If the user wishes to generate these kind of events, he must define the corresponding event with the DEF_VAR_EVENT() function at the beginning of his program. The associated number is user-defined.

2.6.3. EVENT READ

When an event occurs, it is pushed onto the event queue. The event queue contains up to 32 events, after the events are lost. The queue is FIFO type. The user can find out if an event has been pushed with the IS_EVENT() function. The user must read all the events with the READ_EVENT() function to pop them. This is usually done with an independent loop containing these two functions.

READ_EVENT() function offers a complete event report in return. (See paragraph 3.4.2)

2.7. HANDLING THE BUS ARBITRATOR

This version of the library allows you to describe the bus arbitrator program:

- with a ASCII file "ba". The BA_PROG() function download this file onto the PC/WorldFIP board memory.
- with the two programs contained in the configuration file "cnf". They are downloaded with the DOWNLOAD_CNF() function.

Programming the bus arbitrator involves describing the sequence of elementary transactions in the macrocycle. During a running state, the bus arbitrator loops infinitely on this macrocycle. The "instructions" allow you to describe the elementary transactions in the macrocycle. The "commands" control the macrocycle. The library offers four commands for your bus arbitrator:

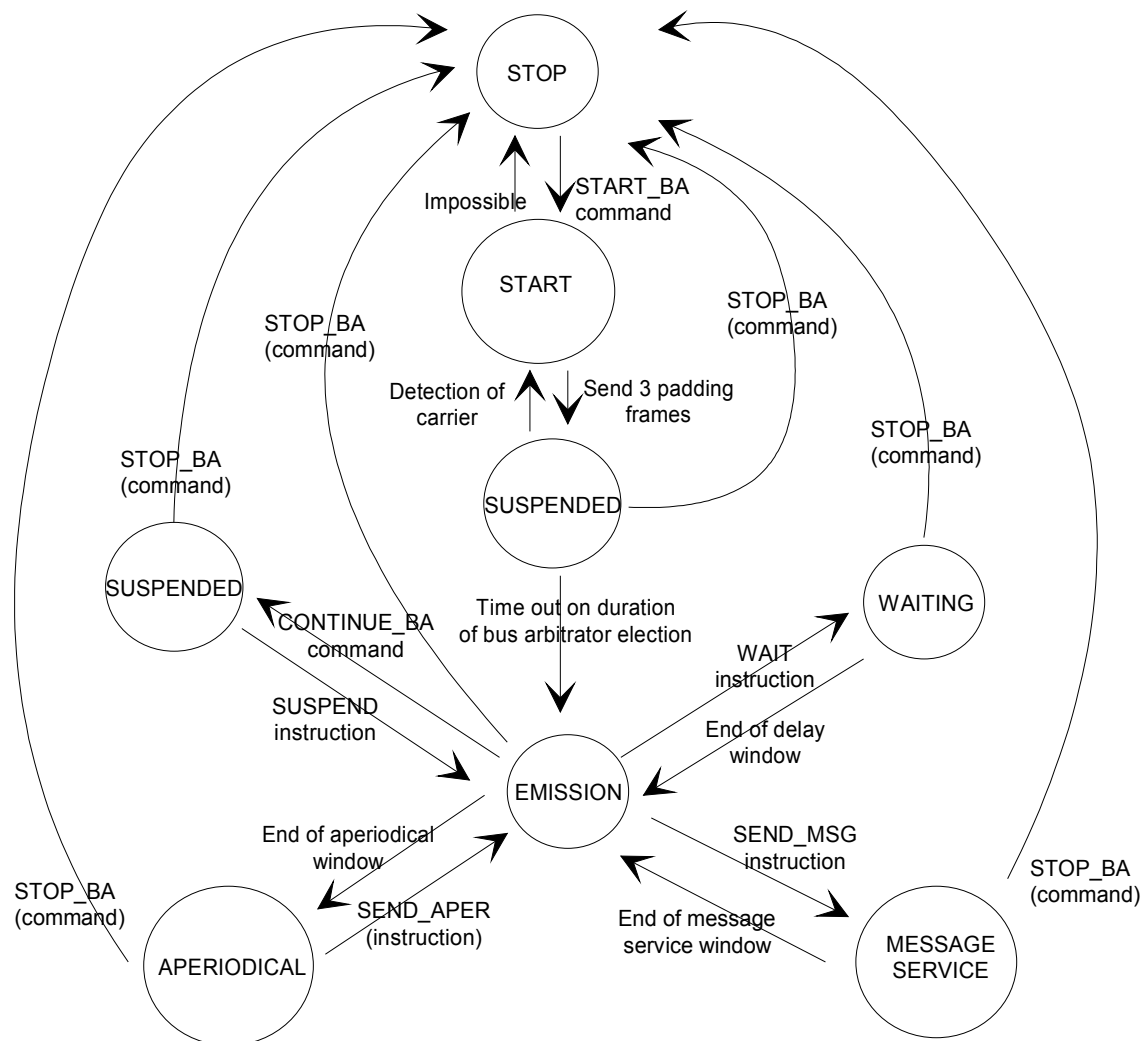
- Start up the bus arbitrator, START_BA(). If there is already a bus arbitrator on the network, the local bus arbitrator goes to the IDLE state (bus arbitrator redundancy).
- Stop the bus arbitrator, STOP_BA(). The local bus arbitrator stops all activity. Nothing is sent on the network.
- CONTINUE_BA() function, allows a bus arbitrator to go from a suspend state to a running state.
- CHANGE_BA() function, allows to change the bus arbitrator program when the current macrocycle is completed.

2.7.1. BUS ARBITRATOR STATES

The bus arbitrator is not only running or stopped, there are in fact eight different states:

- **STOPPED:** the bus arbitrator is not yet started, nothing is sent on the network;
- **STARTING:** the bus arbitrator is starting up, after a calling the `START_BA()` function. At this time, the local bus arbitrator search if another bus arbitrator broadcast on the network.
- **IDLE:** the local bus arbitrator has detected another bus arbitrator already broadcasting. Nothing is sent on the network by the local bus arbitrator, it will automatically switch to the **SENDING** state when the other bus arbitrator stops broadcasting and if the bus arbitrator election procedure is successful.
- **PENDING:** the bus arbitrator is suspend after a `SUSPEND` instruction. In this state, idle frames are transmitted. With the `CONTINUE_BA()` function, the remainder of the program is resumed.
- **WAITING:** the bus arbitrator waits for the specified time elapse. This duration is specified in the `WAIT` instruction. In this state, idle frames are transmitted.
- **SENDING:** the bus arbitrator broadcasts `ID_DAT` and `ID_MSG` frames in its periodic window.
- **APER_WND:** the bus arbitrator broadcasts the aperiodical variable traffic until the specified time has elapsed. This duration is specified in the `SEND_APER` instruction. If the specified time is too short, the remaining traffic will be processed in the next aperiodical variable window. If there are no more aperiodical variables left to broadcast, idle frames are transmitted until the specified time.
- **MSG_WND:** the bus arbitrator broadcasts the aperiodical message traffic until the specified time has elapsed. The duration is specified in the `SEND_MSG` instruction. If the specified time is too short, the remaining traffic will be processed in the next aperiodical message window. If there are no more aperiodical messages left to broadcast, idle frames are transmitted until the specified time.

This document may not be used, copied or redistributed without authorisation of HLP Technologies



These states are closely dependent on the bus arbitrator program instructions. A `BA_REPORT()` function, allows you to access the current state of the local bus arbitrator. This function is important in the management of your bus arbitrator, since you must respect the state machine shown in Figure 1. Thus, if the bus arbitrator is STOPPED, you cannot apply the STOP_BA command; if it is in a state other than STOPPED or STARTING, the START_BA command is possible.

2.7.2. PROGRAMMING THE BUS ARBITRATOR

FipEngine supports a set of instructions which allows the user to program the bus arbitrator cycles.

The instructions are as follows:

- ID_DAT(identifier): Sends an ID_DAT frame of the identifier given as a parameter on the bus.
- ID_MSG(identifier): Sends an ID_MSG frame of the identifier given as a parameter on the bus.
- SEND_MSG(date): Processes the aperiodical messages requests up until the date limit given as a parameter. If all the requests are processed before the date limit, SEND_MSG ends and the program jumps to the next instruction.
- SEND_APER(date): Processes the aperiodical transmission requests up until the date limit given as a parameter. If all the requests are processed before the date limit, SEND_APER ends and the program jumps to the next instruction.
- SUSPEND(event number): This is a special instruction which places the bus arbitrator in a "suspended" state when it is encountered in a program. When the bus arbitrator encounters this instruction it sends back the event number given as a parameter to the event queue. This instruction is designed to synchronise the bus arbitrator on an external clock in order to authorise very long cycles. The bus arbitrator program can be restarted with the help of the CONTINUE_BA command. It is possible to use SUSPEND as a stopping point in the bus arbitrator sequence.
- WAIT(date): Sends idle frames on to the bus up until the date given as a parameter.

Warnings!

- The function parameters are expressed in hexadecimal.
- The date limit is expressed in "slot time" and in hexadecimal. The maximum acceptable value is 0x0fff. Any value higher than this is simply masked using an AND operation with the constant 0x0fff. For more information about slot time value, the user can refer to STATION structure, field "tslot" in the Appendix. If the configuration file (cnf file) originates from FipDesigner software, the user can use this software to find out the slot time value for his configuration.
- All timers are reset at the beginning of each macrocycle.

2.7.3. SIZE OF BUS ARBITRATOR INSTRUCTIONS IN BOARD MEMORY

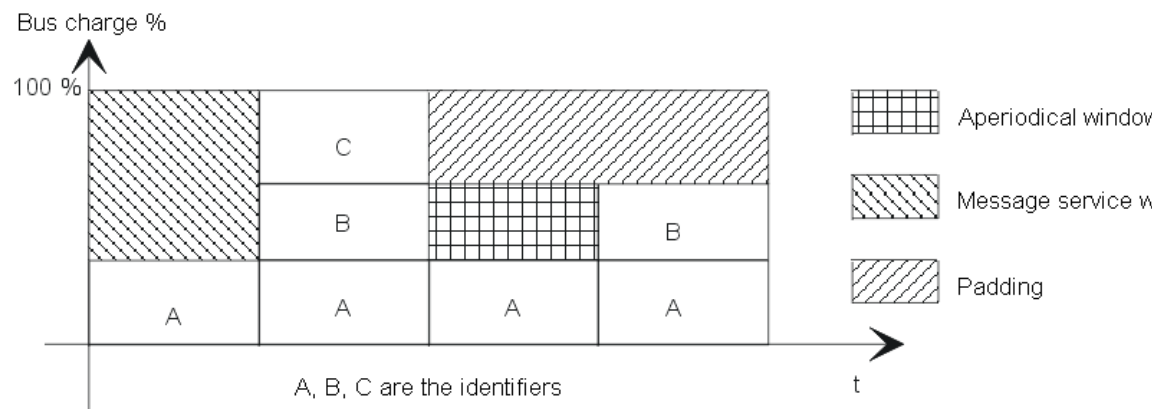
The BA_PROG() function may not have enough memory space for the downloaded program. The memory size reserved for the bus arbitrator is adjustable. If your configuration originates from FipDesigner 3 software, you may adjust this parameter in the "general" menu. Otherwise, the user could increase the "ba" value inside the STATION structure. (see Appendix for STATION structure). It is therefore useful to know the amount of space taken up in the memory by each instruction.

