FIP DEVICE MANAGER
Software Version 4 User Reference Manual

ALS 50278 e-en

First issue: 12-1996
This edition: 01-2001

Meaning of terms that may be used in this document / Notice to readers

WARNING

Warning notices are used to emphasize that hazardous voltages, currents, temperatures, or other conditions that could cause personal injury exist or may be associated with use of a particular equipment.

In situations where inattention could cause either personal injury or damage to equipment, a Warning notice is used.

Caution

Caution notices are used where there is a risk of damage to equipment for example.

Note

Notes merely call attention to information that is especially significant to understanding and operating the equipment.

Index letter	Date	Nature of revision
b	03-1997	Major updates
c	04-2000	Update
d	09-2000	Software revision 4.9 Transmission rate 5 Mbits/s
e	01-2001	Updates throughout the document

Revisions

1. PURPOSE OF MANUAL AND DOCUMENTED VERSION

This document is the user manual for the FIP DEVICE MANAGER software, version 4. This software is one of the components of FIPWARE, a comprehensive technological solution for developing connections to the WorldFIP network. This FIPWARE, developed by ALSTOM Technology, comprises hardware components, software components, development tools and miscellaneous accessories.

As part of this FIPWARE, FIP DEVICE MANAGER is a function library designed to be borne by products connected to the WorldFIP network to facilitate access.

The purpose of this document is to describe all the features offered by this software, as well as the procedures for implementing and using them. Therefore, it describes how to use FIP DEVICE MANAGER Version 4 in order to:

- compile it for a specific target using the various compilers;
- use it to initialize a subscriber and configure the communication exchanges;
- use it to implement the bus arbitrator;
- use it to access the MPS variables and to send and receive messages.

In no case whatsoever may this document be considered a manual for learning the concepts and principles underlying the operation of a WorldFIP network.

Users of this software must already be familiar with the basic mechanisms of a WorldFIP network.

FIP DEVICE MANAGER software Version 4 is designed for developers who develop interfaces with their communication application and with their run-time environment.

2. CONTENT OF THIS MANUAL

This user manual is broken down as follows:

Chapter 1 - General introduction: provides an overview of the software. It describes the supported features and the procedures for using the software.

Chapter 2 – Available functions: contains a detailed description of the available features.

Chapter 3 - Description of user interface primitives: contains a detailed description of each primitive available, as well as of those called by FIP DEVICE MANAGER, to be written by the user.

Chapter 4 – Performance levels: specifies the performances on the user interface and on the line.

Chapter 5 – Error codes: describes the error codes returned following primitive calls.

Chapter 6 – Installation procedure: describes the installation procedures.

Appendix A – Compatibility: describes the compatibility between Version 2 and Version 4 primitives.

Appendix B – Use with C++: describes the FDM use in a C++ environment.

Glossary

3. RELATED PUBLICATIONS

For more information, refer to these publications:

[1] EN50170 partie 3: Normes WorldFIP

[2] FIP Network General Introduction ALS 50249

[3] FIELDUAL User Manual ALS 50273

[4] FIELDRIVE User Reference Manual ALS 50261

[5] FULLFIP2 Component User Reference Manual ALS 50262

[6] Manuel de référence utilisateur du logiciel FIPCODE V6 ALS 50277

4. WE WELCOME YOUR COMMENTS AND SUGGESTIONS

ALSTOM strives to produce quality technical documentation. Please take the time to fill in and return the "Reader's Comments" page if you have any remarks or suggestions.

Reader's comments

ALS 50278 e-en	***FIP DEVICE MANAGER Software Version 4 User Reference Manual***

Your main job is:

- ❑ System designer
- ❑ Distributor
- ❑ System integrator
- ❑ Installer
- ❑ Programmer
- ❑ Maintenance
- ❑ Operator
- ❑ Other (specify below)

If you would like a personal reply, please fill in your name and address below:

COMPANY:.....NAME:.....

ADDRESS:.....

.....COUNTRY:.....

Send this form directly to your ALSTOM sales representative or to this address:

ALSTOM Technology
Technical Documentation Department (TDD)
23-25, Avenue Morane Saulnier
92364 Meudon la Forêt Cedex
France
Fax: +33 (0)1 46 29 10 21

All comments will be considered by qualified personnel.

REMARKS

Reader's comments

Contents

CHAPTER 1 - GENERAL INTRODUCTION

CHAPTER 2 - AVAILABLE FUNCTIONS

Contents

Contents

Tables

Chapter 1 General introduction

1. GENERAL OVERVIEW OF THE SOFTWARE

The FIP DEVICE MANAGER software, Release 4, designed to operate with the FULLFIP2 coprocessor and Release 6 of its FIPCODE microcode, takes the form of a library of primitives to be linked to the user application. Run on the host microprocessor of the communication coprocessor, it can transform the equipment upon which it is integrated into a WorldFIP subscriber, active as a station, bus arbitrator or both.

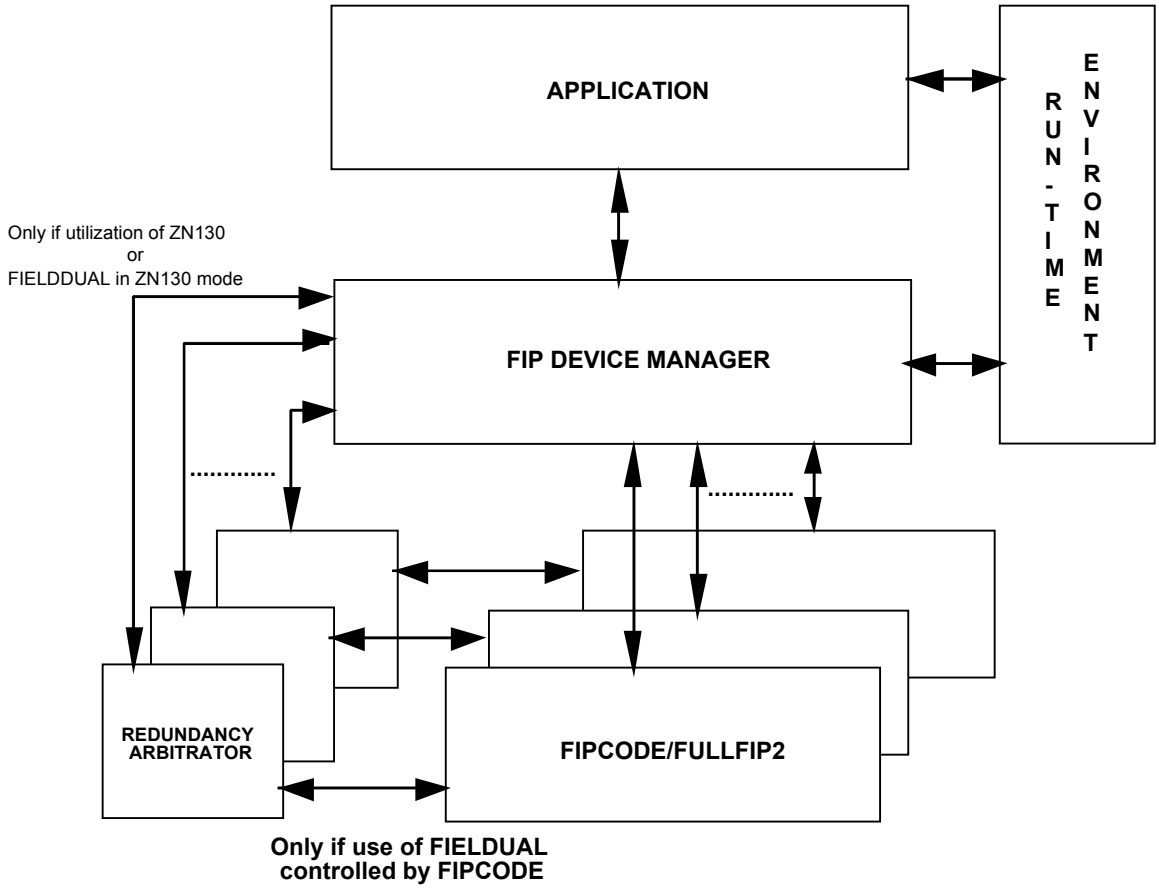
This library contains all the functions required to manage medium redundancy in the case of a hardware target that integrates the FIELDUAL component. All the diagnosis functions of the components belonging to the WorldFIP connection point (excluding the transmission/reception self-test), and the functions to check the integrity of the configuration data of the communicating subscriber, are performed by this software on start-up, and the whole time the subscriber is working.

In relation to the European standard applicable to WorldFIP (EN50170), the interface offered by FIP DEVICE MANAGER Release 4 is at the application layer level for bus arbitrating and variable exchange services, and at the data link layer level for message exchanges. The basic network management functionality is also integrated.

This library is strictly written in ANSI C and ANSI C++ so that it can be used in a large number of environments. It is also strictly independent of all utilization contexts and contains no assumptions made about the run time model of the user application. The user can write the application in C or C++.

The possible implementation in the form of a driver in a given context is the responsibility of the user, who integrates the FIP DEVICE MANAGER library into this context. However, the mechanisms proposed by this library optimize this task thanks, for example, to an attempt to provide a standard interface with a real-time kernel.

FIP DEVICE MANAGER Release 4 can be used to run several FULLFIP2 systems.



2. OVERVIEW OF TARGET HARDWARE CONFIGURATIONS

The WorldFIP network connection points managed by FIP DEVICE MANAGER Release 4 can be built in several ways as regards management of medium redundancy and microprocessor access to the database (i.e. variables and messages) managed by FIPCODE/FULLFIP2.

The following components may be used to help in managing medium redundancy: see [3] for details.

- ZN130 or FIELDUAL in ZN130 mode: intervention of FIP DEVICE MANAGER,
- FIELDUAL in new mode controlled by FULLFIP2/FIPCODE: no intervention of FIP DEVICE MANAGER therefore no code run on the host micro to do this.

The FIP DEVICE MANAGER is informed of the type of management selected by means of a compilation option (see Chapter 6, Section 3.).

The database managed by FULLFIP2/FIPCODE can be accessed in two ways, both dependent on the hardware architecture. See [5] for details.

- via the FULLFIP2/FIPCODE command user interface,
- via access to a memory shared between FULLFIP2/FIPCODE and the host microprocessor. The latter thus has direct access to the database containing the objects it must manipulate (variables, messages, etc.).

The FIP DEVICE MANAGER is informed of the type of access selected by means of a compilation option (see Chapter 6, Section 3.).

3. PRESENTATION OF THE FUNCTIONALITY

The FIP DEVICE MANAGER Release 4 software functionality includes the following tasks:

- **Ensuring system operation**

  This functionality is dedicated to the management of components required to build WorldFIP connection points, i.e. FULLFIP2, medium redundancy circuits and private memories. It consists of the following basic functions:

  - configure and manage one or N FULLFIP2 components,
  - carry out the self-tests of the WorldFIP connection point(s) (optional),
  - process events coming from the network(s),
  - manage memory resources,
  - develop the various time-outs required for other functions,
  - monitor the quality of the medium(s),
  - prevent degradation,
  - prevent latent faults (in double-medium),
  - allow the subscriber to run as FIPIO manager.

- **Managing AE/LE and the MPS variables**

  This functionality enables the application layer services to be performed, which enables the user to have access to the MPS variables through unconnected logic entities called AE/LE. It consists of the following basic functions:

  - create and delete an MPS AE/LE,
  - startup and stop an MPS AE/LE,
  - add a variable in an MPS AE/LE,
  - read a variable consumed in an AE/LE with or without dynamic refresh status,
  - write a variable produced in an AE/LE with or without dynamic refresh status,
  - manage supposed ”pure sync” variables.

- **Managing the bus arbitrator (S)**

  This functionality consists of the following basic functions:

  - load a macrocycle,
  - start or stop a macrocycle,
  - change a macrocycle,
  - resynchronize a macrocycle,
  - manage election of a bus arbitrator.

- **Handling SM-MPS network management variables**

  This functionality enables network management services to be performed and provides the user with access to the SM-MPS variables. It consists of the following basic functions:

  - management of the ”list of equipment present” variables,
  - management of the presence variable,
  - management of the identification variable,
  - management of the report variable,
  - management of the BA sync variable.

- **Managing ”WORLDFIP DATA LINK LAYER” messages**

  This functionality enables the data connection layer services to be performed. These services provide the user with access to messages at ”DLL WorldFIP” level. The functionality consists of the following basic functions:

  - configure messaging,
  - transmit a message,
  - receive a message.

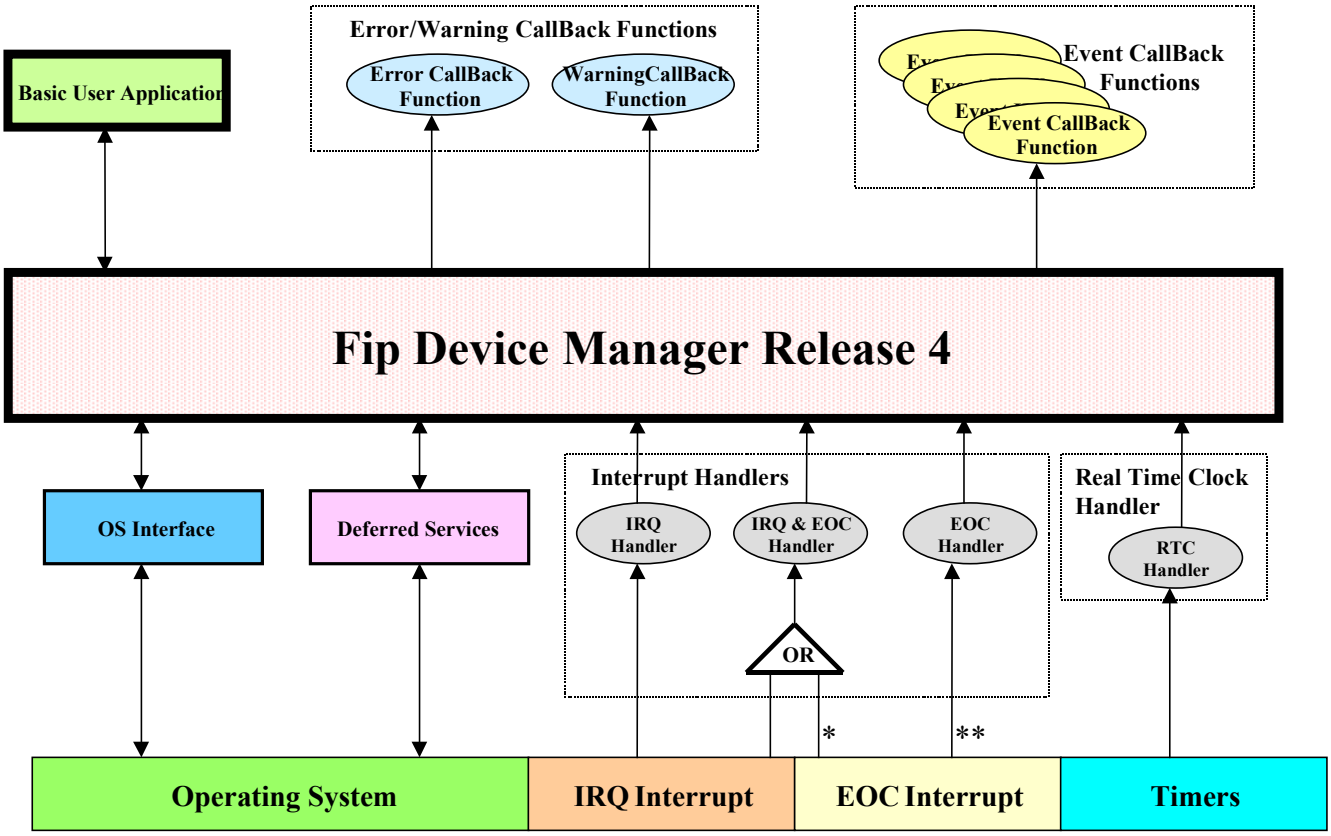
- **Managing the time**

  This functionality consists of the following basic functions:

  - managing election of time variable producer,
  - time variable read,
  - time variable write.

4. OVERVIEW OF THE INTERFACE WITH THE USER APPLICATION

The environment of an application that uses FIP DEVICE MANAGER Release 4 is the following:



* if IRQ and EOC are wired to form an OR gate then EOC should be configured in level mode
** EOC can be configured either in level mode or in pulse mode

Writing a FIP DEVICE MANAGER application means that you have to:

- call the basic FDM services (see Subsection 4.1 of this chapter),
- write the error and warning callback functions (see Subsection 4.2 of this chapter),
- adapt if necessary, some system-dependent objects (semaphores, mutex, etc.) that FDM code uses (see Subsection 4.3 of this chapter),
- write the FULLFIP component interrupt handlers (IRQ and EOC) (see Subsection 4.4 of this chapter),
- write the event callback functions (see Subsection 4.5 of this chapter),
- write the real-time clock handler (see Subsection 4.6 of this chapter),
- implement the mechanisms of the deferred services if you use these services (see Subsection 4.7 of this chapter):
  - access to the SM_MPS variables,
  - access to the aperiodic MPS variables,
  - transmission of messages,
  - reception of messages,
- implement the mechanism for the periodic MPS indications if you create periodic MPS variables (see Subsection 4.8 of this chapter).

4.1 Description of the basic services

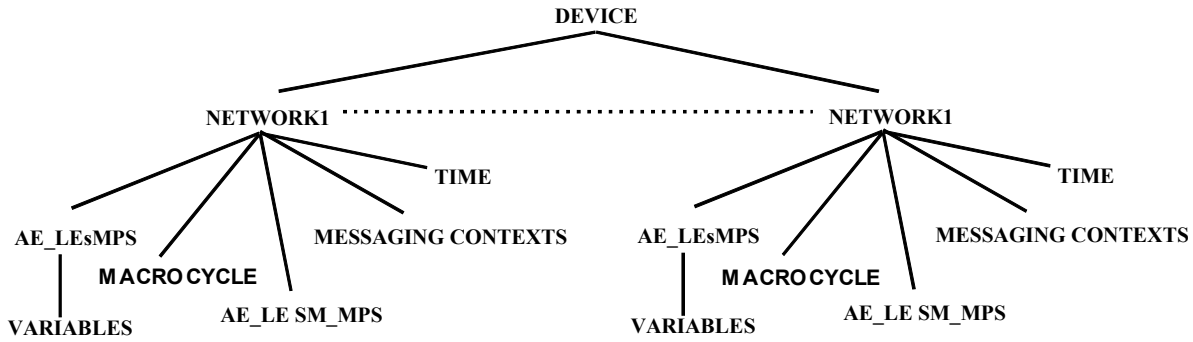
The FIP DEVICE MANAGER Release 4 programming interface contains all the equipment required to set up the network(s), to provide network management facilities and to offer communication services.

This programming interface is arranged around different objects, which can be manipulated using a set of functions, and which are associated with different aspects of network functionality.

The functions provide everything required to:

- create or delete objects, in accordance with their dependency rules,
- modify objects in accordance with the possibilities offered,
- manage operating modes in accordance with the protocol,
- have access to the MPS variables and to transmit and receive messages to communicate.

The list of objects which can be manipulated and their interdependence are shown below:



The indentation which shows the interdependence of the objects indicates, for example, that a variable object required for MPS variable exchanges, can only be created in the context of an existing AE_LE, which itself can only be created in the context of an existing network.

All the creation primitives refer to the object created which must be specified as a parameter of all the primitives that manipulate the object.

The primitives for each object are:

DEVICE: equipment that can control several WorldFIP networks. The primitives regard initialization and overall management of the software.

`fdm_initialize)`	Creation of the device
`fdm_get_version()`	Access to the version of the software
`fdm_ticks_counter()`	Management of internal timing

NETWORK: a WorldFIP network (instance of software to manage it)

`fdm_initialize_network()`	Creation and start-up of network
`fdm_stop_network()`	Deletion of network
`fdm_valid_médium()`	Initialization of medium redundancy management
`fdm_ack_ewd_medium()`	Acknowledgment of the Watch-dog default
`fdm_online_test()`	Activation of online self-tests
`fdm_process_its_fip()`	Processing of IRQ and EOC interrupts
`fdm_process_it_irq()`	Processing of IRQ interrupts
`fdm_process_it_eoc`	Processing of EOC interrupts
`fdm_switch_image()`	Changing the image upon which all the network AE_LEs function
`fdm_get_image()`	Obtaining the image upon which all the network AE_LEs function
`fdm_generic_time_initialize()`	Initialization of time management
`fdm_generic_time_set_priority()`	Setting of the subscriber priority in the time producer election mechanism
`fdm_generic_time_set_candidate_for_election()`	Including or not including the subscriber in the time producer election mechanism
`fdm_get_local_report()`	Local report variable read
`fdm_change_test_medium_ticks()`	Modification of the medium test run time
`fdm_mps_fifo_empty()`	Emptying of aperiodic MPS events
`fdm_smmps_fifo_empty()`	Emptying of SMMPS events
`fdm_msg_send_fifo_empty`	Activation of message transmission function
`fdm_msg_rec_fifo_empty`	Activation of message reception function

AE_LE MPS: Comprehensive application variable units with their projection on the data link layer

`fdm_ae_le_create()`	Creation of an AE_LE
`fdm_ae_le_delete()`	Deletion of an AE_LE
`fdm_ae_le_start()`	Start-up of an AE_LE
`fdm_ae_le_stop()`	Stopping of an AE_LE
`fdm_ae_le_get_state()`	Obtaining the state of an AE_LE

VARIABLES: MPS Variables.

`fdm_mps_var_create()`	Creation of an AE_LE
`fdm_mps_var_change_id()`	Modification of the identifying attribute of a variable
`fdm_mps_var_change_periods()`	Modification of the period attributes of a variable
`fdm_mps_var_change_priority()`	Modification of the priority attribute of a variable
`fdm_mps_var_change_prod_cons()`	Modification of the producer consumer attribute of a variable
`fdm_mps_var_change_RQa()`	Modification of the authorized MPS query attribute of a variable
`fdm_mps_var_change_MSGa()`	Modification of the authorized messaging query attribute of a variable
`fdm_mps_var_write_loc()`	Local variable write
`fdm_mps_var_write_universal()`	Universal variable write
`fdm_mps_var_time_write_loc()`	Local variable write with time
`fdm_mps_var_read_loc()`	Local variable read
`fdm_mps_var_read_universal()`	Universal variable write
`fdm_mps_var_time_read_loc()`	Local variable read with time

AE_LE SM-MPS: All network management variables

`fdm_read_present_list()`	*List of equipment present* variables read
`fdm_read_presence()`	Presence variable read
`fdm_read_report()`	Report variable read
`fdm_read_identification()`	Identification variable read

BUS ARBITRATOR/MACROCYCLE: bus arbitrating function and bus arbitrator program

`fdm_ba_load_macrocycle_fipconfb()`	Loading of a macrocycle built with BA_BUILDER
`fdm_ba_load_macrocycle_manual()`	Loading of a macrocycle built manually
`fdm_ba_delete_macrocycle()`	Deleting of a macrocycle
`fdm_ba_commute_macrocycle()`	Changing of a macrocycle
`fdm_ba_loaded()`	Identification variable read
`fdm_ba_start()`	Start-up of BA function
`fdm_ba_stop()`	Stopping of BA function
`fdm_ba_status()`	BA function status read
`fdm_ba_set_parameters()`	Initialize BA timer
`fdm_ba_set_priority()`	Initialize BA priority
`fdm_ba_external_resync()`	External sync request

MESSAGING CONTEXT:

`fdm_messaging_fullduplex_create`	Creation of messaging context to transmit and receive
`fdm_messaging_to_send_create`	Creation of messaging context to transmit
`fdm_messaging_to_rec_create`	Creation of messaging context to receive
`fdm_channel_create`	Creation of an aperiodic messaging channel
`fdm_channel_delete`	Deletion of an aperiodic messaging channel
`fdm_change_messaging_ acknowledge_type()`	Modification of type of messaging (ACK/NOACK) being transmitted
`fdm_msg_ref_buffer_free`	Freeing the part of the memory containing a message received descriptor
`fdm_msg_data_buffer_free`	Freeing the part of the memory containing a message received
`fdm_msg_send_message`	Request to transmit a message

TIME:

`fdm_generic_time_write_loc()`	Time variable write
`fdm_generic_time_read_loc()`	Time variable read

4.2 Management of the reports and ERROR/WARNING callback functions

Whenever there is a fault of whatever kind, all the user interface primitives return reports with the following standard format:

FDM_OK: if the function was carried out correctly,

FDM_NOK: if an error is detected

except for:

- functions for creating different objects,
- functions implemented in the form of macros: `fdm_get_image, fdm_ba_loaded,`
- functions not associated with a FIP DEVICE MANAGER instance: `fdm_get_version`, `fdm_initialization`, `fdm_initialize_network, fdm_ticks_counter,`
- activation functions: `fdm_mps_fifo_empty, fdm_smmps_fifo_empty, fdm_msg_send_fifo_empty,` et `fdm_msg_rec_fifo_empty,`

Any errors detected by the FIP DEVICE MANAGER primitives are classified in two families: WARNING type errors which do not call the correct operation of the FIP DEVICE MANAGER and the WorldFIP connection into question, and FATAL_ERROR type errors which do.

When the FIP DEVICE MANAGER detects a FATAL_ERROR the `User_Signal_Fatal_Error()` callback function is called. The user provides this function to FDM when he calls the *fdm_initialize_network* function (field of *FDM_CONFIGURATION_HARD* structure - see Chapter 3).

In this procedure, the user must take care to restore all the memory allocated by deleting all the objects created one by one. This procedure must end with the `fdm_stop_network()` function being called (which itself calls the user function `User_Reset_Composent()` to reset FULLFIP2).

The circuit is thus reset: nothing more is transmitted or received from the network and start-up can only occur after reconfiguration (using `fdm_initialize_network()`).

To do so, you must remember to restore, if necessary, any reserved memory, for example in the `User_Signal_Fatal_Error` primitive.

When the FIP DEVICE MANAGER detects a WARNING error, the user callback function `User_Signal_Warning()` is called. The user provides this function to FDM when he calls the *fdm_initialize_network* function (field of *FDM_CONFIGURATION_HARD* structure - see Chapter 3). After this function is called, FIP DEVICE MANAGER continues in sequence.

Following the detection of an error (WARNING) and after the corresponding user function has been called, the function concerned immediately sends the 0xffff report to the caller.

These two callback functions use an input parameter of the *FDM_ERROR_CODE* type described in the *fdm_initialize_network* function (see Chapter 3). The error/warning codes are described in Chapter 5.

4.3 Interface with the operating system

4.3.1 Management of mutual exclusion

This management system involves the mutual exclusion required to protect the following:

- access to the FULLFIP2 circuit, which is non-reentrant: only one user command can be activated at any one time for each FULLFIP2,
- some means of access to the internal FIP DEVICE MANAGER database,
- certain portions of the FIP DEVICE MANAGER code,
- access to the variables (in free access mode).

This is managed with the help of four different semaphores (for each FIP DEVICE MANAGER instance) which makes it possible to separate the means of access.

Flags are used as it involves the appropriate software tools, which are the most frequently available when using a real-time kernel.

However, they can be replaced by (i.e. the corresponding macro can refer to) other mechanisms in order to achieve this mutual exclusion (e.g. masking/unmasking tasks using the primitives which call the flags in question).

These semaphores, manipulated by the FIP DEVICE MANAGER primitives, are to be provided by the user via a set of macros, which he must specify in accordance with his environment.

The files *fdm_os.c* and *fdm_os.h* provide a set of macros for the operating systems pSOS and VxWorks. If you use pSOS or VxWorks and if you want to change these macros or if you use another operating system then you have to change or adapt these macros.

`OS_fdm_sm_create`	To create the semaphores (one call for each)
`OS_fdm_sm_delete`	To delete the semaphores (one call for each)
`OS_fdm_sm_p`	To take the semaphore needed to protect access to FULLFIP2
`OS_fdm_sm_v`	To free the semaphore needed to protect access to FULLFIP2
`OS_fdm_sm_p_bd`	To take the semaphore needed to protect access to FIP DEVICE MANAGER data base
`OS_fdm_sm_v_bd`	To free the semaphore needed to protect access to FIP DEVICE MANAGER data base
`OS_fdm_sm_p_t`	To take the semaphore needed to protect certain portions of FIP DEVICE MANAGER code
`OS_fdm_sm_v_t`	To free the semaphore needed to protect certain portions of FIP DEVICE MANAGER code
`OS_fdm_sm_p_vcom`	To take the semaphore needed to protect access to the variable (free access)
`OS_fdm_sm_v_vcom`	To free the semaphore needed to protect access to the variable (free access)

In addition FIP DEVICE MANAGER calls a procedure which enables it to place the processor it is working on into a state of total non-preemption. It can do this for very short periods of time (a few microinstructions) to protect itself from certain instances of concurrent access (updating pointer without using semaphores). At the same time, the procedure to return to the initial state is also called.

These procedures are:

```
OS_Enter_Region()
```

```
OS_Leave_Region()
```

These procedures can be specified by the user in the form of macros.

The following table shows the FDM functions that use the above primitives.

Note

CA: Classical Access of the host processor to the data base using FULLFIP2 IO ports

FA: Free Access of the host processor to the data base shared between the host processor and FULLFIP2

	OS_sm_v OS_sm_p		OS_sm_v_t OS_sm_p_t		OS_sm_v_bd OS_sm_p_bd		QS_sm_v_vcom OS-sm_p_vcom		OS_enter_region OS_leave_region	
	CA	FA	CA	FA	CA	FA	CA	FA	CA	FA
fdm_initialize										
fdm_get_version										
fdm_ticks_counter	**X**	**X**								
fdm_change_test_medium_ticks										
fdm_initialize_network	**X**	**X**	**X**	**X**					**X**	**X**
fdm_stop_network									**X**	**X**
fdm_valid_medium	**X**	**X**								
fdm_online_test	**X**	**X**						**X**		
fdm_process_its_fip	**X**	**X**								
fdm_process_it_eoc	if status									
fdm_process_it_irq	**X**	**X**								
fdm_ba_load_macrocycle_fipconfb	**X**	**X**								
fdm_ba_load_macrocycle_manual	**X**	**X**								
fdm_ba_delete_macrocycle										
fdm_ba_external_resync	**X**									
fdm_ba_start	**X**	**X**						**X**		
fdm_ba_commute_macrocycle	**X**	**X**						**X**		
fdm_ba_set_priority	**X**	**X**						**X**		
fdm_ba_set_parameters	**X**	**X**						**X**		

	OS_sm_v OS_sm_p		OS_sm_v_t OS_sm_p_t		OS_sm_v_bd OS_sm_p_bd		QS_sm_v_vcom OS-sm_p_vcom		OS_enter_region OS_leave_region	
	CA	FA	CA	FA	CA	FA	CA	FA	CA	FA
fdm_ba_status										
fdm_ba_stop	**X**	**X**								
fdm_ba_loaded										
fdm_read_report	**X**	**X**	**X**	**X**	**X**				**X**	**X**
fdm_read_present_list	**X**	**X**							**X**	**X**
fdm_read_identification	**X**	**X**	**X**	**X**	**X**				**X**	**X**
fdm_read_presence	**X**	**X**	**X**	**X**	**X**				**X**	**X**
fdm_read_ba_synchronize	**X**									**X**
fdm_get_local_report										
fdm_switch_image			**X**	**X**						
fdm_get_image										
fdm_ae_le_create										
fdm_ae_le_start	**X**	**X**	**X**	**X**	**X**				**X**	**X**
fdm_ae_le_delete										
fdm_ae_le_stop	**X**	**X**	**X**	**X**	**X**				**X**	**X**
fdm_ae_le_get_state										
fdm_mps_var_create										
fdm_mps_var_change_periods	**X**				**X**					
fdm_mps-var_change_RQa										
fdm_mps_var_change_MSGa										
fdm_mps_var_change_priority										
fdm_mps_var_change_prod_cons	**X**				**X**					
fdm_mps_var_write_loc	**X**	**X**						**X**		
fdm_mps_var_read_loc	**X**	**X**						**X**		
fdm_mps_var_time_write_loc	**X**	**X**						**X**		
fdm_mps_var_time_read_loc	**X**	**X**						**X**		
fdm_mps_var_write_universal	**X**	**X**						**X**	**X**	**X**
fdm_mps_var_read_universal	**X**	**X**							**X**	**X**
fdm_generic_time_initialize	**X**	**X**	**X**	**X**	**X**				**X**	**X**
fdm_generic_time_read_loc	**X**	**X**						**X**		
fdm_generic_time_write_loc	**X**	**X**						**X**		
fdm_generic_time_set_candidate_for_election										
fdm_generic_time_set_priority										
fdm_ generic_time _delete	**X**	**X**	**X**	**X**	**X**				**X**	**X**
fdm_channel_create	**X**		**X**	**X**	**X**				**X**	**X**
fdm_channel_delete	**X**		**X**	**X**	**X**				**X**	**X**
fdm_change_channel_nr										

	OS_sm_v OS_sm_p		OS_sm_v_t OS_sm_p_t		OS_sm_v_bd OS_sm_p_bd		QS_sm_v_vcom OS-sm_p_vcom		OS_enter_region OS_leave_region	
	CA	FA	CA	FA	CA	FA	CA	FA	CA	FA
fdm_messaging_fullduplex_create	**X**		**X**	**X**	**X**				**X**	**X**
fdm_messaging_to_send_create	**X**		**X**	**X**	**X**				**X**	**X**
fdm_messaging_to_rec_create	**X**		**X**	**X**	**X**				**X**	**X**
fdm_messaging_delete	**X**		**X**	**X**	**X**				**X**	**X**
fdm_send_message									**X**	**X**
fdm_msg_ref_buffer_free									**X**	**X**
fdm_msg_data_buffer_free									**X**	**X**
fdm_msg_rec_fifo_empty									**X**	**X**
fdm_mps_fifo_empty	**X**	**X**							**X**	**X**
fdm_smmps_fifo_empty	**X**	**X**							**X**	**X**
fdm_msg_send_fifo_empty	**X**	**X**							**X**	**X**

4.3.2 Memory management

This management system affects the allocation and freeing of memory zones carried out dynamically by the FIP DEVICE MANAGER.

It is carried out by means of macros, which must be specified by the user in accordance with his environment:

```
OS_Allocate (Memory_Region, type, PA, dim))

OS_Free (Memory_Region,PF)
```

Where:

`Memory_Region`:	Identification of the memory region used. Useful for working with real-time operUseless for DOS for example
`type:`	Name of type of object requested, i.e. of the object which will be contained in the allocated memory zone
`PA:`	Address of the pointer which will contain the address of the allocated zone
`dim:`	Size in bytes of zone requested
`PF:`	Address of the zone to be freed.

This mechanism enables the FIP DEVICE MANAGER to always dynamically manage the memory it needs, while offering the user the opportunity to completely control behavior.

FIP DEVICE MANAGER only uses these mechanisms during the configuration phases.

The files *fdm_gdm.c* and *fdm_gdm.h* provide a set of macros for the operating systems pSOS and VxWorks. If you use pSOS or VxWorks and if you want to change these macros or if you use another operating system then you have to change or adapt these macros.

4.4 Processing IRQ and EOC interrupts

The FULLFIP component issues 2 interrupts:

- **IRQ** - when one of the following events occurs:
  - reception or transmission of an MPS universal-type variable,
  - reception of one of the SM_MPS network management variables, following a request to read this variable,
  - transmission of a produced MPS variable with associated event (A_Sent),
  - reception of a consumed MPS variable with associated event (A_Received),
  - transmission of a produced MPS variable of the universal type or at the time-out of the request,
  - reception of a message,
  - reception of acknowledgement of transmission of a message,
  - start of the TEST_P instruction by a BA, in the case of utilization of ZN130 components or FIELDUAL components in ZN130 mode,
  - stop of the TEST_P instruction by a BA, in the case of utilization of ZN130 components or FIELDUAL components in ZN130 mode,
- **EOC** - when the following event occurs:
  - reception of a synchronization variable (see Subsection 1.3.4.4 of Chapter 2).

The user has to connect the interrupt vectors to the interrupt handlers. This operation depends on the user hardware environment and operating system.

Depending on the hardware environment the user can implement 4 types of interrupt handlers:

- **IRQ Handler:** process of the IRQ interrupt only,
- **EOC in level mode Handler:** process of the EOC interrupt only and if EOC is configured in level mode,
- **EOC in pulse mode Handler:** process of the EOC interrupt only and if EOC is configured in pulse mode,
- **IRQ and EOC in pulse mode Handler:** process of the IRQ and EOC interrupts when they are wired up to form an OR gate and EOC is configured in level mode.

Note

The setting of the pulse mode of EOC is performed when calling the *fdm_initialize_network* function and using the flag EOC_PULSE_MODE of the field *Type* of the structure FDM_CONFIGURATION_SOFT.