INSTRUCTIONS	Number of words(16 bits)
ID_DAT ²	3+2
ID_MSG	3
SEND_MSG	1
SEND_APER	1
NEXT_MACRO2	1
WAIT	1
SUSPEND	1

Each bus arbitrator sequence contains a minimum of one NEXT_MACRO2 instruction. This instruction must be added at the end of each macrocycle. This is done automatically when programming the bus arbitrator using "cnf" files and must be done manually when using an ASCII file.

2.7.4. EXAMPLE OF PROGRAMMING THE BUS ARBITRATOR

Program the following macro cycle:



This cycle will be programmed by the following bus arbitrator sequence:

```

ID_DAT(A);
SEND_MSG(3*T/12);
WAIT(3*T/12);
ID_DAT(A);
ID_DAT(B);
ID_DAT(C);
ID_DAT(A);
SEND_APER(8*T/12);
WAIT(9*T/12);
ID_DAT(A);
ID_DAT(B);
WAIT(T);

```

T is the length of the macro cycle. The time is expressed in multiples of the time slot. This is the shortest programmable duration.

A periodical message is managed in the same way as the transmission of a periodical identifier (for example ID_DAT(A)). The instruction ID_MSG (A) is therefore used. A is the identifier of the variable being used as an emission support for the message concerned (source identifier).

² The first instruction ID_DATXX encountered for a given identifier, takes up five words in the memory. Those which follow take up 3 words.

2.7.5. FORMAT OF THE BUS ARBITRATOR PROGRAMMING FILES

The bus arbitrator programming files are simple text files. Their format is as follows:

Free comment zone

```
BEGIN
ID_DAT(XXXX)
ID_DAT(YYYY)
.
.
.
SEND_MSG(XXXX)
NEXT_MACRO_2()
END
```

BA_PROG() function interprets the files between the keywords **BEGIN** and **END**.

The authorised instructions are those in paragraph 2.7.2. The supplementary instruction **NEXT_MACRO_2()** must be added at the end before the keyword **END**.

It informs BA_PROG() of the end of the macrocycle.

2.8. HANDLING THE COMMUNICATION MEDIUMS

2.8.1. STATUS BYTE FORMAT

In spite of its robustness, the WorldFIP protocol and the connected material could have communication errors at the physical level. The library can know the state of the communication thanks a status byte. Its format is the following:

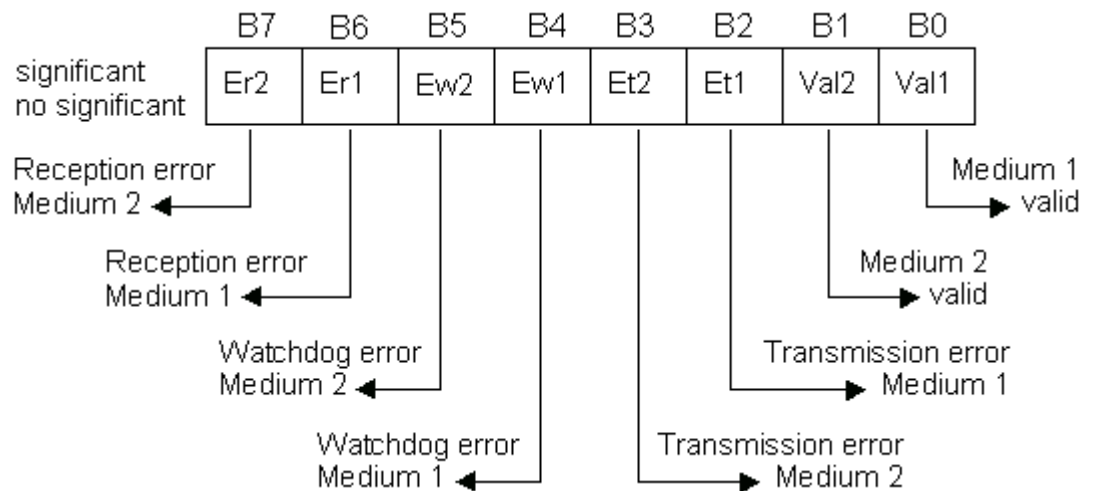


Figure 2: status byte description

The different bits should be interpreted as follows:

- The ErX bits indicate reception errors. It could be a carrier loss, a manchester code error or a cyclic redundancy code error.
- The EwX bits indicate watchdog errors. It is a transmission error, activated when the station transmission is longer than allowed.
- The EtX bits indicate all other transmission errors. It could be a bit which is too long, a transmission level too high or too low, the medium missing.
- The ValX bits indicate if the X channel communication is activated.

2.8.2. AVAILABLE COMMANDS

The user can use several commands to pilot the line drivers in order to manage the medium access. It is done with the `MEDIUM_CONTROL()` function. This function needs a control code, and returns the communication status byte after the command is complete.

The function supports the following commands:

- 0: to obtain the communication medium status byte.
- 1: to erase the reception and transmission errors (only the ErX and EtX bits are concerned).
- 2: to reset medium 1 line drivers.
- 3: to reset medium 2 line drivers.
- 4: to enable medium 1 and disable medium 2.
- 5: to enable medium 2 and disable medium 1.
- 6: to enable both medium 1 and 2.
- 7: to run the test mode (loops transmission on reception).

The user can also reset all the PC/WorldFIP adapter components, and therefore of the line drivers as well.

2.8.3. COMMUNICATION MANAGEMENT

2.8.3.1. Principle

To proceed with communication, the user must have a communication medium enabled (ValX bit to 1), with all error bits cleared. During startup, the user must call the `START_COMMUNICATION()` function to initialise and enable a medium (or both).

As a general rule, it is a good practice to supervise the communication state.

When there is no enabled channel, the test mode (local loop of transmission on reception) is automatically selected. To start communication the user must enable at least one channel.

It is done at start-up by the `START_COMMUNICATION()` function. It proceeds with:

- a reset of the line drivers (commands 2 et 3),
- erasing the transmission and reception errors (command 1),
- enabling the two channels (command 6).

When this function is complete, the two mediums are enabled.

When the two mediums are enabled, the frames are transmitted on both of them. In so far reception is concerned, both channel are watched. The channel which is activated is the channel which first receives data. When data are received at the same time on both channels, channel one is activated.

When a transmission error occurs (EtX bit), the concerned channel is disabled at the end of the current frame. The transmission on the other line is not concerned. If only one medium is enabled when a transmission error occurs, it is not disabled. The EtX bit will remain set as long as command1 is not applied (CLEAR ERROR).

When a watchdog error occurs (EwX bit), the concerned line is automatically disabled. If the error condition stops, the concerned channel automatically

restarts but the EwX bit will remain set as long as commands 2 or 3 are not applied or the START_COMMUNICATION() function executed.

The reception errors have no incidence on the medium activation state. Nevertheless the ErX error bits must be cleared.

2.9. DEFINITIONS, ERROR CODES

The library header file, "FipEngine.h", contains the export function prototypes and a set of definitions useful for the programming.

The definitions cover the following subjects:

- Bus arbitrator states;
- Events handling;
- Error codes.

The meaning of each error code is indicated with the comments. Notice that the error code returned by the functions may be a system error code.

The utility function ERROR_MESSAGE() may be used to get the associated error message. This function manages too the system error codes.

3. FIPENGINE FUNCTION COMPLETE DESCRIPTION

Each function is described with the same format:

- Function prototype.in sequential and multithread modes.
- Function parameters.
- Return value.
- Utility.

The following formalism is used:

DWORD \Leftrightarrow unsigned long,

UINT \Leftrightarrow unsigned int,

USHORT \Leftrightarrow unsigned short,

UCHAR \Leftrightarrow unsigned char,

BOOL \Leftrightarrow int, 0 means FALSE, \neq 1 means TRUE.

A byte is composed of eight bits, b7 is the MSB, b0 the LSB.

3.1. START/STOP CONFIGURATION

3.1.1. START_FIP_ENGINE

- *Prototype* : DWORD START_FIP_ENGINE (USHORT card, BOOL ifThreading) ;
- *Parameters*: this function has two parameters:
 - card: the card number from one to four.
 - IfThreading: if TRUE, the program is a multithread one and if FALSE, the program is sequential.
- *Return value*: unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.
- *Utility*: starts and allocates the objects used by the library. This function must be called before all other library functions. The allocated objects must be freed by the STOP_FIP_ENGINE() function. This is the only function in library with ERROR_MESSAGE which has identical prototypes for the two modes sequential and multithread. For all other functions, the sequential mode prototype needs the card number as argument and all the multithread mode functions are prefixed with _pth.

3.1.2. STOP_FIP_ENGINE

- *Prototype (multithread)*: DWORD STOP_FIP_ENGINE_PTH () ;
- *Prototype (sequential)*: DWORD STOP_FIP_ENGINE (USHORT card) ;
- *Parameters*: this function has one parameter in sequential mode.
 - card: the card number in the range one to four.
- *Return value*: unsigned integer (32 bits) of which the value may be a system error (if b29 is 0) or a library error (if b29 is 1). In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.
- *Utility*: stops and frees the objects used by the library. This function must be called after all other library functions.

3.1.3. DOWNLOAD_CNF

- *Prototype (multithread):* DWORD DOWNLOAD_CNF_PTH(char* PathToConfigFile) ;
- *Prototype (sequential):* DWORD DOWNLOAD_CNF(USHORT card, char* PathToConfigFile) ;
- *Parameter:*
 - PathToConfigFile: a pointer to a character string that contains the complete path name of the configuration file. This file is a FipDesigner "cnf" file. It may be equally well be a FipDesigner 3 format (that is given in the Appendix) or a FipDesigner 4 format.
 - (sequential mode) card: the card number in the range one to four.
- *Return value:* unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.
- *Utility:* this function downloads the configuration needed by the PC/WorldFIP board to be a communicating WorldFIP device. This configuration is read from the file whose name is given in the function parameter. The downloaded configuration contains the general WorldFIP parameters, the variable parameters and the message parameters. In addition to downloading configuration, this function starts up the communication part of your PC/WorldFIP board, so at the end of this function, your board is a real communicating WorldFIP device.

3.1.4. START_COMMUNICATION

- *Prototype (multithread):* DWORD START_COMMUNICATION_PTH() ;
- *Prototype:* DWORD START_COMMUNICATION (USHORT card) ;
- *Parameters:* this function has no parameters, only:
(sequential mode) card: the card number in the range one to four.;
- *Return value:* unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.
- *Utility:* this function allows to a configured board to start WorldFIP communication.

3.2. VARIABLES

3.2.1. FIP_READ_VAR

- *Prototype (multithread):* DWORD FIP_READ_VAR_PTH (USHORT AccessKey,
void* pValue,
UCHAR* pPrompt) ;
 - *Prototype:* DWORD FIP_READ_VAR (USHORT card, USHORT AccessKey,
void* pValue,
UCHAR* pPrompt) ;
 - *Parameters:*
 - AccessKey: variable access key. This value may be given by a previous call to the FIND_VAR_BY_ID() or FIND_VAR_BY_NAME() functions. Only one call to these functions in your application is sufficient since the access key does not change.
 - pValue: a pointer to a memory object allocated by the user. This memory object is used to hold the data read (from 2 up to 128 bytes).
 - pPrompt: a pointer to a byte used to hold the promptness status. If bit b7 is set, the promptness status is required. In this case, bit b5 is meaningful. Else if b7 is reset, b5 has no meaning. If b5 is set, the promptness status is TRUE, else FALSE.
 - *Return value:* unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.
 - *Utility:* this function allows the user to read the variable that has the specified access key. The read value, at "pValue" address, is increased by two or three bytes. In fact, the two first bytes read are always the PDU_TYPE and PDU_LENGTH bytes of the frame. The following read bytes are the value of the variable, but if the refreshment status is required, one more byte is appended - the refreshment status byte. The user knows two useful types of information from the utility functions FIND_VAR_BY_XXX():
 - If the status (refreshment or promptness) is required,
 - the length of the variable value (note: the PDU_TYPE, PDU_LENGTH and refreshment bytes are not included in the length value): it may be up to 125 bytes if a refreshment is required, or up to 126 bytes if refreshment is not required.
- The format of the refreshment status byte is the following:
- When b2 is set, the variable value is a valid value i.e. the value at the consumer level (in the local buffer) has been written from the network at least once.
 - In case of b2 set, b0 is meaningful: if it is set, the refreshment status is true.
 - The other bit values have no meaning.

3.2.2. FIP_WRITE_VAR

- *Prototype (multithread):* DWORD FIP_WRITE_VAR_PTH(USHORT AccessKey, void* pValue) ;
- *Prototype:* DWORD FIP_WRITE_VAR (USHORT card, USHORT AccessKey, void* pValue) ;
- *Parameters:*
 - AccessKey: variable access key. This value may be given with a previous call to the FIND_VAR_BY_ID() or FIND_VAR_BY_NAME() functions. Only one call to these functions in your application is sufficient since the access key does not change.
 - pValue: a pointer to a memory object allocated by the user. This memory object is used to hold the user data to be written (from 1 up to 125 or 126 bytes according to the refreshment byte: 125 if refreshment is required, 126 if not required).
- *Return value:* unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.
- *Utility:* this function allows the user to write the value of the variable that has the specified access key.

3.2.3. SEND_APER

- *Prototype (multithread):* DWORD SEND_APER_PTH (UCHAR Mode, USHORT Identifier) ;
- *Prototype:* DWORD SEND_APER (USHORT card, UCHAR Mode, USHORT Identifier) ;
- *Parameters:*
 - Mode: urgent sending if mode ≠ 0 or normal sending if mode = 0.
 - Identifier: identifier of the aperiodical variable that will be transmitted.
- *Return value:* unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.
- *Utility:* this function allows the user to send a request for an aperiodical variable transmission. This request have two levels of priority depending on the "Mode" parameter value. The necessary condition to send the request to the bus arbitrator is to dispose one **produced variable** with **aperiodical variable transmission request authorised** in the local database. When the bus arbitrator receives the request, it pushes the request onto a queue. The variable transmission will be processed in the next free aperiodical window of the bus arbitrator.

3.3. MESSAGES

3.3.1. FIP_READ_MSG

- *Prototype (multithread):* `DWORD FIP_READ_MSG_PTH (`
`USHORT AccessKey,`
`USHORT* pReceptionTime,`
`USHORT* pMsgLength,`
`USHORT* pDestId,`
`UCHAR* pDestSegment,`
`USHORT* pSourceId,`
`UCHAR* pSourceSegment,`
`UCHAR* pMsgContent) ;`
- *Prototype:* `DWORD FIP_READ_MSG (` `USHORT card,`
`USHORT AccessKey,`
`USHORT* pReceptionTime,`
`USHORT* pMsgLength,`
`USHORT* pDestId,`
`UCHAR* pDestSegment,`
`USHORT* pSourceId,`
`UCHAR* pSourceSegment,`
`UCHAR* pMsgContent) ;`
- *Parameters:*
 - AccessKey: access key of the received message. This key is given by a previous call to the `READ_EVENT()` function.
 - pReceptionTime: unused in this version, reserved for future use.
 - pMsgLength: pointer to an integer that holds the read message length (must be inferior or equal to 250).
 - pDestId: pointer to an integer that holds the destination identifier of the read message.
 - pDestSegment: pointer to an integer that holds the segment number of the previous destination identifier.
 - pSourceId: pointer to an integer that holds the source identifier of the read message.
 - pSourceSegment: pointer to an integer that holds the segment number of the previous source identifier.
 - pMsgContent: pointer to a memory object allocated by the user that holds the read message content (must be inferior or equal than 250).
- *Return value:* unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.
- *Utility:* read a received message. This function needs to know the access key of the received message. This is given by a previous call to the `READ_EVENT()` function.

3.3.2. FIP_WRITE_MSG

- *Prototype (multithread):* `DWORD FIP_WRITE_MSG_PTH (`
`UCHAR ChannelNb,`
`USHORT AccessKey,`
`UCHAR AckMode,`
`UCHAR EventNumber,`
`USHORT MsgLength,`
`USHORT DestId,`
`UCHAR DestSegment,`
`USHORT SourceId,`
`UCHAR SourceSegment,`
`UCHAR* pMsgContent) ;`
- *Prototype:* `DWORD FIP_WRITE_MSG (` `USHORT card,`
`UCHAR ChannelNb,`
`USHORT AccessKey,`
`UCHAR AckMode,`
`UCHAR EventNumber,`
`USHORT MsgLength,`
`USHORT DestId,`
`UCHAR DestSegment,`
`USHORT SourceId,`
`UCHAR SourceSegment,`
`UCHAR* pMsgContent) ;`
- *Parameters:*
 - ChannelNb: number of the transmission queue that will be used. If you wish the transmission to be successful, the following conditions must be fulfilled:
 - 1- There is at least one **produced** variable
 - 2- This produced variable is **authorised for message transmission**. The type, periodical or aperiodical depends on the channel number you choose.
 - 3- This variable is authorised with the **same channel number** as the one specified here.
 - Note: One variable is needed and is sufficient for each messaging type i.e. periodic and aperiodic.
 - The values available for "ChannelNb" are the following:
 - 0: queue number for aperiodical message.
 - 1 to 8: queue number for periodical message. Eight queues are possible in periodical messaging.
 - AccessKey: access key of the transmitted message. This value is given by a previous call to the `FIND_MSG_BY_ID()` or `FIND_MSG_BY_NAME()` functions. One call for your entire application is sufficient since the transmitted message access key does not change.
 - AckMode: sending mode of the message. If `AckMode = 0`, it is sent without acknowledgement. If `AckMode ≠ 0`, it is acknowledged.
 - EventNumber: event number that will be generated on message transmission. The knowledge of this number is important for finding the message sending report provided by the `READ_EVENT()` function.

- MsgLength: length of the message to be transmitted. It is also the length of the memory object pointed to by pMsgContent.
 - DestId: destination identifier of the message to be transmitted.
 - DestSegment: segment number of the previous destination identifier.
 - SourceId: source identifier of the message to transmit.
 - SourceSegment: segment number of the previous source identifier.
 - pMsgContent: pointer to a memory object allocated by the user. This memory object holds the message content to be transmitted and must be inferior or equal to 250 bytes.
- **Return value**: unsigned integer (32 bits) of which the value may be a system error (if b29 is 0) or a library error (if b29 is 1). In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.
 - **Utility**: if all parameters are right, the specified message is sent.

3.4. EVENTS

3.4.1. IS_EVENT

- *Prototype (multithread):* `DWORD IS_EVENT_PTH (BOOL* plsEvent) ;`
- *Prototype:* `DWORD IS_EVENT (USHORT card, BOOL* plsEvent) ;`
- *Parameter:*
 - plsEvent: pointer to a boolean that will be affected by this function. This parameter is set to FALSE if there is no event in the event queue, and TRUE if there is at least one. In the latter case, the user must read the present events with the `READ_EVENT()` function to pop them.
- *Return value:* unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.
- *Utility:* this function determines whether or not the event queue is empty.

3.4.2. READ_EVENT

- *Prototype (multithread):* `DWORD READ_EVENT_PTH (UCHAR* pLifeTime, UCHAR* pMsgSendingReport, UCHAR* pType, UCHAR* pEventNumber, USHORT* pAdditionnalInfo) ;`
- *Prototype:* `DWORD READ_EVENT (UCHAR* pLifeTime, UCHAR* pMsgSendingReport, UCHAR* pType, UCHAR* pEventNumber, USHORT* pAdditionnalInfo) ;`
- *Parameters:*
 - pLifeTime: pointer to a byte containing the event life duration.
 - 0, the event is temporary and will be not be generated at the next occurrence.
 - 1, the event is permanent and will be reactivated at every occurrence.
 - pMsgSendingReport: pointer to a byte that contains the message sending report. This byte is meaningful when the event number set in "pEventNumber" is the same as those set in `FIP_WRITE_MSG()` parameter. In this case, the "pLifeTime" parameter has no meaning. The values set in this byte are the following:
 - 0, messaging not acknowledged.
 - 4, positive acknowledgement without retry.

- 8, no acknowledgement (positive or negative) after retries.
- 12, positive acknowledgement after retries.
- 16, wrong RP_MSG_NOACK frame in transmission.
- 20, negative acknowledgement without retry.
- 28, negative acknowledgement after retries.

Note: you may use the header file definitions for the numerical values stated previously.