1. **IRQ Handler:** process of the IRQ interrupt only

   The interrupt handler processing time has to be short. Therefore, the IRQ interrupt handler has to activate a task. In the body of this task you can process:

   - one event at a time using FDM function *fdm_process_it_irq*(Network_Identification),

     or

   - all the waiting events (produced by IRQs) using the following FDM macro sequence:
     - BEGIN_SMAP_FIP_EVT,
     - SMAP_ON_IRQ(Network_Identification),
     - END_SMAP_FIP_EVT.

2. **EOC in level mode Handler:** process the EOC interrupt if and only if EOC is configured in level mode

   The interrupt handler processing time has to be short. For that the EOC interrupt handler has to activate a task. In the body of this task you can process:

   - one event at a time using FDM function *fdm_process_it_eoc*(Network_Identification),

     or

   - all the waiting events (produced by EOCs) using the following FDM macro sequence:
     - BEGIN_SMAP_FIP_EVT,
     - SMAP_ON_EOC(Network_Identification),
     - END_SMAP_FIP_EVT.

3. **EOC in pulse mode Handler:** process the EOC interrupt if and only if EOC is configured in pulse mode (see Subsection 4.8 of this chapter).

   In this case you don't have to call any FDM functions and you can, for example, activate a task that waits for this synchronization event.

4. **IRQ and EOC in pulse mode Handler:** process the IRQ and EOC interrupts when they are wired up to form an OR gate and EOC is configured in level mode.

   The interrupt handler processing time has to be short. Therefore, the IRQ and EOC interrupt handler has to activate a task. In the body of this task you can process:

   - one event at a time using FDM function *fdm_process_its_fip*(Network_Identification),

     or

   - all the waiting events (produced by IRQs or EOCs) using the following FDM macro sequence:
     - BEGIN_SMAP_FIP_EVT,
     - SMAP_ON_IRQ_EOC(Network_Identification),
     - END_SMAP_FIP_EVT.

Network Identification: FDM_REF type; this handle is returned by the function *fdm_initialize_network*.

Example of implementation

```
        /* for example in the case of pSOS */
FDM_REF      * Network Identification; /* this handle is returned by the function */
        /* fdm_initialize_network. */
static unsigned long  sm_irq;              /* semaphore */
        /* create a task  and a semaphore */
t_create("TIRQ", 100, 4096, 0, T_LOCAL | T_NOFPU, &TaskIRQ);
sm_create("SIRQ", 0, SM_LOCAL | SM_PRIOR, &sm_irq);

        /* IRQ interrupt handler */
static void IRQ_handler(void)
{
   sm_v(sm_irq);        /free the semaphore → wake up of the task TaskIRQ */
}
        /* task that treat the interrupt IRQ */
static void TaskIRQ(void)
{
      for(;;){
            BEGIN_SMAP_FIP_EVT;
            SMAP_ON_IRQ(Network Identification);
            END_SMAP_FIP_EVT;
                  /* wait the next interrupt */
                  sm_p(sm_irq, SM_WAIT, 0);   /* take the semaphore */
      }
}
```

4.5 Processing events

Events are signaled by an interrupt from the circuit. This interrupt must be processed as indicated in the preceding section.

Processing the IRQ handlers as described in the preceding section will activate FDM, which will call the following user callback functions:

- `User_Signal_Mps_Aper()` if the event corresponds to the reception or the transmission of a universal-type variable. The user provides this function to the FDM, when he calls the *fdm_initialize_network* function (a field of *FDM_CONFIGURATION_SOFT* structure),

- `User_Signal_Smmps()` if the event corresponds to the reception of one of the network management variables, following a request to read this variable. The user provides this function to the FDM, when he calls the *fdm_initialize_network* function (a field of *FDM_CONFIGURATION_SOFT* structure),

- `User_Signal_Receive_Msg()` if the event corresponds to the reception of a message. The user provides this function to the FDM, when he calls the *fdm_initialize_network* function (a field of *FDM_CONFIGURATION_SOFT* structure),

- `User_Signal_Send_Msg()` if the event corresponds to the reception of acknowledgement of transmission of a message or to a request for user transmission. The user provides this function to the FDM, when he calls the *fdm_initialize_network* function (a field of *FDM_CONFIGURATION_SOFT* structure),

- `User_Signal_Asent()` if the event corresponds to the transmission of a produced variable with associated event (A_Sent). The user provides this function to the FDM, when he calls the *fdm_mps_var_create* function (a field of FDM_XAE structure),

- `User_Signal_Areceived()` if the event corresponds to the reception of a consumed variable with associated event (A_Received). The user provides this function to the FDM, when he calls the *fdm_mps_var_create* function (a field of FDM_XAE structure),

- `User_Signal_Var_Prod()` if the event corresponds to the transmission of a produced variable of the universal type or at the time-out of the request. The user provides this function to the FDM, when he calls the *fdm_mps_var_create* function (a field of FDM_XAE structure),

Processing the EOC handlers (EOC in level mode) as described in the preceding section will activate FDM, wich will call the following user callback function:

- `User_Signal_Synchro()` if the event corresponds to the reception of a synchronization variable. The user provides this function to the FDM, when he calls the *fdm_mps_var_create* function (a field of FDM_XAE structure),

A more detailed description of the use of these callback functions is given in Subsection 4.7.

4.6 Processing real time clock interrupt

The hardware environment should have a programmable timer that periodically produces interrupts. The user has to connect the interrupt vector to the interrupt handler. This operation depends on the user hardware environment and operating system.

We call the periodicity of the timer interrupt a **tick**.

The real time clock interrupt handler has to activate a task. Inside the body of this task you have to call the FDM function *fdm_ticks_counter*.

The function *fdm_ticks_counter* manages all the time-out aspects of the FIP DEVICE MANAGER. The following parameters of the structure *FDM_CONFIGURATION_SOFT* of the function *fdm_initialize_network* set the values of the FDM time management aspects:

- *Test_Medium_Ticks*: periodicity in ticks of the call of the internal functionality of medium redundancy management,
- *Time_Out_Ticks*: number of ticks to trigger a time-out; it is common to all primitives requiring a time-out except messaging.
- *Time_Out_Msg_Ticks*: number of ticks to trigger a time-out for messaging.
- *OnLine_Tests_Ticks*: periodicity in ticks of the call of the internal self-test functionality.

Example of implementation

```
        /* for example in the case of pSOS */

static unsigned long    sm_rtc;             /* semaphore */

        /* create a task  and a semaphore */

t_create("TRTC", 100, 4096, 0, T_LOCAL | T_NOFPU, &TaskRTC);

sm_create("SRTC", 0, SM_LOCAL | SM_PRIOR, &sm_rtc);



        /* RTC interrupt handler */

static void RTC_handler(void)

{

    sm_v(sm_rtc);          /free the semaphore → wake up of the task TaskRTC */

}

        /* task that treats the real time clock interrupt */
```

```
static void TaskRTC(void)
{
        for(;;){
                fdm_ticks_counter();
                        /* wait the next interrupt */
                sm_p(sm_rtc, SM_WAIT, 0);   /* take the semaphore */
        }
}
```

4.7 Operating model for the deferred services

The execution of some services is deferred by FIP DEVICE MANAGER. This chapter describes the general model to be followed for the use of deferred services.

The main objective of this mechanism is to be able to make, as far as possible, running of the FIP DEVICE MANAGER code independent of the activation context of a service. The user can choose the moment and the context to run the processing code of the services requested, independently of the moment and the context of the request.

For the most time-consuming services, and those that functionally cannot be run immediately, each user request or each indication of the lower level (i.e. FIPCODE) is placed in a queue and the execution environment is informed of this event. The user requests the emptying of the queue (i.e. processing by FIP DEVICE MANAGER) whenever he wishes. Each request causes all the corresponding queues and all that they contain to be emptied when this is possible (in relation to the queues of the lower level).

Each FIP DEVICE MANAGER instance has its own set of queues.

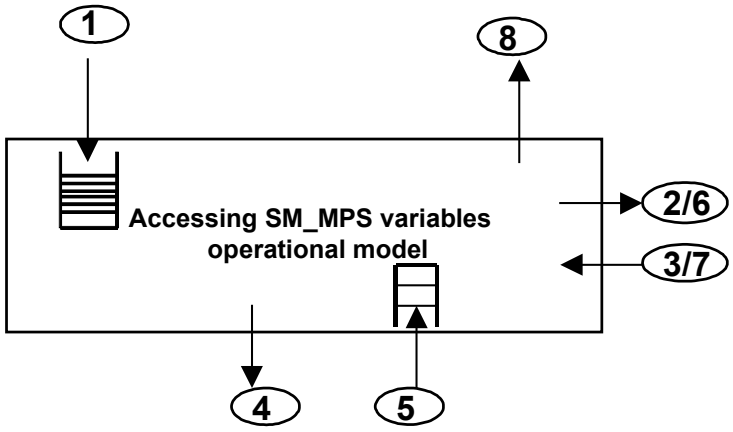
Furthermore, this technique can easily be taken into account in any real-time environment.

The following services are affected by this mechanism:

- access to the SM_MPS variables,
- access to the universal-type MPS variables,
- transmission of messages,
- reception of messages.

4.7.1 Operational model for accessing the SM_MPS variables

The operational model is the following:



1. The user application request is stored in a queue; the requests are the following:
   - fdm_read_present_list,
   - fdm_read_presence,
   - fdm_read_identification,
   - fdm_read_report,
   - fdm_read_ba_synchronize.
2. The FIP DEVICE MANAGER activation manager is warned of the fact that one of the queues is not empty by calling a callback procedure reserved for this purpose, and which the user must write. The address of this procedure is transmitted to FDM by the function *fdm_initialize_network* (field *User_Signal_Smmps* of *FDM_CONFIGURATION_SOFT* structure). The *User_Signal_Smmps* callback function must activate a task that implements the action below.
3. The task activated by the *User_Signal_Smmps* function has to call the *fdm_smmps_fifo_empty* function. This call activates FIP DEVICE MANAGER to empty the queue and process the request.
4. FIP DEVICE MANAGER uses the FIPCODE services to carry out the request.
5. When a SM_MPS variable is received, the FULLFIP component issues an IRQ interrupt; the IRQ handler has to process the event (see Subsection 4.4). Following this processing, FDM stores the received SM_MPS variable in a queue and calls the *User_Signal_Smmps* callback function.
6. The *User_Signal_ Smmps* callback function must activate a task that implements the action below.
7. The task activated by the *User_Signal_Smmps* function has to call the *fdm_smmps_fifo_empty* function. This call activates FIP DEVICE MANAGER to empty the queue and to signal the event to the user.

8. FIP DEVICE MANAGER calls a user callback procedure. The address of this procedure which is transmitted to FDM by the function *fdm_initialize_network*:
   - is the field *User_Present_List*_Prog of FDM_CONFIGURATION_SOFT structure; it is called if the *fdm_read_present_list* function is used,
   - is the field *User_Identification_Prog* of FDM_CONFIGURATION_SOFT structure; it is called if the *fdm_read_identification* function is used,
   - is the field *User_Report_Prog* of FDM_CONFIGURATION_SOFT structure; it is if the *fdm_read_report* function is used,
   - is the field *User_Presence_Prog* of FDM_CONFIGURATION_SOFT structure; it is called if the *fdm_read_presence* function is used,
   - is the field *User_Synchro_BA_Prog* of FDM_CONFIGURATION_SOFT structure; it is called if the *fdm_read_ba_synchronize* function is used.

Note

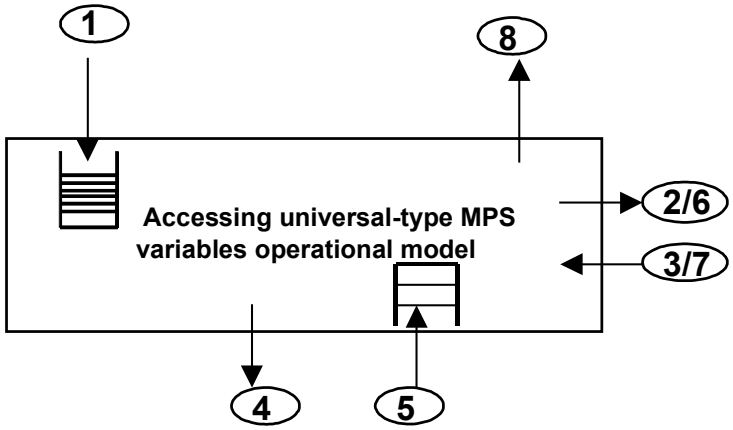
Don't call the *fdm_smmps_fifo_empty* function inside the body of *User_Signal_Smmps* function. The execution context of the *fdm_smmps_fifo_empty* function should be different from the execution context of the *User_Signal_Smmps* function.

Example of implementation of the call of *fdm_smmps_fifo_empty* function

```
        /* for example in the case of pSOS */

FDM_REF      * Network Identification; /* this handle is returned by the function */

                        /* fdm_initialize_network. */

static unsigned long    sm_smmps;               /* semaphore */

        /* create a task and a semaphore */

t_create("TSMP", 100, 4096, 0, T_LOCAL | T_NOFPU, &TaskSMMPS);

sm_create("SSMP", 0, SM_LOCAL | SM_PRIOR, &sm_smmps);



        /* User_Signal_Smmps callback function */

static void  User_Signal_Smmps(struct FDM_REF *Ref)

{

    sm_v(sm_smmps);     /* free the semaphore → wake up of the task TaskSMMPS */

}

        /* task that call fdm_smmps_fifo_empty  function*/
```

```
static void TaskSMMPS(void)
{
        for(;;){
                /* wait for the next event */
                sm_p(sm_smmps, SM_WAIT, 0);   /* take the semaphore */
                fdm_smmps_fifo_empty(Network_Identification);
        }
}
```

4.7.2 Operational model for accessing the universal-type MPS variables



Before using this model you have to create a universal-type MPS variable by the *fdm_mps_var_create* function (the *Type.Scope* field of the structure *FDM_XAE* should be set to 0 - the variable is set as a remote variable).

The operational model is the following:

1. The user application request is stored in a queue; the requests can be the following:
   - *fdm_mps_var_write_universal*,
   - *fdm_mps_var_read_universal*,
2. The FIP DEVICE MANAGER activation manager is warned of the fact that one of the queues is not empty by calling a procedure reserved for this purpose, which the user must write. The address of this procedure is transmitted to FDM by the function *fdm_initialize_network* (field *User_Signal_Mps_Aper* of *FDM_CONFIGURATION_SOFT* structure). The *User_Signal_Mps_Aper* function must activate a task that implements the action below.
3. The task activated by the *User_Signal_Mps_Aper* function has to call the *fdm_mps_fifo_empty* function. This call activates FIP DEVICE MANAGER to empty the queue and process the request.
4. FIP DEVICE MANAGER uses the FIPCODE services to carry out the request.
5. When the MPS variable is sent (or received) then the FULLFIP component issues an IRQ interrupt; the IRQ handler has to process the event (see Subsection 4.4). Following processing, FDM stores the event signaling the production of the MPS variable (or the received MPS variable) in a queue and calls the *User_Signal_Mps_Aper* callback function.
6. The *User_Signal_Mps_Aper* callback function must activate a task that implements the action below.
7. The task activated by the *User_Signal_Smmps* function has to call the *fdm_mps_fifo_empty* function. This call activates FIP DEVICE MANAGER to empty the queue and signal the event to the user.
8. FIP DEVICE MANAGER calls a user callback procedure. The address of this procedure, which is transmitted to FDM when the MPS variable is created by the function *fdm_mps_var_create* is:
   - the field *User_Signal_Var_Prod* of *FDM_XAE* structure; it is called if the *fdm_mps_var_write_universal* function is used,
   - the field *User_Signal_Var_Cons* of *FDM_XAE* structure; it is called if the *fdm_mps_var_read_universal* function is used.

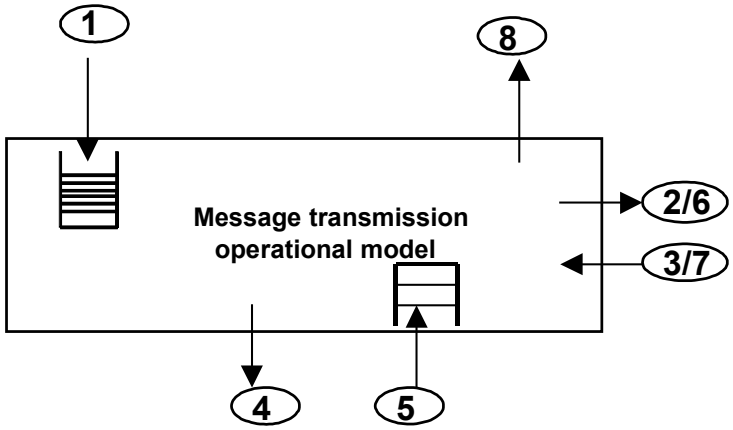
Note

Don't call the *fdm_mps_fifo_empty* function inside the body of the *User_Signal_Mps_Aper* function. The execution context of the *fdm_mps_fifo_empty* function should be different from the execution context of the *User_Signal_Mps_Aper* function.

Example of implementation of the call of *fdm_mps_fifo_empty* function

```
        /* for example in the case of pSOS */
FDM_REF         * Network Identification; /* this handle is returned by the function */
                        /* fdm_initialize_network. */
static unsigned long    sm_mps;             /* semaphore */
        /* create a task and a semaphore */
t_create("TMPS", 100, 4096, 0, T_LOCAL | T_NOFPU, &TaskMPS);
sm_create("SMPS", 0, SM_LOCAL | SM_PRIOR, &sm_mps);

        /* User_Signal_Mps_Aper callback function */
static void  User_Signal_Mps_Aper(struct FDM_REF *Ref)
{
          sm_v(sm_mps);     /* free the semaphore → wake up of the task TaskMPS */
}
        /* task that call fdm_mps_fifo_empty  function*/
static void TaskMPS(void)
{
            for(;;){
                    /* wait the next event */
                    sm_p(sm_mps, SM_WAIT, 0);   /* take the semaphore */
                    fdm_mps_fifo_empty(Network_Identification);
            }
}
```

4.7.3 Operational model for the transmission of messages



Before using this model you have to create a messaging context by calling the *fdm_messaging_fullduplex_create* or *fdm_messaging_to_send_create* function. There are 8 channels for the periodic messages (Channel 1 to 8) and 1 channel for the aperiodic messages (Channel 0).

After the creation of a messaging context, you can use the following model each time when you want to send a message.

Note

You don't have to create a new messaging context each time you want to send a message. This messaging context can be created once for the entire life of your application. At the end, when you no longer need to send messages on this messaging context, you can delete it by calling *fdm_messaging_delete*.

The operational model is the following:

1. Allocate a buffer to store the message to transmit.

   The user application request is stored in a queue (there is a queue for each channel); the request is *fdm_send_message*.

Note

Don't free the allocated buffer till step 8.

2. The FIP DEVICE MANAGER activation manager is warned of the fact that one of the queues is not empty by calling a callback procedure reserved for this purpose, and which the user must write. The address of this procedure is transmitted to FDM by the function *fdm_initialize_network* (*User_Signal_Send_Msg* field of the *FDM_CONFIGURATION_SOFT* structure). The *User_Signal_Send_Msg* callback function must activate a task that implements the action below.

3. The task activated by the *User_Signal_Send_Msg* callback function has to call the *fdm_msg_send_fifo_empty* function. This call activates the FIP DEVICE MANAGER to empty the queue and process the request.

4. The FIP DEVICE MANAGER uses the FIPCODE services to carry out the request.

5. When the message is sent, the the FULLFIP component issues an IRQ interrupt; the IRQ handler has to process the event (see Subsection 4.4). Following processing FDM stores the event signaling the sending of the message in a queue and calls the *User_Signal_Send_Msg* callback function.

6. The *User_Signal_Send_Msg* callback function must activate a task that implements the action below.

7. The task activated by the *User_Signal_Send_Msg* callback function has to call the *fdm_msg_send_fifo_empty* function. This call activates FIP DEVICE MANAGER to empty the queue and signal the event to the user.

8. FIP DEVICE MANAGER calls a user callback procedure when it sends the message. The address of this procedure is transmitted to FDM when a messaging context is created by calling the *fdm_messaging_fullduplex_create* or *fdm_messaging_to_send_create* function:

   - field *User_Msg_Ack_Proc* of the *FDM_MESSAGING_FULLDUPLEX* structure in the case of a creation of a messaging context by calling the *fdm_messaging_fullduplex_create* function,
   - field *User_Msg_Ack_Proc* of the *FDM_MESSAGING_TO_SEND* structure in the case of a creation of a messaging context by calling the *fdm_messaging_to_send_create* function,

Now you can free the allocated buffer used to store the message to transmit.

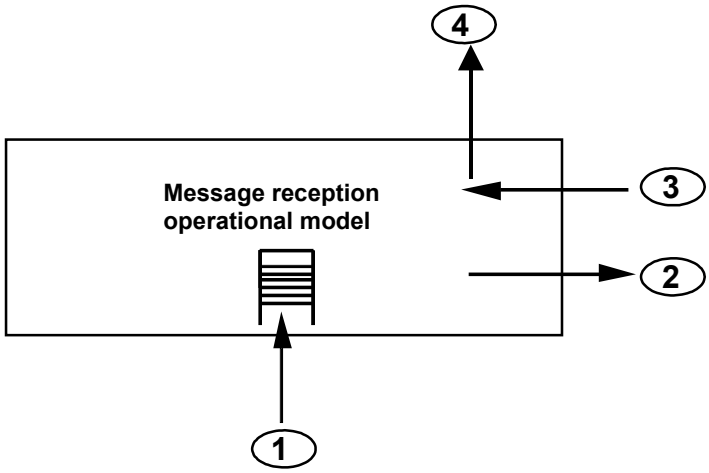
Note

Don't call the *fdm_msg_send_fifo_empty* function inside the body of the *User_Signal_Send_Msg* function. The execution context of the *fdm_msg_send_fifo_empty* function should be different from the execution context of the *User_Signal_Send_Msg* function.

Example of implementation of the call of *fdm_msg_send_fifo_empty* function

```
        /* for example in the case of pSOS */
FDM_REF         * Network Identification; /* this handle is returned by the function */
                        /* fdm_initialize_network. */
static unsigned long        sm_msg_send;            /* semaphore */
        /* create a task  and a semaphore */
t_create("TMSD", 100, 4096, 0, T_LOCAL | T_NOFPU, &TaskMsgSend);
sm_create("SMSD", 0, SM_LOCAL | SM_PRIOR, &sm_msg_send);

        /* User_Signal_Send_Msg callback function */
static void  User_Signal_Send_Msg(struct FDM_REF *Ref)
{
         sm_v(sm_msg_send); /* free the semaphore → wake up of the task TaskMsgSend */
}
        /* task that call fdm_msg_send_fifo_empty  function*/
static void TaskMsgSend(void)
{
            for(;;){
                    /* wait the next event */
                    sm_p(sm_msg_send, SM_WAIT, 0);   /* take the semaphore */
                    fdm_msg_send_fifo_empty(Network_Identification);
            }
}
```

4.7.4 Operational model for the reception of messages



Before using this model you have to create a messaging context by calling the *fdm_messaging_fullduplex_create* or *fdm_messaging_to_rec_create* function.

After the creation of a messaging context you can use the following model each time you want to receive a message.

Note

You don't have to create a new messaging context each time when you want to receive a message. This messaging context can be created once for the entire life of your application. At the end when you no longer need to receive messages on this messaging context you can delete it by calling *fdm_messaging_delete*.

The operational model is the following:

1. The indication received by FIP DEVICE MANAGER, in the form of a call to a primitive, is stored in a queue.
2. The FIP DEVICE MANAGER activation manager is warned of the fact that one of the queues is not empty by calling a procedure reserved for this purpose, and which the user must write. The address of this procedure is transmitted to FDM by the function *fdm_initialize_network* (field *User_Signal_Rec_Msg* of *FDM_CONFIGURATION_SOFT* structure). The *User_Signal_Rec_Msg* function must activate a task that implements the action below.
3. The task activated by the *User_Signal_Rec_Msg* function has to call the *fdm_msg_rec_fifo_empty* function. This call activates FIP DEVICE MANAGER to empty the queue and process the indication.
4. FIP DEVICE MANAGER calls the callback procedure provided by the user to inform him of the event. The address of this procedure is transmitted to FDM when a messaging context is created by calling the *fdm_messaging_fullduplex_create* or *fdm_messaging_to_rec_create* function:
   - field *User_Msg_Rec_Proc* of *FDM_MESSAGING_FULLDUPLEX* structure in the case of a creation of a messaging context by calling the *fdm_messaging_fullduplex_create* function.
   - field *User_Msg_Rec_Proc* of *FDM_MESSAGING_TO_REC* structure in the case of a creation of a messaging context by calling the *fdm_messaging_to_rec_create* function.

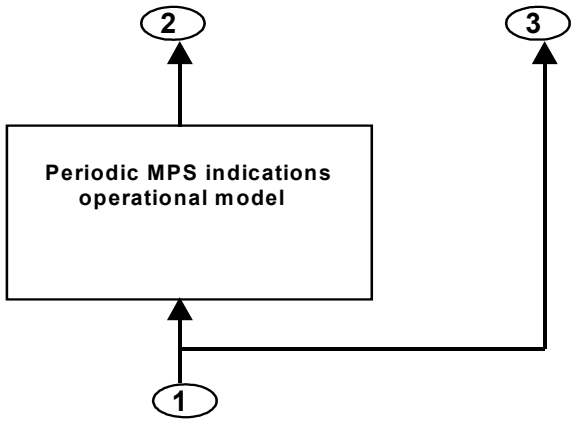
Note

Don't call the *fdm_msg_rec_fifo_empty* function inside the body of the *User_Signal_Rec_Msg* function. The execution context of the *fdm_msg_rec_fifo_empty* function should be different from the execution context of the *User_Signal_Rec_Msg* function.

Example of implementation of the call of *fdm_msg_rec_fifo_empty* function

```
        /* for example in the case of pSOS */
FDM_REF         * Network Identification; /* this handle is returned by the function */
                        /* fdm_initialize_network. */
static unsigned long        sm_msg_rec;             /* semaphore */
        /* create a task  and a semaphore */
t_create("TMRE", 100, 4096, 0, T_LOCAL | T_NOFPU, &TaskMsgRec);
sm_create("SMRE", 0, SM_LOCAL | SM_PRIOR, &sm_msg_rec);

        /* User_Signal_Rec_Msg callback function */
static void  User_Signal_Rec_Msg(struct FDM_REF *Ref)
{
          sm_v(sm_msg_rec); /* free the semaphore → wake up of the task TaskMsgRec */
}
        /* task that call fdm_msg_rec_fifo_empty  function*/
static void TaskMsgRec(void)
{
            for(;;){
                    /* wait for the next event */
                    sm_p(sm_msg_rec, SM_WAIT, 0);   /* take the semaphore */
                    fdm_msg_rec_fifo_empty(Network_Identification);
            }
}
```

4.8 Operating model for the local MPS indications



Before using this model you have to create a local MPS variable using the function *fdm_mps_var_create* (the field *Type.Scope* of the structure *FDM_XAE* should be set to 1 - the variable is set as a local variable).

The type of the variable (produced, consumed, etc.) is set by the *Type.Communication* field of the *FDM_XAE* structure.

Don't forget to enable the indications by using the *Type.Indication* field of the *FDM_XAE* structure.

In the case that you configure the EOC interrupt in pulse mode (see Subsection 4.4), then connect your interrupt vector corresponding to the EOC interrupt on a "User Interrupt Handler" (your interrupt handler).

Note

EOC mode is set by the *Type field* of the *FDM_CONFIGURATION_SOFT* structure when you call the *fdm_initialize_network* function.

The operational model is the following:

1. The indication is received by the FIP DEVICE MANAGER from FIPCODE. If the indication is:
   - transmission of a produced variable, then action 2 is performed,
   - reception of a consumed variable, then action 2 is performed,
   - reception of a synchronisation variable and the EOC interrupt has been configured in level mode, then action 2 is performed,
   - reception of a synchronisation variable and the EOC interrupt has been configured in pulse mode, then action 3 is performed,
2. FIP DEVICE MANAGER calls the associated procedure provided by the user to inform him of the indication. The address of this procedure, which is transmitted to FDM when the MPS variable is created by the function *fdm_mps_var_create*:
   - is field *User_Signal_Asent* of *FDM_XAE* structure in case of reception by FDM of an indication on the transmission of a produced variable; you can, for example, in the body of the *User_Signal_Asent* procedure, call the function *fdm_mps_var-write_loc* to write your actual value,
   - is field *User_Signal_Areceived* of *FDM_XAE* structure in case of reception by FDM of an indication on the reception of a consumed variable; you can, for example, in the body of the *User_Signal_Areceived* procedure, call the function *fdm_mps_var-read_loc*,
   - is field *User_Signal_Synchro* of *FDM_XAE* structure in case of reception by FDM of an indication on the reception of a synchronisation variable and if the EOC interrupt has been configured in level mode.
3. FIP DEVICE MANAGER calls the User Interrupt Handler in case of reception of a synchronisation variable and if the EOC interrupt has been configured in the pulse mode.

4.9 Create an FDM application

To create an FDM application you have to:

Build FIP DEVICE MANAGER library:

1. Adapt the *user_opt* file (choosing the right options) to your environment (operating system, processor, compiler type, see Chapter 6.3).
2. Adapt if necessary some system-dependent objects (semaphores, mutex, etc.) that FDM uses by modifying the files *fdm_os.c* and *fdm_os.h.*
3. Adapt if necessary the memory management to your environment. The files *fdm_gdm.c* and *fdm_gdm.h* provide a set of macros for operating systems pSOS, VxWorks, etc.
4. Build the *fipman.lib* library (you can choose another name if you want).

Write your application:

1. Connect the interrupt vectors to your interrupt handlers for the IRQ, EOC and real-time clock.
2. Initialize FDM by calling the function *fdm_initialize* (see Chapter 3).
3. Create a context for using FDM on a network (this context is related to one FULLFIP component):
   - allocate and fill the structures FDM_CONFIGURATION_HARD, FDM_CONFIGURATION_SOFT and FDM_IDENTIFICATION; then call the function *fdm_initialize_network* (see Chapter 3),
   - memorize the returned handle; this handle is then used as parameter in the other functions.

Note

For each FULLFIP component that your FDM has to drive, you have to call the *fdm_initialize_network* function with the right parameters.

4. Validate the medium(s) by calling the function *fdm_valid_medium.*
5. Create and initialise the tasks that will process the interrupts (see Subsections 4.4 and 4.6 of this chapter).
6. Implement the Error/Warning callback functions (see Subsection 4.2 of this chapter and Chapter 3 for the description of the *FDM_ERROR_CODE* structure in the *fdm_initialize_network* function and Chapter 5 for the list of the errors/warnings).
7. If you want to access the SM_MPS variables (see Subsection 1.5 of Chapter 2 for their description), then follow the model described in Subsection 4.7.1 of this chapter.
8. If your application manages MPS variables, before creating them you have to create the AE_LE objects (see Subsection 1.3 of Chapter 2) by calling the function *fdm_ae_le_create* (see the Chapter 3).
9. If you want to send and receive universal-type MPS variables:
   - create the universal-type MPS variables by calling the function *fdm_mps_var_create* (see the Chapter 3); for each variable to create you have to call the function *fdm_mps_var_create*,
   - follow the model described in Subsection 4.7.2 of this chapter.

10. If you want to send and receive local MPS variables:
    - create the local MPS variables by calling the function *fdm_mps_var_create* (see Chapter 3); for each variable to create you have to call the function *fdm_mps_var_create,*
    - you can read and write local variables using the functions *fdm_mps_var_read_loc* and *fdm_mps_var_write_loc.*
    - if you want to receive indications after a write of a variable or when you receive a variable, follow the model described in Subsection 4.8 of this chapter.
11. If you want to send and receive messages (see Subsection 1.6 of Chapter 2 for an overview):
    - if you want to use the periodic messaging then you have to create a periodic channel by calling the function *fdm_channel_create* (the channel number should be 1 to 8),
    - create a messaging context used only for the transmission of messages by calling the function *fdm_messaging_to_send_create* (see Chapter 3); if the channel is aperiodic then the *Channel_Nr* field of the structure *FDM_MESSAGING_TO_SEND* should be 0; otherwise if it is periodic, this field must be set to the value used when creating the periodic channel using the function *fdm_channel_create,*
    - create a messaging context used only for the reception of messages by calling the function *fdm_messaging_to_rec_create* (see Chapter 3),
    - create a messaging context used for both the transmission and the reception of messages by calling the function *fdm_messaging_fullduplex_create* (see Chapter 3); if the transmission channel is aperiodic then the *Channel_Nr* field of the structure *FDM_MESSAGING_FULLDUPLEX* should be 0. Otherwise, if it is periodic, this field must be set to the value used when creating the periodic channel using the function *fdm_channel_create,*
    - if you want to send messages then you have to follow the model described in Subsection 4.7.3 of this chapter,
    - if you want to receive messages then you have to follow the model described in Subsection 4.7.4 of this chapter.
12. If you want to manage a bus arbitrator (see Chapter 2, Subsection 1.4 and Chapter 3):
    - to load a bus arbitrator program then call:
        - *fdm_ba_load_macrocycle_manual*: loading a simple bus arbitrator generated manually by yourself,
        - *fdm_ba_load_macrocycle_fipconfb*: loading a bus arbitrator generated by an external tool.
    - start the bus arbitrator by calling the *fdm_ba_start* function,
    - link the application with the FDM library.

The **Programs\Fdm_R4.X\demo** subdirectory of the CD contains a C++ application.

Chapter 2 Available Functions

1. DESCRIPTION OF FUNCTIONS

1.1 Self-tests

1.1.1 Aims

This function is used to run the ”In Circuit” test of FULLFIP2 and its associated components.

The test is run in two stages:

- one OFF LINE stage, when the equipment is being powered up
- one ON LINE stage (FULLFIP2 configured and connected to the network).

The limits are those set by the testing possibilities of the FULLFIP2 system (possibility of accessing the ”core” of the components).

There are no line tool tests.

The following are required:

- adequate coverage has to be established,
- faulty elements have to be detected accurately,
- performance levels have to be reached which do not penalise the user application,
- a special microcode must be available for certain tests (included in FIPCODE V6).

Due to critical sections in the ON LINE test stage, the tests are divided into basic blocks whose run time does not affect the user application. This breakdown requires that the sequencing of the various tasks be closely monitored.

1.1.2 Offline test functions

Offline test functions comprise all the test functions which are called before FULLFIP2 is connected to the network.

They are included in the test stage when a FIP DEVICE MANAGER instance is created and initialised.

There are three types of functions:

- complete test of the private FULLFIP2 RAM: the aim of this test is to thoroughly test the private FULLFIP2 RAM by checking:
  - PDA[15:0] and PA [19,16] address buses/private data,
  - control signals WRn and RDn.

This is done by using an algorithm known as the ”count jump method” whose sequencing is as follows:

  - complete write of the volume,
  - check of this volume.

- test of user interface and certain parts of the core of the system: the aim of this test is to carry out an electrical test (detection of sticking at 1 or 0) of the user interface by running address bit, data and control tests, the functional test of the internal hardware indicators and the register and FIFO tests, as well as the test of certain parts of the system core which are accessible.
  - Ustate register bit test:

    Bit IC: carries out a CLOSE command when no other transaction is in progress,
    Bit FE: physical read of more than 128 bytes,
    Bit AE: read of FIFO when no other transaction is in progress,
    Bit SV: no time-out on a function of the physical read or write type by filtering the Busy bit,
    Bit FR: FIFO full in write and empty in read,
    Bit IRQ: forces generation by FULLFIP2,
    Bit EOC: forces generation by FULLFIP2.

  - user register test:

    Registers UFLAG and VAR_STATE are tested in read and KEY_L, KEY_H,TIMER_L, TIMER_H in write. The general test principle is signature analysis.

- FIPCODE integrity test.

  The aim of the test is to check integrity of the microcode after it has been loaded into the private FULLFIP2 memory. Calculating the check-sum of each page of microcode and comparing it with the check-sum supplied for reference does this.

- check of interrupt lines IRQn and EOC of FULLFIP2: checks that there is no discrepancy between the value of signals IRQn and EOC transmitted by FULLFIP2 and the value seen by the user processor.

1.1.3 Online test functions

Online test functions comprise all the test functions, which are called after FULLFIP2 has been connected to the network. There are four types of functions:

- ON LINE integrity test of FIPCODE: calculates check-sum of each page of microcode to be tested and compares it with the original. A thorough check is carried out page-by-page,

- partial test of private FULLFIP2 RAM: adds a descriptor of a variable and of the corresponding value zone to the FULLFIP2/FIPCODE database. The test is carried out in this zone by simulating a variable write using the descriptor. The page in question is thus circulated. This makes it possible to test the entire private RAM containing the pages used to store variable values or message contents.

These functions are processed in a primitive which runs through the test blocks one after the other so that the user application is not penalised. This primitive can either be called automatically by FIP DEVICE MANAGER, or be called by the user from a background task, for example. The activation type is selected when FIP DEVICE MANAGER is created.