- pType: pointer to a byte that contains the event type. The values set in this byte are the following:
 - 0: aperiodical: A aperiodical variable transmission request has been sent to the bus arbitrator. In this case, the "pLifeTime" parameter has no meaning.
 - 1: transmitted: a variable or a message has been sent on the network.
 - 2: received: a variable or a message has been received from the network.
 - 3: bus arbitrator: The bus arbitrator has been suspended (the local bus arbitrator sends idle frames). In this case, the "pLifeTime" parameter has no meaning.
- pEventNumber: pointer to a byte that contains the associated event number. Different possible cases:
 - it may be a predefined event number (See paragraph 2.6.1),
 - it may be a event that has been set with DEF_VAR_EVENT() function,
 - it may be a event that has been set with FIP_WRITE_MSG() function,
 - it may be a event that has been set with the SUSPEND bus arbitrator instruction.
- pAdditionnalInfo: pointer to a byte that contains some additional information. The values set in this byte are the following:
 - the number of the identifier contained in the transmitted request if the event type is aperiodical,
 - the access key of the variable or message associated with the event if the event type is received or transmitted. The access key is necessary for aperiodical reading with FIP_READ_VAR() or FIP_READ_MSG() functions.
 - the instruction address of SUSPEND in PC/WorldFIP board memory if the event type is a bus arbitrator. This address is necessary to resume bus arbitrator with CONTINUE_BA() function. Note: this address must be modified to become a CONTINUE_BA() parameter (see paragraph 3.5.4).

- **Return value**: unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.
- **Utility**: read and pop an event from the event queue.

3.4.3. DEF_VAR_EVENT

- *Prototype (multithread):* `DWORD DEF_VAR_EVENT_PTH (UCHAR LifeTime, UCHAR Type, UCHAR EventNumber, USHORT AccessKey);`
- *Prototype:* `DWORD DEF_VAR_EVENT (UCHAR LifeTime, UCHAR Type, UCHAR EventNumber, USHORT AccessKey);`
- *Parameters:*
 - LifeTime: event life duration.
 - 0, the event is temporary and will not be generated at the next occurrence.
 - 1, the event is permanent and will be reactivated at every occurrence.
 - Type: the event type to be defined.
 - 0, no event will be configured.
 - 1, the event will be generated at the variable transmission.
 - 2, the event will be generated at the variable receiving.
 - EventNumber: event number associated with the variable.
 - AccessKey: access key of the variable associated with this event. This value is given by a previous call to the `FIND_VAR_BY_ID()` or `FIND_VAR_BY_NAME()` functions.
- *Return value:* unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.
- *Utility:* define an event for variable transmission or receiving.

3.5. BUS ARBITRATOR

3.5.1. BA_PROG

- *Prototype (multithread) :* `DWORD BA_PROG_PTH (char* PathToBaAsciiFile);`
- *Prototype:* `DWORD BA_PROG (char* PathToBaAsciiFile);`
- *Parameter:*
 - PathToBaAsciiFile: pointer to a character string that contains the complete path name of the bus arbitrator ASCII file.
- *Return value:* unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.

- *Utility:* this function downloads a bus arbitrator program in the PC/WorldFIP board memory. This program comes from an ASCII file and its format is given in paragraph 2.7.5. This function does not allow the bus arbitrator to start up. For that you need the START_BA() function.

3.5.2. START_BA

- *Prototype (multithread):* DWORD START_BA_PTH (USHORT Padding) ;
- *Prototype:* DWORD START_BA (USHORT Padding) ;
- *Parameter:*
 - Padding: identifier used by the bus arbitrator for its idle frame.
- *Return value:* unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.
- *Utility:* starts the bus arbitrator. Note: this function must be used only if the bus arbitrator is in the STOPPED state. On the other hand, if the bus arbitrator is still STOPPED after a START_BA(), try once again (not more than once).

3.5.3. STOP_BA

- *Prototype (multithread):* DWORD STOP_BA_PTH () ;
- *Prototype:* DWORD STOP_BA () ;
- *Parameters:* this function has no parameters.
- *Return value:* unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.
- *Utility:* stops the bus arbitrator. Note: this function is only used if the bus arbitrator state is different to STOPPED or STARTING.

3.5.4. CONTINUE_BA

- *Prototype (multithread):* DWORD CONTINUE_BA_PTH (UCHAR NewMacroCycle, USHORT NewProgramAdd) ;
- *Prototype:* DWORD CONTINUE_BA (UCHAR NewMacroCycle, USHORT NewProgramAdd) ;
- *Arguments:*
 - NewMacroCycle: Unused in this version, must be set to 0. Reserved for future use.
 - NewProgramAdd: restart address of the bus arbitrator. This address is given by the sum of the "pAdditionnalInfo" value from the READ_EVENT() function (if the event type is "bus arbitrator") and the value "16".

- *Return value:* unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.
- *Utility:* this function allows the bus arbitrator to pass again into an activate state. Note: This function must only be used if the bus arbitrator state is PENDING.

3.5.5. BA_REPORT

- *Prototype (multithread):* DWORD BA_REPORT_PTH (UCHAR* pStatus);
- *Prototype:* DWORD BA_REPORT (UCHAR* pStatus);
- *Parameter:*
 - pStatus: pointer to a byte that contains the bus arbitrator state. The PC/WorldFIP board is a bus arbitrator when bit 3 is set to 1. The values set in this byte are the following (see paragraph 2.7.1 for more information about these states):
 - 8, SENDING;
 - 9, STOPPED;
 - 10, STARTING;
 - 11, IDLE;
 - 12, MSG_WND;
 - 13, APER_WND;
 - 14, WAITING;
 - 15, PENDING;
 - Other values: the PC/WorldFIP board is not a bus arbitrator.
- *Return value:* unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.
- *Utility:* this function allows the user to find out the current state of the bus arbitrator. This function is essential for managing the set of bus arbitrator functions.

3.5.6. CHANGE_BA

- *Prototype (multithread):* DWORD CHANGE_BA_PTH (USHORT NewProgramAdd);
- *Prototype:* DWORD CHANGE_BA (USHORT NewProgramAdd);
- *Parameter:*
 - NewProgramAdd: next bus arbitrator macrocycle start address.
- *Return value:* unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.

- *Utility:* indicates the next bus arbitrator address. By default, the DOWNLOAD_CNF() function uses the address of the first bus arbitrator program contained in the "cnf" file. The GET_BA_PROGRAM_ADDR() function indicates the first and second program addresses. By calling the CHANGE_BA() function with one of the address returned by GET_BA_PROGRAM_ADDR(), we can toggle from a program to the other. The toggling occurs at the end of the running macrocycle.

3.6. TOOLS FUNCTIONS

3.6.1. PURGE

- *Prototype (multithread):* DWORD PURGE_PTH (UCHAR Parameter);
- *Prototype:* DWORD PURGE (UCHAR Parameter);
- *Parameter:*
 - Parameter: this byte empties the queue.
 - 0 to 8: message transmission queue (recall: 0 means aperiodical transmission; 1 to 8 means periodical transmission).
 - 9: aperiodical variable transmission request queue.
- *Return value:* unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.
- *Utility:* empties the specified queue.

3.6.2. FIND_MSG_BY_ID

- *Prototype (multithread):* BOOL FIND_MSG_BY_ID_PTH (USHORT SourceIdentifier, UCHAR SourceSegment, USHORT DestIdentifier, UCHAR DestSegment, char *Name, USHORT* pAccessKey);
- *Prototype:* BOOL FIND_MSG_BY_ID (USHORT SourceIdentifier, UCHAR SourceSegment, USHORT DestIdentifier, UCHAR DestSegment, char *Name, USHORT* pAccessKey);
- *Parameters:*
 - SourceIdentifier: source identifier of the message.
 - SourceSegment: segment number of the previous identifier.
 - DestIdentifier: destination identifier of message.
 - DestSegment: segment number of the previous identifier.
 - Name: name of the message found.
 - pAccessKey: pointer to an integer that contains the message access key if the function returns TRUE and if the message is a transmitted type. If the message is a received type, the access key is only known by a READ_EVENT() function. In the case of a received message, the function sets the value by pointed to by "pAccessKey" to a meaningless value equal to 0xffff.

This document may not be used, copied or redistributed without authorisation of HLP Technologies

- *Return value:* TRUE if the message is found. Otherwise FALSE.
- *Utility:* gives the message access key and other information from the message identifiers

3.6.3. FIND_MSG_BY_NAME

- *Prototype:* (multithread): BOOL FIND_MSG_BY_NAME_PTH (char* Name, USHORT* pAccessKey, USHORT* pSourceIdentifier, UCHAR* pSourceSegment, USHORT* pDestIdentifier, UCHAR* pDestSegment) ;
- *Prototype:* BOOL FIND_MSG_BY_NAME (char* Name, USHORT* pAccessKey, USHORT* pSourceIdentifier, UCHAR* pSourceSegment, USHORT* pDestIdentifier, UCHAR* pDestSegment) ;
- *Parameters:*
 - Name: pointer to a character string that contains the message name (32 characters max).
 - pAccessKey: pointer to an integer that contains the message access key if the function returns TRUE and if the message is a transmitted type. If the message is a received type, the access key is only known by a READ_EVENT() function. In the case of a received message therefore, the function sets the value pointed to by pAccessKey to a meaningless value equal to 0xffff.
 - pSourceIdentifier: pointer to an integer that contains the message source identifier if the function returns TRUE. This integer has no meaning if the function returns FALSE.
 - pSourceSegment: pointer to an integer that contains the segment number of the previous identifier if the function returns TRUE. This integer has no meaning if the function returns FALSE.
 - pDestIdentifier: pointer to an integer that contains the message destination identifier if the function returns TRUE. This integer has no meaning if the function returns FALSE.
 - pDestSegment: pointer to an integer that will contain the segment number of the previous identifier if the function returns TRUE. This integer has no meaning if the function returns FALSE.
- *Return value:* TRUE if the message is found. Otherwise FALSE.
- *Utility:* gives the message access key and other information, such as the source and destination identifiers, from the message name.

3.6.4. FIND_VAR_BY_ID

- *Prototype (multithread):* BOOL FIND_VAR_BY_ID_PTH (USHORT Identifier, USHORT* pAccessKey, char* Name,

This document may not be used, copied or redistributed without authorisation of HLP Technologies

- *Prototype:* `BOOL FIND_VAR_BY_ID (USHORT Identifier,
 USHORT* pAccessKey,
 char* Name,
 USHORT* pValueLength,
 UCHAR* pStatus);`

- **Parameters:**
 - Identifier: variable identifier.
 - pAccessKey: pointer to an integer that contains the variable access key if the function returns TRUE. This integer has no meaning if the function returns FALSE.
 - Name: name of the variable found. Be careful! The user must allocate enough bytes for this information. The length of a name cannot exceed MAX_NAME_LENGTH.
 - pValueLength: pointer to an integer that contains the length of the variable value. This length does not include the PDU_LENGTH, PDU_TYPE and refreshment bytes.
 - pStatus: pointer to an integer that allows the user to find out whether time status is required or not:
if b0 is set, refreshment status is required;
if b1 is set, promptness status is required.
- **Return value:** TRUE if the message is found. Otherwise FALSE.
- **Utility:** gives the variable access key and other information, from the variable identifier.