1.2 Managing the medium (or media) of each configured network

There are 2 important elements that are used for management of medium redundancy:

- a threshold of medium defaults expressed in % of transactions without errors compared to transactions with errors (this threshold is set by the *Default_Medium_threshold* field of the *FDM_CONFIGURATION_SOFT* structure when you call the *fdm_initialize_network* function)

- a period of calling an internal procedure (called TEST_MEDIUM in FDM R2). The *fdm_ticks_counter* function handles the call of the TEST_MEDIUM procedure. The period of calling the TEST_MEDIUM function is set by the *Test_Mediums_Ticks* field of the *FDM_CONFIGURATION_SOFT* structure when you call the *fdm_initialize_network* function). Subsection 4.6 of Chapter 1 describes how to activate the *fdm_ticks_counter* function by the real-time clock interrupt handler.

Medium test operation

The aim is to:

- avoid using a medium on which too many faults have been detected,

- use a previously unusable medium from which all faults have been eliminated,

- indicate to the user the status of the media (i.e. through which channel(s) the communication is passing).

The functional summary of redundancy management, executed by the TEST_MEDIUM function, that is called periodically by FIP DEVICE MANAGER, is as follows:

Assessment of reception quality

If the number of reception faults (read on the FULLFIP2/FIPCODE counters) on channel x since the last TEST_MEDIUM call exceeds the configured threshold on three consecutive calls of the same procedure, then **Channel_x_Down_On_Reception = true.**

(Faults are filtered to prevent reaction to the slightest interference).

If the number of reception faults on channel x is less than the threshold over three consecutive calls of the TEST_MEDIUM procedure, then **Channel_x_Down_On_Reception = false**.

(The return to normal is filtered).

In all other cases, there is no change.

Assessment of transmission quality

If the transmission status of channel x, specified by the medium redundancy arbitrator, is positioned on error over three consecutive calls of the TEST_MEDIUM procedure, then **Channel_x_Down_On_Transmission = true.**

(Faults are filtered to prevent reaction to the slightest interference).

If the transmission status of channel x, specified by the medium redundancy arbitrator, is not positioned on error over three consecutive calls of the TEST_MEDIUM procedure, then **Channel_x_Down_On_Reception = false.**

(The return to normal is filtered).

In all other cases, there is no change.

Channel validation decision-making

If **Channel_x_Down_On_Reception = true** or **Channel_x_Down_On_Transmission = true** and the other channel is valid, channel x is invalidated: there can be no more transmission or reception on this channel.

(Regardless of their status, two channels can never be invalidated at the same time).

If **Channel_x_Down_On_Reception = false** and **Channel_x_Down_On_Transmission = false,** channel x is revalidated.

Special cases of active Bus Arbitrator

If, over three consecutive calls of the TEST_MEDIUM procedure, both channels have to be invalidated *(this is never done, see above),* the Bus Arbitrator function ceases to operate.

(Filtering occurs so that the BA is not halted spuriously).

If both channels have been declared valid over consecutive calls of the TEST_MEDIUM procedure, the Bus Arbitrator function resumes operation.

(The return to normal is filtered).

In all other cases, there is no change.

Indication to the user

Updating of bits concerning channel status in report variable. See Subsection 1.5.1.1 of this chapter.

Note

Some rules should be respected:

- the recommended values for the periodicity of calling TEST_MEDIUM are:
  - 100 ms: for a 5 Mbits/s network,
  - 200 ms: for a 2.5 Mbits/s network,
  - 500 ms: for a 1 Mbit/s network,
  - 1.6 s: for a 31.25 kbits/s network.
- at least 2 TEST_P instructions should be executed by the Bus Arbitrator between two successive calls of the TEST_MEDIUM procedure,
- the recommended value of the threshold is 5%.

1.3 Managing AE/LE and MPS variables

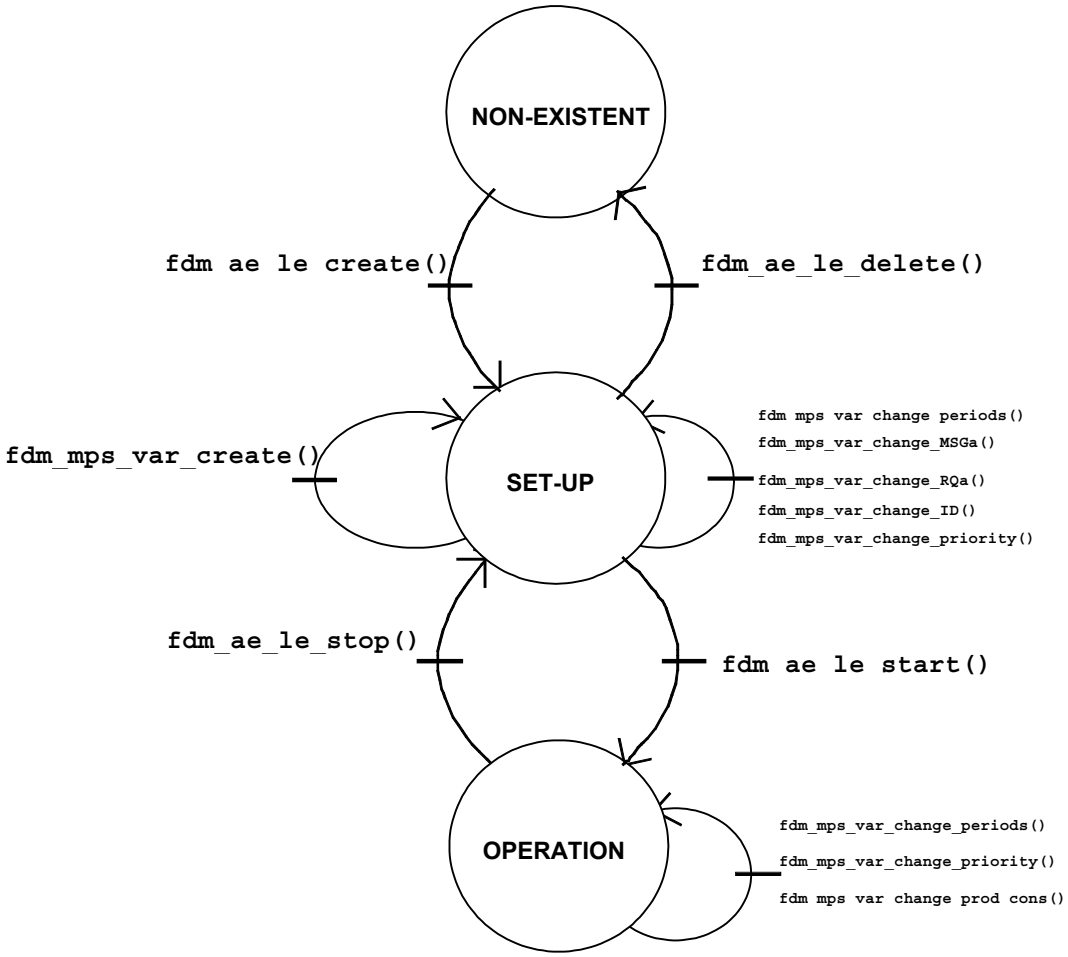
1.3.1 General

This function is used to manage the operating modes of an AE/LE, set it up, set its parameters, adjust it and access the MPS variables it contains.

The AE/LE such as FIP DEVICE MANAGER handling them are unconnected MPS variables associated with their projections on buffers identified by a data link layer identifier.

1.3.2 Management of AE/LE operating modes

1.3.2.1 Diagram of AE/LE operating modes



1.3.2.2 Associated functions

The functions required to manage the operating modes of an AE/LE as defined above are as follows:

- creation and deletion: a creation request creates the envelope, which receives all the variables of this AE/LE. A deletion request deletes the envelope,
- start-up and shutdown: start-up causes the variables contained in the AE/LE to be loaded into the FULLFIP2/FIPCODE database. A shutdown request does the opposite,
- change of operating image(s): you can switch from one image to another in all states except NON-EXISTENT.

1.3.2.3 Setting up and adjusting an AE/LE and setting its parameters

Setting up an AE/LE consists in creating all the variables it has to contain, along with their associated attributes.

Setting the parameters of an AE/LE consists in modifying the attributes of the variables it contains when the AE/LE is in the CONFIGURATION state (it has not been started up). The attributes, which can be modified in this case, are:

- authorised aperiodic MPS queries,
- authorised messaging queries,
- priority of aperiodic MPS queries,
- prompt period,
- refresh period,
- value of associated identifier.

Adjustment of an AE/LE consists in modifying the attributes of the variables it contains when the AE/LE is in the OPERATION state (it has been started up). The attributes, which can be modified in this case, are:

- authorised aperiodic MPS queries,
- authorised messaging queries,
- priority of aperiodic MPS queries,
- prompt period,
- refresh period,
- produced or consumed.

1.3.3 Managing two AE/LE images

FIP DEVICE MANAGER can manage two images for each AE/LE in which the same variables can have different attributes, which can be combined as follows:

(Produced implies with or without transmission and Consumed implies with or without reception)

IMAGE 1	IMAGE 2
nil	produced
nil	consumed
produced	produced
produced	nil
consumed	consumed
consumed	nil
pure event	pure event
nil	pure event
pure event	nil

The idea of “image 1” and “image 2” (or Master/Slave or Normal/Backup) as regards the variables of an AE/LE is only meaningful in terms of what can be transmitted or received on the network; under no circumstances can it refer to what is produced or read by the user.

The idea of an AE/LE image implies the image seen from the network. The user has only one image of the AE/LEs, which comprises all the variables declared in the AE/LE, regardless of the image seen, from the network to which they belong.

This means that once a variable exists in an AE/LE, regardless of the image(s) of the AE/LE to which it belongs, it is always accessible, and in the case of a produced variable in particular, it is always updated.

The following table shows the different possible cases, indicating what is happening on the network and what the user can do.

It should be noted that the transmission or reception indication depends on what is happening on the network and therefore produced or consumed implies with or without a transmission indication.

Variable image	Produced or consumed	MPS variable or Sync EVT	AE_LE OPERATES WITH IMAGE 1		AE_LE OPERATES WITH IMAGE 2	
			USER	NETWORK	USER	NETWORK
IMAGE 2	PRODUCED	MPS VAR	Writes variable and corresponding buffer is effectively updated in FULLFIP DB	Nothing transmitted even if corresponding ID is circulating	Writes variable and corresponding buffer is effectively updated in FULLFIP DB	Variable value transmission on network
IMAGE 2	CONSUMED	MPS VAR	Reads variable: value is last one received when operating with previous image 2. Prompt status false	No variable value updating even if corresponding RP_DAT has been received	Reads variable: value read is last update by network	Updating of variable value when corresponding RP_DAT has been received
IMAGE 1 and IMAGE2	PRODUCED	MPS VAR	Writes variable and corresponding buffer is effectively updated in FULLFIP DB	Variable value transmission on network	Writes variable and corresponding buffer is effectively updated in FULLFIP DB	Variable value transmission on network
IMAGE 1	PRODUCED	MPS VAR	Writes variable and corresponding buffer is effectively updated in FULLFIP DB	Variable value transmission on network	Writes variable and corresponding buffer is effectively updated in FULLFIP DB	Nothing transmitted even if corresponding ID is circulating
IMAGE 1 and IMAGE2	CONSUMED	MPS VAR	Reads variable: value read is last update by network	Updating of variable value when corresponding RP_DAT has been received	Reads variable: value read is last update by network	Updating of variable value when corresponding RP_DAT has been received
IMAGE 1	CONSUMED	MPS VAR	Reads variable: value read is last update by network	Updating of variable value when corresponding RP_DAT has been received	Reads variable: value is last one received when operating with previous image 1. Prompt status false	No variable value updating even if corresponding RP_DAT has been received
IMAGE 1 and IMAGE2	CONSUMED	SYNC EVT	Receives sync event when it appears	Takes sync event into account to inform user of it	Receives sync event when it appears	Takes sync event into account to inform user of it
IMAGE 2	CONSUMED	SYNC EVT	Does not receive sync event when it appears	Does not take sync event into account	Receives sync event when it appears	Takes sync event into account to inform user of it
IMAGE 1	CONSUMED	SYNC EVT	Receives sync event when it appears	Takes sync event into account to inform user of it	Does not receive sync event when it appears	Does not take sync event into account

Table 2-1: Management of two AE/LE images

1.3.4 Managing MPS variables

1.3.4.1 MPS variables

There are two types of MPS variable access services:

- Local: the variable can be read or written directly in the local database.
- Remote: circulation of the variable has to be requested of the Bus Arbitrator via an aperiodic request prior to the read or following the write.

The type of access service to be used for a given variable is a configuration parameter of the variable in question. Similarly, in the case of a remote service, the priority of the request is also a configuration parameter of the variable.

Access to the MPS variables contained in the AE/LE therefore requires the following services:

- *Local write*: Local write of a variable with immediate confirmation.
- *Local read*: Local read of a variable with immediate confirmation.
- *Universal read*: This service makes it possible to read the value of a variable without having to know which type of read service to use. This service offers immediate confirmation that execution is possible and delayed confirmation is given with the value of the variable.
- *Universal write*: This service makes it possible to write the value of a variable without having to know which type of write service to use. This service offers immediate confirmation that execution is possible, with delayed confirmation. As far as universal services are concerned, the anticipation factor for the same variable is 1. All these services guarantee the integrity of the variable value obtained.

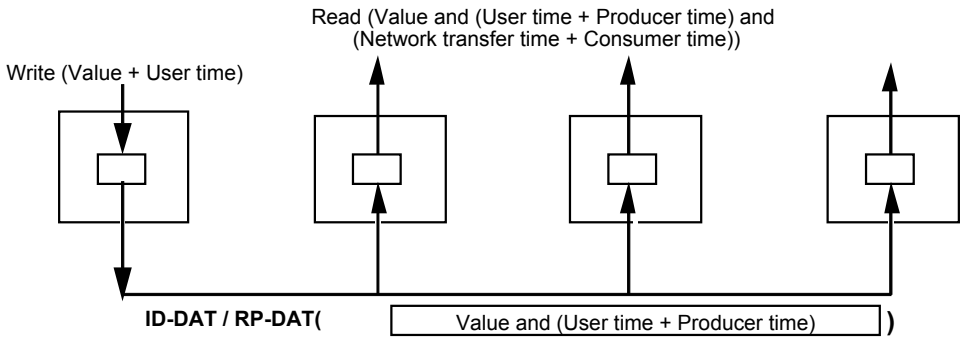
When read, consumed MPS variables may be produced accompanied by a status indicating the transmission validity and production validity of a variable. They are:

- **PROMPTNESS:** Promptness relates to the validity of the value of a variable being made available to the user by the network. This validity is generated from a maximum consumption period set by the user. The status is generated the moment the variable is consumed (read) on the network. If the value of the status is true, the value of the variable has been updated within a period of time less than or equal to the specified consumption period.
- **REFRESHMENT:** Refreshment relates to the validity of the value of a variable being made available to the network by a producer user. This validity is generated from a maximum production period set by the user. The status is generated the moment the variable is produced on the network. If the value of the status is true, the value of the variable has been updated within a period of time less than or equal to the specified production period.
- **SIGNIFICANCE:** This status is generated the moment the variable is produced on the network. If the value of the status is true, the value of the variable has been written at least once.

1.3.4.2 MPS variables with dynamic refreshment

The value of the variable supplied to the WorldFIP network is then handled by the network which delivers to the consumer the length of time this value stays in the producer's local buffer before it is produced, to which must be added the time taken to transfer it to the network and the length of time it spent with the consumer before being read.

Basic diagram:



The selected implementation consists in defining a new MPS variable, with the *with_time_var* attribute. Processing the variable shows which type it is:

- When the variable is read by the consumer(s):

  The time difference between the reception time from the network and the time it is read by the user is supplied with the contents of the variable.

  The contents comprises the value written by the producer followed by four bytes corresponding to the production difference. The time difference is also expressed in microseconds.

- When the variable is produced on the network:

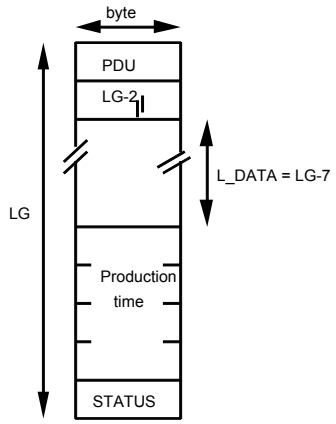
  The time difference between the moment it is updated by the producer in the database and the moment it is produced on the network is added to the value of the variable in the four bytes intended for this purpose.

  The four bytes in question may already contain a time difference (in the case of bridges, for example).

  These time differences are expressed in microseconds.

  The refresh status is automatically calculated in the same way as for a conventional variable.

Format of variable produced on the network



Note

L_DATA must be a multiple of 2

The accuracy of the calculation of the two time differences corresponds to a value of 2*Tslot (time unit used internally by the system), for example 125µs for FULLFIP2 at 64 MHz.

It follows from the above that if this type of variable is to be managed correctly, it must be possible to set it up, the producer must be able to write it and the consumer must be able to read it.

1.3.4.3 Time variable

The solution adopted consists in changing only the time in an MPS variable produced by the item of equipment with the reference time.

The time variable is configured automatically, if requested at the time of general configuration. The variable selected is that associated with identifier 9802H.

It is created as a *variable with dynamic refreshment* type.

The format of this variable is as follows: (normalised contents).

40H	Type of PDU
11H	Length of PDU
OBH	Duration
80H	POSIX time field
09H	Length of POSIX time field
XXH	Meaning: 0=global-reference, 2=local-reference
(1) XX XX XX XX H	Number of seconds since 1/1/70 (Unsigned 32)
(1) XX XX XX XX H	Number of nanoseconds since beginning of current second (Unsigned 32)
(1) XX XX XX XX H	Dynamic refresh status in µs (Unsigned 32)
XXH	MPS refresh status

(1) in the order PF...pf

The value of the reference time supplied to the WorldFIP network in the requested form (POSIX) is handled by the network, which delivers to the consumer the length of time this value has stayed in the producer's local buffer before being produced, to which must be added the time taken to transfer it to the network and the length of time it stays with the consumer before it is read.

Thus, after a time variable is read, the application has an absolute time supplied directly by the FIP DEVICE MANAGER which adds to the value given by the producer and the producer and consumer waiting times.

When this time variable is read or written, the operations are the same as those for variables with dynamic refresh status (see Subsection 1.3.4.2.).

1.3.4.4 Synchronisation variables

Synchronisation variables are used to signal events generated on reception of a frame of the ID_DAT type with no associated RP_DAT. This notion:

- eliminates the need for a producer: there is no associated user variable,
- is only visible locally.

The signal mode of the event in question is the mode that consists in being alerted directly by a special physical interrupt signal from the FULLFIP2 system. The semantics of this signal are therefore: "pure sync event reception".

This mode makes use of the following mechanism:

- the EOC signal from the system is used as an interrupt source,
- it is set on reception of a frame of the ID_DAT type with no associated response if the descriptor of the identifier concerned contains the ”signal mode” parameter set to the corresponding value,
- no additional data are associated with this signal,
- the interrupt is either a pulse lasting approximately 310 ns, in which case it needs no external acknowledgement, or it operates at various levels and is acknowledged by FIP DEVICE MANAGER after the user procedure used in this context has been called. The choice of interrupt type can be configured:
- Pulse duration is: 375 µs if FULLFIP2 @ 40 MHz,
  234 µs if FULLFIP2 @ 64 MHz,
  188 µs if FULLFIP2 @ 80 MHz.

1.3.5 Managing time variable producer redundancy

Several potential time variable producers can exist on one network segment. However, at any given moment, only one should be the active producer on the network.

The various subscriber statuses with regard to the time production mechanism are:

- potential time producer,
- active time producer (a selected potential producer),
- active non-producing consumer,
- active Bus Arbitrator: it elects the active producer,
- not involved in time management: neither producer nor consumer.

The principle implemented to manage time producer redundancy is as follows:

- a potentially time-producing subscriber:
  - consumes the MPS time variable (ID 9802) if required,
  - periodically transmits its STATUS via the SUBMMS messaging service *Information_Report* to the group of potential Bus Arbitrators,
  - consumes a COMMAND transmitted by the active Bus Arbitrator via the SUBMMS messaging service *Information_Report*,
  - becomes an active producer if the contents of the COMMAND so require,
- an active time-producing subscriber:
  - produces the MPS time variable (ID 9802),
  - periodically transmits its STATUS via the SUBMMS messaging service *Information_Report* to the group of potential Bus Arbitrators,

  - consumes a COMMAND transmitted by the active Bus Arbitrator via the SUBMMS messaging service *Information_Report*,
  - ceases to be an active producer if the contents of the COMMAND so require,
- an active non-producing consumer subscriber consumes the MPS time variable (ID 9802),
- the active Bus Arbitrator:
  - consumes the STATUSES transmitted by the time producers (potential and active) via the SUBMMS messaging service *Information_Report*,
  - builds the COMMAND according to the STATUS contents,
  - produces the COMMAND via the SUBMMS messaging service *Information_Report*. The active producer is designated in a random manner from among the potential producers standing by to be elected.

The messaging addresses retained are:

- 0xA00000 for the address of the Bus Arbitrator group: STATUS reception (LSAP = A000, num.segment = 0),
- 0xA00100 for the address of potential producers: COMMAND reception (LSAP = 0xA001, num.segment = 0),
- 0xFy0000 for the address of the COMMAND and STATUS transmitter (y = physical address of the transmitting subscriber) (LSAP = 0xFy00, num.segment = 0.).

Note

The active producer cannot consume the time variable: if it is read, its value is meaningless.

The user application is informed when a potential producer becomes an active producer by calling a procedure it supplied to FIP DEVICE MANAGER when the mechanism was started up and initialised.

The user application on the potential producer side may (must) continue to update the time variable so that the correct value is produced if it is elected.

A potential producer which consumes the time variable indicates in STATUS the refresh status value of the variable so that the active Bus Arbitrator can decide to elect another active producer should there be a problem.

The procedure for managing time producer redundancy is activated periodically on both the active Bus Arbitrator and on the time producers (active or potential). The activation period is set when the mechanism is started up and initialised.

Subscribers which are merely time consumers (not potential producers) do not produce STATUSES and do not consume COMMANDS.

1.4 Managing the Bus Arbitrator(s)

1.4.1 Setting up a Bus Arbitrator program

Bus Arbitrator programs may be supplied as FIP DEVICE MANAGER inputs in two different forms:

- identical to that supplied by the BA_BUILDER (FIPCONFB) tool and to that of remote BA loading via SMS,
- simplified to create a manual BA program (i.e. with no tools) in a C program.

The instructions that can be used are as follows:

SEND_LIST:	Requests that a list of identifiers is transmitted in the form ID_DAT or ID_MSG
SEND_APER:	Requests that a window is opened to process aperiodic MPS requests
SEND_MSG:	Requests that a window is opened to process aperiodic messaging requests
BA_WAIT:	Requests internal synchronisation
SYN_WAIT:	Requests external synchronisation
NEXT_MACRO:	Reloops the macrocycle
TEST_P:	Requests a test of subscribers present
END_BA:	Denotes the end of a BA program

Note:

The behaviour of the NEXT_MACRO instruction depends on the bit rate of the network:

- at **31.25 kbits/s**, after the NEXT_MACRO instruction and the end of the macrocycle, a time T composed of *Silences* and *Stuffing* will elapse. This time depends on the `FDM_WITH_OPTIMIZED_BA` compilation option (in the *user_opt.h* file):
  - if `FDM_WITH_OPTIMIZED_BA` is set to NO then:

    T = (`MAX_Subscriber` + 2)*T0 + 6260

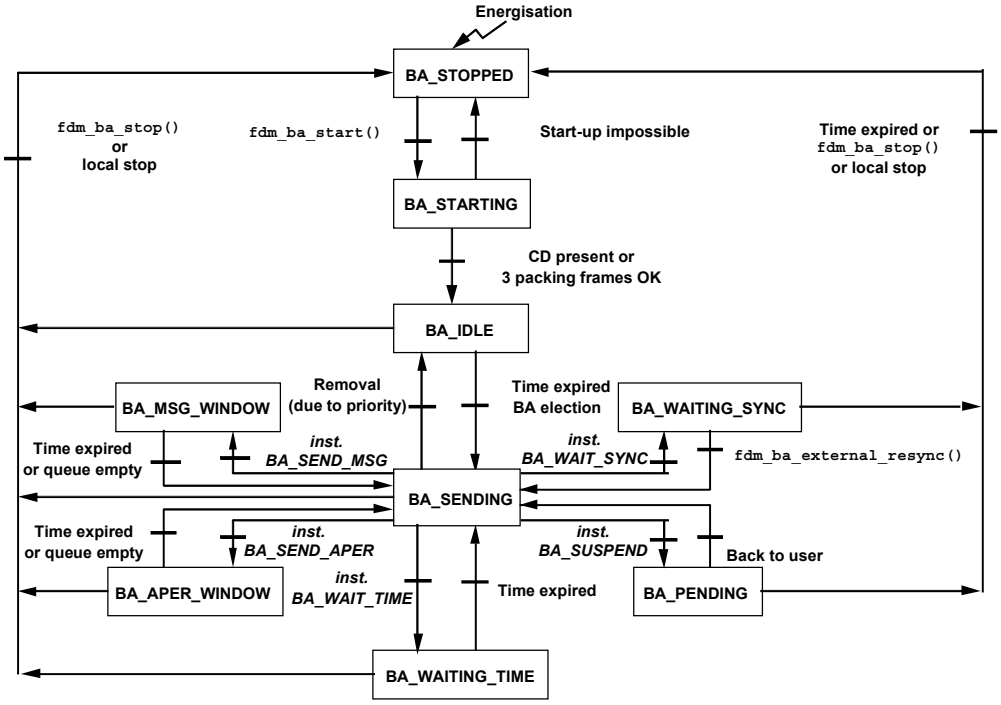
    where
    - `MAX_Subscriber` parameter is set by the user in the `fdm_ba_set_parameters()` function
    - T0 is *Silence Time-Out* and its value is 4096 ms (for details see **[1]**)
  - if `FDM_WITH_OPTIMIZED_BA` is set to YES then:

    T = (GreatestNumberOfBusArbitrator + 2)*T0 + 6260

    where
    - *GreatestNumberOfBusArbitrator* is a parameter of the FDM_CONFIGURATION_HARD structure set by the user in the `fdm_initialize_network()` function
    - T0 is *Silence Time-Out* and its value is 4096 ms (for details see **[1]**)
- at **1 Mbit/s, 2.5 Mbits/s and 5 Mbits/s**, after the NEXT_MACRO instruction and the end of the macrocycle, a time that corresponds to one *Stuffing ID_DAT/Silence* will elapse; the elapsed time depends on the bit rate.

1.4.2 Managing Bus Arbitrator operating modes

1.4.2.1 Diagram of Bus Arbitrator operating modes



Status description:

BA_STOPPED: The Bus Arbitrator is completely disconnected from the network and is standing by for a (re)start-up command.

BA_STARTING: The user start-up command is being processed. Three packing frames are transmitted on the network if no activity has been detected beforehand.

BA_IDLE: In this standby status, the Bus Arbitrator does not transmit on the network but monitors activity, ready to trigger its election procedure should activity on the network cease.

BA_SENDING: The Bus Arbitrator is active and processes BA program instructions such as BA_SEND_ID_DAT, BA_SEND_ID_MSG and NEXT_MACRO.

BA_WAITING_TIME: Status in which the Bus Arbitrator transmits packing frames until the specified time has elapsed. This makes it possible to synchronise execution of the BA program and set a fixed macrocycle duration.

BA_WAITING_SYNC: Status in which the Bus Arbitrator transmits packing frames until the specified time has elapsed or requests user restart-up (external resynchronisation).

BA_PENDING: Status in which the Bus Arbitrator transmits packing frames until the user asks to take over or the monitoring time elapses.

BA_APER_WINDOW: In this status, the Bus Arbitrator processes the urgent and non-urgent aperiodic requests stored in its queues. It leaves this status when the queues are empty or when the time allocated to the window has elapsed.

BA_MSG_WINDOW: In this status, the Bus Arbitrator processes the aperiodic messaging requests stored in its queues. It leaves this status when the queue is empty or when the time allocated to the window has elapsed.

1.4.2.2 Bus Arbitrator associated functions

The functions required to manage the Bus Arbitrator operating modes, as defined above, are as follows:

- *Start-up*: The sequence of actions is as follows:
  - launch of a time-out known as *Start-up Time-out*, the value of which is greater than the longest *Election Time-out* possible for the network concerned.
  - if an activity is detected during this period, the Bus Arbitrator starts up, passing directly to the IDLE status where it activates its election time-out.
  - if no activity is detected during the start-up period, three packing frames are transmitted. If a fault is detected during transmission, the Bus Arbitrator returns to the STOPPED status, otherwise it passes to the IDLE status where it activates its election time-out.
  - when the election period has elapsed, and the Bus Arbitrator has not detected any activity on the network, it becomes active by executing the required BA program.
- *Stop*: The stop command causes a return to the STOPPED status. It is taken into account from any other status,
- *External resynchronisation of a macrocycle*: this command is used to position the user sync pulse that makes the Bus Arbitrator resume execution of the macrocycle,
- *Macrocycle switching*: This function is used to select another macrocycle in order to change the nature of the flow on the network without losing control. The change of macrocycle is only effective at the end of the current macrocycle execution.

1.4.3 Setting the parameters of the Bus Arbitrator Start-up function

The sequence of operation for starting up a Bus Arbitrator is described in Subsection 1.4.2.2. of this chapter.

The start-up of a Bus Arbitrator function uses the following two time-outs: *Start-up Time-out* and *Election Time-out*.

The parameters that are used to calculate these two time-outs are set by user with the following functions:

- `fdm_initialize_network()`,
- `fdm_ba_set_parameters()`,
- `fdm_ba_set_priority()`.

The two time-outs are calculated with the following formula:

Start-up Time-Out = [(**GNOBA** + 1)*(**MP** +1) + **ETOTPI** + 3]*2

Election Time-Out = [(**GNOBA** + 1)***PL** + **ETOTPI** + **BANO** + 3]*2

where

- **GNOBA**: highest number of the BAs on the network.

  Its value depends on the `FDM_WITH_OPTIMIZED_BA` compilation option (in the *user_opt.h* file) and the optimization level (`STANDARD`, `OPTIMIZE_1`, `OPTIMIZE_2` and `OPTIMIZE_3`) set by the function `fdm_ba_set_parameters()`

  - if FDM_WITH_OPTIMIZED_BA is set to NO then:
    - if the optimization level is `STANDARD` then GNOBA = 255; this value is constant and cannot be changed by the user.
    - else if the optimization level is `OPTIMIZE_1` or `OPTIMIZE_2` or `OPTIMIZE_3` then the GNOBA = `MAX_Subscriber` parameter set by the user in the `fdm_ba_set_parameters()` function.
  - else if `FDM_WITH_OPTIMIZED_BA` is set to YES then:
    - GNOBA = GreatestNumberOfBusArbitrator parameter of the FDM_CONFIGURATION_HARD structure set by the user in the `fdm_initialize_network()` function.

- **MP**: maximum priority of the Bus Arbitrators connected on the network

  Its value depends on the optimization level (`STANDARD`, `OPTIMIZE_1`, `OPTIMIZE_2` and `OPTIMIZE_3`) set by the `fdm_ba_set_parameters()` function

  - if the optimization level is `STANDARD` or `OPTIMIZE_1` then MP = 15; this value is constant and cannot be changed by the user.
  - if the optimization level is `OPTIMIZE_2` or `OPTIMIZE_3` then MP = `Max_Priority` parameter set by the user in the `fdm_ba_set_parameters()` function.

- **PL**: priority level.

  Its value is PL = `Priority_Level` parameter set by the user in the `fdm_ba_set_priority()` function.

- **BANO**: bus arbitrator number

  Its value depends on the `FDM_WITH_OPTIMIZED_BA` compilation option (in the *user_opt.h* file)

  - if FDM_WITH_OPTIMIZED_BA is set to NO then:
    - BANO = `K_PHYADR` parameter of the `FDM_CONFIGURATION_HARD` structure set by the user in the `fdm_initialize_network()` function
  - else if `FDM_WITH_OPTIMIZED_BA` is set YES then:
    - BANO = NumberOfThisBusArbitrator parameter of the FDM_CONFIGURATION_HARD structure set by the user in the `fdm_initialize_network()` function.

- **ETOTPI**: execution time (in microseconds) of the TEST_P instruction of the BA

  Its value depends on the `FDM_WITH_OPTIMIZED_BA` compilation option (in the *user_opt.h* file) and the optimization level (`STANDARD`, `OPTIMIZE_1`, `OPTIMIZE_2` and `OPTIMIZE_3`) set by the `fdm_ba_set_parameters()` function

  - if `FDM_WITH_OPTIMIZED_BA` is set to NO then:
    - if the optimization level is `OPTIMIZE_3` then ETOTPI = 0; this value is constant and cannot be changed by the user.
    - else if the optimization level is `STANDARD` or `OPTIMIZE_1` or `OPTIMIZE_2` then ETOTPI = 257*T0 where T0 is *Silence Time-Out* and depends on the bit rate of your network (for details see [1]); the ETOTPI value depends on the bit rate of your network and cannot be changed by the user.
  - else if `FDM_WITH_OPTIMIZED_BA` is set to `YES` then:
    - if the optimization level is `OPTIMIZE_3` then ETOTPI = 0; this value is fixed and cannot be changed by the user.
    - else if the optimization level is STANDARD or OPTIMIZE_1 or OPTIMIZE_2 then :
      - ETOTPI = 10000 if the bit rate is 1 Mbit/s or 2.5 Mbits/s or 5 Mbits/s,
      - ETOTPI = 50000 if the bit rate is 31.25 kbits/s.

Note

You cannot use the optimization level `OPTIMIZE_3` for a double medium topology of your network. You can only use it for a single medium topology.

1.5 Managing SM-MPS network management variables

1.5.1 Format of network management variables

The network management services known as SM_MPS come from an MPS specialisation: the exchange principles are also based on the ”buffer transfer” service of the FIP data link layer.

SM_MPS offers several types of specialised variables characterised by:

- values with special syntax and semantics,
- reserved identifiers,
- as in the MPS, these variables are handled using special read and write primitives,
- there are limited possibilities for the MPS variables (for example there is no associated FIP status).

Via its FIP DEVICE MANAGER software, ALSTOM Technology has chosen to support a variable sub-assembly proposed in the standard:

- presence variable: each item of equipment sends a data item indicating its presence on the network,
- identity variable: each item of equipment can be clearly identified remotely,
- report variable: each item of equipment supplies potential users with a set of default counters,
- presence control variable: the presence of all the items of equipment can be monitored by one particular item of equipment,
- BA synchronisation variable: indicates the current status of the active Bus Arbitrator.

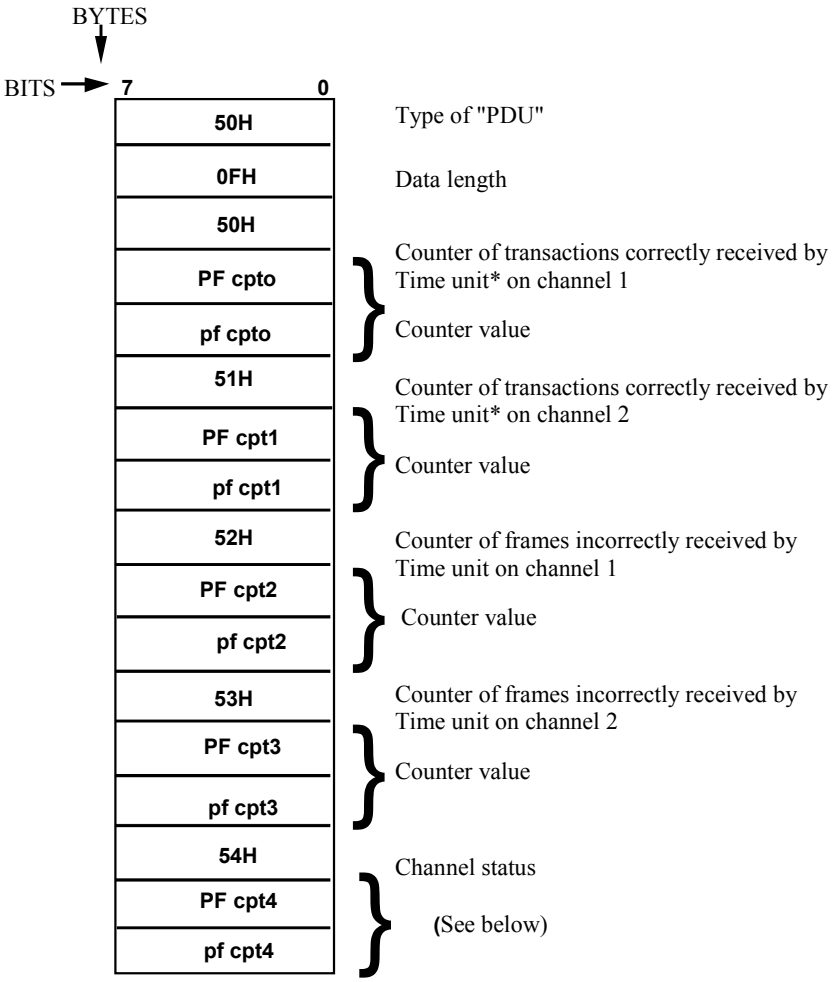
1.5.1.1 Report variable

This variable is linked to one physical allocation identifier per station, built in the following way:

ID = 11xyH where xy = physical address of the subscriber

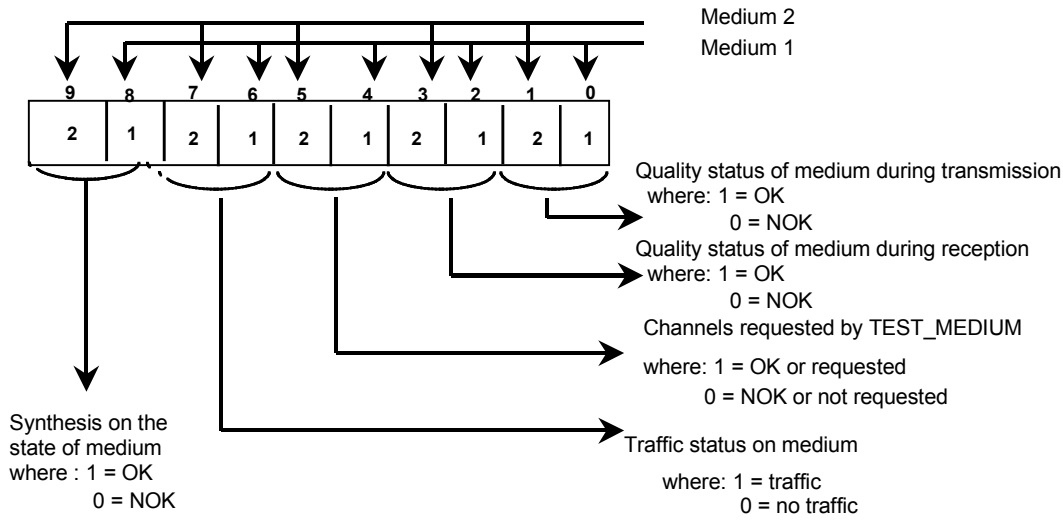
It can be produced by an item of equipment as soon as it is powered up in reply to a report identifier linked to its own station number.

No FIP status is associated with this variable. As soon as it is produced, it is always valid and meaningful.



*time unit = medium test function call time.

Format of counter 4

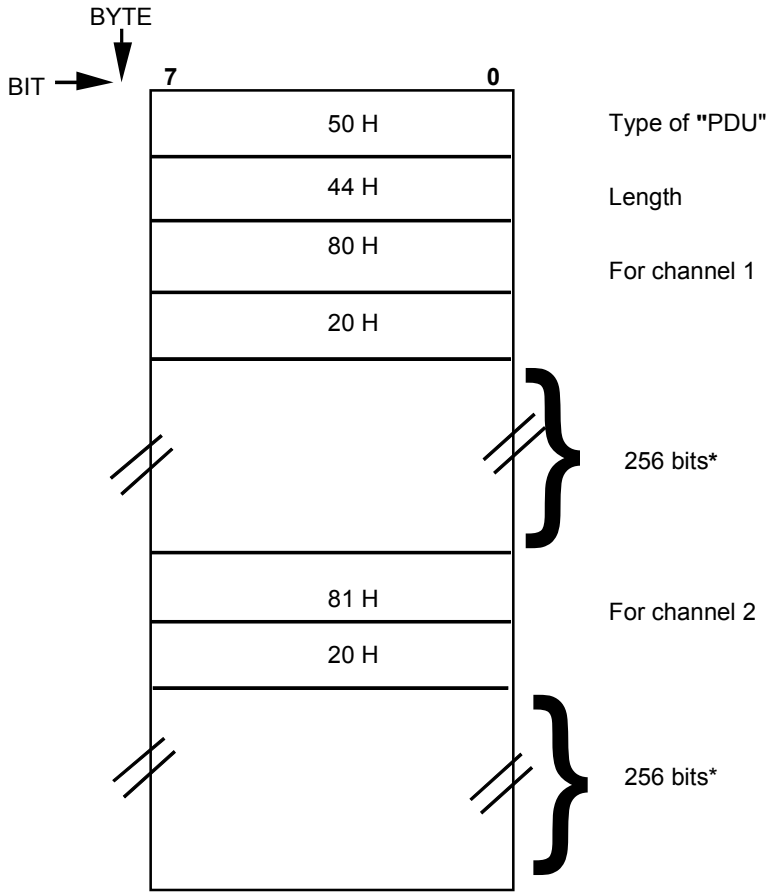


1.5.1.2 List of equipment present variable

The *list of equipment present* variable (or presence monitoring variable) is a universal network management variable associated with identifier 9002 H.

No FIP status is associated with this variable. As soon as it is produced, the variable is valid and significant.

However, this variable, produced by the active Bus Arbitrator, is only valid and significant when the Bus Arbitrator has finished updating.



* i.e. 32 bytes: subscriber i present on channel x: bit number (i mode 8) of byte number (i div 8) = 1. Absence: the same bit = 0

When operation is with a single medium card, all the bits on channel 2 are at 0 (no subscribers present). When operation is with a double medium card, but with a single channel validated, all the bits on the channel not used are at 0 (no subscribers present).

When operation is in FIPIO mode, only the part concerning channel 1 is contained in this variable.

1.5.1.3 Presence variable

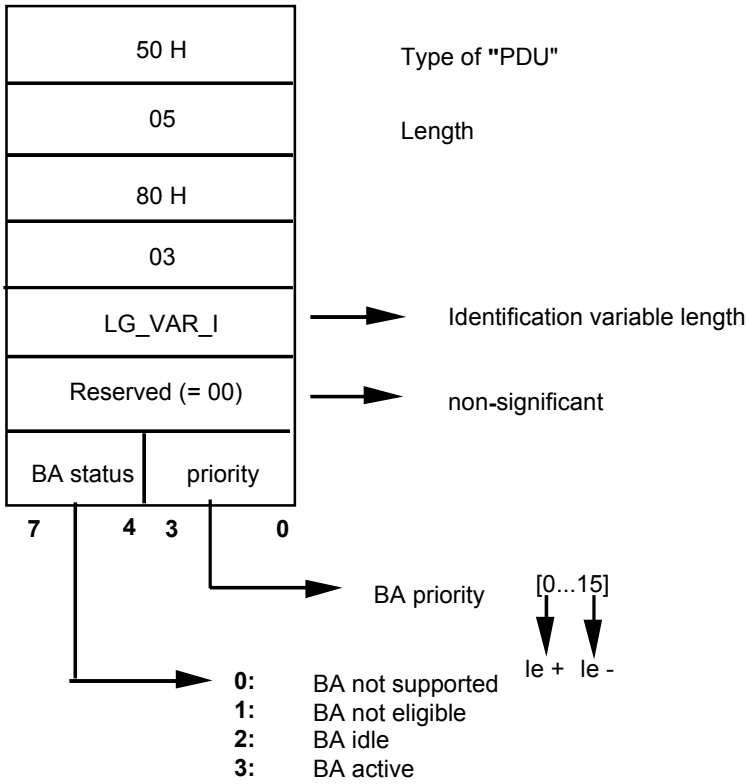
This variable is linked to one physical allocation identifier per station built in the following way:

ID = 14xy H where xy: physical address of the subscriber

A station can produce it as soon as it has been powered up in reply to the presence indicator linked to its own station number.

The active Bus Arbitrator to build the list of equipment present consumes it.

No FIP status is associated to this variable. As soon as it is produced, it is always valid and significant.



1.5.1.4 Identification variable

The identification (identity) variable is a variable linked to the physical address of each item of equipment. The identifier used is built as follows:

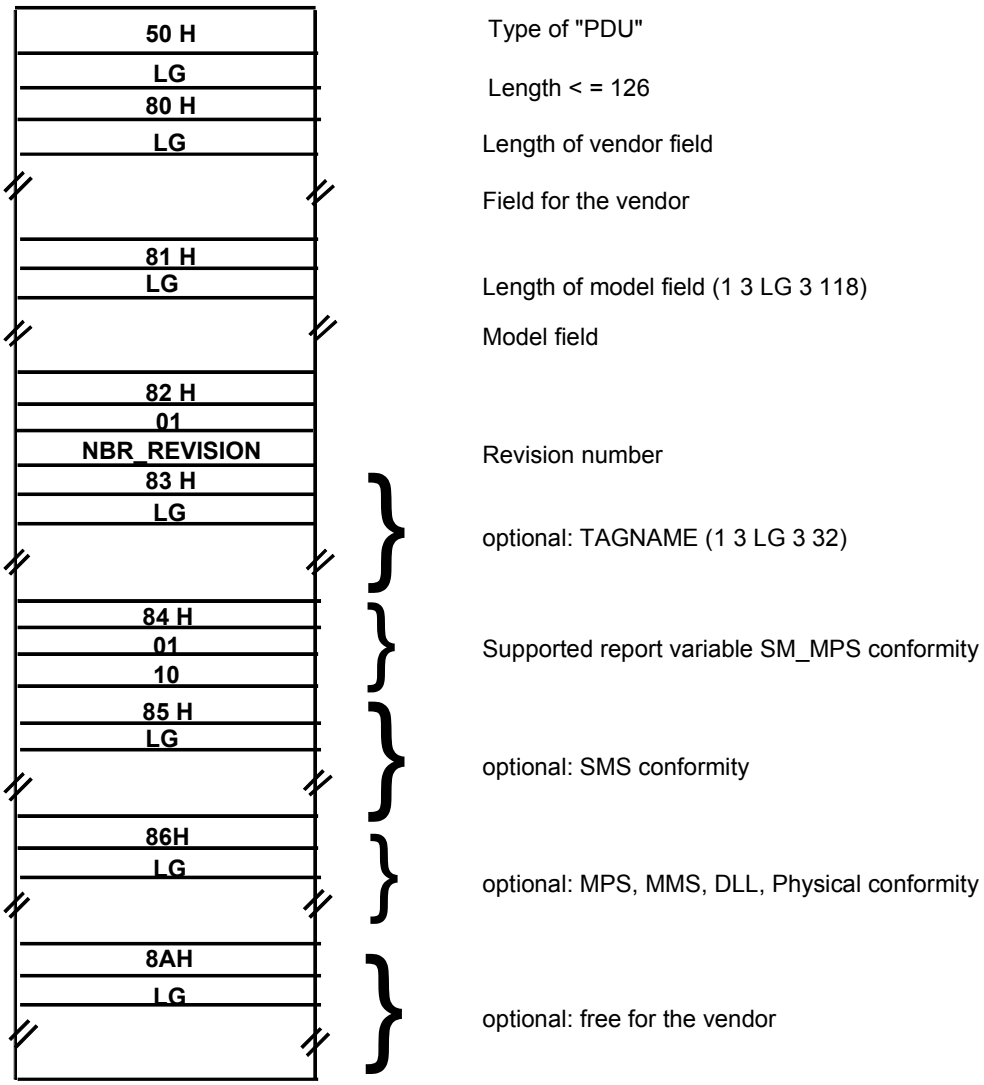
ID = 10xy H where xy = physical address of subscriber.

It may be produced by an item of equipment as soon as it has been powered up, in reply to the identification identifier linked to its own station number.

No FIP status is associated with this variable.

This variable is always valid and significant.

The conformity in question has to be interpreted as the software potential.

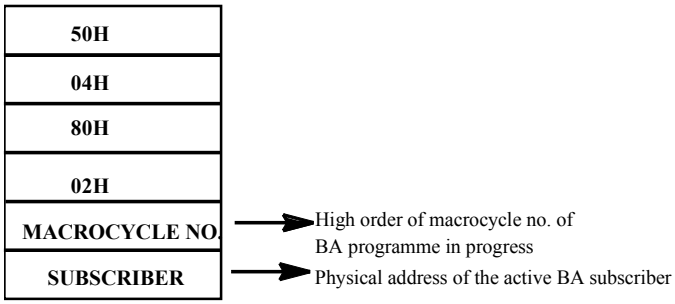


1.5.1.5 BA synchronisation variable

The *BA sync* variable is a universal network management variable associated with identifier 9003H.

No FIP status is associated with this variable. As soon as it is produced, it is valid and significant.

The active Bus Arbitrator produces this variable.



1.5.1.6 Segment Parameters variable

This variable is only produced on a subscriber that runs in FIPIO mode (i.e. as a FIPIUO manager). It could not be consumed by subscribers build with FIP DEVICE MANAGER but only FIPIO Agent (not supplied by ALSTOM).

1.5.2 Managing network management variables

The operations to be carried out on these variables are as follows:

- **Creation**: they are automatically created by the FIP DEVICE MANAGER for each controlled network, when the corresponding FIP DEVICE MANAGER instance is created.

- **Initialisation**: the presence and identification variables are initialised when they are created (the identification variable with the values supplied by the user). The report variable is initialised by FIP DEVICE MANAGER the first time the medium test function is called. The list of equipment present variable is initialised by the active Bus Arbitrator subscriber as the potential network subscribers are scanned (however, it is not produced until all the subscribers have been scanned at least once).

- **Updating**: the identification variable is never updated. The presence variable is updated (the last byte) as a function of the possible change in characteristics of the local Bus Arbitrator function. The report variable is updated each time the medium test function is activated. The list of equipment present variable is updated by the active Bus Arbitrator each time a subscriber is scanned.

- **Read**: each of these variables is read by means of a universal read request. If the variable to be read is not that of the subscriber making the request, an aperiodic request is made. Hence the need for an aperiodic MPS window in the Bus Arbitrator program. The user is informed of the arrival of the variable requested by calling a procedure written by it.

1.5.3 Creation of list of equipment present variable

This variable is only built (and therefore produced) by the active Bus Arbitrator subscriber whose BA program (the series of frames to be circulated) comprises at least one instruction of the TEST_P type (see **[4]** Subsection 12.1).

There are two ways of processing this instruction depending on which equipment is used to manage medium redundancy:

- either FIP DEVICE MANAGER is used with a ZN130 component or a FIELDUAL system used in a mode compatible with ZN130. In this case, the subscribers present test is partly managed by the FIP DEVICE MANAGER and partly by FIPCODE

- or FIP DEVICE MANAGER is used with a FIELDUAL system used in a ”new” mode (which can be controlled by FULLFIP2). In this case, the subscribers present test is entirely managed by FIPCODE which creates the list of equipment present variables.

1.5.3.1 ZN130 or FIELDUAL in compatible mode

Each time a TEST_P instruction is given, FIP DEVICE MANAGER interrogates a subscriber and asks it to produce its presence variable in order to check its presence.

Transmitting on each channel alternately carries out this interrogation: complete information about a subscriber is only obtained after two TEST_P instructions have been executed.

This also makes it possible to find out the status of each subscriber channel, and detect any ”latent faults” by forcing reception on each channel of all the connected subscribers.

The list of equipment present is updated as a function of the replies, but the list of equipment present variable is only updated after all subscribers whose presence is being tested have been scanned.

The active Bus Arbitrator also produces the list of equipment present variables each time the TEST_P instruction is executed.

The detailed operation of TEST_P instruction processing is as follows:

- Decoding by the FULLFIP2 system (microcode of Bus Arbitrator function) of a special BA instruction used to tell FIP DEVICE MANAGER, via a special event associated with an IT of the IRQ type, that it has to carry out the test of equipment present. The user BA program is no longer executed: packing frames are transmitted by the system (to prevent the election of another BA),

- The interrupt which signals the *equipment present test to be carried out* event is processed by the `fdm_process_its_fip()` procedure as follows:
  - memorising of return address in interrupted BA program,
  - creation and loading of BA program containing instructions for equipment present test,
  - positioning of channel to be used, if double medium,
  - connection to equipment present test BA program.

- After execution of the equipment present test BA program, the FULLFIP2 system (microcode of Bus Arbitrator function) decodes a special BA instruction which it uses to tell FIP DEVICE MANAGER, via a special event associated with an IT of the IRQ type, that it has to process the end of the equipment present test,

- No further BA programs are executed: packing frames are transmitted by the system (to prevent the election of another BA),

- The interrupt which signals the *end of equipment present test to be processed* is processed by the `fdm_process_its_fip()` procedure as follows:
  - updating of list of equipment present as a function of reply obtained,
  - restoration of channel to pre-equipment present test status,
  - return to user BA program.

1.5.3.2 FIELDUAL in new mode

The list of equipment present is created entirely by FIPCODE; noting is done by FIP DEVICE MANAGER.

1.6 Managing WorldFIP data link layer messages

1.6.1 Introduction

The structures of the data and mechanisms used for messaging management are specified so that a complete stack of messages can be developed as easily as possible.

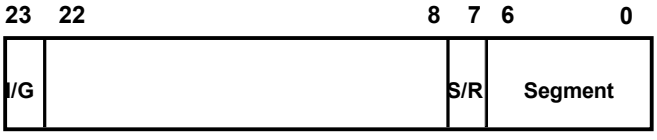
The principle retained is based on the idea of messaging context. Depending on its type, a messaging context is used either to transmit, to receive or to transmit and receive.

Each context is characterised by:

- two addresses on 24 bits in messaging data link format:
  `Local_DLL_Address` and `Remote_DLL_Address`
  - in the case of a context configured for transmission, *Local_DLL_Address* represents the source address and *Remote_DLL_Address* represents the target address for messages transmitted on this context.
  - in the case of a context configured for reception, the messages received on this context have a target address corresponding to *Local_DLL_Address* and a source address at *Remote_DLL_Address*.
  - in the case of a context configured for reception and transmission, messages which have been received on this context have a target address corresponding to *Local_DLL_Address* and a source address at *Remote_DLL_Address* and *Local_DLL_Address* represents the source address and *Remote_DLL_Address* represents the target address of messages transmitted on this context.
- a user procedure (to be written by the user) which will be called by FIP DEVICE MANAGER to signal the arrival of a received message intended for this context, or the end of transmission (line acknowledgement or time-out or fault) of a message transmitted on this context,
- two standard pointers intended for use by the user. They are transmitted to the user by the procedures mentioned above,
- the channel number in the case of a transmission or transmission/reception context,
- the memory zone intended to contain the message in the case of a reception or transmission/reception context.

Note

The message exchange type at the Data Link Level is acknowledged or not acknowledged according to the contents of the address.



I/G	S/R	Segment	Address Type	Message exchange
0	0	00 to 7F	Individual address	Acknowledged *
1	0	00 to 7F	Group address on the segment	Not Acknowledged
X	1	7F	Group address on all the segments	Not Acknowledged

* If you want to exchange unacknowledged messages on an individual address, then use the function *fdm_change_message_acknowledge_type* (see Chapter 3) to change the type of message exchange.

1.6.2 Transmission configuration

The configuration of the Message transmission function consists in creating periodic and aperiodic transmission channels.

For periodic messaging channels, the configuration parameters for a channel are as follows:

- associated identifier,
- channel number: from 1 to 8, a number cannot be associated with more than one identifier.

In the case of aperiodic messaging, there can be only one channel with number 0. It is configured at the same time as the MPS variables (see Page 3-29 `fdm_mps_var_create()` primitive).

1.6.3 Reception configuration

The reception configuration consists in indicating the addresses (WorldFIP 24 bits) to be used for receiving messages.

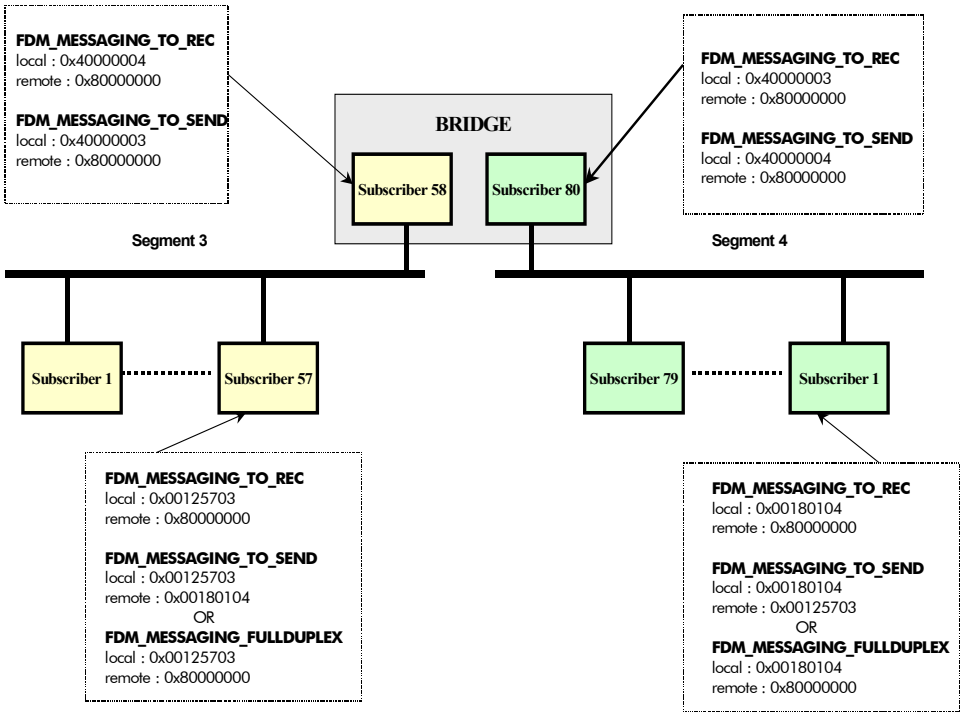
There are two address types for one message: target and source.

Target addresses have to be associated with memory zones allocated to store messages received at this address.

1.6.4 User application that plays the role of a bridge

If you want to use a station as a bridge (at least 2 networks), then you have to write an application that plays the role of a bridge.

The following figure shows you a network with 2 segments and a bridge.



For the subscribers you can create:

- a half-duplex context used only for message transmission (see Chapter 3 - *fdm_messaging_to_send_create* function and the *FDM_MESSAGING_TO_SEND* structure).
- a half-duplex context used only for the reception of messages (see Chapter 3 - *fdm_messaging_to_rec_create* function and the *FDM_MESSAGING_TO_REC* structure).
- a full-duplex context used for the transmission and the reception of messages (see Chapter 3 - *fdm_messaging_fullduplex_create* function and the *FDM_MESSAGING_FULLDUPLEX* structure).

For the bridge functionality you can create:

- a half-duplex context used only for message transmission of (see Chapter 3 - *fdm_messaging_to_send_create* function and the *FDM_MESSAGING_TO_SEND* structure)

- a half-duplex context used only for message reception (see Chapter 3 - *fdm_messaging_to_rec_create* function and the *FDM_MESSAGING_TO_REC* structure)

Note

It is forbidden to create a full-duplex context for the bridge functionality.

The figure shows you an example of how to configure the local and remote addresses of subscribers and of a bridge. In the example subscriber 57 of the segment 3 dialogues with subscriber 1 of segment 4.

Chapter 3 Description of user interface primitives

1. PROVIDED PRIMITIVES, TO BE CALLED BY THE USER

The following conventions have been adopted to describe the primitives:

- the constants written in **BOLD** are those contained in the *fdm.h* file, therefore they can be used directly.
- the lines of C code are written in `courrier`.

1.1 fdm_initialize()

SUMMARY:

FIP DEVICE MANAGER general initialisation function

PROTOTYPE:

```
void fdm_initialize(void);
```

DESCRIPTION:

This function carries out the general initialisation operations for the FIP DEVICE MANAGER, in particular those regarding the management of timers.

It is totally independent of the managed networks and must be the first function called by any application using FIP DEVICE MANAGER Version 4.

INPUT PARAMETERS:

None.

REPORT:

None.

1.2 fdm_get_version()

SUMMARY:

Function for obtaining the software versions used.

PROTOTYPE:

const FDM_VERSION * fdm_get_version (void);

DESCRIPTION:

This function is for finding out which version of FIP DEVICE MANAGER is being used and which version of the FIPCODE software is included.

INPUT PARAMETERS:

None

REPORT:

This function produces a report of the `FDM_VERSION` type, which gives the version, the revision number and any prototype references for both the software packages.

In all the marketed product versions, the prototype reference is equal to 0. If the prototype reference is not 0, this means the version or revision is not yet stabilised.

Definition of type: `FDM_VERSION`

```
Typedef struct {
     FDM_VERSION_ELEMENT fdm;
     FDM_VERSION_ELEMENT fipcode;
} FDM VERSIONS;
Typedef struct {
     char Version;
     char Revision;
     char Indice;
} FDM_VERSION_ELEMENT;
```

1.3 fdm_ticks_counter()

SUMMARY:

Time management function.

PROTOTYPE:

```
void fdm_ticks_counter (void);
```

DESCRIPTION:

This is the function which manages the time-outs of those aspects of the FIP DEVICE MANAGER functionality aspects which use them and which periodically call (periods given in the primitive `fdm_initialize_network()` the internal functionality of medium redundancy management (ex `TEST_MEDIUM()` in Version 2) or, if necessary, the internal self-test functionality. (This is the equivalent of the `IT_HTR_FIP()` function of Version 2.)

The unit of time will therefore be the same for all the networks controlled by the FIP DEVICE MANAGER (i.e. all FIP DEVICE MANAGER instances).

INPUT PARAMETERS:

None

REPORT:

None

1.4 fdm_initialize_network()

SUMMARY:

Function for creating a FIP DEVICE MANAGER utilisation context.

PROTOTYPE:

```
FDM_REF *fdm_initialize_network     (
     const   FDM_CONFIGURATION_SOFT  *User_Soft_Definition,
             FDM_CONFIGURATION_HARD  *User_Hard_Definition,
     const   FDM_IDENTIFICATION      *User_Ident_Param );
```

DESCRIPTION:

This function is for creating a context for using the FIP DEVICE MANAGER for each network (i.e. FULLFIP2/FIPCODE) managed. It must be called as many times as there are networks to manage.

The parameter supplied on output is the reference to be given, directly or indirectly, each time an operation is to be carried out on the corresponding network.

INPUT PARAMETERS:

`User_Soft_Definition:`	User variable of the `FDM_CONFIGURATION_SOFT` type containing the FIP DEVICE MANAGER parameters relating to database configuration.
`User_Hard_Definition:`	User variable of the `FDM_CONFIGURATION_HARD` type containing the FIP DEVICE MANAGER parameters relating to external interfaces.
`User_Ident_Param:`	User variable of the `FDM_IDENTIFICATION` type containing the identification variable parameters.

Description of type `FDM_CONFIGURATION_HARD:`

```
typedef struct {  Unsigned char              K_PHYADR;
                  Char                       MySegment;
                  #if (FDM_WITH_OPTIMIZED_BA == YES)
                  Unsigned char              NumberOfThisBusArbitrator;
                  Unsigned char              GreatestNumberOfBusArbitrator;
                  #else
                  Unsigned char              Reserved[2];
                  #endif
                  #if (FDM_WITH_CHAMP_IO == YES)
                  __Port_type__              LOC_FIP[8];
                  __Port_type__              LOC_FIPDRIVE[4];
                  #else
```

```
                Unsigned char               *volatile LOC_FIP[8];
                Unsigned char               *volatile LOC_FIPDRIVE  [4];
                #endif
                Unsigned short volatile * FREE_ACCES_ADDRESS;

            void (*User_Reset_Component) (struct _FDM_CONFIGURATION_HARD*);
            void (*User_Signal_Fatal_Error)(Struct _FDM_REF * Ref,
            FDM_ERROR_CODE);
            void (*User_Signal_Warning)(Struct _FDM_REF * Ref,
            FDM_ERROR_CODE);

            MEMORY_RN                   *Memory_Management_Ref;
            struct _FDM_REF             *Ptr_Autotests;
} FDM_CONFIGURATION_HARD;
```

`K_PHYADR:`	Physical address of the subscriber on the WorldFIP network in the interval [0..255].
`My_Segment:`	Number of segment used for messaging other than 0 [1..127]; 0 if not used.
`NumberOfThisBusArbitrator`	When `FDM_WITH_OPTIMIZED_BA` is set to YES in the *user.opt.h* file, this parameter represents the number of the Bus Arbitrator of your station [0…255].
`GreatestNumberOfBusArbitrator`	When `FDM_WITH_OPTIMIZED_BA` is set to YES in the *user.opt.h* file, this parameter represents the highest number of the Bus Arbitrators connected on your network.
`LOC_FIP:`	Contains the addresses for accessing the FULLFIP2 circuit registers.
`LOC_FIPDRIVE:`	Contains the addresses for accessing the ZN130 circuit or FIELDUAL circuit registers when it is used in ZN130 compatible mode.
`FREE_ACCESS_ADDRESS:`	Address of the beginning of the memory zone shared between the host microprocessor and FIPCODE/FULLFIP2 if free-access mode is used.
`User_Reset_Component:`	Pointer to a user procedure which is called to carry out hardware resetting of the component.
`User_Signal_Fatal_Error:`	Pointer to the user function which will be called by the FIP DEVICE MANAGER if there is an error.
`User_Signal_Warning:`	Pointer to the user function which will be called by the FIP DEVICE MANAGER if there is a warning signal.
`Memory_Management_Ref:`	Pointer to a structure required when using divided memory manager. Must be initialised at NULL_PTR if divided memory manager not used.
`Ptr_Autotests:`	Pointer required for interrupt self-tests. Must always be initialised at NULL_PTR. The `fdm_initialize_network()` function will provide its value.

Description of type `FDM_ERROR_CODE:`

```
typedef struct {          ENUM CODE_ERROR Fdm_Default;
                          union {

                             struct {
                                      unsigned          _Ustate: 3;
                                      unsigned          Var_State: 8;
                             } Fipcode_Report;
                             unsigned long Additional_Report;

                          } Information;
} FDM_ERROR_CODE
Fdm_Default:
```

Code of the detected fault on an unsigned integer. The list of codes is given in Chapter 5 of this document.

`Additional_Report:`	Supplementary information on `Fdm_Default` when `Fdm_Default` is different from 0x100.
`Fipcode_Report:`	Supplementary information on Fdm_Default when `Fdm_Default` equals 0x100: Errors detected at FIPCODE level.

Description of type: FDM_CONFIGURATION_SOFT:

```
typedef struct {      unsigned short Type;
                      enum _FULLFIP_Mode_List Mode;
                      unsigned short Tslot;
                      unsigned short NB_OF_USER_MPS_VARIABLE
                      unsigned short BA_Dim;
                      unsigned long  FULLFIP_RAM_Dim;
                      unsigned short NB_OF_DIFFERENT_ID_PROG_BA;
                      unsigned short Nr_Of_Repeat;
                      unsigned short Nr_Of_Tx_Buffer [9];
                      void (*User_Present_List_Prog)(struct _FDM_REF *Ref,
                      FDM_PRESENT_LIST *Ptr_Present_List);
                      unsigned short (*User_Identification_Prog)
                      (struct _FDM_REF *Ref, FDM_IDENT_VAR*Ptr_Ident_Var);
                      unsigned short (*User_Report_Prog)(struct _FDM_REF
                      *Ref, FDM_REPORT_VAR*Ptr_Report_Var);
                      unsigned short (*User_Presence_Prog)(struct _FDM_REF
                      *Ref, FDM_PRESENCE_VAR*Ptr_Presence_Var);
                      void (*User_Synchro_BA_Prog) (struct _FDM_REF *Ref,
                      FDM_SYNCHRO_BA_VAR *Ptr_Synchro_BA_Var);
                      unsigned short Test_Medium_Ticks;
                      unsigned short Time_Out_Ticks;
                      unsigned short Time_Out_Msg_Ticks;
                      unsigned short Online_Tests_Ticks;
                      unsigned short Default_Medium_threshold;
```

```
                    #if (FDM_WITH_BA = YES)&& (FDM_WITH_FIPIO == YES)
                    unsigned short Segment_Paramers_Ticks;
                    #endif
                    struct {
                         unsigned short TIMER_CNT_REGISTER;
                         unsigned short MODE_REGISTER;
                    } User_Responsability;
                    void (*User_Signal_Mps_Aper)  (struct _FDM_REF *Ref);
                    void (*User_Signal_Smmps)     (struct _FDM_REF *Ref);
                    void (*User_Signal_Send_Msg)  (struct _FDM_REF *Ref);
                    void (*User_Signal_Rec_Msg)   (struct _FDM_REF *Ref);
                    void *User_Ctxt;
} FDM_CONFIGURATION_SOFT;
```

`Type:` 16-bit word associated with the basic communication configuration. The logic OR of the following flags must be carried out to initialize it. The flags are defined in *fdm.h.*

MESSAGE_RECEPTION_AUTHORIZED:

The message reception queue is automatically created with 64 message descriptors.

TWO_BUS_MODE:

Use of a double medium card with a medium redundancy management component ZN130 or FIELDUAL.

TWO_IMAGE_MODE:

Configuration mode for the communication database: operation with two images (1 and 2).

EOC_PULSE_MODE:

Operating mode for the EOC interrupt of FULLFIP2: pulse mode (otherwise level mode).

IMAGE_2_STARTUP:

The active image on start-up is Image 2.

TEST_RAM_STARTUP:

Selection of complete RAM test on start-up.

EXTERNAL_TXCK:

Position the TXCK signal of the FULLFIP2 circuit on input (Standard use: on output. Therefore this flag not used).

IRQ_CONNECTED:

If the FIP DEVICE MANAGER is compiled with the ”FDM_WITH_DIAG” option set to **YES**, the use of this flag indicates that the FULLFIP2 IRQn interrupt signal test will be carried out during the initialization function (self-test off line).

EOC_CONNECTED:

If the FIP DEVICE MANAGER is compiled with the FDM_WITH_DIAG option set to **YES**, the use of this flag indicates that the FULLFIP2 EOC interrupt signal test will be carried out during the initialisation function (self-test off line).

The logic OR, designed to initialize the *Type* field is to be carried out with the flags corresponding to the options which are to be implemented. Obviously, the ”Type” field must be initialized at 0 beforehand.

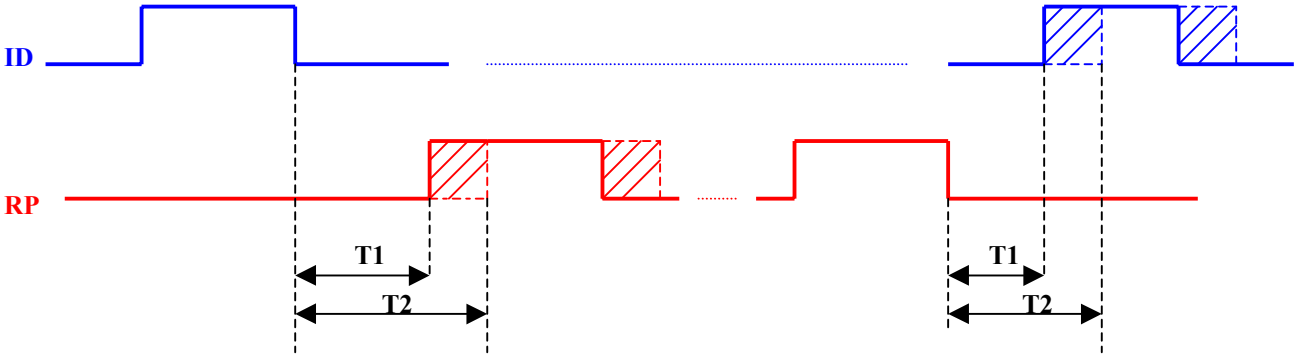
`Mode:` Mode of transmission on the network. Its value must be taken from the following list (FIP = UTE physical layer, WorldFIP = IEC 1158 physical layer):

WORLD_FIP_31	’WorldFIP’-type network using a FULLFIP2 component at the speed of 31.25 kbit/s with a 40 MHz clock.
WORLD_FIP_1000	’WorldFIP’-type network using a FULLFIP2 component at the speed of 1 Mbit/s with a 64 MHz clock.
WORLD_FIP_2500	’WorldFIP’-type network using a FULLFIP2 component at the speed of 5 Mbit/s with a 80 MHz clock.
WORLD_FIP_5000	’WorldFIP’-type network using a FULLFIP2 component at the speed of 2.5 Mbit/s with a 80 MHz clock.
FIP_31	’FIP’-type network using a FULLFIP2 component at the speed of 31.25 kbit/s with a 40 MHz clock.
FAST_FIP_1000	’fast FIP’-type network using a FULLFIP2 component at the speed of 1 Mbit/s with a 64 MHz clock.

FIP_2500	'FIP'-type network using a FULLFIP2 component at the speed of 2.5 Mbit/s with a 80 MHz clock.
FIP_5000	'FIP'-type network using a FULLFIP2 component at the speed of 5 Mbit/s with a 80 MHz clock.
Slow_FIP_1000	'slow FIP'-type network using a FULLFIP2 component at the speed of 1 Mbit/s with a 64 MHz clock.
FDM_OTHER	To set other values than the default ones for the turnaround and silence time. In this case you have to set the fields of the User_Responsability structure.

Note

The default values of the parameters that characterise the transmission on the network are given in the table below.