3.6.5. FIND_VAR_BY_NAME

- **Prototype (multithread):**

```

BOOL FIND_VAR_BY_NAME_PTH (
char* Name,
                                USHORT* pAccessKey,
                                USHORT* pIdentifier,
                                USHORT* pValueLength,
                                UCHAR* pStatus );

```
- **Prototype:**

```

BOOL FIND_VAR_BY_NAME ( char* Name,
                                USHORT* pAccessKey,
                                USHORT* pIdentifier,
                                USHORT* pValueLength,
                                UCHAR* pStatus );

```
- **Parameters:**
 - Name: pointer to a character string that contains the variable name (32 characters max).
 - pAccessKey: pointer to an integer that contains the variable access key if the function returns TRUE. This integer has no meaning if the function returns FALSE.
 - pIdentifier: pointer to an integer that contains the variable identifier if the function returns TRUE.
 - pValueLength: pointer to an integer that contains the length of the variable value. This length does not include the PDU_LENGTH, PDU_TYPE and refreshment bytes.
 - pStatus: pointer to an integer that allows the user to find out whether time status is required or not:
if b0 is set, refreshment status is required;
if b1 is set, promptness status is required.
- **Return value:** TRUE if the message is found. Otherwise FALSE.

- *Utility*: gives the variable access key and other information, from the variable name.

This document may not be used, copied or redistributed without authorisation of HLP Technologies

3.6.6. GET_BA_PROGRAM_ADDR

- *Prototype (multithread):* void GET_BA_PROGRAM_ADDR_PTH (USHORT* pAdd1, USHORT* pAdd2);
- *Prototype:* void GET_BA_PROGRAM_ADDR (USHORT* pAdd1, USHORT* pAdd2);
- *Parameters:*
 - pAdd1: first bus arbitrator macrocycle address. This program was downloaded with the "cnf" file by the DOWNLOAD_CNF() function.
 - pAdd2: second bus arbitrator macrocycle address. This program was downloaded with the "cnf" file by the DOWNLOAD_CNF() function.
- *Return value:* none.
- *Utility:* returns the addresses of the two bus arbitrator macrocycles. They can be used as parameters with the CHANGE_BA() and the CONTINUE_BA() functions. If the pAdd1 or pAdd2 pointer is null when the function returns, it means there is no associated macrocycle.

3.6.7. GET_TRANSMISSION_QUEUE

- *Prototype (multithread):* DWORD GET_TRANSMISSION_QUEUE_PTH(UINT Mode, QUEUE_INFO* pVarList, DWORD AllowedVarNb, UINT* pVarNbRequired);
- *Prototype:* DWORD GET_TRANSMISSION_QUEUE (UINT Mode, QUEUE_INFO* pVarList, DWORD AllowedVarNb, UINT* pVarNbRequired);
- *Parameters:*
 - Mode: indicates which kind of queue is to searched for:
 - 0: aperiodical messages,
 - 1: periodical messages,
 - 2: aperiodical variables.
 - pVarList: pointer to an array of QUEUE_INFO structures (described in Appendix). This structure contains the identifier of the variable and the number of the associated transmission

channel. The user has to allocate the array. "AllowedVarNb" is the size of the allocated buffer. If this size large enough, the array will be filled. Whatever the "AllowedVarNb" value, the required count is returned in the "pVarNbRequired" parameter.

- AllowedVarNb: size of the array (in QUEUE_INFO units) allocated by the user.
- pVarNbRequired: count of QUEUE_INFO structures to be returned.

- **Return value**: unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.

- **Utility**: allows to the user to know the variables enabled for messages or aperiodical transfers and the associated channel. This information is useful to:

- affect the "ChannelNb" parameter of the FIP_WRITE_MSG() function
- check that a variable has an aperiodical variable request transmission channel before using the SEND_APER() function.

3.6.8. MEDIUM_CONTROL

- **Prototype (multithread)**: DWORD MEDIUM_CONTROL_PTH (UCHAR Command, UCHAR* pStatus);
- **Prototype**: DWORD MEDIUM_CONTROL (UCHAR Command, UCHAR* pStatus);

- **Parameters**:

- Command: command to apply:
 - 0: to obtain the communication medium status byte.
 - 1: to erase the reception and transmission errors (only the ErX and EtX bits are concerned).
 - 2: to reset medium 1 line drivers.
 - 3: to reset medium 2 line drivers.
 - 4: to enable medium 1 and disable medium 2.
 - 5: to enable medium 2 and disable medium 1.
 - 6: to enable both medium 1 and 2.
 - 7: to run the test mode (loops transmission on reception).

- pStatus: pointer to the returned status byte.

The status byte format is the following (bit activated when 1):

- b7 = Er2 (reception error channel 2)
- b6 = Er1 (reception error channel 1).
- b5 = Ew2 (watchdog error channel 2).
- b4 = Ew1 (watchdog error channel 1).
- b3 = Et2 (transmission error channel 2).
- b2 = Et1 (transmission error channel 1).
- b1 = Val2 (channel 2 enabled).
- b0 = Val1 (channel 1 enabled).

- *Return value:* unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.
- *Utility:* permits to the user to perform commands, and to retrieve medium(s) state after the command application. For more details, see the chapter 2.8.

3.6.9. ERROR_MESSAGE

- *Prototype:* DWORD ERROR_MESSAGE (DWORD ErrorCode) ;
- *Parameters:*
 - ErrorCode: error code in.
- *Return value:* unsigned integer (32 bits) of which the value may be a system error or a library error. In the latter case, the possible return values are given by the FipEngine header file. A null value indicates success.
- *Utility:* gives the error message associated with the error code. If the error code is a system error code, the language is the system language. If the error code is a library error code, the language is the same as the library.

4. APPENDIX

4.1. DETAILED FORMAT OF "CNF" FILES FROM FIPDESIGNER VERSION 3.

This format is compatible with the DOWNLOAD_CNF() function and depends on the following C structures:

```
typedef struct
{
    unsigned short    frequence,    // unused with DOWNLOAD_CNF( )
                     vitesse,      // 0 □ 31,25 kb/s
                                     // 1 □ 1 Mb/s
                                     // 2 □ 2,5 Mb/s
    tslot,            // parameter for unit time used for the durations:
                     // 0 □ 100µs at 31,25 kb/s; 62.5µs at 1Mb/s; 50µs at 2.5
                     Mb/s.
                     // 1 □ 400µs at 31,25 kb/s; 250µs at 1Mb/s; 200µs at 2.5
                     Mb/s.
                     // 2 □ 1000µs at 31,25 kb/s; 625µs at 1Mb/s; 500µs at 2.5
                     Mb/s.
                     // 3 □ 4000µs at 31,25 kb/s; 2500µs at 1Mb/s; 2000µs at
                     2.5 Mb/s.
    turn_around,      // this parameter is usually set to 0.
                     // parameter used for the turn-around time:
                     // turn-around time (µs)= turn_around x a + 1.25 x Tbit
                     // where:
                     // at 31.25 kbit/s:
                     //     a = 128; Tbit = 32µs; min value for turn_around
                     = 3;
                     //     max value for turn_around = 63.
                     // at 1Mb/s, if "clktype" = 0:
                     //     a = 0.625; Tbit = 1 µs; min value for turn_around
                     = 14;
                     //     max value for turn_around = 63.
                     // at 1Mb/s, if "clktype" = 1:
                     //     a = 4; Tbit = 1 µs; min value for turn_around = 3;
                     //     max value for turn_around = 63.
                     // at 2.5Mb/s:
                     //     a = 1.6; Tbit = 0.4 µs; min value for turn_around
                     = 20;
                     //     max value for turn_around = 63.
    silence,          // see clktype parameter in STATION.
                     // parameter used for the silence time:
                     // silence time (µs)= a + (silence x b)
                     // (note: (silence x b) will be use for the bus arbitrator
    election_time      // and start-up time)
                     // where
                     // at 31.25 kbit/s:
                     //     a = 64µs; b = 1024µs; min value for silence = 4;
                     //     max value for silence = 63.
                     // at 1Mb/s, if "clktype" = 0:
                     //     a = 40µs; b = 5 µs; min value for silence = 22;
                     //     max value for silence = 63.
                     // at 1Mb/s, if "clktype" = 1:
                     //     a = 40µs; b = 32 µs; min value for silence = 4;
                     //     max value for silence = 63.
                     // at 2.5Mb/s:
                     //     a = 32µs; b = 12.8 µs; min value for silence = 5;
                     //     max value for silence = 63.
    messages,         // maximum number of messages that the configuration
    can_handle.        // must be a power of 2.
```

This document may not be used, copied or redistributed without authorisation of HLP Technologies

```

// from 16 to 2048 for DOWNLOAD_CNF( ).
variables, // maximum number of variables that the configuration
can handle. // must be a power of 2.

ba, // from 16 to 2048 for DOWNLOAD_CNF( ).
// bus arbitrator program size in kilowords.
// usually set to 5 kw for current program.
// Could be more, there are 128kw on PC/FIP board.
// See paragraph 2.7.3 for more information.
idbourrage; // padding identifier. Usually set to 0x7530

long ba_electime, // parameter for bus arbitrator election time,
// see the previous silence field for (silence x b) value,
// bus arbitrator election time (µs)= ba_electime x
(silence x b)
// min value for ba_electime = 518
// max value for ba_electime = 8708
ba_startime; // parameter for bus arbitrator start up time,
// see the previous silence field for (silence x b) value,
// bus arbitrator start up time (µs) =
// ba_startime x (silence x b)
// the bus arbitrator start-up time shall be superior to the
greatest value
// of the bus arbitrator election time of any workstation on
the network.
// The recommended value for ba_startime is 8712.

char clktype, // 0 or 1, used for modify the duration span of silence and
turn around. // usually set to 0 for small duration value.

frametype; // 1 □ WorldFIP frame delimiters and CRC
// 0 □ FIP frame delimiters and CRC

unsigned short carte_adr, // unused with DOWNLOAD_CNF( )
interruption, // unused with DOWNLOAD_CNF( )
memoire; // unused with DOWNLOAD_CNF( )

}STATION; // structure that holds the general WorldFIP parameters

typedef struct
{
unsigned short identifieur; // WorldFIP variable identifier

char nom[30], // variable name (null-terminated)
format[30]; // unused with DOWNLOAD_CNF( )

short pdu, // PDU; usually set to 0x40 for a user variable and
// 0x50 for a network management variable.

longueur, // length of user data field: 1 up to 125 bytes if
refreshment status is // required; 1 up to 126 bytes if not required.

msg_channel, // 0 □ the aperiodical message transmission queue n°0
// has been defined on this variable.
// 1 up to 8 □ the aperiodical message transmission
queue n°1 up to 8
// has been defined on this variable.
// other values: no queue has been defined
evenement; // event number associated with the variable (optional)

char type, // 1 □ consumed variable
// 0 □ produced variable
refresh, // 1 □ refreshment required
// 0 □ no refreshment
prompt, // 1 □ promptness required
// 0 □ no promptness