Mnemonic	Speed	Type of frame delimiters and CRC	Minimum Turnaround Time T1	Maximum Turnaround Time T2	Silence Time
FIP_31	31.25 kbits/s	UTE	424µs	440µs	4096 µs
SLOW_FIP_1000	1 Mbit/s	UTE	41 µs	115 µs	290 µs
FAST_FIP_1000	1 Mbit/s	UTE	10 µs	33 µs	150 µs
FIP_2500	2.5 Mbit/s	UTE	13.5 µs	40 µs	96 µs
FIP_5000	5 Mbit/s	UTE	31.75 µs	40 µs(*)	92 µs
WORLD_FIP_31	31.25 kbits/s	IEC	424 µs	440 µs	4096 µs
WORLD_FIP_1000	1 Mbit/s	IEC	10 µs	33 µs	150 µs
WORLD_FIP_2500	2.5 Mbits/s	IEC	13.5 µs	40 µs	96 µs
WORLD_FIP_5000	5 Mbits/s	IEC	31.75 µs	40 µs(*)	92 µs

Table 3.1 – Default values for the network transmission

(*) If the time variable (ID 9802) is used then this time is 72 µs.

`Tslot:` Value of the unit of time used as a basis for all time calculations. Its value in microseconds must be selected from the following list:

100, 400, 1000, 4000	if Mode = WORLD_FIP_31 or FIP _31
62, 250, 625, 2500	if Mode = FAST_ FIP _1000 or SLOW_ FIP _1000 or WORLDFIP_1000
50, 200, 500, 2000	if Mode = WORLD_FIP_2500, WORLD _FIP_5000, FIP_2500 or FIP_5000.

`NB_OF_USER_MPS_VARIABLE:`	Maximum number of MPS variables that will be created for the FDM instance being defined.
`BA_Dim:`	For a station intended to be a Bus Arbitrator, this is the maximum size (in number of 16-bit words) reserved for the arbitrating tables. The value must be a multiple of **400**H in the range [0..**FFFF**H].
`FULLFIP_RAM_Dim:`	Size of the private RAM of the FULLFIP2 component (in 16-bit words). The value must be between [**6000**H..**100000**H].
`NB_OF_DIFFERENT_ID_PROG_BA:`	Maximum number of different identifiers used in a BA macrocycle.
`Nr_Of_Repeat:`	Maximum number of repetitions authorised in acknowledged messaging after the initial attempt, at the data link layer. The value must be between [0 and 3]. "0" corresponds to only one attempt (no repetition) and "3" corresponds to 4 attempts (i.e. a maximum of 3 repetitions).
`Nr_Of_Tx_Buffer:`	Number of buffers reserved for message transmission. The FIPCODE software makes it possible to consider a maximum of 32 messages for each of the 9 channels.
`User_Present_List_Prog:`	Pointer to a user function which will be called by the FIP DEVICE MANAGER after the user has requested a list of equipment present.
`User_Synchro_BA_Prog:`	Pointer to a user function which will be called by the FIP DEVICE MANAGER after the user has requested a sync variable.
`User_Presence_Prog:`	Pointer to a user function which will be called by the FIP DEVICE MANAGER after the user has requested a presence variable.
`User_Identification_Prog:`	Pointer to a user function which will be called by the FIP DEVICE MANAGER after the user has requested an identification variable.
`User_Report_Prog:`	Pointer to a user function which will be called by the FIP DEVICE MANAGER after the user has requested a report variable.

Note

The above three user functions must provide the following values in response to the call: **NO_VAR_DELETE** if FDM is not meant to delete the variable, and **VAR_DELETE** if it is.

`Test_Medium_Ticks:`	Periodicity of call of internal function for testing the mediums. One ”ticks” = one call of fdm_ticks_counter().
`Time_Out_Ticks:`	Number of ticks to trigger a time-out: common to all primitives requiring idling time except messaging.
`Time_Out_Msg_Ticks:`	Number of ticks to trigger a time-out for messaging function.
`Online_Tests_Ticks:`	Number of ticks to trigger execution of the online self-test primitive. If it equals 0, then the user is responsible for calling this function.
`Default_Medium_threshold:`	Threshold of medium default between two calls of internal function that tests the mediums used to declare medium NOK (for reception). The value must be given as a percent (%) of transactions without errors compared to transactions with errors
`Segment_Paramers_Ticks:`	Only in FIPIO mode: periodicity of the transmission of the Segment_Parameters SM_MPS variable by the FIPIO manager
`User_Responsability:`	Data structure which, if the ”Mode” field is **FDM_OTHER**, makes it possible to configure the internal registers with values compatible with the operating modes of earlier versions which are no longer taken into account with V4 (see the FIPCODE User Manual **[6]** Chapter 2, Subsection 1.1)
`Timer_CNT_Register:`	see description of FIP_Timer_Cnt parameter in FIPCODE User Manual **[6]** Chapter 2, Subsection 1.1
`Mode_Register:`	see description of FIP_Extend_Par parameter in FIPCODE User Manual **[6]** Chapter 2, Subsection 1.1.
`User_Signal_Mps_Aper:`	Pointer to a user function which will be called by the FIP DEVICE MANAGER following reception or transmission of the value of a *universa*” type variable.
`User_Signal_Smmps:`	Pointer to a user function which will be called by the FIP DEVICE MANAGER following reception of the value of a network management variable (list of equipment present, presence, report, identification or BA sync).
`User_Signal_Rec_Msg:`	Pointer to a user function which will be called by the FIP DEVICE MANAGER following reception of a message.
`User_Signal_Send_Msg:`	Pointer to a user function which will be called by the FIP DEVICE MANAGER following a transmission request or on reception of acknowledgement of transmission.
`User_Ctxt:`	User identification of his application.

Description of type `FDM_PRESENT_LIST`

```
typedef struct {
                        unsigned char           Report;
                        unsigned char           Nop;
                        unsigned char           Bus1_P[32];
                        unsigned char           Bus2_P[32];
} FDM_PRESENT_LIST;
```

`Report:`	Execution report of the *list of equipment present* variable read request
VAR_TRANSFERT:	The list has been received, therefore the following two parameters are significant.
NO_VAR_TRANSFERT:	The list of equipment present has not been transferred, therefore the following two parameters are not significant.

`Bus1_P:` Table containing a list of bits indicating that it is present on Channel 1 for each subscriber: bit = 1 if present)

`Bus1_P[0] bit 0:`	`Subscriber 0`
`Bus1_P[0] bit 7:`	`Subscriber 7`
`Bus1_P[1] bit 0:`	`Subscriber 8`
`Bus1_P[1] bit 7:`	`Subscriber 15`
`Bus1_P[2] bit 0:`	`Subscriber 16`
`Bus1_P[31] bit 0:`	`Subscriber 248`
`Bus1_P[31] bit 7:`	`Subscriber 255`

`Bus2_P:` idem Bus1_P for Channel 2

Description of type `FDM_PRESENCE_VAR:`

```
typedef struct {        unsigned char          Report;
                        unsigned char          Subscriber;
                        unsigned char          Ident_Length;
                        struct {
                           unsigned char       BA_Status;
                           unsigned char       BA_Priority;
                        } BA_Information;
} FDM_PRESENCE_VAR;
```

`Report:` Execution report of the presence variable read request:

VAR_TRANSFERT: The presence variable has been received, therefore the following three parameters are significant.

NO_VAR_TRANSFERT: No transfer of the variable, therefore the following two parameters are not significant.

`Subscriber:` Number of the subscriber (physical address) from which the presence variable was requested.

`Ident_Length:` Length in bytes in the interval [0..256] of the identification variable of the subscriber from which the identification variable was requested.

`BA_Status:` Status of Bus Arbitrator function on the subscriber interrogated.

000b:	The subscriber is not a potential Bus Arbitrator.
001b:	The subscriber is a potential Bus Arbitrator but the Bus Arbitrator function has not been started up.
010b:	The subscriber is a potential Bus Arbitrator, the Bus Arbitrator function has been started up, but the subscriber is not elected active Bus Arbitrator.
011b:	The subscriber is a potential Bus Arbitrator, the Bus Arbitrator function has been started up, and the subscriber is elected active Bus Arbitrator.

BA_Priority:	Level of priority in the range [0..15] of the subscriber interrogated. 0 corresponds to the maximum level of priority and 15 to the minimum. This level of priority is taken into account in the Bus Arbitrator election time-slot calculation.

Description of type FDM_IDENT_VAR:

```
typedef struct {        unsigned char Report;
                        unsigned char Subscriber;
                        unsigned short Identification_Length;
                        unsigned char Identification [128];
} FDM_IDENT_VAR
```

Report:	Execution report of the identification variable read request:
VAR_TRANSFERT:	The identification variable has been received, therefore the following two parameters are significant.
NO_VAR_TRANSFERT:	No transfer of the identification variable, therefore the following two parameters are not significant.
Subscriber:	Number of the subscriber (physical address) from which the identification variable was requested.
Identification_length:	Size in bytes of the identification variable.
Identification:	Content of the identification variable.

Description of type FDM_REPORT_VAR:

```
typedef struct {        unsigned char       Report;
                        unsigned char       Subscriber;
                        unsigned short      Nb_Of_Transaction_Ok_1;
                        unsigned short      Nb_Of_Transaction_Ok_2;
                        unsigned short      Nb_Of_Frames_Nok_1;
                        unsigned short      Nb_Of_Frames_Nok_2;
                        unsigned short      Activity_status;
} FDM_REPORT_VAR;
```

Report:	Execution report of the report variable read request:
VAR_TRANSFERT:	The report variable has been received, therefore the following two parameters are significant.
NO_VAR_TRANSFERT:	No transfer of the report variable therefore the following two parameters are not significant.
Subscriber:	Number of the subscriber (physical address) from which the identification variable has been requested.
Nb_Of_Transaction_OK_1:	Number of transactions without errors per unit of time on Channel 1.
Nb_Of_Transaction_OK_2:	Number of transactions without errors per unit of time on Channel 2.
Nb_Of_Frame_NOK_1:	Number of frames received with errors per unit of time on Channel 1.
Nb_Of_Frame_NOK_2:	Number of frames received with errors per unit of time on Channel 2.

`Activity_Status:`	Bit 0: status of the quality of Channel 1 in transmission mode	(0=Nok, 1=Ok)
	Bit 1: status of the quality of Channel 2 in transmission mode	(0=Nok, 1=Ok)
	Bit 2: status of the quality of Channel 1 in reception mode	(0=Nok, 1=Ok)
	Bit 3: status of the quality of Channel 2 in reception mode	(0=Nok, 1=Ok)
	Bit 4: status of the validation of Channel 1	(0=invalidated, 1=validated)
	Bit 5: status of the validation of Channel 2	(0=invalidated, 1=validated)
	Bit 6: status of traffic on Channel 1	(0=no traffic, 1=traffic)
	Bit 7: status of traffic on channel 2	(0=no traffic, 1= traffic)
	Bit 8: status of channel 1	(0=Nok, 1=Ok) OK if (Bit 0 and Bit 2 and Bit 4 and Bit 6) 00
	Bit 9: status of channel 2	(0=Nok, 1=Ok) OK if (Bit 1 and Bit 3 and Bit 5 and Bit 7) 00

Note

The unit of time used is the interval between two calls of the function for testing the mediums using the FIP DEVICE MANAGER. (cf `CONFIGURATION_SOFT.Test_Medium_Ticks`)

Description of type `FDM_IDENTIFICATION:`

```
typedef struct {        char *Vendor_Name;
                        char *Model_Name;
                        char *Revision;
                        char *Tag_Name;
                        char *SM_MPS_Conform;
                        char *SMS_Conform;
                        char *PMDP_Conform;
                        char *Vendor_Field;
} FDM_IDENTIFICATION;
```

where

`Vendor_Name`:	Pointer to a string of characters which contains the name of the equipment supplier. This string must end with the character '\0'. The length of this string is limited by the size of the identification variable and must be within the [2..119] range.
`Model_Name`:	Pointer to a character string which contains the name of the equipment at the supplier's premises. This string must end with the character '\0'. The length of this string is limited by the size of the identification variable and must be within the [2..119] range.
`Revision`:	Pointer to a character string which contains the revision reference of the supplier's equipment. The value of this reference must be between [0..255].
`Tag_Name`:	Pointer to a character string which contains the Tag-Name of the equipment in the context of the user application. This string must end with the character '\0'. The length of this string is limited by the size of the identification variable and must be within the [2..33] range.
`SM_MPS_Conform`:	Pointer to a byte which contains the SMMPS conformity class (network management) of the equipment. The conformity class will be equal to **10**H: thus a report variable will be created automatically in the two images (**images 1** & **2)** of the communication database.
`SMS_Conform`:	Pointer to a byte table that contains the network management conformity class based on SMS messaging. The value of the conformity class is encoded on an unspecified number of bytes, between [1..115] limited by the maximum size of the identification variable. The first element in the table (unsigned 8-bit integer) contains the size of this value, which can take a value from the [2..116] range.
`PMDP_Conform`:	Pointer to a byte table which contains the class of conformity to MPS and SubMMS standards, data link, and physical layer. The value of the conformity class is encoded on an unspecified number of bytes, between [1..115] and limited by the maximum size of the identification variable. The first element in the table (unsigned 8-bit integer) contains the size of this value, which can be a value from the [2..116] range.
`Vendor_Field`:	Pointer to a string of characters which contains additional supplier-specific information about the equipment. This string must end with the character '\0'. The length of this string is limited by the size of the identification variable and must be within the [2..33] range.

Description du type: `FDM_SYNCHRO_BA_VAR` :

```
typedef struct {        unsigned char Report;
                        unsigned char Hi_MC_Nr;
                        unsigned char Subscriber;
} FDM_SYNCHRO_BA_VAR;
```

`Report`	Execution report of the Synchro_BA variable read request.
VAR_TRANSFERT	The Synchro_BA variable has been received, therefore the following parameters are significant.
NO_VAR_TRANSFERT	The Synchro_BA variable has not been received, therefore the following parameters are not significant.
`Hi_MC_Nr`:	High order byte of the BA macrocycle number that is running.
`Subscriber`:	Physical address of the active Bus Arbitrator that produced the Synchro_BA variable.

REPORT:

The `fdm_initialize_network()` primitive returns a pointer to a `FDM_REF`-type structure.

If initialisation could not take place correctly, the `NULL_PTR` pointer is returned.

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

1.5 fdm_stop_network()

SUMMARY:

Function for stopping/deleting a FIP DEVICE MANAGER utilisation context (i.e. stopping the network corresponding to this instance.).

PROTOTYPE:

```
void fdm_stop_network   (FDM_REF    *Ptr_Network_Identification);
```

DESCRIPTION:

This primitive makes it possible to end the use of a FIP DEVICE MANAGER context, created by the `fdm_initialize_network()` function by restoring all the allocated data (memory).

Each object which is dependent on the context to be deleted must be deleted first (AE/LE, variables, messaging etc.)

INPUT PARAMETERS:

`Ptr_Network_Identification:` Pointer to a data structure of the `FDM_REF` type.

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

1.6 fdm_online_test

SUMMARY:

Function for running self-diagnosis functions during operation.

PROTOTYPE:

```
unsigned shortfdm_online_test (
       FDM_REF          *Ptr_Network_Identification);
```

DESCRIPTION:

This function enables the FIP DEVICE MANAGER to run a part of the self-diagnosis functions during operation each time a call is made.

This function can be used if `CONFIGURATION_SOFT.Online_Tests_Ticks` = 0. Otherwise it is the FIP DEVICE MANAGER which automatically activates the self-tests during operation at a frequency equal to the `CONFIGURATION_SOFT.Online_Tests_Ticks` parameter.

INPUT PARAMETERS:

`Ptr_Network_Identification:` Pointer to a data structure of the `FDM_REF` type

REPORT:

This function produces a report that reads:

FDM_OK if the function has been carried out correctly

FDM_NOK: if the function has not been carried out correctly; in this case the user function `User_Signal_Warning` has been called at least once to warn of an FDM_ERROR_CODE type fault that is specified in Chapter 5 of this document.

1.7 fdm_valid_medium()

SUMMARY:

Function for initialising medium redundancy management

PROTOTYPE:

```
unsigned short  fdm_valid_medium    (
       FDM_REF          *Ptr_Network_Identification,
       enum MEDIUM_DEF  Bus_Mode);
```

DESCRIPTION:

This function initialises the medium management functionality by indicating which channels are used and what fault threshold to take into account (see Chapter 2, Subsection 1.2. of this document).

INPUT PARAMETERS:

`Ptr_Network_Identification`: Pointer to a data structure of the `FDM_REF` type.

`Bus_Mode:`	_ MEDIUM_1:	Selection of Channel 1
	_ MEDIUM_2:	Selection of Channel 2
	_ MEDIUM_1_2:	Selection of Channels 1 and 2

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

Description of type `MEDIUM_DEF:`

```
enum MEDIUM_DEF {       _MEDIUM_1       = 1, /
                        _MEDIUM_2       = 2,
                        _MEDIUM_1_2     = 3
}
```

REPORT:

This function submits a report as specified by the primitive `fdm_online_test`().

Note

This function must be called on all the stations with a double-medium card and on all stations with a single-medium card which are potential Bus Arbitrators.

Note

Precision for the different types of initialisation required to manage medium redundancy in accordance with the configurations used:

Double-medium card and double-medium network:

`CONFIGURATION_SOFT Type = TWO_BUS_MODE ...`"other possible flags" `VALID_MEDIUM (3, threshold);`

Double-medium card and single-medium network:

`CONFIGURATION_SOFT Type = TWO_BUS_MODE |...`"other possible flags" `VALID_MEDIUM (1, threshold);`
or
`VALID_MEDIUM (2, threshold);`

Mono-medium card:

with or without `TWO_BUS_MODE` to build `CONFIGURATION_SOFT.Type`
`VALID_MEDIUM (1, threshold)`

1.8 fdm_process_its_fip()

SUMMARY:

Function for processing EOC and IRQ interrupts

PROTOTYPE:

```
unsigned short fdm_process_its_fip (
     FDM_REF *Ptr_Network_Identification);
```

DESCRIPTION:

This is the function for processing EOC interrupts, if it is configured by level and IRQ interrupts when they are together on the same interrupt level. This function must, in this case, be called each time an EOC or IRQ interrupt occurs.

INPUT PARAMETERS:

`Ptr_Network_Identification:` Pointer to a data structure of the `FDM_REF` type.

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report of the following type:

EOC_IRQ_TO_PROCESS: if there is still at least one more interrupt to process

NO_EOC_IRQ_TO_PROCESS: if there are no more left to process

Note

EOC must be configured by level.

1.9 fdm_process_it_eoc()

SUMMARY:

Function for processing EOC interrupt only.

PROTOTYPE:

```
unsigned short fdm_process_it_eoc
       (FDM_REF *Ptr_Network_Identification);
```

DESCRIPTION:

This function is for processing EOC interrupts. In this case it must be called each time an EOC interrupt occurs if it is configured by level.

INPUT PARAMETERS:

`Ptr_Network_Identification:` Pointer to a data structure of the `FDM_REF` type

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report of the following type:

EOC_TO_PROCESS: if there is still at least one interrupt to process

N**O_EOC_TO_PROCESS:** if there are no more interrupts to process

1.10 fdm_process_it_irq()

SUMMARY:

Function for processing IRQ interrupts only.

PROTOTYPE:

```
unsigned short fdm_process_it_irq (
       FDM_REF *Ptr_Network_Identification);
```

DESCRIPTION:

This function is for processing IRQ interrupts. In this case it must be called each time an IRQ interrupt occurs.

INPUT PARAMETERS:

`Ptr_Network_Identification:` Pointer to a data structure of the `FDM_REF` type.

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report of the following type:

IRQ_TO_PROCESS: if there is still at least one interrupt to process

NO_IRQ_TO_PROCESS: if there are no more interrupts to process

1.11 fdm_switch_image()

SUMMARY:

Function for switching operation from the AE/LEs of one image, to the other.

PROTOTYPE:

```
unsigned short fdm_switch_image     (
     FDM_REF             *Ptr_Network_Identification,
     enum IMAGE_NR       Image_Nr);
```

DESCRIPTION:

This function is for switching operation from the AE/LE of one image, to the other for all AE/LEs of the FIP DEVICE MANAGER instance involved.

INPUT PARAMETERS:

`Ptr_Network_Identification:` Pointer to a data structure of the `FDM_REF` type.

`Image_Nr` Identification of the image to which the switch must be made: `IMAGE_1` or `IMAGE_2`

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report as specified by the primitive `fdm_stop_network`().

1.12 fdm_get_image()

PROTOTYPE:

```
enum IMAGE NR fdm_get_image          (
     FDM_REF*Ptr_Network_Identification);
```

DESCRIPTION:

This function makes it possible to know which image all the AE/LEs of the same FIP DEVICE MANAGER instance are operating on.

INPUT PARAMETERS:

`Ptr_Network_Identification:` Pointer to a data structure of the `FDM_REF` type.

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report indicating which image is active, stating:

IMAGE_1: if the function has been executed correctly, and AE/Ls are operating on Image 1.

IMAGE_2: if the function has been executed correctly, and AE/Ls are operating on Image 2.

1.13 fdm_change_test_medium_ticks()

PROTOTYPE:

```
fdm_change_test_medium_ticks        (
     FDM_REF            *Ptr_Network_Identification,
     unsigned long      New_Value;);
```

DESCRIPTION:

This function is for modifying the value of the frequency with which the FIP DEVICE MANAGER calls the medium testing function.

INPUT PARAMETERS:

`Ptr_Network_Identification:` Pointer to a data structure of the `FDM_REF` type.

`New_Value:` New value of the frequency for calling the medium testing function.

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

None

1.14 fdm_ae_le_create()

SUMMARY:

Function for creating an AE/LE

PROTOTYPE:

```
FDM_AE_LE_REF * fdm_ae_le_create    (
       FDM_REF          *Ptr_Network_Identification,
       int              Nb_Vcom,
       enum FDM_ALLOWED Type);
```

DESCRIPTION:

This function enables an AE/LE to be created. On creation, this AE/LE does not contain any variables.

INPUT PARAMETERS:

`Nb_Vcomr`: Max. number of MPS variables that the AE/LE could contain.

`Ptr_Network_Identification:` Pointer to a data structure of the `FDM_REF` type.

Type:	CHANGE_ALLOWED:	The P/C attribute of the variables contained in this AE/LE may be modified
	CHANGE_NOT_ALLOWED:	The P/C attribute of the variables contained in this AE/LE may not be modified

Description of type `FDM_REF`:

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function produces either:

- a NULL_PTR pointer if it has not been able to run correctly, in which case the `User_Signal_Warning()` function has been called at least once.
- or a pointer to an `FDM_AE_LE_REF` type structure (inside FIP DEVICE MANAGER: not manipulated by the user).

1.15 fdm_ae_le_delete()

SUMMARY:

Function for deleting an AE/LE.

PROTOTYPE:

```
unsigned short fdm_ae_le_delete     (
       FDM_AE_LE_REF *Ptr_AE_LE);
```

DESCRIPTION:

This function enables an AE/LE to be deleted. The AE/LE to be deleted must be in CONFIGURATION mode. (It must not be in OPERATION mode).

INPUT PARAMETERS:

`Ptr_AE_LE:` Pointer to a data structure of the `FDM_AE_LE_REF` type.

Description of type `FDM_AE_LE_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report as specified by the primitive `fdm_online_test()`.

1.16 fdm_ae_le_start()

SUMMARY:

Function for starting up an AE/LE.

PROTOTYPE:

```
unsigned short fdm_ae_le_start      (
       FDM_AE_LE_REF *Ptr_AE_LE);
```

DESCRIPTION:

This function enables an AE/LE to be started up.

INPUT PARAMETERS:

`Ptr_AE_LE:` Pointer to a data structure of the `FDM_AE_LE_REF` type.

Description of type `FDM_AE_LE_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report as specified by the primitive `fdm_online_test()`.

1.17 fdm_ae_le_get_state()

SUMMARY:

Function for obtaining the state of an AE/LE.

PROTOTYPE:

```
enum FDM_AE_LE_STATE  fdm_ae_le_get_state (FDM_AE_LE_REF *Ptr_AE_LE);
```

DESCRIPTION:

This function makes it possible to know the operating state of an AE/LE.

INPUT PARAMETERS:

`Ptr_AE_LE:` Pointer to a data structure of the `FDM_AE_LE_REF` type.

Description of type `FDM_AE_LE_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function produces a report that reads:

AE_LE_NOT_EXIST: if the function has been carried out correctly: AE/LE non-existent.

AE_LE_CONFIG: if the function has been carried out correctly: AE/LE being configured.

AE_LE_RUNNING: if the function has been carried out correctly: AE/LE in operation.

1.18 fdm_ae_le_stop()

SUMMARY:

Function for shutting down an AE/LE.

PROTOTYPE:

```
unsigned short fdm_ae_le_stop       (
       FDM_AE_LE_REF *Ptr_AE_LE);
```

DESCRIPTION:

This function enables an AE/LE to be shut down.

INPUT PARAMETERS:

`Ptr_AE_LE:` Pointer to a data structure of the `FDM_AE_LE_REF` type.

Description of type `FDM_AE_LE_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function produces a report as specified in Chapter 1, Subsection 4.2. of this document.

1.19 fdm_mps_var_create()

SUMMARY:

Function for creating an MPS variable.

PROTOTYPE:

```
FDM_MPS_VAR_REF *fdm_mps_var_create (
       FDM_AE_LE_REF *Ptr_AE_LE,
       const FDM_XAE    *Var);
```

DESCRIPTION:

This function is for creating a variable and loading it into an AE/LE.

INPUT PARAMETERS:

`Ptr_AE_LE:`	Pointer to a data structure of the `FDM_AE_LE_REF` type.
`Var:`	Pointer to a data structure in the `FDM_XAE` format, describing all the parameters of a variable.

Description of type `FDM_AE_LE_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

Definition of type: `FDM_XAE`

```
typedef struct {
                  struct {
                    unsigned Reserved        :5;
                    unsigned Position        :2;
                    unsigned Communication   :2;
                    unsigned Scope           :1;
                    unsigned With_Time_Var   :1;
                    unsigned Refreshment     :1;
                    unsigned Indication      :1;
                    unsigned Priority        :1;
                    unsigned Rqa             :1;
                    unsigned MSGa            :1;
                    } Type
                    unsigned short   ID;
                    unsigned short   Var_Length;
                  unsigned long  Refreshment_Period;

                  unsigned long  Promptness_Period;
                  int         Rank;
                  struct {
```

```
                    void (*User_Signal_Synchro)   (struct _FDM_REF*);
                    void (*User_Signal_Asent)     (struct _FDM_MPS_VAR_REF *);
                    void (*User_Signal_Areceived) (struct _FDM_MPS_VAR_REF*);
                    void (*User_Signal_Var_Prod)  (struct _FDM_MPS_VAR_REF *,
                                                   unsigned short);
                    void (*User_Signal_Var_Cons)  (struct _FDM_MPS_VAR_REF*,
                                          FDM_MPS_READ_STATUS
                                                   FDM_MPS_VAR_DATA * );
                } Signals;

} FDM_XAE;
```

`Type Reserved:` Unused. Must be initialised at 0.

`Type Position:` Position of the variable in relation to the different images of the AE-LE.

- 0: AE-LE single image
- 1: AE-LE double image. The variable is only specified in Image 1.
- 2: AE-LE double image. The variable is only specified in Image 2.
- 3: AE-LE double image. The variable is specified in Image 1 and Image 2.

`Type.Communication:` Type of variable

- **VAR_SYNCHRO:** Synchronisation variable
- **VAR_PRODUCED:** Produced variable
- **VAR_CONSUMED:** Consumed variable
- **VAR_CONS_PROD:** Consumed variable on creation, but which may then change type.

`Type.Scope:` only used for MPS variables. Initialised at 0 for a synchronisation event.

- `0:` Remote variable. Only ”universal” reads or writes possible. The variable must be specified in Image 1.
- `1:` Local variable. Local reads and writes can be carried out if the variable is specified in Image 1 or Image 2. The ”universal” reads and writes are possible if the variable is specified in Image 1.

`Type.With_Time_Var:`

- `0:` The variable is not a variable with dynamic refresh status.
- `1:` The variable is a variable with dynamic refresh status (see Subsection 1.3.4.2. of Chapter 2).

`Type.Refreshment:` Used only for communication variables. Must be initialised at 0 for a synchronisation event. Makes it possible to know if there is a refresh status to be read for a consumed variable.

- `0:` no refresh status to be read
- `1:` refresh status to be read
  Note: can only be set to 1 if `Type.Communication` = **VAR_CONSUMED** at the **VAR_CONS_PROD**.

`Type.Indication:` Preparation of a transmission indication (A_SENT) or reception indication (A_RECEIVE) following a network transaction involving this variable. Must be set to 0 if `Type.Communication` = **VAR_SYNCHRO**

- `0:` no indication requested
- `1:` indication requested

`Type.Priority:` Level of priority of MPS aperiodic requests that can be addressed to the Bus Arbitrator using the identifier associated with that variable. Must be at zero if `Type.Communication` = **VAR_PRODUCED** or if `Type.With_Time_Var` = 1

- `0:` Non urgent
- `1:` Urgent

`Type.RQa:` Authorisation to carry out MPS aperiodic requests using the identifier associated with that variable. Must be at 0 if `Type.Communication` = **VAR_PRODUCED** or if `Type.With_Time_Var` = 1

- `0:` MPS aperiodic requests not authorised
- `1:` MPS aperiodic requests authorised

`Type.MSGa:` Authorisation to carry out messaging aperiodic requests using the identifier associated with that variable. Must be at 0 if `Type.Communication` = **VAR_PRODUCED** or if `Type.With_Time_Var` = 1

- `0:` not authorised
- `1:` Authorised

`ID:` Value of the identifier associated with the variable. This value belongs to one of the following value ranges:

[0000H	...	0FFFH]
[3000H	...	8FFFH]
[9800H	...	9FFFH]
[B000H	...	FFFFH]

.

`Var_Length:`

- for a single MPS variable, this is the length, in bytes, of the data for the user to read or write. This length is between [1..125] if the refresh status is requested, [1..126] if it is not.
- In the case of 5 Mbits/s the length should be greater than 6 bytes.
- must be initialised at 0 for a synchronisation event.
- for a produced variable with dynamic refresh status, this is the length in bytes of the data to be written. This size is between [12..120] and must be even.
- for a consumed variable with dynamic refresh status, this is the length in bytes of the data to be read. This size is between [12..120] and must be even. In this case, this size does not include the size of the time differential calculated by the consumer, but only that of the time differential calculated by the producer.

`Rank:` Rank of the variable in the AE/LE (user identification of the variable).

`Refreshment_Period:` Refresh period if the produced variable is expressed in microseconds.

`Promptness_Period:` Promptness period if consumed variable is expressed in microseconds

`User_Signal_Synchro:` User procedure which will be called on reception of a synchronisation variable if the EOC interrupt is configured in ”level” mode.

`User_Signal_Asent:` User procedure that will be called on production on the network of a produced local variable if an event is associated with it.

`User_Signal_Areceive:`	User procedure that will be called on reception from the network of a consumed local variable if an event is associated with it.
`User_Signal_Var_Prod:`	User procedure which will be called: • on production on the network following a universal-type write of this variable (if produced variable) • on expiry of the time-out associated with the previous one.
`User_Signal_Var_Cons:`	User procedure which will be called: • on reception from the network following a universal-type read of this variable (if consumed variable) • on expiry of the time-out associated with the previous one
`User_Get_Value`	User procedure which will be called by the write primitive of a variable with dynamic refresh (`fdm_mps_var_time_write_loc`) if `Type.With_Time_Var = 0`. This parameter must then be set to `NULL_PTR` to request the user to supply the address of the byte table containing the value to be transmitted.

Definition of type: `FDM_MPS_VAR_DATA`

```
typedef struct FDM_XAE_REF{
                    unsigned char       Pdu_Type;
                    unsigned char       Pdu_Length;
                    unsigned char       Fbuffer[126];
} FDM_MPS_VAR_DATA;
```

`Pdu_Type:`	Type of PDU: 40H for MPS variables, 50H for SM_MPS
`Pdu_Length:`	Length of PDU. Equivalent to: • size of the variable if no refresh status configured • size of the variable if refresh status
`FBuffer[126]`	Value of the variable in the form of a byte string

Definition of type: `FDM_MPS_VAR_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function produces either:

- a NULL_PTR pointer if it has not been able to run correctly, in which case the `User_Signal_Warning()` function has been called at least once.
- or a pointer to an `FDM_MPS_VAR_REF` type structure.

1.20 fdm_mps_var_change_id()

SUMMARY:

Function for modifying the value of the ID associated with a variable

PROTOTYPE:

```
unsigned short fdm_mps_var_change_id (
       FDM_MPS_VAR_REF  *Ptr_MPS_VAR,
       unsigned short   New_ID);
```

DESCRIPTION:

This function modifies the value of the identifier associated with an MPS variable when it has already been created but when the AE/LE to which it belongs has not yet started up.

INPUT PARAMETERS:

`Ptr_MPS_VAR:`	Pointer to a data structure of the `FDM_MPS_VAR_REF` type. This parameter enables the variable on which the primitive must act, to be identified.
`New_ID:`	New value of identifier to be associated with the variable.

Description of type `FDM_MPS_VAR_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function produces a report as specified in Chapter 1 Subsection 4.2. of this document.

1.21 fdm_mps_var_change_periods()

SUMMARY:

Function for modifying the refresh or prompt periods of a variable.

PROTOTYPE:

```
unsigned short fdm_mps_var_change_periods (
       FDM_MPS_VAR_REF  *Ptr_MPS_VAR,
       unsigned long    Refreshment_Period,
       unsigned long    Promptness_Period);
```

DESCRIPTION:

This function makes it possible to modify the value of the prompt and refresh windows associated with an MPS variable.

INPUT PARAMETERS:

`Ptr_MPS_VAR:`	Pointer to a data structure of the `FDM_MPS_VAR_REF` type. This parameter enables the variable on which the primitive must act, to be identified.
`Refreshment_Period:`	New value of the period to be associated with the variable when it is produced (in microseconds).
`Promptness_Period:`	New value of the period to be associated with the variable when it is consumed (in microseconds).

Description of type `FDM_MPS_VAR_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user**.**

REPORT:

This function produces a report as specified in Chapter 1 Subsection 4.2. of this document.

1.22 fdm_mps_var_change_priority()

SUMMARY:

Function for modifying the priority attribute of a variable.

PROTOTYPE:

```
unsigned short fdm_mps_var_change_priority (
       FDM_MPS_VAR_REF  *Ptr_MPS_VAR,
       unsigned         New_Priority);
```

DESCRIPTION:

This function makes it possible to modify the value of the priority of aperiodic transfer requests which is liable to support an MPS variable.

INPUT PARAMETERS:

`Ptr_MPS_VAR:`	Pointer to a data structure of the `FDM_MPS_VAR_REF` type. This parameter enables the variable on which the primitive must act, to be identified.
`New_Priority:`	New value of the priority to be associated with the variable (for any aperiodic transfer requests) **FIFO_URGENT** or **FIFO_NORMAL.**

Description of type `FDM_MPS_VAR_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function produces a report as specified in Chapter 1 Subsection 4.2. of this document.

1.23 fdm_mps_var_change_prod_cons()

SUMMARY:

Function for modifying the P/C attribute of a variable.

PROTOTYPE:

```
unsigned short fdm_mps_var_change_prod_cons(
       FDM_MPS_VAR_REF              *Ptr_MPS_VAR,
       enum _FDM_CHANGE_TYPE        New_Image1_Prod_Cons,
       enum _FDM_CHANGE_TYPE        New_Image2_Prod_Cons);
```

DESCRIPTION:

This function makes it possible to modify the value of the Produced/Consumed attribute associated with an MPS variable when the AE/LE to which it belongs is started up.

INPUT PARAMETERS:

`Ptr_MPS_VAR:`	Pointer to a data structure of the `FDM_MPS_VAR_REF` type. This parameter enables the variable on which the primitive must act, to be identified.
`New_Image1_Prod_Cons:`	New value of the produced/consumed attribute for Image 1 of the variable: **CONSUMED** or **PRODUCED.**
`New_Image2_Prod_Cons:`	New value of the produced/consumed attribute for Image 2 of the variable: **CONSUMED** or **PRODUCED.**

Description of type `FDM_MPS_VAR_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function produces a report as specified in Chapter 1 Subsection 4.2. of this document.

1.24 fdm_mps_var_change_rqa()

SUMMARY:

Function for modifying the RQa attribute of a variable.

PROTOTYPE:

```
unsigned short fdm_mps_var_change_RQa (
       FDM_MPS_VAR_REF  *Ptr_MPS_VAR,
       ENUM FDM_FLAGS   New_RQa);
```

DESCRIPTION:

This function makes it possible to modify the value of the RQa attribute associated with an MPS variable when it has already been created but the AE/LE to which it belongs has not yet started up.

INPUT PARAMETERS:

`Ptr_MPS_VAR:`	Pointer to a data structure of the `FDM_MPS_VAR_REF` type. This parameter enables the variable on which the primitive must act, to be identified.
`New_RQa:`	New value of the RQa attribute of the variable identified by `Ptr_MPS_VAR:` **FLAG_OFF** or **FLAG_ON.**

Description of type `FDM_MPS_VAR_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function produces a report as specified in Chapter 1 Subsection 4.2. of this document.

1.25 fdm_mps_var_change_msga()

SUMMARY:

Function for modifying the MSGa attribute of a variable.

PROTOTYPE:

```
unsigned short fdm_mps_var_change_MSGa(
       FDM_MPS_VAR_REF  *Ptr_MPS_VAR,
       ENUM FDM_FLAGS   New_MSGa);
```

DESCRIPTION:

This function makes it possible to modify the value of the MSGa attribute associated with an MPS variable when it has already been created, but the AE/LE to which it belongs has not yet started up.

INPUT PARAMETERS:

`Ptr_MPS_VAR:`	Pointer to a data structure of the `FDM_MPS_VAR_REF` type. This parameter enables the variable on which the primitive must act, to be identified.
`New_MSGa:`	New value of the MSGa attribute of the variable identified by `Ptr_MPS_VAR:` **FLAG_OFF** or **FLAG_ON.**

Description of type `FDM_MPS_VAR_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function produces a report as specified in Chapter 1 Subsection 4.2. of this document.

1.26 fdm_mps_var_write_loc()

SUMMARY:

Write function of a local MPS variable.

PROTOTYPE:

```
unsigned short fdm_mps_var_write_loc (
       FDM_MPS_VAR_REF              *Ptr_MPS_VAR,
       USER_BUFFER_TO_READ          Data_Buffer);
```

DESCRIPTION:

This function makes it possible to write a local MPS variable.

INPUT PARAMETERS:

`Ptr_MPS_VAR:` Pointer to a data structure of the `FDM_MPS_VAR_REF` type.

`Data_Buffer:` Pointer to a read-only data structure of indeterminate type.

Description of type `FDM_MPS_VAR_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

Description of type `USER_BUFFER_TO_READ`

```
typedef const void * USER_BUFFER_TO_READ
```

REPORT:

This function produces a report as specified in Chapter 1 Subsection 4.2. of this document.

1.27 fdm_mps_var_write_universal()

SUMMARY:

Write function of a *universal* type variable.

PROTOTYPE:

```
unsigned short fdm_mps_var_write_universal(
       FDM_MPS_VAR_REF              *Ptr_MPS_VAR,
       USER_BUFFER_TO_READ          Data_Buffer);
```

DESCRIPTION:

This function makes it possible to write an MPS variable whether it is local or remote. If the variable is remote, an aperiodic request is made.

INPUT PARAMETERS:

`Ptr_MPS_VAR:` Pointer to a data structure of the `FDM_MPS_VAR_REF` type.

`Data_Buffer:` Pointer to a read-only data structure of indeterminate type.

Description of type `FDM_MPS_VAR_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

Description of type `USER_BUFFER_TO_READ:`

```
Typedef const void * USER_BUFFER_TO_READ
```

REPORT:

This function produces a report as specified in Chapter 1 Subsection 4.2. of this document.

1.28 fdm_mps_var_time_write_loc()

PROTOTYPE:

```
unsigned short fdm_mps_var_time_write_loc      (
       FDM_MPS_VAR_REF              *Ptr_MPS_VAR);
       unsigned char * (* User_Get_Value) (void)
       const unsigned long          * Delta
```

DESCRIPTION:

INPUT PARAMETERS:

`Ptr_MPS_VAR:` Pointer to a data structure of the `FDM_MPS_VAR_REF` type

`User_Get_Value:` User function that would be called by FIP DEVICE MANAGER to get the variable value just before writing it in the FIPCODE data base. This function returns an array of char that contains the variable value to be written. This function call is encapsulated between `OS_Enter_Region ()` and `OS_Leave_Region()` calls. So the execution time of these user-written functions must be as short as possible.

`Delta:` Pointer to the constant representing the time differential already provided by the user.

REPORT:

This function produces a report as specified in Chapter 1 Subsection 4.2. of this document.

1.29 fdm_mps_var_read_loc()

SUMMARY:

Read function of a local variable.

PROTOTYPE:

```
FDM_MPS_READ_STATUS fdm_mps_var_read_loc (
       FDM_MPS_VAR_REF  *Ptr_MPS_VAR,
       FDM_MPS_VAR_DATA *Storage_Buffer );
```

DESCRIPTION:

This function makes it possible to read a local variable, obtaining in return the value read and any MPS status.

INPUT PARAMETERS:

`Ptr_MPS_VAR:`	Pointer to a data structure of the `FDM_MPS_VAR_REF` type.
`Storage_Buffer:`	Pointer to a data structure of the `FDM_MPS_VAR_DATA` type.

Definition of type: `FDM_MPS_VAR_DATA:`

See primitive `fdm_mps_var_create()`.

Description of type `FDM_MPS_VAR_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function produces a report of the `FDM_MPS_READ_STATUS` type.

Definition of type: `FDM_MPS_READ_STATUS`

```
typedef struct FDM_MPS_READ_STATUS{
                          Enum                  Report                   :3;
                          _FDM_MPS_READ_CNF
                          Unsigned              Non_Significant          :1;
                          unsigned              Promptness_false         :1;
                          unsigned              Significance_status_false :1;
                          unsigned              Refresment_false         :1;
} FDM_MPS_READ_STATUS;
```

`Report:`	Report of the `FDM_MPS_READ_CNF` type, the possible values of which are: **_FDM_MPS_READ_OK** = correct variable read. **_FDM_MPS_USER_ERROR** = incorrect read because parameters are wrong, or produced variable read. **_FDM_MPS_STATUS_FAULT** = incorrect read because one of the statuses is false.
`Non_Significant:`	If = 1 the data contained in the zone pointed to by `Storage_Buffer` are not significant, nor are the following three bits.
`Promptness_false:`	Prompt status: 0 = false, 1 = true
`Refreshment_false:`	Refresh status: 0 = false, 1 = true
`Significance_status_false:`	Significance status: 0 = false, 1 = true

1.30 fdm_mps_var_read_universal()

SUMMARY:

Read function for a *universal* variable

PROTOTYPE:

```
unsigned short fdm_mps_var_read_universal(
       FDM_MPS_VAR_REF  *Ptr_MPS_VAR);
```

DESCRIPTION:

This function makes it possible to read an MPS variable whether it is local or remote. If the variable is remote, an aperiodic request is made.

INPUT PARAMETERS:

`Ptr_MPS_VAR:` Pointer to a data structure of the `FDM_MPS_VAR_REF` type.

Description of type `FDM_MPS_VAR_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function produces a report as specified in Chapter 1 Subsection 4.2. of this document.

1.31 fdm_mps_var_time_read_loc()

SUMMARY:

Read function for a local variable.

PROTOTYPE:

```
FDM_MPS_READ_STATUS fdm_mps_var_time_read_loc  (
       FDM_MPS_VAR_REF              *Ptr_MPS_VAR,
       Void (*User_Set_Value) (FDM_MPS_VAR_TIME_DATA *),
       FDM_MPS_VAR_TIME_DATA        *Variable  );
```

DESCRIPTION:

This function makes it possible to read a local MPS variable.

INPUT PARAMETERS:

`Ptr_MPS_VAR:`	Pointer to a data structure of the `FDM_MPS_VAR_REF` type.
`User_Set_Value`	User function that would be called by FIP DEVICE MANAGER to indicate that the variable value is available in the `FDM_MPS_VAR_TIME_DATA` type structure. This function call is encapsulated between `OS_Enter_Region()` and `OS_Leave_Region()` calls. So the execution time of these user-written functions must be as short as possible.
`Variable:`	Pointer to the data structure containing the variable and its additional information.

Definition of type: `FDM_MPS_VAR_TIME_DATA`:

```
typedef struct {
     unsigned long       Network_Delay;
     unsigned char       Pdu_Type;
     unsigned char       Pdu_Length;
     unsigned char       FBuffer[126];
} FDM_MPS_VAR_TIME_DATA;
```

`Pdu_Type:`	Value of the ”PDU type” field of the frame received.
`Pdu_Length:`	Value of the length field of the frame received. It is equal to the effective length of the data (contained in `Rx_Data`) +7.
`Network_Delay:`	Sum of the variable transit delays: initial delay when user writes the variable + production delay + consumption delay + delay for transfer onto the network in microseconds.
`FBuffer:`	Array of byte containing the value of the variable.

REPORT:

This function produces a report of the `FDM_MPS_READ_STATUS` type as specified by the primitive `fdm_mps_var_read_loc()`

1.32 fdm_generic_time_initialize()

SUMMARY:

Initialisation and start-up of the procedure for managing redundancy of the MPS time variable producers (ID 9802) and the management of that variable (production/consumption).

PROTOTYPE:

```
FDM_GENERIC_TIME_REFERENCE fdm_generic_time_initialize (
    FDM_REF                                  *Ptr_Network_Identification,
    Const FDM_GENERIC_TIME_DEFINITION        *User_Definition  );
```

DESCRIPTION:

Creation of the MPS time variable, the messaging LSAPs required for the time producer redundancy management protocol to be operated and the possible start up of this protocol.

INPUT PARAMETERS:

`Ptr_Network_Identification:`	Pointer to a data structure of the `FDM_REF` type.
`User_Definition:`	Pointer to a data structure containing the initialisation and operating parameters for the time producer redundancy management procedure.

Definition of type: `FDM_GENERIC_TIME_DEFINITION`:

```
typedef struct {
enum BOOLEAN           With_Choice_Producer;
enum BOOLEAN           With_MPS_Var_Produced;
enum BOOLEAN           With_MPS_Var_Consumed;
enum _FDM_MSG_IMAGE    Image;
unsigned long          Refreshment;
unsigned long          Promptness;;
const unsigned long *Delta_Time_Location;
void                   (*User_Signal_Mode) (const int Sens);
unsigned short         Ticks_Election;
unsigned short         Channel_Nr;
 } FDM_GENERIC_TIME_DEFINITION;
```

`With_Choice_Producer:`	**YES:**	Protocol for electing time producer is active
	NO:	Protocol for electing time producer is not active
`With_MPS_Var_Produced:`	**YES:**	the time variable can be produced: potential producer
	NO:	the time variable cannot be produced
`With_MPS_Var_Consumed:`	**YES:**	the time variable can be consumed
	NO:	the time variable cannot be consumed
`Image:`	1:	Time producer redundancy and time can only be managed if the operating image is Image 1.
	2:	Time producer redundancy and time can only be managed if the operating image is Image 2.
	3:	Time producer redundancy and time can be managed whatever the operating image (1 or 2).

`Refreshment:`	Time variable refresh period in µs if `With_MPS_Var_Produced = YES`
`Promptness:`	Time variable prompt period in µs if `With_MPS_Var_Consumed = YES`
`User_Signal_Mode:`	User function called when moving from potential producer to active producer or vice-versa.
`Channel_Nr`:	Number of transmission channel chosen to transmit messages from services linked to the time producer redundancy management protocol. This channel has to have been created.
`Delta_Time_Location:`	Pointer to the zone where the time supplied by the user is to be recovered.
`Ticks_Election:`	Activation period of the procedure for electing the time producer.

Description of type: `FDM_REF`:

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function places either:

- a `NULL_PTR` pointer if it has not been able to run correctly. In this case the `User_Signal_Warning()` function has been called at least once
- or a pointer to an `FDM_GENERIC_TIME_REFERENCE` type structure (inside FIP DEVICE MANAGER: not manipulated by the user).

1.33 fdm_generic_time_write_loc()

SUMMARY:

Time variable write (ID 9802H).

PROTOTYPE:

```
unsigned short fdm_generic_time_write_loc (
     Const FDM_GENERIC_TIME_REFERENCE          *Ptr_Time_Identification,
```

DESCRIPTION:

This function enables the user to write the value of the time variable (i.e. write the time). The time value is transmitted to the FIP DEVICE MANAGER on request. Calling the user-written procedure (`fdm_generic_time_get_value`) makes this request.

This ensures the time transmitted is as accurate as possible.

To use this function, the compiling switch `FDM_WITH_GT` or `FDM_WITH_GT_ONLY_PRODUCED` must be set to `YES`.

INPUT PARAMETERS:

`Ptr_Time_Identification:`	Pointer to a data structure of the `FDM_GENERIC_TIME_REFERENCE` type.

Description of type `FDM_GENERIC_TIME_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function produces a report as specified in Chapter 1 Section 4.2. of this document.

Description of the function: `fdm_generic_time_get_value()`

```
FDM_GENERIC_TIME_VALUE fdm_generic_time_get_value (void);
```

The `FDM_GENERIC_TIME_VALUE` type is described on page 3-51.

This procedure must be written by the user to provide the time to be transmitted by the FIP DEVICE MANAGER. This is common to type all the FIP DEVICE MANAGER instances.

1.34 fdm_generic_time_read_loc()

SUMMARY:

Time variable read (ID 9802H)

PROTOTYPE:

```
FDM_MPS_READ_STATUS fdm_generic_time_read_loc (
     Const FDM_GENERIC_TIME_REFERENCE          *Ptr_Time_Identification);
```

DESCRIPTION:

This function enables the user to read the value of the time variable (i.e. read the time).

The time value is transmitted to the caller by FIP DEVICE MANAGER by calling the user-written function `fdm_generic_time_give_value()`.

The FIP DEVICE MANAGER directly adds the total differential to the value of the variable.

To use this function, the compiling switch `FDM_WITH_GT` or `FDM_WITH_GT_ONLY_CONSUMED` must be set to `YES`.

INPUT PARAMETERS:

`Ptr_Time_Identification:` Pointer to a data structure of the `FDM_GENERIC_TIME_REF` type.

Description of type : `FDM_GENERIC_TIME_REFERENCE:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

Description of the function: `fdm_generic_time_give_value()`

```
FDM_GENERIC_TIME_VALUE fdm_generic_time_give_value (void);
```

Definition of type: `FDM_GENERIC_TIME_VALUE`: 3

```
typedef INOUT struct {
     unsigned long      Significance;
     unsigned long      Number_Of_Second;
     unsigned long      Number_Of_Nanosecond;
} FDM_GENERIC_TIME_VALUE;
```

Significance:

x00: time written by an overall external reference.

x02: time written locally by the producer.

`Number_Of_Second:` Number of seconds since 1/01/70.

`Number_Of_Nanosecond:` Number of nanoseconds since the beginning of the current second.

REPORT:

This function produces a report of the `FDM_MPS_READ_STATUS` type as specified by the primitive `fdm_mps_var_read_loc()`.

1.35 fdm_generic_time_delete()

SUMMARY:

This function enables the user to stop the procedure for managing redundancy of the MPS time variable producers (ID 9802) and the management of that variable (production/consumption).

PROTOTYPE:

```
Unsigned short fdm_generic_time_delete
        (FDM_GENERIC_TIME_REFERENCE *Ptr_Time_Identification);
```

DESCRIPTION:

Deletion of the MPS time variable, the messaging LSAPs required for the time producer redundancy management protocol to be operated, and stopping of this protocol.

INPUT PARAMETERS:

`Ptr_Time_Identification:` Pointer to a data structure of the `FDM_GENERIC_TIME_REFERENCE` type.

Description of type: `FDM_GENERIC_TIME_REFERENCE:` Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function produces a report as specified in Chapter 1 Subsection 4.2. of this document.

1.36 fdm_generic_time_set_priority()

SUMMARY:

This function enables the user to define the subscriber priority in the time producer election mechanism.

PROTOTYPE:

```
Unsigned short fdm_generic_time_set_priority
       (FDM_GENERIC_TIME_REFERENCE  *Ptr_Time_Identification,
       GT_PRIORITY                  prio);
```

DESCRIPTION:

After initialisation, and before calling this function the priority is set to 0 (minimum).

To use this function, the compiling switch `FDM_WITH_GT` must be set to `YES`.

INPUT PARAMETERS:

`Ptr_Time_Identification:`	Pointer to a data structure of the `FDM_GENERIC_TIME_REFERENCE` type.
`Prio:`	Value of the priority to be set. 0 is the value for the minimum priority, 15 is the value for the maximum priority.

Description of type: `FDM_GENERIC_TIME_REFERENCE:` Inside FIP DEVICE MANAGER: not manipulated by the user.

Description of type: `GT_PRIORITY: Typedef struct {unsigned Val: 4;} GT_PRIORITY.`

REPORT:

This function produces a report as specified in Chapter 1 Subsection 4.2. of this document.

1.37 fdm_generic_time_set_candidate_for_election()

SUMMARY:

This function enables the user to include or to exclude the subscriber in the procedure for managing redundancy of the MPS time variable producers (ID 9802).

PROTOTYPE:

```
Unsigned short fdm_generic_time_set_candidate_for_election
       (FDM_GENERIC_TIME_REFERENCE  *Ptr_Time_Identification,
       enum FDM_BOOLEAN             Etat);
```

DESCRIPTION:

The subscriber is to be included in the procedure if `Etat = FDM_TRUE`, and to be excluded from the procedure if `Etat = FDM_FALSE`

To use this function, the compiling switch `FDM_WITH_GT` must be set to `YES`.

INPUT PARAMETERS:

`Ptr_Time_Identification:`	Pointer to a data structure of the `FDM_GENERIC_TIME_REFERENCE` type.
`Etat:`	`FDM_TRUE` or `FDM_FALSE`

Description of type `FDM_GENERIC_TIME_REFERENCE:` Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function produces a report as specified in Chapter 1 Subsection 4.2. of this document.

1.38 fdm_mps_fifo_empty()

SUMMARY:

The user calls this function to activate FIP DEVICE MANAGER for processing the events regarding universal read or write requests for MPS variables.

PROTOTYPE:

```
fdm_mps_fifo_empty
        (FDM_REF *Ptr_Network_Identification);
```

DESCRIPTION:

This function enables the user to find out the events received regarding universal read or write requests for MPS variables by emptying the corresponding file. Activation of this procedure by the user leads to activation by the FIP DEVICE MANAGER of the `User_Signal_Var_xxx` user procedure (`xxx=cons` or `prod` depending on the case) provided it is done on configuration of the variable.

This function must be called after that the user–written function `User_Signal_Mps_Aper`() is called by FIP DEVICE MANAGER.

INPUT PARAMETERS:

`Ptr_Network_Identification:` Pointer to a data structure of the `FDM_REF` type.

Description of type `FDM_REF:` Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

None.

1.39 fdm_ba_load_macrocycle_fipconfb()

PROTOTYPE:

```
FDM_BA_REF * fdm_ba_load_macrocycle_fipconfb (
     FDM_REF                          *Ptr_Network_Identification,
     const unsigned short             *Programme);
```

DESCRIPTION:

This function is for loading a Bus Arbitrator program presented in a format like that of the BA_BUILDER output (automatic tool for building Bus Arbitrator program).

INPUT PARAMETERS:

`Ptr_Network_Identification:`	Pointer to a data structure of the `FDM_REF` type.
`Programme:`	Pointer to a table of elements of the unsigned short type, with the BA program output format written by BA_BUILDER.

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

Description of type `FDM_BA_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function places either:

- a `NULL_PTR` pointer if it has not been able to run correctly. In this case the `User_Signal_Warning()` function has been called at least once.
- or a pointer to an `FDM_BA_REF` type structure.

1.40 fdm_ba_load_macrocycle_manual()

PROTOTYPE:

```
FDM_BA_REF * fdm_ba_load_macrocycle_manual(
     FDM_REF                          *Ptr_Network_Identification,
     Integer                          Nb_Of_Lists,
     integer                          Nb_Of_Instructions,
     unsigned short                   Label,
     const PTR_LISTS                  *Ptr_Lists,
     const PTR_INSTRUCTIONS           *Ptr_Instructions);
```

DESCRIPTION:

This function is for loading a Bus Arbitrator program presented in a format which differs from that of the BA_BUILDER output (automatic tool for building Bus Arbitrator program) but which is easy to manipulate using C.

INPUT PARAMETERS:

`Ptr_Network_Identification:`	Pointer to a data structure of the `FDM_REF` type.
`Label:`	Internal macrocycle label.
`Nb_Of_Lists:`	Number of lists described.
`Nb_Of_Instructions:`	Number of instructions described.
`Ptr_Lists:`	Pointer to a data structure of the `PTR_LISTS` type, described above, containing the sequences of identifiers to be circulated.
`Ptr_instructions:`	Pointer to a data structure of the `PTR_INSTRUCTIONS` type, described above, containing the sequences of instructions to be executed.

Description of type : `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

Description of type: `FDM_BA_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user**.**

Definition of types: PTR_LISTS et PTR_INSTRUCTIONS:

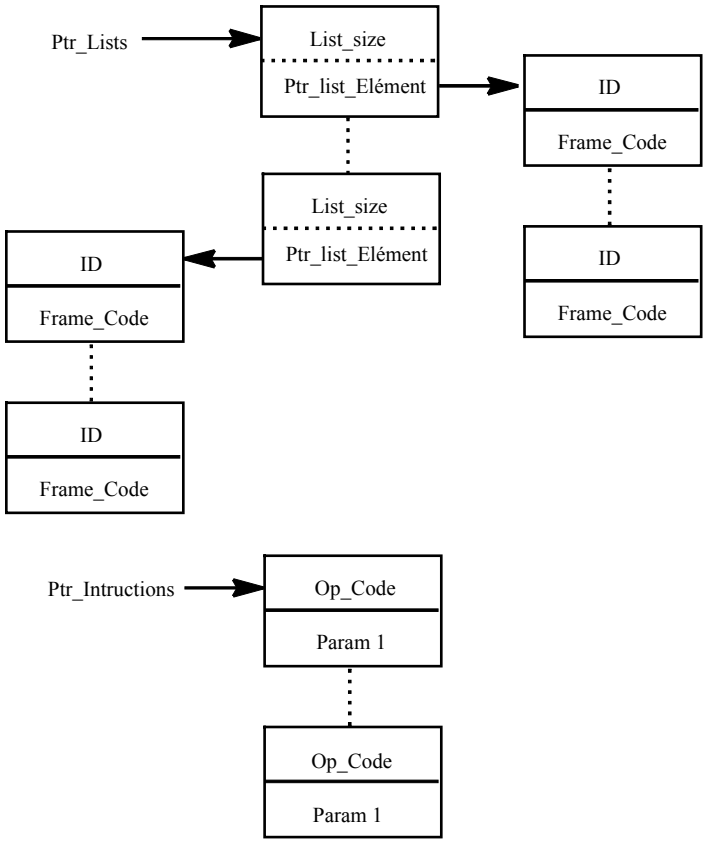
```
typedef struct {
  unsigned short            List_Size;
  LIST_ELEMENT              *Ptr_List_Element;
} PTR_LISTS

typedef struct {
  unsigned short ID;
  unsigned short Frame_Code;
} LIST_ELEMENT

typedef struct {
  unsigned short            Op_Code;
  unsigned long             Param1;
} PTR_INSTRUCTIONS
```

List_Size:	Size in bytes of the described list;
Ptr_List_Element:	Pointer to the list of identifiers.
ID:	Value of the identifier concerned. Only the packing identifier cannot be used (value = 9080H).
Frame_Code:	Used to indicate with which type of frame the identifier is to be circulated, and which are the aperiodic requests which can be processed by the Bus Arbitrator: 03 H: ID-DAT 05 H: ID-MSG
Op_Code:	Code of the Bus Arbitrator program instruction which must be chosen from the following :

2:	NEXT_MACRO:	Macrocycle re-looping.
4:	SEND_MSG:	Opening of aperiodic messaging window.
5:	SEND_APER:	Opening of aperiodic MPS variable window.
6:	BA_WAIT:	Internal synchronisation waiting window.
100:	SEND_LIST:	Periodic processing.
103:	TEST_P:	Test of subscribers present.
104:	SYN_WAIT:	External synchronisation waiting window with stuffing.
105	SYN_WAIT_SILENT	External synchronisation waiting window without stuffing. **This instruction must be used with caution because of potential BA election problems.**

`Param 1:` Significance dependent on Op_Code parameter

`SEND_LIST:`	Number of the list (i.e.: index in the list array).
`NEXT_MACRO, END_BA, TEST_P,:`	Param1 = 0.
`SEND_MSG:`	End time in relation to beginning of macrocycle (Time 0) of the aperiodic messaging window which is going to be opened (in microseconds).
`SEND_APER:`	End time in relation to beginning of macrocycle (Time 0) of the MPS aperiodic window which is going to be opened (in microseconds).
`BA_WAIT:`	Internal sync time in relation to beginning of macrocycle (Time 0) (in microseconds).
`SYN_WAIT_SILENT:`	Time of end of external sync time-out in relation to beginning of macrocycle (Time 0) in microseconds.
`SYN_WAIT:`	Time of end of external sync time-out in relation to beginning of macrocycle (Time 0) in microseconds.



REPORT:

This function places either:

- a `NULL_PTR` pointer if it has not been able to run correctly. In this case the `User_Signal_Warning()` function has been called at least once.
- or a pointer to an `FDM_BA_REF` type structure.

EXAMPLE

```
const LIST_ELEMENT L0 [] = {
            0xE001,ID_DAT
};
const LIST_ELEMENT L1 [] = {
            0x9802, ID_DAT,
            0x0010, ID_DAT,
            0x0012, ID_DAT,
            0x0013, ID_DAT,
            0x0014, ID_DAT,
            0x0100, ID_DAT,
            0x0101, ID_DAT,
};
const LIST_ELEMENT L2 [] = {
            0x1301, ID_MSG,
            0x1302, ID_MSG,
            0x1303, ID_MSG,
            0x1304, ID_MSG,
            0x1305, ID_MSG,
            0x1306, ID_MSG,
            0x1307, ID_MSG,
            0x1308, ID_MSG,
            0x1309, ID_MSG,
            0x130a, ID_MSG,
            0x130b, ID_MSG,
            0x130c, ID_MSG,
            0x130d, ID_MSG,
            0x130e, ID_MSG,
            0x130f, ID_MSG,
            0x1300, ID_MSG,

};
```

```
PTR_LISTS Listes_BA [] = {
     (Ushort)sizeof(L0), (LIST_ELEMENT*)L0,
     (Ushort)sizeof(L1), (LIST_ELEMENT*)L1,
     (Ushort)sizeof(L2), (LIST_ELEMENT*)L2,
};
Const PTR_INSTRUCTIONS Prg_BA_A [] = {
            SEND_LIST,       0,
            BA_WAIT,         5000,
            SEND_LIST,       1,
            SEND_LIST,       2,
            TEST_P,          0,

            SEND_APER,       200000,
            SEND_MSG,        200000,
            BA_WAIT,         200000,
            NEXT_MACRO,      0
};
FDM_BA_REF * MON_BA;
FDM_REF * MON_RESEAU;
MON_BA = fdm_ba_load_macrocycle_manual (
            Mon_Reseau,
            3,
            10,
            0,
            &Listes_BA,
            &Prg_BA_A);
```

1.41 fdm_ba_delete_macrocycle()

SUMMARY:

Function for deleting a macrocycle.

PROTOTYPE:

```
unsigned short fdm_ba_delete_macrocycle(
            FDM_BA_REF  *Ptr_BA);
```

DESCRIPTION:

This function is for removing a macrocycle from the FULLFIP2/FIPCODE database regardless of how it was loaded.

INPUT PARAMETERS:

`Ptr_BA:` Pointer to a data structure of the `FDM_BA_REF` type.

Description of type `FDM_BA_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report as specified by the primitive `fdm_stop_network()`.

1.42 fdm_ba_set_priority()

SUMMARY:

Function for modifying the priority of a potential Bus Arbitrator.

PROTOTYPE:

```
unsigned short fdm_ba_set_priority (
       FDM_REF          *Ptr_Network_Identification,
       unsigned char    Priority_Level);
```

DESCRIPTION:

This function makes it possible to modify the priority of a potential Bus Arbitrator.

INPUT PARAMETERS:

`Ptr_Network_Identification:`	Pointer to a data structure of the `FDM_REF` type.
`Priority_Level:`	Priority level of the Bus Arbitrator between [0..15], 0 being the highest priority.

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report as specified by the primitive `fdm_stop_network()`.

1.43 fdm_ba_set_parameters()

SUMMARY:

Function for modifying the time-outs used for the calculation of election and start-up of a Bus Arbitrator.

PROTOTYPE:

```
unsigned short fdm_ba_set_parameters (
     FDM_REF             *Ptr_Network_Identification,
     enum                _BA_SET_MODE,
     unsigned char       MAX_Subscriber,
     unsigned char       MAX_Priority);
```

DESCRIPTION:

This function makes it possible to modify the time-outs of a potential Bus Arbitrator.

INPUT PARAMETERS:

`Ptr_Network_Identification:` Pointer to a data structure of the `FDM_REF` type.

`_BA_SET_MODE`: This parameter is used for the calculation of *Start-up Time-out* and *Election Time-out* used for the startup of a BA. You can enter one of the following values: `STANDARD`, `OPTIMIZE_1`, `OPTIMIZE_2`, `OPTIMIZE_3`. To understand the impact of these values in the calculation of the time-outs, see Chapter 2, Subsection 1.4.3.

Note

- `STANDARD` corresponds to the calculation of the time-outs in compliance with the WorldFIP standard.
- `OPTIMIZE_3`: this option can be used only in the case of a single-medium network typology.

`Max_Subscriber:` Max. value of physical addresses used by subscribers present on the network (to optimise the calculation of BA time-outs): [0..255].

`Max_Priority:` Max. value of the priority of BAs present on the network used (to optimise the calculation of BA time-outs): [0..15].

Description of type `FDM_BA_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report as specified by the primitive `fdm_stop_network()`.

1.44 fdm_ba_start()

SUMMARY:

Function for starting up a potential Bus Arbitrator.

PROTOTYPE:

```
unsigned short fdm_ba_start (FDM_BA_REF *Ptr_Ba);
```

DESCRIPTION:

This function enables a potential Bus Arbitrator to be started up, for which a program has already been loaded. If necessary, it will become the active Bus Arbitrator, but only after the Bus Arbitrator election procedure has acted.

INPUT PARAMETERS:

`Ptr_Ba:` Pointer to a data structure of the `FDM_BA_REF` type.

Description of type: `FDM_BA_REF`**:** Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report as specified by the primitive `fdm_stop_network()`.

1.45 fdm_ba_stop()

SUMMARY:

Function for shutting down a potential Bus Arbitrator.

PROTOTYPE:

```
unsigned short fdm_ba_stop (
       FDM_REF          *Ptr_Network_Identification);
```

DESCRIPTION:

This function makes it possible to request the shutdown of the Bus Arbitrator function. This shutdown will be effective as soon as a new BA program instruction is processed.

INPUT PARAMETERS:

`Ptr_Network_Identification:` Pointer to a data structure of the `FDM_REF` type.

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report as specified by the primitive `fdm_stop_network()`.

1.46 fdm_ba_status()

SUMMARY:

Function for obtaining the state of a potential Bus Arbitrator.

PROTOTYPE:

```
unsigned short fdm_ba_status
       (FDM_REF         *Ptr_Network_Identification,
       BA_INF_STATUS    *Ptr_BA_Status);
```

DESCRIPTION:

This function makes it possible to find out the operating state of the Bus Arbitrator functionality of the station.

INPUT PARAMETERS:

`Ptr_Network_Identification:`	Pointer to a data structure of the `FDM_REF` type.
`Ptr_BA_Status:`	Pointer to a data structure of the `BA_INF_STATUS` type specified previously and containing the state of the BA function as well as the label of the program underway if running.

Description of type: `FDM_BA_REF:` Inside FIP DEVICE MANAGER: not manipulated by the user.

Definition of type `BA_INF_STATUS`

```
typedef struct {
     enum _BA_INF          Actif_Running;
     unsigned short        Status_Fiphandler;
     unsigned short        Label;
} BA_INF_STATUS
```

Actif_Running:	Overall status of the BA
NOT_BA:	The subscriber is not a potential BA, or the subscriber is a potential BA but the BA function is not running.
LOADED_BA:	The BA program is loaded.
ACTIVE:	The BA function is in the **BA_STARTING** or **BA_IDLE** or **BA_STOPPED** mode.
RUNNING:	The BA function is in the **BA_SENDING**, **BA_PENDING**, **BA_WAIT_TIME**, **BA_WAIT_SYNC**,**BA_WAIT_SYNC_SILENT**, **BA_MSG_WINDOW** or **BA_APER_WINDOW** mode.

`Status_Fiphandler:`	Detailed status of the BA.
BA_SENDING:	Periodic window.
BA_STOPPED:	Stopped.
BA_STARTING:	Starting up.
BA_IDLE:	Waiting for election.
BA_MSG_WINDOW:	Aperiodic messaging window.
BA_APER_WINDOW:	Aperiodic MPS window.
BA_WAIT_TIME:	Waiting for internal synchronisation.
BA_WAIT_SYNC:	Waiting for external synchronisation without stuffing on the line.
BA_WAIT_SYNC_SILENT	Waiting for external synchronisation with stuffing on the line
BA_PENDING:	Waiting for external synchronisation (TEST_P).
`Label:`	Label (name) of the program running.

REPORT:

This function submits a report as specified by the primitive `fdm_stop_network()`.

1.47 fdm_ba_commute_macrocycle()

SUMMARY:

Function for changing macrocycle.

PROTOTYPE:

```
unsigned short fdm_ba_commute_macrocycle (
            FDM_BA_REF *Ptr_BA);
```

DESCRIPTION:

This function makes it possible to change the macrocycle to be run by the local Bus Arbitrator whether active or idle.

INPUT PARAMETERS:

`Ptr_BA:` Pointer to a data structure of the `FDM_BA_REF` type.

Description of type `FDM_BA_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report as specified by the primitive `fdm_stop_network()`.

1.48 fdm_ba_external_resync()

PROTOTYPE:

```
unsigned short fdm_ba_external_resync (
       (FDM_REF         *Ptr_Network_Identification);
```

DESCRIPTION:

This function is for sending the Bus Arbitrator functionality an external resynchronisation order.

INPUT PARAMETERS:

`Ptr_Network_Identification:` Pointer to a data structure of the `FDM_REF` type.

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report as specified by the primitive `fdm_stop_network()`.

Note

Only if operating in conventional access. In the case of free access, the function is implemented with a hardware device.

1.49 fdm_ba_loaded()

PROTOTYPE:

```
int fdm_ba_loaded (FDM_BA_REF       *Ptr_BA,);
```

DESCRIPTION:

This function makes it possible to find out if a Bus Arbitrator program is loaded.

INPUT PARAMETERS:

`Ptr_BA:` Pointer to a data structure of the `FDM_BA_REF` type .

Description of type `FDM_BA_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function produces one of the following reports:

BA_LOADED:	BA program loaded
BA_NOT_LOADED:	No BA program loaded

1.50 fdm_read_present_list()

SUMMARY:

Read function of the list of equipment present.

PROTOTYPE:

```
unsigned short fdm_read_present_list (
       FDM_REF          *Ptr_Network_Identification);
```

DESCRIPTION:

This function is for the SM_MPS ”list of equipment present” variable read request.

When FIP DEVICE MANAGER will receive the variable, it will call the `User_Present_List_Prog()` function. This function is a user-written function given in the configuration parameter (`FDM_CONFIGURATION_SOFT` in the `fdm_Initialize_Network()` function).

INPUT PARAMETERS:

`Ptr_Network_Identification:` Pointer to a data structure of the `FDM_REF` type.

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report as specified by the primitive `fdm_stop_network()`.

Note

The BA program should contain at least one TEST_P instruction.

1.51 fdm_read_presence()

SUMMARY:

Read function of the presence variable of a subscriber.

PROTOTYPE:

```
unsigned short fdm_read_presence (
       FDM_REF          *Ptr_Network_Identification,
       unsigned char    Subscriber);
```

DESCRIPTION:

This function is for requesting the subscriber presence SM_MPS variable read (including itself). This is carried out by means of an aperiodic request regarding the variable in question.

When FIP DEVICE MANAGER will receive the variable, it will call the `User_Presence_Prog()` function. This function is a user-written function given in the configuration parameter (`FDM_CONFIGURATION_SOFT` in the `fdm_Initialize_Network()` function)

INPUT PARAMETERS:

`Ptr_Network_Identification:`	Pointer to a data structure of the `FDM_REF` type.
`Subscriber:`	Physical address on the network of the subscriber whose presence variable is to be read.

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report as specified by the primitive `fdm_stop_network()`.

Note

The BA program should contain an aperiodic window (instruction SEND_APER).

1.52 fdm_read_identification()

SUMMARY:

Read function of the subscriber identification variable.

PROTOTYPE:

```
unsigned short fdm_read_identification (
       FDM_REF          *Ptr_Network_Identification,
       unsigned char    Subscriber);
```

DESCRIPTION:

This function is for requesting the subscriber identification SM_MPS variable read (including itself). This is carried out by means of an aperiodic request regarding the variable in question.

When FIP DEVICE MANAGER will receive the variable, it will call the `User_Identification_Prog()` function. This function is a user-written function given in the configuration parameter (`FDM_CONFIGURATION_SOFT` in the `fdm_Initialize_Network()` function).