```

```

requests are
    messagerie; // If b0 = 1, the aperiodical message transmission
                // authorised on this variable. In this case, the
                // "type" field
                // must be 0.
                // If b1 = 1, the message receiving is authorised on this
                // variable,
                // i.e. it will be possible to receive any message
                // whose
                // two first bytes of the destination address field is
                // equal to
                // the identifier value of this variable.
                // If b6 = 1, the aperiodical variable transmission requests
                // are
                // authorised on this variable. In this case, the
                // "type" field
                // must be 0.

    unsigned short    period_r, // refreshment period in milliseconds
    period_p;         // promptness period in milliseconds

} VARIABLE; // structure that holds a variable configuration

typedef struct
{
    unsigned short    source, // message source identifier (note: segment = 0 in this
version 3)
    destination, // message destination identifier (note: segment = 0)
    value_ptr, // unused
    evenement; // event number associated with the message (optional)

    char    nom[30], // message name (null-terminated)
    format[30], // unused with DOWNLOAD_CNF( )
    acquittement, // 0, messaging service without acknowledgement;
                // 1 with acknowledgement.
    mode, // 0 □ periodical message transmission.
                // Be careful, the database must have one
                // with a msg_channel field set from 1 up to 8.
                // 1 □ aperiodical message transmission.
                // Be careful, the database must have one
                // with a msg_channel field set to 0.
                // note: the mode field is unused for a received message.
    type; // 1 □ transmitted message.
                // Be careful, the database must have one
                // with a "messagerie" field with b0 = 1. This
                // variable is the
                // same than the source identifier of the message.
                // 0 □ received message.
                // Be careful, the database must have one
                // variable (produced
                // or consumed) with a "messagerie" filed with b1
                // = 1.
                // This variable is the same than the destination
                // identifier of
                // the message.

} MESSAGE; // structure that holds a message configuration.

typedef struct
{
    char    op_code[15]; // character string that contains the instruction

} INSTRUCTION; // structure that holds a bus arbitrator instruction

```

// in ASCII format.

The configuration file format is the following:

- Header: 11 bytes that holds the "LOCOM 3.00" (and null character) character string.
- Bus arbitrator origin: 128 bytes.
- Bus arbitrator first sequence:
 - A short integer (2 bytes) that holds n number of instructions.
 - n INSTRUCTION structures.
- Bus arbitrator second sequence:
 - A short integer (2 bytes) that holds n number of instructions.
 - n INSTRUCTION structures.
- Message configurations:
 - A short integer (2 bytes) that holds n number of message.
 - n MESSAGE structures.
- General configuration of the device: one STATION structure.
- Variable configurations:
 - A short integer (2 bytes) that holds n number of variable.

This document may not be used, copied or redistributed without authorisation of HLP Technologies

4.2. THE MULTI THREAD POSIX LIBRARY

POSIX is the acronym for **P**ortable **O**perating **S**ystem **I**nterface **(for Unix)**.

It is the name of a family of related standards specified by the IEEE to define the application programming interface (API) for software compatible with variants of the Unix operating system, although the standard can apply to any operating system.

POSIX 1c is associated with Threads extensions (IEEE Std 1003.1c-1995)

The latest version is known as IEEE Std 1003.1, 2004 Edition.

The library is included in the most of LINUX distributions.

To use the **Pthreads** library, you must include the file *"pthread.h"*. If you use semaphores, you must too include the file *"semaphore.h"* and if you use the function *sched_yield*, you must too include the file *"sched.h"*.

To load the library, you must use the option *-lpthread* in your makefile.

Generally, the functions of the library return an integer, 0 if successful or an error code like:

- ESRCH : No process matches the specified process ID.
- EDEADLK: Deadlock avoided.
- EINVAL : Invalid argument
- EAGAIN: Resource temporarily unavailable.
- EPERM: Operation not permitted
- ENOMEM: No memory available

Threads created by the same process share its resources but each maintains its own stack pointer, scheduling properties, registers, set of pending and blocked signals, specific data.

Because threads share resources, synchronization and protection mechanisms are necessary. The **Pthread** library implement mechanisms like: mutex, semaphores, condition variables, barriers, spinlocks, read_write locks.

You can access by the Linux command **man** the documentation for each function.

4.2.1. CREATION/DESTRUCTION

Once created via the function *pthread_create* by a main process, a thread may create other threads.

A thread may be terminated in several different ways:

- the thread has finished to run its routine.
- the thread has called *pthread_exit*.
- the thread is canceled by another thread via the *pthread_cancel*.
- the main process is terminated.

Creation/Destruction	
Data Structure	
<i>pthread_t</i>	Thread identifier
<i>pthread_attr_t</i>	Thread Attributes Structure
Functions	
<i>int</i> <i>pthread_create(pthread_t* tid, const ,</i> <i>pthread_attr_t* tattr, void*(*start_routine)</i> <i>(void*), void* arg)</i>	Creates a thread. If return is successful, the thread with identifier <i>tid</i> starts running the function <i>start_routine</i> . If <i>tattr</i> is NULL, the default attributes are used.
<i>void</i> <i>pthread_exit(void *status)</i>	The calling thread terminates. If a process is waiting for the end of this thread on a <i>pthread_join</i> function, it will get status as argument at return.
<i>int</i> <i>pthread_cancel(pthread_t thread)</i>	The calling thread wants the thread of identifier <i>thread</i> to be cancelled.
<i>int</i> <i>pthread_detach(pthread_t tid)</i>	This function indicates to the application that storage for the thread <i>tid</i> can be reclaimed when the thread terminates.
<i>int</i> <i>pthread_setcancelstate(int state, int</i> <i>*oldstate)</i>	This function allows to enable or disable cancellation.
<i>void</i> <i>pthread_testcancel(void)</i>	At this point, the calling thread can be cancelled by another thread.
<i>int</i> <i>pthread_setcanceltype(int type, int</i> <i>*oldtype)</i>	This function allows to set the cancellation type: deferred or asynchronous.
<i>int</i> <i>pthread_join(pthread_t tid, void **status)</i>	The calling thread is going to wait for the termination of the thread with identifier <i>tid</i> . The thread must not be detached.

4.2.2. UTILITIES

Utilities	
Data Structure	
<i>sched_param</i>	This structure contains scheduling parameters like <i>sched_priority</i> .
Functions	
<i>pthread_t</i> <i>pthread_self(void)</i>	This function returns the identifier of the calling thread.
<i>int</i> <i>pthread_equal(pthread_t tid1, pthread_t tid2)</i>	This function returns 0 when <i>tid1</i> and <i>tid2</i> are equal.
<i>int</i> <i>pthread_once(pthread_once_t *once_control, void (*init_routine)(void))</i>	This function allows to perform an initialisation routine for a thread. It can be called only once.
<i>int</i> <i>sched_yield(void)</i>	This function causes the current thread to yield its execution in favor of another thread.
<i>int</i> <i>pthread_setschedparam(pthread_t tid, int policy, const struct sched_param *param)</i>	This function is used to modify the scheduling policy and scheduling parameters of a thread. Supported policies are <i>SCHED_FIFO</i> , <i>SCHED_RR</i> and <i>SCHED_OTHER</i> .
<i>int</i> <i>pthread_getschedparam(pthread_t tid, int *policy, struct sched_param *param)</i>	This function is used to set the scheduling policy and scheduling parameters of a thread.
<i>int</i> <i>pthread_setschedprio(pthread_t tid, int prio)</i>	This function is used to set the scheduling priority of a thread.
<i>int</i> <i>pthread_kill(pthread_t tid, int sig)</i>	This function send the signal <i>sig</i> to the thread <i>tid</i> . The thread <i>tid</i> must be in the same process as the calling thread.
<i>int</i> <i>pthread_sigmask(int how, const sigset_t *new, sigset_t *old)</i>	This function is used to modify or examine the signal mask of the calling thread.
<i>int</i> <i>pthread_atfork(void (*prepare) (void, void (*parent) (void), void (*child) (void))</i>	This function declares <i>fork()</i> handlers that are called before and after <i>fork()</i> in the context of the thread that called <i>fork()</i> .

This document may not be used, copied or redistributed without authorisation of HLP Technologies

4.2.3. THREAD SPECIFIC DATA

Threads can own private data. Each thread private data item is associated with a key.

Thread Specific Data	
Data Structure	
<i>pthread_key_t</i>	
Functions	
<i>int</i> <i>pthread_key_create(pthread_key_t *keyl,</i> <i>void (*destructor)(void*));</i>	This function allocates a key used to identify the thread-specific data in a process.
<i>int</i> <i>pthread_key_delete(pthread_key_t key);</i>	This function destroys an existing thread-specific data key.
<i>int</i> <i>pthread_setspecific(pthread_key_t key,</i> <i>const void *value);</i>	This function is used to bind specific data and key.
<i>void*</i> <i>pthread_getspecific(pthread_key_t key);</i>	This function is used to get the specific data associated with a key.

4.2.4. ATTRIBUTES

Attributes are used to give at threads a behaviour different from the default behaviour. Attributes are used at thread creation time. An attribute object is opaque and cannot be accessed directly.

Attributes have default values. So, if the structure *pthread_attr_t* is NULL at the creation of a thread, default values are used.