INPUT PARAMETERS:

`Ptr_Network_Identification:`	Pointer to a data structure of the `FDM_REF` type.
`Subscriber:`	Physical address on the network of the subscriber whose identification variable is to be read.

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report as specified by the primitive `fdm_stop_network()`.

Note

The BA program should contain an aperiodic window (instruction SEND_APER).

1.53 fdm_read_report()

SUMMARY:

Read function of the subscriber report variable.

PROTOTYPE:

```
unsigned short fdm_read_report (
       FDM_REF          *Ptr_Network_Identification,
       unsigned char    Subscriber);
```

DESCRIPTION:

This function is for requesting the subscriber report SM_MPS variable read (including itself). This is carried out by means of an aperiodic request regarding the variable in question.

When FIP DEVICE MANAGER will receive the variable, it will call the `User_Report_Prog()` function. This function is a user-written function given in the configuration parameter (`FDM_CONFIGURATION_SOFT` in the `fdm_Initialize_Network()` function).

INPUT PARAMETERS:

`Ptr_Network_Identification:`	Pointer to a data structure of the `FDM_REF` type.
`Subscriber:`	Physical address on the network of the subscriber whose report variable is to be read.

Description of type `FDM_REF`:

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report as specified by the primitive `fdm_stop_network()`.

Note

The BA program should contain an aperiodic window (instruction SEND_APER).

1.54 fdm_read_ba_synchronize()

SUMMARY:

Read function of the BA sync variable.

PROTOTYPE:

```
unsigned short fdm_read_ba_synchronize(FDM_REF *
Ptr_Network_Identification);
```

DESCRIPTION:

This function is for requesting the BA synchronisation SM_MPS variable read. This is carried out by means of an aperiodic request regarding the variable in question. The active Bus Arbitrator produces this variable.

When FIP DEVICE MANAGER will receive the variable, it will call the `User_Synchro_BA_Prog()` function. This function is a user-written function given in the configuration parameter (`FDM_CONFIGURATION_SOFT` in the `fdm_Initialize_Network()` function).

INPUT PARAMETERS:

`Ptr_Network_Identification:` Pointer to a data structure of the `FDM_REF` type.

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER not manipulated by the user.

REPORT:

This function produces a report as specified by the primitive `fdm_stop_network()`.

Note

The BA program should contain an aperiodic window (instruction SEND_APER).

1.55 fdm_get_local_report()

SUMMARY:

Local report variable read.

PROTOTYPE:

```
unsigned short fdm_get_local_report (
     FDM_REF * Ptr_Network_Identification)
     FDM_REPORT_VAR * Ptr_Report_Var);
```

DESCRIPTION:

This function enables the user to read the local report variable (i.e. from itself). No traffic is generated on the network. The value is obtained immediately.

INPUT PARAMETERS:

`Ptr_Network_Identification:`	Pointer to a data structure of the `FDM_REF` type.
`Ptr_Report_Var:`	Pointer to a data structure of the `FDM_REPORT_VAR` type in which the value of the report variable read will be stored.

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER not manipulated by the user.

Description of type `FDM_REPORT_VAR:`

See the primitive `fdm_initialize_network()`.

REPORT:

This function produces a report as specified by the primitive `fdm_stop_network()`.

1.56 fdm_smmps_fifo_empty()

SUMMARY:

Function for emptying the event file regarding SM_MPS requests.

PROTOTYPE:

```
fdm_smmps_fifo_empty (
       FDM_REF *Ptr_Network_Identification);
```

DESCRIPTION:

This function enables the user to find out about received events regarding SM_MPS variable read requests by emptying the corresponding queue. When the user activates the procedure, the FIP DEVICE MANAGER activates the `User_yyy_prog` user procedure (`yyy=Present_list` or `Presence` or `Identification` or `Report` or `Synchro_BA` depending on the case) provided to do this on configuration of the variable.

INPUT PARAMETERS:

`Ptr_Network_Identification:` Pointer to a data structure of the `FDM_REF` type.

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

None

1.57 fdm_messaging_fullduplex_create()

SUMMARY:

Creation of a full duplex messaging context.

PROTOTYPE:

```
FDM_MESSAGING_REF * fdm_messaging_fullduplex_create(
     FDM_REF                          *Ptr_Network_Identification,
     FDM_MESSAGING_FULLDUPLEX         *Messaging_Fullduplex);
```

DESCRIPTION:

This function enables a full-duplex messaging context to be created which can be used in transmission and reception mode.

INPUT PARAMETERS:

`Ptr_Network_Identification:`	Pointer to a data structure of the `FDM_REF` type.
`Messaging_Fullduplex:`	Pointer to a data structure of the `FDM_MESSAGING_FULLDUPLEX` type.

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

Description of type `FDM_MESSAGING_FULLDUPLEX:`

```
typedef struct {
     enum _FDM_MSG_IMAGE Position;
     struct {
        void (*User_Msg_Ack) (FDM_MESSAGING_REF*, FDM_MSG_TO_SEND*);
        void *User_Qid;
        void *User_Ctxt;
        unsigned short Channel_Nr;
     } sending;
     struct {
        void (*User_Msg_Rec_Proc)(FDM_MESSAGING_REF*, FDM_MSG_RECEIVED *);
        void *User_Qid;
        void *User_Ctxt;
        int Number_Of_Msg_Desc;
        int Number_Of_Msg_Block;
     } receiving;
     unsigned long Local_DLL_Address;
     unsigned long Remote_DLL_Address;
} FDM_MESSAGING_FULLDUPLEX;
```

`Position:`	`_FDM_MSG_IMAGE_1:`	The messaging context created can only be used if the AE/LEs are operating in Image 1.
	`_FDM_MSG_IMAGE_2:`	The messaging context created can only be used if the AE/LEs are operating in Image 2.
	`_FDM_MSG_IMAGE_1_2:`	The messaging context created can be used whether the AE/LEs are operating in Image 1 or Image 2.

`User_Msg_Ack:` Pointer to the user function which will be called by the FIP DEVICE MANAGER to send the user the transmission acknowledgement or the time-out if necessary.

`User_Qid:` Pointer retransmitted to the user for his private use.

`User_Ctxt:` Pointer retransmitted to the user for his private use.

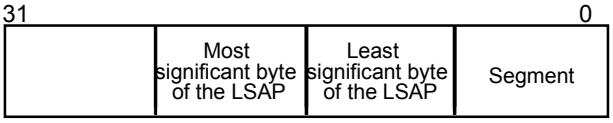
`Channel_Nr:` Transmission channel number to be used.

`User_Msg_Rec_Proc:` Pointer to the user function that will be called by FIP DEVICE MANAGER to send the user a message received in this context.

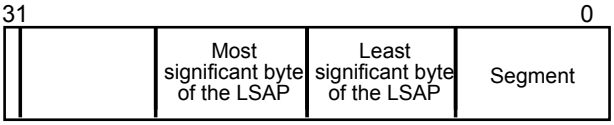
`Number_Of_Msg_Desc:` Number of message descriptors received (of the `FDM_MSG_RECEIVED` type specified below) which can be used to receive messages with the same destination address as that specified below.

`Number_Of_Msg_Block:` Number of message blocks received (of the `FDM_MSG_R_DESC` type specified below) which can be used to receive messages with the same destination address as that specified below.

`Local_DLL_Address:` If in transmission mode, then it is the source address of the message transmitted.
If in reception mode, then it is the destination address of the message received.



`Remote_DLL_Address:` If in reception mode then it is the source address of the message received.
If in transmission mode, then it is the destination address of the message transmitted:



If bit 31= 1: `Remote_DLL_Address` is unknown (in general used for broadcast : transmission for all the stations and reception from all the stations).

If bit 31= 0: `Remote_DLL_Address` is known, i.e. a message is received only if `Local_DLL_Address` and `Remote_DLL_Address` correspond to the address field of this message.

Note

It is forbidden to configure the station as a bridge (at least 2 networks) in full-duplex mode. A station can be configured as a bridge only in half-duplex mode (see functions *fdm_messaging_to_send_create* and *fdm_messaging_to_rec_create*).

Description of type `FDM_MSG_TO_SEND:`

```
typedef struct _FDM_MSG_TO_SEND{
   struct_FDM_MSG_TO_SEND          *Next;
   struct_FDM_MSG_TO_SEND          *Prev;
   integer                         Nr_Of_Blocks;
   FDM_MSG_T_DESC                  *Ptr_Block;
   Unsigned long                   Local_DLL_Address
   unsigned long                   Remote_DLL_Address;
   FDM_MSG_SEND_SERVICE_REPORT     Service_Report;
   FDM_PRIVATE_DLI_T               Fdm_Msg_T_Private;
} FDM_MSG_TO_SEND;
```

`Next:` Pointer of concatenation on another data structure of the same type (front concatenation).

`Prev:` Pointer of concatenation on another data structure of the same type (back concatenation).

`Nr_Of_Blocks:` Number of `FDM_MSG_T_DESC` type blocks that make up the message to be transmitted.

`Ptr_Block:` Pointer to the first block of the `FDM_MSG_T_DESC` type which contains the message to be transmitted.

`Local_DLL_Address:`

`Remote_DLL_Address:` Same definition as for the `Local_DLL_Address` parameter (above).

`Service_Report:` Message transmission report given by the FIP DEVICE MANAGER when the user procedure intended to acknowledge it, and the reference which is given by the aforementioned `User_Msg_Ack_Proc` parameter, is called. The parameter of this user procedure is a pointer to a data structure of the `FDM_MSG_TO_SEND` type: the same as that used for transmission.

`Fdm_Msg_T_Private:` Data structure in the `FDM_PRIVATE_DLI_T` format which only the FIP DEVICE MANAGER can use.

Description of type: FDM_MSG_T_DESC

```
typedef struct _FDM_MSG_T_DESC{
     integer                    Nr_Of_Bytes;
     unsigned char              *Ptr_Data;
     struct _FDM_MSG_T_DESC     *Next_Block;
} FDM_MSG_T_DESC;
```

Nr_Of_Bytes:	Number of bytes contained in the block pointed to by Ptr_Data.
Ptr_Data:	Pointer to the zone containing data.
Next_Block:	Pointer to the following block of the same type.

Description of type FDM_MSG_SEND_SERVICE_REPORT:

```
typedef struct {
     enum _FDM_MSG_SND_CNF          Valid;
     enum _FDM_MSG_USER_ERROR_LIST  msg_user_soft_report;
     enum _FIP_MSG_SND_REP          Way;
};

enum _FDM_MSG_SND_CNF {
     _FDM_MSG_SEND_OK    = 0,
     _FDM_MSG_USER_ERROR,
     _FDM_DATA_LINK_ERROR
};
```

_FDM_MSG_SEND_OK	No error indication.
_FDM_MSG_USER_ERROR	The error type is indicated in the _FDM_MSG_USER_ERROR_LIST field.
_FDM_DATA_LINK_ERROR	The error type is indicated in the _FIP_MSG_SND_REP field.

```
enum _FDM_MSG_USER_ERROR_LIST {
     _FDM_MSG_REPORT_OK      = 0,
     _FDM_MSG_REPORT_MSG_NOT_ALLOWED,
     _FDM_MSG_REPORT_CHANNEL_NOT_ALLOWED,
     _FDM_MSG_REPORT_ERR_LG_MSG,
     _FDM_MSG_REPORT_ERR_MSG_INFOS,
     _FDM_MSG_REPORT_INTERNAL_ERROR
 };
```

`_FDM_MSG_REPORT_OK`	No indication of an error of this type.
`_FDM_MSG_REPORT_MSG_NOT_ALLOWED`	Transmission has been requested during operation on image 2.
`_FDM_MSG_REPORT_CHANNEL_NOT_ALLOWED`	Channel on which transmission took place does not exist.
`_FDM_MSG_REPORT_ERR_LG_MSG`	Message size error: = 0 or > 256.
`_FDM_MSG_REPORT_ERR_MSG_INFOS`	`FDM_MSG_TO_SEND.Nr_Of_Blocks = 0` or `FDM_MSG_TO_SEND.Ptr_Blocks = NULL.`
`_FDM_MSG_REPORT_INTERNAL_ERROR`	Problem of access to component: User_Signal_Warning called to provide details.

```
enum _FIP_MSG_SND_REP {
     _FIP_NOACK_BAD_RP_MSG = 0x4,
     _FIP_ACK_NO_REC_AFTER_RETRY = 0x2,
     _FIP_ACK_NEG_AFTER_RETRY       = 0x5,
     _FIP_ACK_NEG_NO_RETRY = 0x7,
     _FDM_TIME_OUT = 0x8
};
```

`_FIP_NOACK_BAD_RP_MSG`	Message transmission error if messaging not acknowledged.
`_FIP_ACK_NO_REC_AFTER_RETRY`	No acknowledgement after retry.
`_FIP_ACK_NEG_AFTER_RETRY`	Negative acknowledgement after retry.
`_FIP_ACK_NEG_NO_RETRY`	Negative acknowledgement with no retry.
`_FDM_TIME_OUT`	Time-out waiting to be transmitted.

Description of type `FDM_PRIVATE_DLI_T:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

Description of type: FDM_MSG_RECEIVED

```
typedef struct _FDM_MSG_RECEIVED{
     struct _FDM_MSG_RECEIVED      *Next;
     struct _FDM_MSG_RECEIVED      *Prev;
     Integer                       Nr_Of_Blocks
     FDM_MSG_R_DESC                *Ptr_Block;
     unsigned long                 Local_DLL_Address;
     unsigned long                 Remote_DLL_Address;
     Void                          *User_Qid;
     Void                          *User_Ctxt;
     FDM_MESSAGING_REF             *Ref;
} FDM_MSG_RECEIVED;
```

Next:	Pointer for concatenation to another data structure of the same type (front concatenation).
Prev:	Pointer for concatenation to another data structure of the same type (back concatenation).
Nr_Of_Blocks:	Number of FDM_MSG_R_DESC type blocks used to store the received message, (Always equal to 1 in the case of FIP DEVICE MANAGER).
User_Qid et User_Ctxt:	Pointers retransmitted to the user for his private use. They are those which were given on creation of the context. Optional user information.
Local_DLL_Address: Remote_DLL_Address:	See Local_DLL_Addresses.

This field is used when Bit 31 of the Remote_DLL_Address field, of the context on which the message is received, has been initialised at 1 (Remote_DLL_Address unknown).

Ref:	Reference of the context on which the message was received. Can be used to transmit a response if the context in question is of the FDM_MESSAGING_FULLDUPLEX type.

Description of type: FDM_MSG_R_DESC

```
typedef struct _FDM_MSG_R_DESC{
     integer                    Nr_Of_Bytes;
     unsigned char              *Ptr_Data;
     struct _FDM_MSG_R_DESC     *Next_Block;
     unsigned char              Data_Buffer[256];
} FDM_MSG_R_DESC;
```

Nr_Of_Bytes:	Number of significant bytes contained in the Data_Buffer block.
Ptr_Data:	Pointer to the Data_Buffer block containing the data.
Next_Block:	Pointer to the next block of the same type: is always NULL_PTR in the case of FIP DEVICE MANAGER.
Data_Buffer:	Zone intended to hold the received messages.

REPORT:

This function produces either:

- a NULL_PTR pointer if it has not been able to run correctly, in which case the User_Signal_Warning() function has been called at least once,
- or a pointer to an FDM_MESSAGING_REF type structure.

Description of type FDM_MESSAGING_REF:

Inside FIP DEVICE MANAGER: not manipulated by the user.

1.58 fdm_messaging_to_send_create()

SUMMARY:

PROTOTYPE:

```
FDM_MESSAGING_REF * fdm_messaging_to_send_create(
     FDM_REF                        *Ptr_Network_Identification,
     FDM_MESSAGING_TO_SEND          *Messaging_To_Send);
```

DESCRIPTION:

This function is for creating a messaging context that can be used in transmission mode.

INPUT PARAMETERS:

`Ptr_Network_Identification:`	Pointer to a data structure of the `FDM_REF` type.
`Messaging_To_Send:`	Pointer to a data structure of the `FDM_MESSAGING_TO_SEND` type.

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

Description of type `FDM_MESSAGING_TO_SEND:`

```
typedef struct
     enum _FDM_MSG_IMAGE Position;
     struct {
          void (*User_Msg_Ack)(FDM_MESSAGING_REF*, FDM_MSG_TO_SEND*);
          void *User_Qid;
          void *User_Ctxt;
          unsigned short Channel_Nr;
     } sending;
     unsigned long Local_DLL_Address;
     unsigned long Remote_DLL_Address;
} FDM_MESSAGING_TO_SEND;
```

Position:	**_FDM_MSG_IMAGE_1:**	The messaging context created can only be used if the AE/LEs are operating in Image 1
	_FDM_MSG_IMAGE_2:	The messaging context created can only be used if the AE/LEs are operating in Image 2
	_FDM_MSG_IMAGE_1_2:	The messaging context created can be used whether the AE/LEs are operating in Image 1 or Image 2.

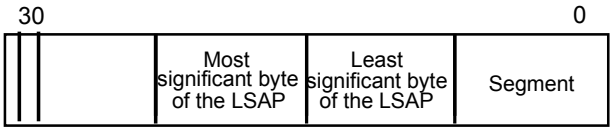
`User_Msg_Ack:` Pointer to the user function which will be called by the FIP DEVICE MANAGER to send the user the transmission acknowledgement or the time-out if necessary.

`User_Qid:` Pointer retransmitted to the user for his private use.

`User_Ctxt:` Pointer retransmitted to the user for his private use.

`Channel_Nr:` Transmission channel number to be used.

`Local_DLL_Address:` Source address of the message transmitted.

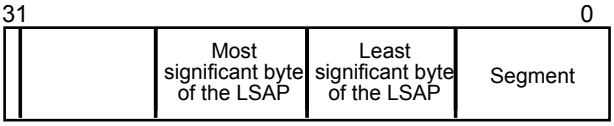


Note

If you write an application that plays the role of a bridge (at least 2 networks - see the subsection 1.6.4 of Chapter 2) then you have to set:

- bit 30 = 1,
- Segment byte = segment number of the source address,
- the 2 bytes describing the LSAP = any value.

`Remote_DLL_Address:` destination address of the message transmitted:



If bit 31= 1: Remote_DLL_Address is unknown (in general used for broadcast : transmission for all the stations)

If bit 31= 0: Remote_DLL_Address is known

Note

If you write an application that plays the role of a bridge (at least 2 networks) then you have to set:

- bit 31 = 1,
- the Segment byte and the 2 bytes describing the LSAP = any value.

Description of type `FDM_MSG_TO_SEND:`

See the primitive `fdm_messaging_fullduplex_create()`.

REPORT:

This function produces either:

- a `NULL_PTR` pointer if it has not been able to run correctly. In this case the `User_Signal_Warning()` function has been called at least once,
- or a pointer to an `FDM_MESSAGING_REF` type structure.

Description of type `FDM_MESSAGING_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

1.59 fdm_messaging_to_rec_create()

PROTOTYPE:

```
FDM_MESSAGING_REF * fdm_messaging_to_rec_create(
     FDM_REF                        *Ptr_Network_Identification,
     FDM_MESSAGING_TO_REC           *Messaging_To_Rec);
```

DESCRIPTION:

This function is for creating a messaging context that can be used in reception mode.

INPUT PARAMETERS:

`Ptr_Network_Identification:`	Pointer to a data structure of the `FDM_REF` type.
`Messaging_To_Rec:`	Pointer to a data structure of the `FDM_MESSAGING_TO_REC` type.

Description of type `FDM_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

Description of type `FDM_MESSAGING_TO_REC:`

```
typedef struct
     enum _FMD_MSG_IMAGE Position,;
     struct {
         void (*User_Msg_Rec_Proc)(FDM_MESSAGING_REF*, FDM_MSG_RECEIVED*);
         void *User_Qid;
         void *User_Ctxt;
         integer Number_Of_Msg_Desc;
         integer Number_Of_Msg_Block;
     } receiving;
     unsigned long Local_DLL_Address;
     unsigned long Remote_DLL_Address;
} FDM_MESSAGING_TO_REC;
```

Position:	**_FDM_MSG_IMAGE_1:**	The messaging context created can only be used if the AE/LEs are operating in Image 1.
	_FDM_MSG_IMAGE_2:	The messaging context created can only be used if the AE/LEs are operating in Image 2.
	_FDM_MSG_IMAGE_1_2:	The messaging context created can be used whether the AE/LEs are operating in Image 1 or Image 2.

`User_Msg_Rec_Proc:` Pointer to the user function that will be called by FIP DEVICE MANAGER to send the user a message received in this context.

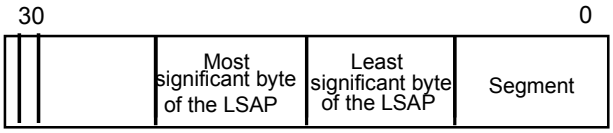
`User_Qid:` Pointer retransmitted to the user for his private use.

`User_Ctxt:` Pointer retransmitted to the user for his private use

`Number_Of_Msg_Desc:` Number of message descriptors received (of the `FDM_MSG_RECEIVED` type - see *fdm_messaging_fullduplex_create*) which can be used to receive messages with the same destination address as that specified below.

`Number_Of_Msg_Block:` Number of message blocks received (of the `FDM_MSG_R_DESC` type - see *fdm_messaging_fullduplex_create*) which can be used to receive messages with the same destination address as that specified below.

`Local_DLL_Address:` destination address of the message received.

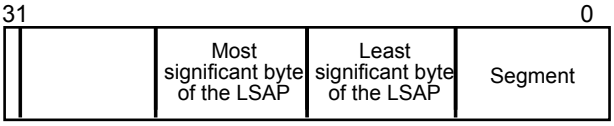


Note

If you write an application that plays the role of a bridge (at least 2 networks - see Chapter 2, Subsection 1.6.4) then you have to set:

- bit 30 = 1,
- Segment byte = segment number of the destination address,
- the 2 bytes describing the LSAP = any value.

`Remote_DLL_Address:` source address of the message received.



If bit 31= 1: `Remote_DLL_Address` is unknown (in general used for broadcast: reception from all the stations).

If bit 31= 0: `Remote_DLL_Address` is known, i.e. a message is received only if `Local_DLL_Address` and `Remote_DLL_Address` correspond to the address field of this message.

Note

If you write an application that plays the role of a bridge (at least 2 networks) then you have to set:

- bit 31 = 1,
- the Segment byte and the 2 bytes describing the LSAP = any value.

Description of type: `FDM_MSG_RECEIVED`

See the primitive `fdm_messaging_fullduplex_create()`.

REPORT:

This function produces either:

- a `NULL_PTR` pointer if it has not been able to run correctly. In this case the `User_Signal_Warning()` function has been called at least once,
- or a pointer to an `FDM_MESSAGING_REF` type structure.

Description of type `FDM_MESSAGING_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

1.60 fdm_messaging_delete()

PROTOTYPE:

```
unsigned short fdm_messaging_delete (
     FDM_MESSAGING_REF   *Ptr_Messaging);
```

DESCRIPTION:

This function deletes a messaging context.

INPUT PARAMETERS:

`Ptr_Messaging:` Pointer to the messaging context to be deleted.

Description of type `FDM_MESSAGING_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report as specified by the primitive `fdm_stop_network()`.

1.61 fdm_channel_create()

SUMMARY:

Creation of a periodic messaging channel.

PROTOTYPE:

```
unsigned short fdm_channel_create (
     FDM_REF             *Ptr_Network_Identification,
     FDM_CHANNEL_PARAM   *Ptr_Channel);
```

DESCRIPTION:

This function is for creating a periodic messaging channel. Eight periodic messaging channels numbered from 1 to 8 can be selected.

When the station receives an ID_MSG frame with an identifier corresponding to a produced variable intended to support a periodic channel, it takes the message to be transmitted from the queue associated with this channel.

INPUT PARAMETERS:

`Ptr_Network_Identification:`	Pointer to a data structure of the `FDM_REF` type.
`Ptr_Channel:`	Pointer to a data structure of the `FDM_CHANNEL_PARAM` type specified above and containing the transmission channel configuration parameters to be created.

Description of type `FDM_REF`:

Inside FIP DEVICE MANAGER: not manipulated by the user.

Description of type: `FDM_CHANNEL_PARAM`

```
typedef struct {
     enum _FDM_MSG_IMAGE        Position
     unsigned short             Channel_Nr;
     unsigned short             Identifier;
} FDM_CHANNEL_PARAM;
```

`Position:`	**FDM_MSG_IMAGE_1** or **FDM_MSG_IMAGE_2** or **FDM_MSG_IMAGE_1_AND_2**
`Identifier:`	Identifier corresponds to a produced variable intended to support the transmission channel. If this identifier does not exist in any AE/LE it is automatically created with the produced attribute.
`Channel_Nr:`	Periodic messaging channel number.

REPORT:

This function submits a report as specified by the primitive `fdm_stop_network()`.

1.62 fdm_channel_delete()

SUMMARY:

Deleting a periodic messaging channel.

PROTOTYPE:

```
unsigned short fdm_channel_delete (
     FDM_REF             *Ptr_Network_Identification,
     unsigned short      Channel_Nr);
```

DESCRIPTION:

This function is for deleting a periodic messaging channel.

INPUT PARAMETERS:

`Ptr_Network_Identification:`	Pointer to a data structure of the `FDM_REF` type.
`Channel_Nr:`	See the `fdm_channel_create()` primitive.

Description of type `FDM_REF` :

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report as specified by the primitive `fdm_stop_network()`.

1.63 fdm_send_message()

SUMMARY:

Request to transmit a message.

PROTOTYPE:

```
unsigned short fdm_send_message (
     FDM_MESSAGING_REF   *Ptr_Messaging,
     FDM_MSG_TO_SEND     *Ptr_Msg_To_Send);
```

DESCRIPTION:

This function makes it possible to request the transmission of a message on a messaging context which has already been configured for this purpose.

INPUT PARAMETERS:

`Ptr_Messaging:`	Pointer to a data structure of the `FDM_MESSAGING_REF` type.
`Ptr_Msg_To_Send:`	Pointer to a data structure of the `FDM_MSG_TO_SEND` type specified by the primitive `fdm_messaging_fullduplex_create()` and containing the transmission parameters of the message, as well as the reference to the message itself.

Description of type `FDM_MESSAGING_REF:`

Inside FIP DEVICE MANAGER: not manipulated by the user.

Description of type: `FDM_MSG_TO_SEND:`

See the primitive `fdm_messaging_fullduplex_create()`.

REPORT:

This function submits a report as specified by the primitive `fdm_stop_network()`.

Note

In the case of a 5 Mbits/s network, the user message should be greater than 4 bytes.

1.64 fdm_msg_send_fifo_empty()

SUMMARY:

Activation of processing of message in transmission mode.

PROTOTYPE:

```
void fdm_msg_send_fifo_empty
              (FDM_REF *Ptr_Network_Identification);
```

DESCRIPTION:

This function enables the user to:

- activate the effective transmission of a message, the request for which has been sent by the `fdm_send_message()` primitive.
- find out about received events regarding acknowledgement of messages, the transmission of which has been requested by the `fdm_send_message()` primitive (line acquired or time-out).

In the second case, when the user activates this procedure, the FIP DEVICE MANAGER activates the `User_Msg_Ack()` user procedure, provided for this purpose on configuration of the messaging context on which the message is sent.

INPUT PARAMETERS:

`Ptr_Network_Identification:` Pointer to a data structure of the `FDM_REF` type.

Description of type `FDM_REF`:

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

None.

1.65 fdm_msg_rec_fifo_empty()

SUMMARY:

Activation of processing of message in reception mode.

PROTOTYPE:

```
void fdm_msg_rec_fifo_empty
            (FDM_REF *Ptr_Network_Identification);
```

DESCRIPTION:

This function enables the user to find out about received messages after their reception has been acknowledged.

When the user activates this procedure, the FIP DEVICE MANAGER activates the `User_Msg_Rec_Proc()` user procedure, provided for this purpose on configuration of the messaging context on which the message was received.

INPUT PARAMETERS:

`Ptr_Network_Identification:` Pointer to a data structure of the `FDM_REF` type.

REPORT:

None.

1.66 fdm_msg_ref_buffer_free()

PROTOTYPE:

```
unsigned short fdm_msg_ref_buffer_free (
       FDM_MSG_RECEIVED *Ptr_Msg_Received);
```

DESCRIPTION:

This function enables the FIP DEVICE MANAGER to free the memory zone that contains a received message descriptor, once it is no longer in use.

INPUT PARAMETERS:

`Ptr_Msg_Received:` Pointer to a data structure of the `FDM_MSG_RECEIVED` type.

Description of type `FDM_MSG_RECEIVED`:

See the primitive `fdm_messaging_fullduplex_create()`.

REPORT:

This function submits a report as specified by the primitive `fdm_stop_network()`.

Note

The freeing operations must be carried out in the following order:

1. `fdm_msg_data_buffer_free()`
2. `fdm_msg_ref_buffer_free()`

1.67 fdm_msg_data_buffer_free()

PROTOTYPE:

```
unsigned short fdm_msg_data_buffer_free (
       FDM_MSG_R_DESC   *Ptr_Block);
```

DESCRIPTION:

This function enables the FIP DEVICE MANAGER to free the memory zone that contains a received message, once it is no longer in use.

INPUT PARAMETERS:

`Ptr_Block:` Pointer to a data structure of the `FDM_MSG_R_DESC` type.

Description of type `FDM_MSG_R_DESC`:

See the primitive `fdm_messaging_full_duplex_create()`.

REPORT:

This function submits a report as specified by the primitive `fdm_stop_network()`.

Note

The freeing operations must be carried out in the following order:

1. `fdm_msg_data_buffer_free()`
2. `fdm_msg_ref_buffer_free()`

1.68 fdm_change_messaging_acknowledge_type()

SUMMARY:

Function for modifying the type of messaging used in transmission mode (with or without acknowledgement).

PROTOTYPE:

```
unsigned short fdm_change_messaging_acknowledge_type(
     FDM_MESSAGING_REF* Ptr_Messaging,
     enum _FDM_MSG_ACK_TYPE
        (*User_Msg_Ack_Type) (struct _FDM_REF*, Unsigned Long Remote_Adr));
```

DESCRIPTION:

This function makes it possible to indicate, context by context, whether it is necessary to modify the type of messaging (acknowledged or not) to be used for transmission.

For a given context, if this function is not called, acknowledged messaging will be used if the destination address is an individual address and non-acknowledged messaging will be used if it is a group address.

If this function is called for a context, each time a message is transmitted, the `User_Msg_Ack_Type` procedure will be called with the destination address as the input parameter, in order to ask which type of messaging is to be used.

INPUT PARAMETERS:

`Ptr_Messaging:`	Pointer to the messaging context.
`User_Msg_Ack_Type:`	User procedure that will be called by FIP DEVICE MANAGER for each message transmitted on this context, to find out which type of messaging to be used.

Description of type: `FDM_MESSAGING_REF`

Inside FIP DEVICE MANAGER: not manipulated by the user.

REPORT:

This function submits a report as specified the primitive `fdm_stop_network`().

2. CALLBACK PRIMITIVES TO BE PROVIDED BY THE USER

The table below is a review of the callback procedures which the user has to write, and which are called by the FIP DEVICE MANAGER to warn the user of various events. In this document, all these functions are prefixed by `User_`. However, the user may name them as he sees fit.

The FIP DEVICE MANAGER recognises these functions by means of a pointer.

PROCEDURE	Reference transmitted to the FIP DEVICE MANAGER by	Called by the FIP DEVICE MANAGER by or during
`User_Reset_Component()`	`Fdm_initialize_network()` in the `FDM_CONFIGURATION_HARD` type parameter	`fdm_stop_network()`
`User_Signal_Fatal_Error ()`	`Fdm_initialize_network()` in the `FDM_CONFIGURATION_HARD` type parameter	Detection of serious errors which lead to shutdown
`User_Signal_Warning()`	`Fdm_initialize_network()` in the `FDM_CONFIGURATION_HARD` type parameter	Warning of non-serious errors
`User_Present_List_Prog ()`	`Fdm_initialize_network()` in the `FDM_CONFIGURATION_SOFT` type parameter	`fdm_smmps_fifo_empty()` for transmitting the ”list of equipment present” variable value
`User_Presence_Prog()`	`Fdm_initialize_network()` in the `FDM_CONFIGURATION_SOFT` type parameter	`fdm_smmps_fifo_empty()` for transmitting the presence variable value
`User_Identification_Prog()`	`Fdm_initialize_network()` in the `FDM_CONFIGURATION_SOFT` type parameter	`fdm_smmps_fifo_empty()` for transmitting the identification variable value
`User_Report_Prog()`	`Fdm_initialize_network()` in the `FDM_CONFIGURATION_SOFT` type parameter	`fdm_smmps_fifo_empty()` for transmitting the report variable value
`User_Synchro_BA_Prog()`	`Fdm_initialize_network()` in the `FDM_CONFIGURATION_SOFT` type parameter	`fdm_smmps_fifo_empty()` for transmitting the BA sync variable value
`User_Signal_Mps_Aper()`	`Fdm_initialize_network()` in the `FDM_CONFIGURATION_SOFT` type parameter	`fdm_process_its_fip()` for indicating a variable has been transmitted or received following a ”universal” type request
`User_Signal_Smmps()`	`Fdm_initialize_network()` in the `FDM_CONFIGURATION_SOFT` type parameter	`fdm_process_its_fip()` for indicating an SM-MPS variable has been received, following a read request
`User_Signal_Rec_Msg()`	`Fdm_initialize_network()` in `FDM_CONFIGURATION_SOFT`	`fdm_process_its_fip()` for indicating a message has been received.
`User_Signal_Send_Msg()`	`Fdm_initialize_network()` in the `FDM_CONFIGURATION_SOFT` type parameter	`fdm_send_message()` and `fdm_process_its_fip()` for indicating a message transmission has been requested or that acknowledgement has been received

PROCEDURE	Reference sent to the FIP DEVICE MANAGER by	Called by the FIP DEVICE MANAGER by or during
User_Signal_Synchro()	fdm_mps_var_create() in the FDM_XAE type parameter	fdm_process_its_fip() to indicate that a sync event has been received
User_Signal_Var_Prod()	fdm_mps_var_create() in the FDM_XAE type parameter	fdm_process_its_fip()to indicate that a universal variable has been transmitted
User_Signal_Var_Cons()	fdm_mps_var_create() in the FDM_XAE type parameter	fdm_process_its_fip()to indicate that a universal variable has been received
User_Signal_Asent()	fdm_mps_var_create() in the FDM_XAE type parameter	fdm_process_its_fip()to indicate that a produced variable has been transmitted
User_Signal_Areceived()	fdm_mps_var_create() in the FDM_XAE type parameter	fdm_process_its_fip() to indicate that a consumed variable has been received.
User_Signal_Mode()	fdm_generic_time_initialize in the FDM_GENERIC_TIME_DEFINITION type parameter	Management of time producer redundancy if switching from prod. to cons. or vice versa.
User_Get_Value()	fdm_mps_var_create() in the FDM_XAE type parameter	fdm_mps_var_time_write_loc() to request the write value of the variable with dynamic refresh status
User_Set_Value()	Fdmçmps_var_create() in the FDM_XAE type parameter.	fdm_mps_var_time_read_loc() to give the read value of the variable with dynamic refresh status.
User_Msg_Rec_Proc()	fdm_messaging_fullduplex_create fdm_messaging_to_rec_create() in the FDM_MESSAGING_FULLDUPLEX and FDM_MESSAGING_TO_REC type parameters	fdm_msg_rec_fifo_empty () to transmit received message
User_Msg_Ack_Proc()	fdm_messaging_fullduplex_create fdm_messaging_to_send_create() in the FDM_MESSAGING_FULLDUPLEX and FDM_MESSAGING_TO_SEND type parameters	fdm_msg_send_fifo_empty() to transmit received acknowledgement
User_Msg_Ack_Type()	fdm_change_messaging_acknowledge_type	The transmission of a message if this primitive was called for the context concerned.
fdm_generic_time_give_value()	Unique for all FIP DEVICE MANAGER instances	fdm_generic_time_read_loc() to give time value
fdm_generic_time_get_value()	Unique for all FIP DEVICE MANAGER instances	fdm_generic_time_write_loc() to request time value

Chapter 4 Performance levels

1. USER PRIMITIVE RUN TIME

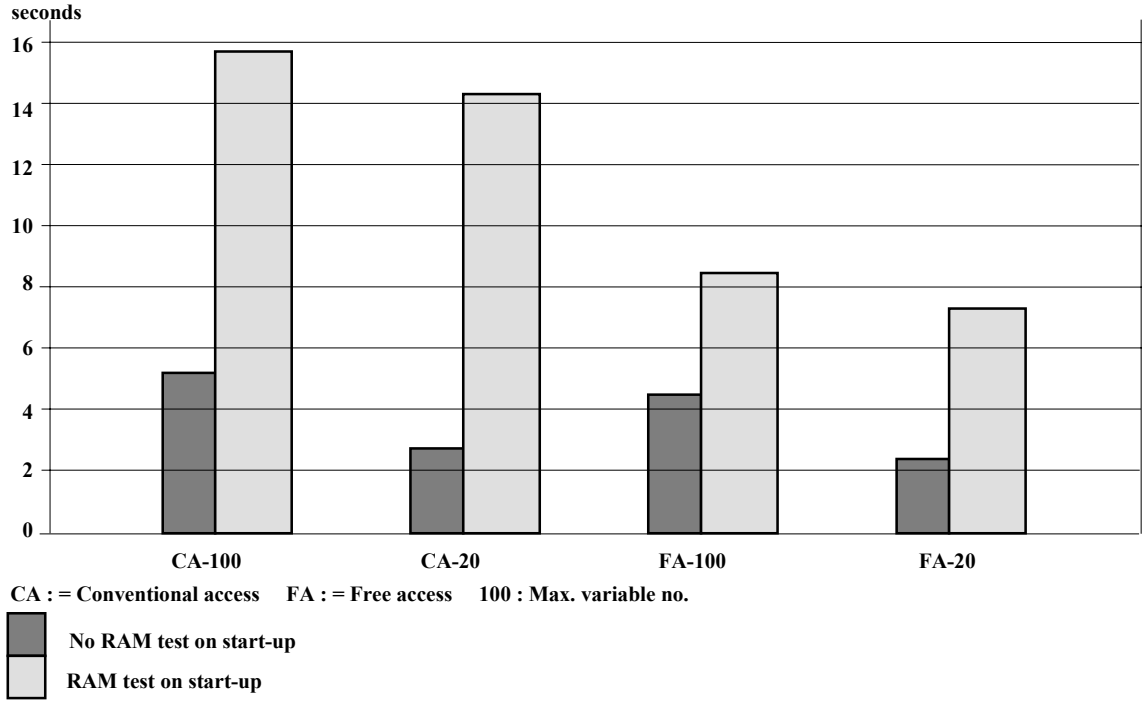
1.1 Introduction

Performance levels are measured using a:

- CC117 board: target with an i960SA microprocessor at 16 MHz with a code generated with no particular optimisation options (with a 1 Mbit/s network, therefore FULLFIP2 64 MHz) and pSOS+,
- CC121 board: ISA board (1 MBytes/s) with a FULLFIP2 64 MHz (1 Mbit/s),
- CC139 board: PCI board (132 MBytes/s) with a FULLFIP2 64 MHz (1 Mbit/s).