Attributes	Default Value
scope	PTHREAD_SCOPE_PROCESS
detachstate	PTHREAD_CREATE_JOINABLE
stackaddr	NULL
stacksize	0
priority	0
inheritsched	PTHREAD_EXPLICIT_SCHED
schedpolicy	SCHED_OTHER
guardsize	PAGESIZE

Thread Attributes	
Data Structures	
<i>pthread_attr_t</i>	Thread attributes.
Functions	
<i>pthread_attr_init(pthread_attr_t *tattr);</i>	Initialize the attributes to their default values.
<i>int pthread_attr_destroy(pthread_attr_t *tattr);</i>	Destroy the attributes and free the allocated resources.
<i>int pthread_attr_setdetachstate(pthread_attr_t *tattr, int detachstate);</i>	This function sets the thread with <i>tattr</i> attributes in the detached state.
<i>int pthread_attr_getdetachstate(pthread_attr_t *tattr, int *detachstate);</i>	This function gets the thread create state which can be detached or joined.
<i>int pthread_attr_setguardsize(pthread_attr_t *tattr, size_t guardsize);</i>	With this function we can set the <i>guardsize</i> attribute of the <i>tattr</i> object.
<i>int pthread_attr_getguardsize(pthread_attr_t *tattr, size_t *guardsize);</i>	With this function we can get the <i>guardsize</i> attribute of the <i>tattr</i> object.
<i>int pthread_attr_setscope(pthread_attr_t *tattr, int scope);</i>	This function sets the scope of a thread either process private (intraprocess) or system wide (interprocess)
<i>int pthread_attr_getscope(pthread_attr_t *tattr, int *scope);</i>	This function gets the scope of a thread.
<i>int pthread_attr_setschedpolicy(pthread_attr_t *tattr, int policy);</i>	This function sets the scheduling policy: <i>SCHED_FIFO</i> , <i>SCHED_RR</i> (Round-Robin) or <i>SCHED_OTHER</i> .
<i>int pthread_attr_getschedpolicy(pthread_attr_t *tattr, int *policy);</i>	This function gets the scheduling policy.
<i>int pthread_attr_setinheritsched(pthread_attr_t *tattr, int inheritsched);</i>	This function sets the inherited scheduling policy: and priority if <i>inheritsched</i> value is <i>PTHREAD_INHERIT_SCHED</i> , the thread inherits scheduling policy and priority from its creator process and if <i>inheritsched</i> value is <i>PTHREAD_EXPLICIT_SCHED</i> , scheduling policy and priority to use are in the attributes structure.
<i>int pthread_attr_getinheritsched(pthread_attr_t *tattr, int *inheritsched);</i>	This function gets the inherited scheduling policy: and priority
<i>int pthread_attr_scheduledparam(pthread_attr_t *</i>	This function sets the scheduling parameters.

<i>tattr, const struct sched_param param);</i>	
<i>int pthread_attr_getschedparam(pthread_attr_t* tattr, const struct sched_param* param);</i>	This function gets the scheduling parameters.
<i>int pthread_attr_setstacksize(pthread_attr_t* tattr, size_t size);</i>	This function sets the thread stack size (in bytes). The constant <i>PTHREAD_STACK_MIN</i> defines the amount of stack space required to start a thread.
<i>int pthread_attr_getstacksize(pthread_attr_t* tattr, size_t *size);</i>	This function gets the thread stack size (in bytes).
<i>int pthread_attr_setstack(pthread_attr_t* tattr, void* stackaddr, size_t stacksize);</i>	This function sets the thread stack address and size.
<i>int pthread_attr_getstack(pthread_attr_t* tattr, void** stackaddr, size_t *stacksize);</i>	This function gets the thread stack address and size.
<i>int pthread_attr_setconcurrency(int new_level);</i>	This function is used to inform the system of the desired concurrency level.
<i>int pthread_attr_getconcurrency(void);</i>	This function returns the current concurrency level.

4.2.5. CONDITION VARIABLES

With condition variables it is possible to block threads until a particular condition is true. Condition variables are usually used together with a mutex lock.

The scheduling policy determines how blocking threads are awakened.

The attributes for condition variables must be set and initialized before the condition variables can be used or default values are used.

pthread_cond_wait() blocks the calling thread until the specified condition is signalled. This routine should be called after mutex is locked.

pthread_cond_signal routine is used to signal another thread waiting on the condition variable. This routine should be called before the associated mutex is unlocked.

pthread_cond_broadcast should be used if more than one thread is in a blocking wait state.

The condition variables	
Data Structures	
<i>pthread_condattr_t</i>	Attributes for the condition variable.
<i>pthread_cond_t*</i>	Condition variable.
Functions	
<i>pthread_condattr_init(pthread_condattr_t *cattr);</i>	Initialize attributes to their default values.
<i>int pthread_condattr_destroy(pthread_condattr_t *cattr);</i>	Destroy the attributes and free the allocated resources.
<i>int pthread_condattr_setpshared(pthread_condattr_t *cattr, int pshared);</i>	This function sets the scope of a either process private (intraprocess) or system wide (interprocess)
<i>int pthread_condattr_getpshared(pthread_condattr_t cattr, int* pshared)</i>	This function gets the current value of the scope.
<i>int pthread_condattr_setclock(pthread_condattr_t *cattr, clockid_t clock_id)</i>	With this function we can set the clock attribute. The clock attribute is the <i>clockID</i> of the clock used to measure the timeout service of <i>pthread_cond_timedwait</i> : it can be <i>CLOCK_REALTIME</i> or <i>CLOCK_MONOTONIC</i> .
<i>int pthread_condattr_getclock(pthread_condattr_t *cattr, clockid_t clock_id)</i>	With this function we can get the clock attribute value..
<i>pthread_cond_init(pthread_cond_t* cv, const pthread_condattr_t* cattr)</i>	With this function, we initialize the condition variable. If <i>cattr</i> is NULL, the default attributes are used.
<i>int pthread_cond_wait(pthread_cond_t* cv, pthread_mutex_t* mutex)</i>	This function blocks until the condition <i>cv</i> is signaled.
<i>int pthread_cond_signal(pthread_cond_t* cv,)</i>	This function unblocks one thread that is blocked on the condition variable <i>cv</i> .
<i>int pthread_cond_timedwait(pthread_cond_t* cv, pthread_mutex_t* mutex, const struct timespec* abstime)</i>	This function blocks until the condition <i>cv</i> is signaled. or the time <i>abstime</i> is reached.
<i>int pthread_cond_reltimedwait_np(pthread_cond_t* cv, pthread_mutex_t* mutex, const struct timespec* abstime)</i>	This function blocks until the condition <i>cv</i> is signaled. or the time <i>abstime</i> passed.
<i>int pthread_cond_broadcast(pthread_cond_t* cv)</i>	Calling this function unblocks all threads blocked on the condition.
<i>int pthread_cond_destroy(pthread_cond_t* cv)</i>	This function destroys all the resources used by the condition <i>cv</i> .

4.2.6. SEMAPHORES

To use semaphores you must include the file "semaphore.h". The semaphores of the POSIX threads library are counting semaphores. The semaphore count is initialized to the number of free resources. Threads then atomically increment the count when resources are added (function *sem_post*) and decrement the count when resources are removed (return from *sem_wait* or *sem_trywait*).

Semaphores	
Data Structures	
<i>sem_t</i>	<i>Semaphore</i> structure
Functions	
<i>int</i> <i>sem_init(sem_t *sem, int pshared, unsigned int value)</i>	Initializes a semaphore. If <i>pshared</i> is zero, the semaphore cannot be shared between processes.
<i>int</i> <i>sem_post(sem_t *sem)</i>	Increments the semaphore pointed to by <i>sem</i> .
<i>int</i> <i>sem_wait(sem_t *sem)</i>	The calling thread is blocked until the semaphore count becomes greater than zero.
<i>int</i> <i>sem_trywait(sem_t *sem)</i>	This function is the nonblocking version of the precedent.
<i>int</i> <i>sem_destroy(sem_t *sem)</i>	Destroys the semaphore.

4.2.7. SPINLOCKS

Spin locks are a low level synchronization mechanism. If a thread requires a spin lock already held by another thread, it spins in a loop to test if the lock is freed.

Spinlock	
Data Structures	
<i>pthread_spinlock_t</i>	Spin lock structure.
Functions	
<i>int pthread_spin_init(pthread_spinlock_t *lock, int pshared);</i>	Initializes a spin lock to an unlocked state. Possible values for <i>pshared</i> are: - <i>PTHREAD_PROCESS_SHARED</i> - <i>PTHREAD_PROCESS_PRIVATE</i> .
<i>int pthread_spin_lock(pthread_spinlock_t *lock);</i>	Use this function to lock a spin lock. If the lock is free, the calling thread gets it. If the lock is not free, the calling thread is blocked until it is free.
<i>int pthread_spin_trylock(pthread_spinlock_t *lock);</i>	This function is a non blocking version of the precedent. If the lock is not free, it return the error code <i>EBUSY</i> .
<i>int pthread_spin_unlock(pthread_spinlock_t *lock);</i>	The calling thread releases the locked spin lock.
<i>int pthread_spin_destroy(pthread_spinlock_t *lock);</i>	The spin lock is destroyed.

4.2.8. MUTEX

Mutex objects implement mutual exclusion locks. They ensure that only one thread at a time executes a critical section of code.

Mutex must be initialized before use and destroyed after. They have default attributes which can be modified.

Only one thread can lock or own a mutex variable at any time. When several threads compete for a mutex, the losers block at the call `pthread_mutex_lock` but a no blocking call is possible with `pthread_mutex_trylock`. Some functions allow to use a timeout.

Mutex Attributes	
Data Structures	
<i>pthread_mutexattr_t</i>	Mutex attributes.
Functions	
<i>int</i> <i>pthread_mutexattr_init(pthread_mutexattr_t *mattr);</i>	Initialize a mutex attributes structure with the default values for <i>pshared</i> , <i>type</i> , <i>protocol</i> , <i>robustness</i> : <i>PTHREAD_PROCESS_PRIVATE</i> <i>PTHREAD_MUTEX_DEFAULT</i> <i>PTHREAD_PRIO_NONE</i> <i>PTHREAD_MUTEX_STALLED_NP</i> <i>prioceiling</i> is inherited from the existing priority range.
<i>int</i> <i>pthread_mutexattr_destroy(pthread_mutexattr_t *mattr);</i>	Destroys the mutex attributes and free allocated resources..
<i>int</i> <i>pthread_mutexattr_setpshared(pthread_mutexattr_t *restrict mattr, int *restrict pshared);</i>	Sets the <i>pshared</i> attribute at the value <i>PTHREAD_PROCESS_PRIVATE</i> or <i>PTHREAD_PROCESS_SHARED</i> .
<i>int</i> <i>pthread_mutexattr_getpshared(pthread_mutexattr_t *restrict mattr, int *restrict pshared);</i>	Gets the <i>pshared</i> attribute.
<i>int</i> <i>pthread_mutexattr_settype(pthread_mutexattr_t *mattr, int type);</i>	Sets the mutex <i>type</i> attribute at the specified value. Possible values are: <i>PTHREAD_MUTEX_NORMAL</i> , <i>PTHREAD_MUTEX_ERRORCHECK</i> , <i>PTHREAD_MUTEX_RECURSIVE</i> , <i>PTHREAD_MUTEX_DEFAULT</i> .
<i>int</i> <i>pthread_mutexattr_gettype(pthread_mutexattr_t *restrict mattr, int *restrict type);</i>	Gets the mutex <i>type</i> attribute.
<i>int</i> <i>pthread_mutexattr_setprotocol(pthread_mutexattr_t *mattr, int protocol);</i>	Sets the mutex <i>protocol</i> attribute at the specified value. Possible values are: <i>PTHREAD_PRIO_NONE</i> , <i>PTHREAD_PRIO_INHERIT</i> , <i>PTHREAD_PRIO_PROTECT</i> .
<i>int</i> <i>pthread_mutexattr_getprotocol(const pthread_mutexattr_t *restrict mattr, int *restrict protocol);</i>	Gets the mutex <i>protocol</i> attribute.

<code>int pthread_mutexattr_setprioceiling(pthread_m utexatt_t *attr, int prioceiling) int pthread_mutexattr_getprioceiling(const pthread_mutexatt_t *restrict attr, int *restrict prioceiling);</code>	<p>Sets the mutex <i>priorityceiling</i> attribute. The priority ceiling defines the minimum priority level at which the critical section guarded by the mutex is executed.</p>
<code>int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t *restrict mutex, int *restrict prioceiling);</code>	<p>Returns the priority ceiling in <i>priorityceiling</i>.</p>

Mutex	
Data Structures	
<i>pthread_mutex_t</i>	Mutex structure
Functions	
<i>int</i> <i>pthread_mutex_init(pthread_mutex_t *restrict mp, const pthread_mutexattr_t *restrict mattr);</i>	Initializes the mutex pointed at by <i>mp</i> to its default value if <i>mattr</i> is NULL.
<i>int</i> <i>pthread_mutex_lock(pthread_mutex_t *mutex);</i>	This function locks the mutex pointed to by <i>mutex</i> . The calling thread becomes the mutex owner if it is free or it blocks until the mutex is free.
<i>int</i> <i>pthread_mutex_unlock(pthread_mutex_t *mutex);</i>	This function unlocks the mutex referenced by <i>mutex</i> . How the mutex is freed depends on the mutex's <i>type</i> attribute.
<i>int</i> <i>pthread_mutex_trylock(pthread_mutex_t *mutex);</i>	This function attempts to lock the mutex pointed by <i>mutex</i> and returns immediately if the mutex is already locked.
<i>int</i> <i>pthread_mutex_timedlock(pthread_mutex_t *restrict mutex, const struct timespec *restrict abs_timeout);</i>	This function attempts to lock the mutex pointed by <i>mutex</i> and returns after <i>timeout</i> is reached if the mutex is already locked.
<i>int</i> <i>pthread_mutex_reltimedlock_np(pthread_mutex_t *restrict mutex, const struct timespec *restrict rel_timeout);</i>	This function acts like the precedent but waits until a specified amount of time instead of a time.
<i>int</i> <i>pthread_mutex_destroy(pthread_mutex_t *mp);</i>	Destroys the mutex and free the allocated resources.
<i>int</i> <i>pthread_mutex_setprioceiling(pthread_mutex_t *restrict mutex, int prioceiling, int *restrict old_ceiling);</i>	This function modifies the priority ceiling.
<i>int pthread_mutex_getprioceiling(const pthread_mutex_t *restrict mutex, int *restrict prioceiling);</i>	This function gets the priority ceiling.

4.2.9. READ-WRITE LOCKS

A read-write lock is an object that can be locked in read or write mode. So concurrent reads and exclusive writes to a protected shared resource are possible.

Read-Write Locks	
Data Structures	
<i>pthread_rwlockattr_t</i>	Read-Write lock structure.
<i>pthread_rwlock_t</i>	Read-Write lock attributes.
Functions	
<i>int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);</i>	Initialize a read-write lock attribute structure.
<i>int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);</i>	Destroys a read-write lock attribute structure.
<i>int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr, int pshared);</i>	Use this function to set the process-shared read-write lock attribute. The <i>pshared</i> lock attribute can have one of the value: <i>PTHREAD_PROCESS_SHARED</i> or <i>PTHREAD_PROCESS_PRIVATE</i> .
<i>int pthread_rwlockattr_getpshared(const pthread_rwlockattr_t *restrict attr, int *restrict pshared);</i>	Use this function to get the process-shared read-write lock attribute.
<i>int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock, const pthread_rwlockattr_t *restrict attr);</i>	Initialize a read-write lock <i>rwlock</i> with the attributes <i>attr</i> . If <i>attr</i> is NULL, the default read-write lock attributes are used.
<i>int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);</i>	This function applies a read lock on the read-write lock object <i>rwlock</i> .
<i>int pthread_rwlock_timedrdlock(pthread_rwlock_t *restrict rwlock, const struct timespec *restrict abs_timeout);</i>	This function applies a read lock on the read-write lock object <i>rwlock</i> but if the lock cannot be acquired without waiting, this wait will be terminated when the specified <i>timeout</i> expires..
<i>int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);</i>	This function tries to apply a read lock on the read-write lock object <i>rwlock</i> but if the lock cannot be acquired it does not block but fails.
<i>int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);</i>	This function applies a write lock on the read-write lock object <i>rwlock</i> .
<i>int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);</i>	This function tries to apply a write lock on the read-write lock object <i>rwlock</i> but if the lock cannot be acquired it does not block but fails.

<pre>int pthread_rwlock_timedwrlock(pthread_rwlock_t *restrict rwlock, const struct timespec *restrict abs_timeout);</pre>	<p>This function applies a write lock on the read-write lock object <i>rwlock</i> but if the lock cannot be acquired without waiting, this wait will be terminated when the specified <i>timeout</i> expires..</p>
<pre>int pthread_rwlock_unlock (pthread_rwlock_t *rwlock);</pre>	<p>This function releases a lock on the read-write lock object <i>rwlock</i>.</p>
<pre>int pthread_rwlock_destroy(pthread_rwlock_t **rwlock);</pre>	<p>This function destroys a read-write lock and free its resources.</p>

4.2.10. BARRIERS

We use barriers when several tasks have to be completed before another task begins. The barrier must first be created with the number of threads that are synchronizing on the barrier. When the last thread reaches the barrier, all the waiting threads on the barrier resume their execution.

Barriers	
Data Structures	
<i>pthread_barrierattr_t</i>	Attributes for barriers
<i>pthread_barrier_t</i>	Barrier structure
Functions	
<i>int pthread_barrierattr_init(pthread_barrierattr_t *attr);</i>	This function initializes barrier attributes.
<i>int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr, int pshared);</i>	This function sets the process-shared attribute in a <i>pthread_barrierattr_t</i> object. The process-shared attribute can have the value: <i>PTHREAD_PROCESS_PRIVATE</i> or <i>PTHREAD_PROCESS_SHARED</i> .
<i>int pthread_barrierattr_getpshared(const pthread_barrierattr_t *restrict attr, int *restrict pshared);</i>	This function gets the process-shared attribute in a <i>pthread_barrierattr_t</i> object.
<i>int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);</i>	Use this function to destroy an initialized <i>pthread_barrierattr_t</i> object.
<i>int pthread_barrier_init(pthread_barrier_t *barrier, const pthread_barrierattr_t *restrict attr, unsigned count);</i>	This function allocates resources for a barrier and initialize its attributes. If <i>attr</i> is NULL, default attributes are used (process-share value is <i>PTHREAD_PROCESS_PRIVATE</i>). Count is the maximum number of threads expected to wait at the barrier.
<i>int pthread_barrier_wait(pthread_barrier_t *barrier);</i>	The calling thread blocks until the required number of threads have called this function.
<i>int pthread_barrier_destroy(pthread_barrier_t *barrier);</i>	Use this function to destroy a barrier and free its resources.

This document may not be used, copied or redistributed without authorisation of HLP Technologies

4.3. THE PROGRAMS OF EXAMPLE

Sample1 is a sequential program. It uses the card 1 configured with the file "Station1.cnf". After it has configured and connected the card, the program starts the bus arbitrator, reads the variable var2, writes 111 in the variable var1, displays the events, sends the message " ABCDEF " to the station2 and then loops until the user has entered the key 'q' or the station 2 has sent a message. By this time, the program reads and displays var2 and the state of the bus arbitrator.

Sample1 must be launched with a second station configured with "Station2.cnf" (for example, Sample5 or Sample5_pth).

Launch from a terminal window in the "Sample1" directory: "./Sample1".

Sample2 is a multi-threaded program. The main process creates two threads running the same function fipengine. This process creates too a semaphore, a spinlock and a barrier then it waits until the end of threads 1 and 2 on a pthread_join.

The barrier blocks the two threads until the bus arbitrator has started. Then, the two stations start a loop with a local counter. The thread 1 reads and displays the state of the bus arbitrator and waits for the semaphore. When unblocked, it reads var2 and displays its value. After, it writes the value of the counter in var1.

The thread 2 associated with card 2 runs a similar process but it increments the semaphore. So, in a tour of loop, each card reads and writes the associated variables.

Here are traces of this program execution

Card 1 is waiting semaphore

Thread 2 k = 251 Var Status= 0xa0 Trame :0x40 0x5 0x0 0xfe 0xfe 0xfe 0xfd

Thread 2 Var Value :0 254 254 254] Refresh:0x5

Thread 2 Value k= 251

Card 2 Send semaphore

Thread 1 k = 251 Var Status= 0xa0 Trame :0x40 0x5 0x0 0xfc 0xfc 0xfc 0xff

Thread 1 Var Value :0 252 252 252] Refresh:0x5

Thread 1 Value k= 251

BA Status c

BA is MSG_WND

Card 1 is waiting semaphore

Thread 2 k = 250 Var Status= 0xa0 Trame :0x40 0x5 0x0 0xfc 0xfc 0xfc 0xfd

Thread 2 Var Value :0 252 252 252] Refresh:0x5

Thread 2 Value k= 250

Loop Thread2 sends k =250

Card 2 Send semaphore

Thread 1 k = 250 Var Status= 0xa0 Trame :0x40 0x5 0x0 0xfc 0xfc 0xfc 0xff

Thread 1 Var Value :0 252 252 252] Refresh:0x5

Thread 1 Value k= 250

Loop Thread1 sends k =250

BA Status 8

BA is SENDING

Sample5 is a sequential and interactive program. The user must enter the card number and this card is configured with the associated .cnf file. Most of FipEngine commands can be executed in this program but the user must take care of consistency, (for example, you must connect the network before starting the bus arbitrator and before any other action).

Sample5_ptb is the multi-threaded version of sample5.. The main process creates only one thread.