The main aim is to determine the difference in performance levels between *conventional access* and *free access* operation.

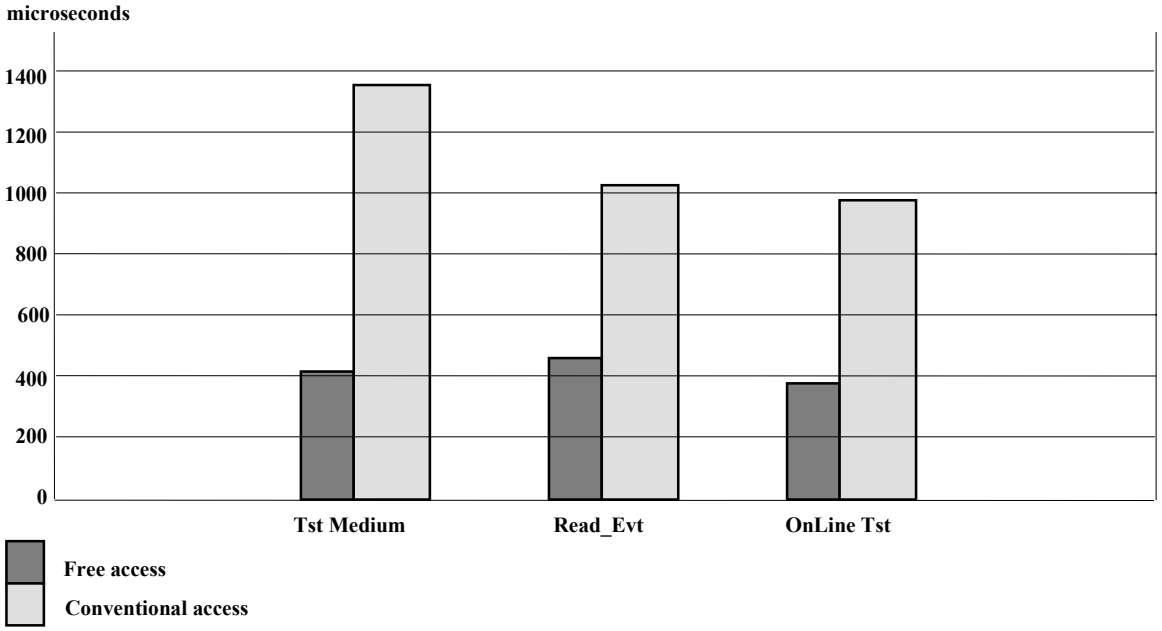
1.2 Start-up: fdm_initialize_network() primitive



The test above has been performed on a CC117 board.

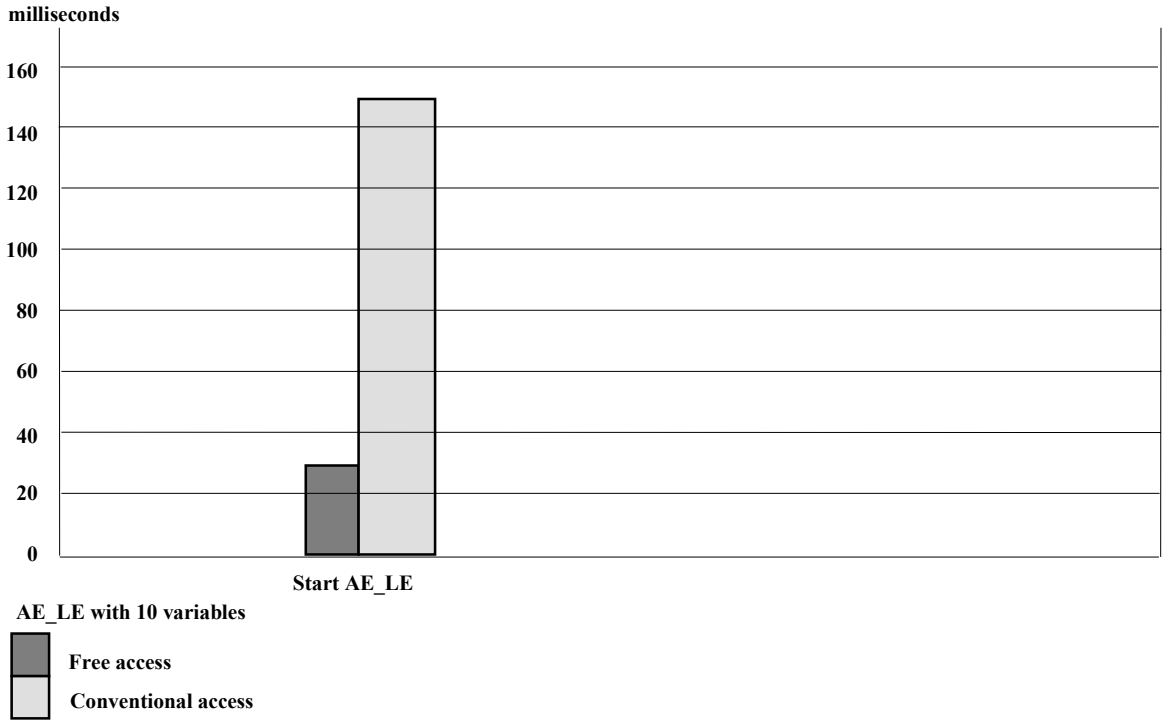
1.3 Miscellaneous functions

`Tst_Medium:`	Periodic medium redundancy management by FIP DEVICE MANAGER (frequency is determined by the user).
`Read_Evt:`	Time taken to process an interrupt and associated processing time (user procedure). This is the mean value of all possible cases.
`On_line_Tst:`	Online Autotest. Duration of a call by FIP DEVICE MANAGER, or by the user (depending on set-up).



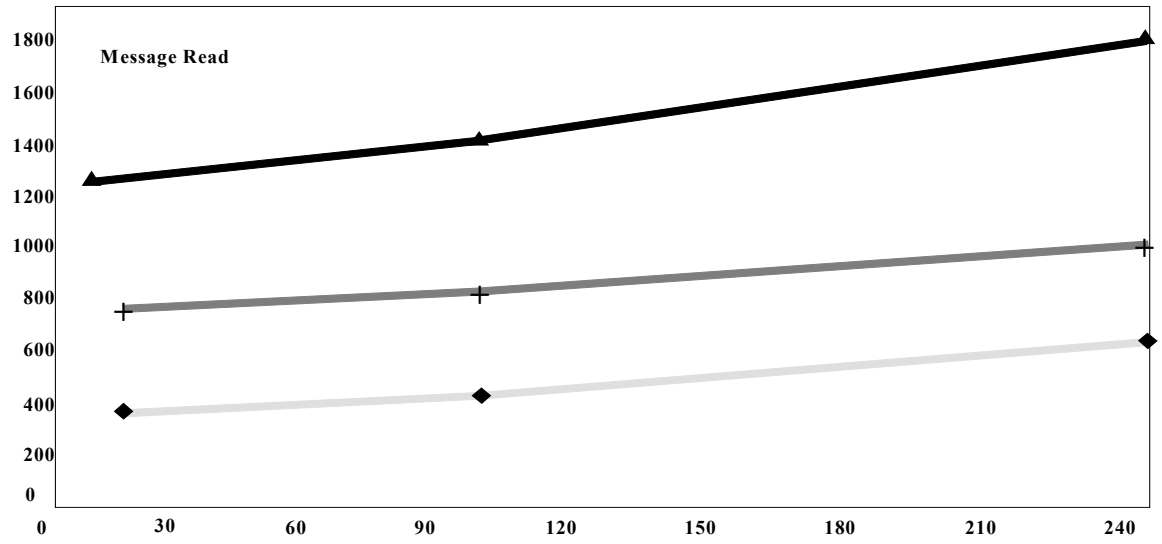
The test above has been performed on a CC117 board.

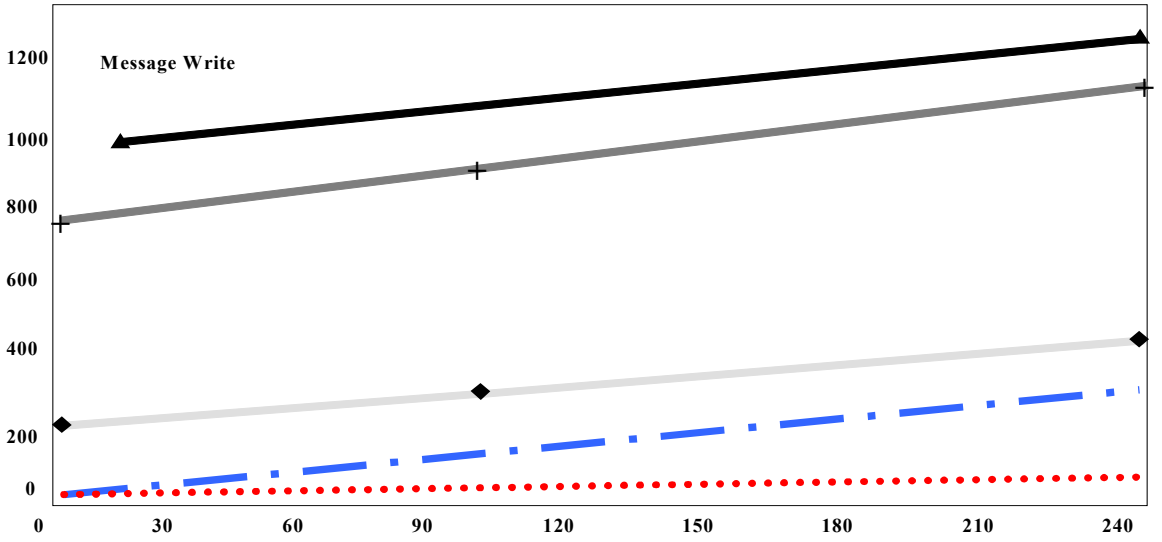
1.4 AE_LE start-up

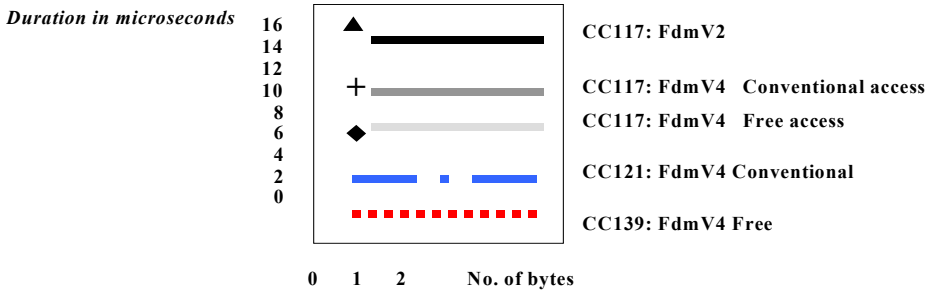


The test above has been performed on a CC117 board.

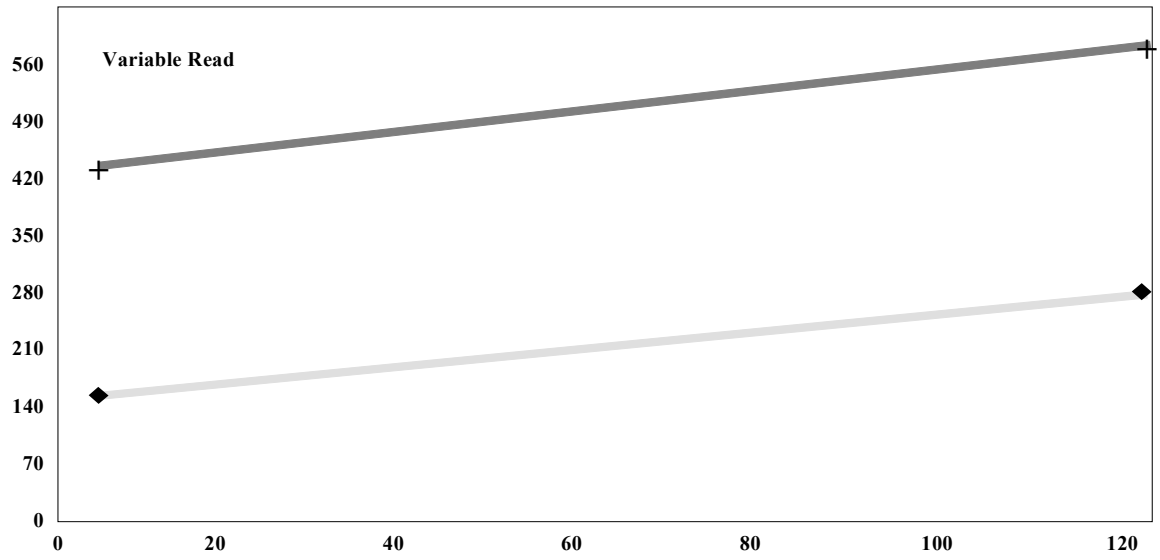
1.5 Messaging

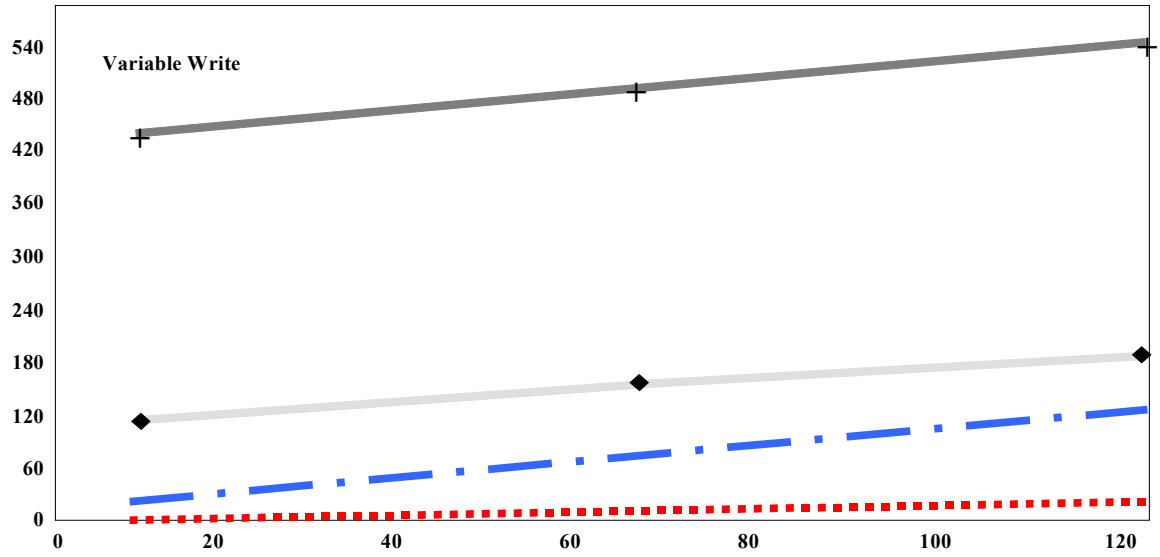


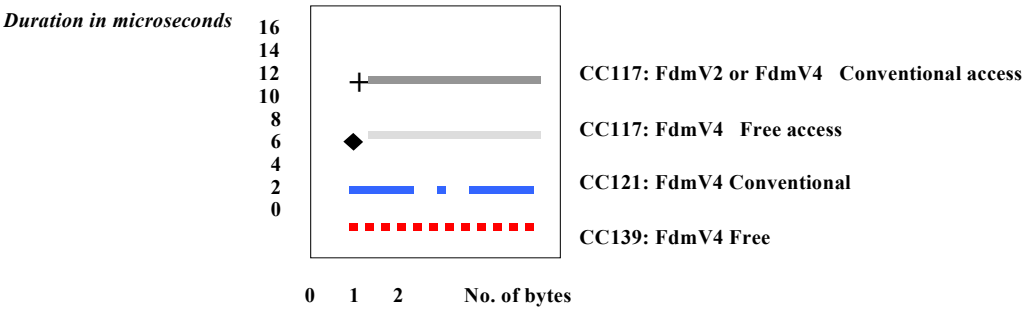




1.6 MPS variables







2. MEMORY CAPACITY REQUIRED

2.1 Code size and modularity

Modularity is implemented using compilation switches (described in Section 3, Chapter 6).

The capacities given for reference are those of the libraries (i.e of the object code) obtained for the INTEL 80x86 (16 bits) target with a BORLAND string.

Implemented Services	• Periodic MPS	• Periodic MPS • Periodic DLL messaging	• Periodic MPS • Aperiodic MPS • Periodic and Aperiodic DLL messaging	• Periodic MPS • Aperiodic MPS • Periodic and Aperiodic DLL messaging • Bus Arbitrator
FIP DEVICE MANAGER	34 KB	40 KB	51 KB	62 KB
FIPCODE	8 KB	14 KB	20 KB	48 KB
	You have to compile the FIPCODE files from **ST1** directory (see Chapter 6)	You have to compile the FIPCODE files from **ST2** directory (see Chapter 6)	You have to compile the FIPCODE files from **ST3** directory (see Chapter 6)	You have to compile the FIPCODE files from **ST4** directory (see Chapter 6)
TOTAL	42 KB	54 KB	71 KB	110 KB

Two independent options can also be used:

- Autodiag: 4 KB,
- Double medium: 3 KB.

This leads to the following maximum and minimum capacities:

	Max. capacity	Min. capacity
FIP DEVICE MANAGER	69 KB	34 KB
FIPCODE	48 KB	8 KB
TOTAL	117 KB	42 KB

Therefore, for a station which has all the functions, the total capacity is: **117 KB.** (by way of comparison, version 2, which has fewer functions, has a capacity of 100 KB)

The minimum set-up (MPS with common functions) has a capacity of **42 KB.**

2.2 Data size

The size of the data zone required for FIP DEVICE MANAGER operation (size on FULLFIP host microprocessor) can be broken down as follows:

stack size: Procedure `fdm_initialize_network()` requires the largest stack size: approximately 1 KB.

heap size: (in KB)

2		
+ 2.4		for electing time producer
+ 0.1	*	number of MPS variables
+ 0.1	*	number of simultaneous active requests for `SM_MPS`
+ 0.06	*	number of messaging contexts
+ (0.036	*	number of received message descriptors) + (0.268 * number of received message blocks)
+ 0.012	*	number of macrocycles

The capacities given are those required by FIP DEVICE MANAGER and which it allocates to itself. They do not include user capacities (buffer containing transmitted messages).

All dynamic memory allocations made by FIP DEVICE MANAGER are allocated on initialisation or by the object creation primitives.

2.3 Memory capacity for FULLFIP2

This is the capacity of the FIPCODE database, depending on the set-up required.

The FIPCODE database comprises 5 separate zones:

Zone A:	microcode + specific data
Zone B:	frame descriptors
Zone C:	variable descriptors = queue descriptors for messaging
Zone D:	BA program and associated queues
Zone E:	message and variable data

Zone capacity constraints:

Zone A capacity < = 24 k words

Zone B capacity < = 64 k words (192 k words in the case of double image operation)

Zone C capacity < = 64 k words

Zone D capacity < = 64 k words

Zone E capacity < = 512 k words

`CONFIGURATION_SOFT.BA_DIM` = **zone D capacity**

`CONFIGURATION_SOFT.FULLFIP_RAM_DIM` = zone A capacity + zone B capacity + zone C capacity + zone D capacity + zone E capacity

Total memory capacity required =

Zone A capacity + zone B capacity + zone C capacity + zone D capacity + zone E capacity

Calculation of zone A capacity:

```
if (FDM_WITH_BA= YES) or (FDM_WITH_FREE_ACCES = YES)
    zone A capacity = 48 KB
otherwise
    if FDM_WITH_APER = YES then
        zone A capacity = 20 KB
    otherwise
        if FDM_WITH_MESSAGING = YES then
            zone A capacity = 14 KB
        otherwise
            zone A capacity = 8 KB
        endif
    endif
endif
```

Calculation of zone B capacity:

Zone B capacity =	256 words	* 3	if MPS variable number + 5	< 16
Zone B capacity =	512 words	* 3	if MPS variable number + 5	< 32
Zone B capacity =	1024 words	* 3	if MPS variable number + 5	< 64
Zone B capacity =	2048 words	* 3	if MPS variable number + 5	< 128
Zone B capacity =	4096 words	* 3	if MPS variable number + 5	< 256
Zone B capacity =	8192 words	* 3	if MPS variable number + 5	< 512
Zone B capacity =	16384 words	* 3	if MPS variable number + 5	< 1024
Zone B capacity =	32768 words	* 3	if MPS variable number + 5	< 2048
Zone B capacity =	64536 words	* 3	if MPS variable number + 5	< 4096

Calculation of zone C capacity

zone C capacity = TVAR + TMSGE + TMSGR

TVAR = 160 + (16 * number of MPS variables) words

```
TMSGE =    16 *        (CONFIGURATION_SOFT.Nr_Of_Tx_Buffer [0] +
                        CONFIGURATION_SOFT.Nr_Of_Tx_Buffer [1] +
                        CONFIGURATION_SOFT.Nr_Of_Tx_Buffer [2] +
                        CONFIGURATION_SOFT.Nr_Of_Tx_Buffer [3] +
                        CONFIGURATION_SOFT.Nr_Of_Tx_Buffer [4] +
                        CONFIGURATION_SOFT.Nr_Of_Tx_Buffer [5] +
                        CONFIGURATION_SOFT.Nr_Of_Tx_Buffer [6] +
                        CONFIGURATION_SOFT.Nr_Of_Tx_Buffer [7] +
                        CONFIGURATION_SOFT.Nr_Of_Tx_Buffer [8])
```

TMSGR = 1024 words

Calculation of zone D capacity

zone D capacity =

1000 + (8 * `CONFIGURATION_SOFT.NB_OF_DIFFERENT_ID_PROG_BA`)
+ (3 * total number of instructions)

Note: for a SEND_LIST instruction, the actual number of instructions is equal to the number of IDs on the list.

Calculation of zone E capacity

zone E capacity = TDATAV + TDATAME + TDATAMR

if (FDM_WITH_FREE_ACCES = YES then

TDATAV = (128 * number of variables) + 1216 words

otherwise

TDATAV = (64 * number of variables) + 704 words

endif

```
TDATAME =    135 *       (CONFIGURATION_SOFT.Nr_Of_Tx_Buffer [0] +
                          CONFIGURATION_SOFT.Nr_Of_Tx_Buffer [1] +
                          CONFIGURATION_SOFT.Nr_Of_Tx_Buffer [2] +
                          CONFIGURATION_SOFT.Nr_Of_Tx_Buffer [3] +
                          CONFIGURATION_SOFT.Nr_Of_Tx_Buffer [4] +
                          CONFIGURATION_SOFT.Nr_Of_Tx_Buffer [5] +
                          CONFIGURATION_SOFT.Nr_Of_Tx_Buffer [6] +
                          CONFIGURATION_SOFT.Nr_Of_Tx_Buffer [7] +
                          CONFIGURATION_SOFT.Nr_Of_Tx_Buffer [8]) words
```

TDATAMR = 8775 words

Chapter 5 Error codes

1. ERROR MESSAGES

Corresponds to contents of `FDM_ERROR_CODE.FDM_DEFAULT` which is of the `enum Code_Error` type. E= error indicated by `User_Signal_Fatal_Error`, W= warning indicated by `User_Signal_Warning` (in 3rd column of table).

VALUE	MNEMONIC	CONTEXT	E/W	PROBABLE CAUSE
0x000	_NO_ERROR			
0x100	_FIPCODE ERROR	All primitives	W	See `Fipcode_Report`
0x201	_TIME_OUT	All primitives	W	See `Fipcode_Report`

0x202	_FIPCODE_RESPONSE_IMPOSSIBLE	`fdm_mps_var_read_loc`
		`fdm_mps_var_write_loc`
		`fdm_mps_var_write_universal`
			W	FIPCODE replies that a variable cannot be produced on a produced variable write or that a variable cannot be consumed on a consumed variable read

0x203	_FIPCODE_FAIL	`fdm_initialize_network`	E	FIPCODE cannot be loaded
0x204	_INIT_TIMER_FAIL	`fdm_initialize_network`	E	`fdm_Initialize` not called
0x205	_INIT_ABORTED	`fdm_initialize_network`	E	All warnings during `fdm_initialize_network` execution are upgraded

0x206	_RN_FAILED	`fdm_initialize_network`
		`fdm_ae_le_create`
		`fdm_mps_var_create`
		`fdm_load_macrocycle_fipconfb`
		`fdm_load_macrocycle_manual`
		`fdm_messaging_fullduplex_create`
		`fdm_messaging_to_send_create`
		`fdm_messaging_to_rec_create`
			E
			W
			W
			W
			W
			W
			W
			W
				Problem to allocate or restore memory when a divided memory manager is used

0x207	_SM_FAILED	`fdm_initialize_network`	E	Problem to create system flag

0x208	_CIRCUIT_ACCES_FAILED	`fdm_ba_load_macrocycle_manual`
		`fdm_load_macrocycle_fipconfb`
		`fdm_initialize_network`
		`fdm_messaging_to_rec_create`
		`fdm_messaging_fullduplex_create`
		`fdm_messaging_to_send_create`
			W
			W
			E
			W
			W
			W
				P roblem with physical access to component

0x209	_ILLEGAL_POINTER	`fdm_msg_send_fifo_empty`	E	Use of memory resources
0x20A	MEDIUM_EWD_FAILED	`fdm_ticks_counter`	W	Hardware error
0x20B	FIFO_ACCES_FAILED	`fdm_msg_send_fifo_empty`		If the model of the transmission of messages (see Chapter 1, Subsection 4.7.3) is not respected

VALUE	MNEMONIC	CONTEXT	E/W	PROBABLE CAUSE
0x401	_ALLOCATE_MEMORY_FAULT	fdm_initialize_network fdm_ae_le_create fdm_mps_var_create fdm_load_macrocycle_fipconfb fdm_load_macrocycle_manual fdm_messaging_fullduplex_create fdm_messaging_to_send_create fdm_messaging_to_rec_create fdm_ae_le_start fdm_read_ident fdm_read_report fdm_read_presence	E W W W W W W W W W W W	Problem with memory allocation
0x402	_PRIVATE_FULLFIP_RAM_OVERFLOW	fdm_initialize_network fdm_ae_le_create fdm_messaging_fullduplex_create fdm_messaging_to_send_create fdm_messaging_to_rec_create fdm_ae_le_start fdm_read_ident fdm_read_report fdm_read_presence	E W W W W W W W W	The configuration requested does not fit into the private FULLFIP2 memory available.
0x403 0x404 0x405	_FRAME_DESCRIPTOR_TABLE_OVERFLOW _VARIABLE_DESCRIPTOR_TABLE_OVERFLOW VARIABLE_TABLE_OVERFLOW	fdm_initialize_network fdm_messaging_fullduplex_create fdm_messaging_to_send_create fdm_messaging_to_rec_create fdm_ae_le_start fdm_read_ident fdm_read_report fdm_read_presence fdm_create_channel	E W W W W W W W W	Parameter FDM_CONFIGURATION_SOFT..NB_OF_USER _MPS_VARIABLE is too small
0x406	_VARIABLE_ALREADY_EXIST	fdm_ae_le_start	W	Variable has already been defined
0x407	_ILLEGAL_IDENTIFICATION_PARAMETER	fdm_initialize_network	E	Contents of FDM_IDENTIFICATION incorrect
0x408	_ILLEGAL_IDENTIFICATION_LENGTH	fdm_initialize_network	E	Contents of FDM_IDENTIFICATION incorrect
0x409	_INTEGRITY_DATA_BASE_FAULT	fdm_initialize_network fdm_messaging_fullduplex_create fdm_messaging_to_send_create fdm_messaging_to_rec_create fdm_ae_le_start fdm_read_ident fdm_read_report fdm_read_presence	E E E E E E E E	FIP DEVICE MANAGER self-test
0x40A	_SYNCHRO_VAR_ALREADY_EXIST	fdm_ae_le_start	W	Sync variable must be unique
0x40B	_INDICATION_PROG_MISSING	fdm_mps_var_create	W	User_Signal_Asent or User_Signal_Areceive absent
0x40C	_CONFLICT_CONFIGURATION	fdm_initialize_network fdm_ae_le_start	E W	FIPIO problem between VCOM and LSAP
0x501	_GEN_AE_LE_ILLEGAL_LENGTH	fdm_ae_le_create	W	Variable number declared for AE/LE is either <1 or > max.

VALUE	MNEMONIC	CONTEXT	E/W	PROBABLE CAUSE
0x502	_GEN_AE_LE_ILLEGAL_STATE	fdm_mps_var_create fdm_ae_le_start fdm_ae_le_stop fdm_ae_le_delete fdm_mps_var_change_id fdm_mps_var_read_loc fdm_mps_var_write_loc fdm_mps_var_write_universal fdm_mps_var_read_universal fdm_read_ident fdm_read_presence fdm_read_report	W W W W W W W W W W W W W	AE/LE is not in a state where primitive execution is possible
0x503	_GEN_AE_LE_ILLEGAL_RANK	fdm_ae_le_create	W	Rank greater than size of AE/LE
0x504	_GEN_AE_LE_ILLEGAL_POSITION	fdm_mps_var_create	W	Variable declared in image 2 if single image configuration
0x505	_GEN_AE_LE_ILLEGAL_MODIFICATION	fdm_mps_var_change_MSGa fdm_mps_var_change_RQa fdm_mps_var_change_priority fdm_mps_var_change_id fdm_mps_var_change_periods fdm_mps_var_change_prod_cons	W W W W W W	AE/LE is not in a state where modification requested can be made
0x506	_GEN_AE_LE_START_NOT_POSSIBLE	fdm_ae_le_start	W	Parameters inconsistent or missing
0x507	_GEN_AE_LE_TIME_PROG_MISSING	fdm_mps_var_create	W	User_Get_Value() procedure absent
0x508	_GEN_AE_LE_SYNCHRO_PROG_MISSING	fdm_mps_var_create	W	User_Synchro() procedure absent
0x509	_GEN_AE_LE_SYNCHRO_PROG_ILLEGAL	fdm_mps_var_create	W	User_Synchro() procedure superfluous
0x50A	_GEN_AE_LE_ILLEGAL_CONFIGURATION	fdm_mps_var_create fdm_generic_time_initialize	W W	Problem creating variable with dynamic refresh
0x601	_GEN_BA_CHECKSUM_ERROR	fdm_ba_load_macrocycle_fipconfb fdm_ba_load_macrocycle_manual	W W	Checksum incorrect
0x602	_GEN_BA_LIST_ERROR	fdm_ba_load_macrocycle_fipconfb fdm_ba_load_macrocycle_manual	W W	Content of a list is incorrect
0x603	_GEN_BA_END_BA_NOT_FOUND	fdm_ba_load_macrocycle_fipconfb fdm_ba_load_macrocycle_manual	W W	Error in BA program: no END_BA
0x604	_GEN_BA_NOT_EXISTED_INSTRUCTION	fdm_ba_load_macrocycle_fipconfb fdm_ba_load_macrocycle_manual	W W	Error in BA program: use of non-existent operating code
0x605	_GEN_BA_NOT_VALID_TIME	fdm_ba_load_macrocycle_fipconfb fdm_ba_load_macrocycle_manual	W W	Error in BA program: time parameter incorrect
0x606	_GEN_BA_NOT_ID_IN_LIST	fdm_ba_load_macrocycle_fipconfb fdm_ba_load_macrocycle_manual	W W	Error in BA program: use of empty ID list
0x607	_GEN_BA_DEF_DICHO	fdm_ba_load_macrocycle_fipconfb fdm_ba_load_macrocycle_manual	W W	Problem with memory allocation
0x608	_GEN_BA_DEF_DEM_MEM_MC	fdm_ba_load_macrocycle_fipconfb fdm_ba_load_macrocycle_manual	W W	No space in FULLFIP2 RAM for macrocycle
0x609	_GEN_BA_DEF_DIM_FILE_FILES	fdm_initialize_network	W	Parameter
0x901	_LEVEL_PRIORITY_BA	fdm_ba_set_priority fdm_ba_set_parameters	W W	Incorrect parameter
0x902	_MAX_SUBSCRIBER_BA	fdm_ba_set_priority fdm_ba_set_parameters	W W	Incorrect parameter

VALUE	MNEMONIC	CONTEXT	E/W	PROBABLE CAUSE
0x903	_DELETE_MC_IN_USE	fdm_ba_delete_macrocycle	W	Macrocycle in "RUN" state
0x904	_USER_NB_OF_USER_MPS_VARIABLE	fdm_initialize_network	E	FDM__CONFIGURATION_SOFT.Desc_Par incorrect
0x905	_USER_MODE_FAULT	fdm_initialize_network	E	Parameter FDM_CONFIGURATION_SOFT. Timer incorrect
0x906	_VAR_PERIOD_FAULT	fdm_ae_le_start fdm_mps_var_change_periods	W W	Refresh or prompt time-out value is incorrect
0x907	_LENGTH_VAR_FAULT	fdm_ae_le_start	W	
0x908	_USER_NR_OF_TX_BUFFER_FAULT	fdm_initialize_network	E	Number of buffers for message transmission incorrect
0x909	_USER_SWITCH_FAULT	fdm_ae_le_switch_image	W	Switch attempt during single image operation
0x90A	_NON_CONSUMING_VARIABLE	fdm_mps_var_read_loc fdm_mps_var_read_universal	W W	Produced variable read attempt
0x90B	_NON_PRODUCING_VARIABLE	fdm_mps_var_write_loc fdm_mps_var_write_universal	W W	Consumed variable write attempt
0x90C	IMAGE_FAULT_ON_REQUEST	fdm_mps_var_write_universal fdm_mps_var_read_universal	W W	If running on IMAGE 2
0x90D	_REQUEST_RAN_ALWAYS	fdm_mps_var_write_universal fdm_mps_var_read_universal	W W	Previous remote access to same variable is still in progress
0x90E	_NOT_TIME_VARIABLE	fdm_mps_var_time_read_loc fdm_mps_var_time_write_loc	W W	Accessed variable is not a time variable
0x90F	_INDICATION_VARIABLE_IN_USE	fdm_mps_var_write_universal fdm_mps_var_read_universal	W W	Universal access prohibited on variables with Asent ou Areceive
0x910	_BA_STOPPED_ON_TIME_OUT	fdm_licks_conter	W	Impossible to send potential BA
0xA01	_CHANNEL_ALREADY_EXIST	fdm_channel_create	W	Creation of an existing channel
0xA02	_INVALID_NO_CHANNEL	fdm_channel_create fdm_channel_nr_modif fdm_channel_delete	W W W	Number of channel specified is < 1 or > 8
0xA03	_CHANNEL_NOT_CONFIGURED	fdm_send_message fdm_channel_delete	W W	Number of channel specified does not exist
0xA04	_NOT_TX_BUFFER_ON_CHANNEL	fdm_channel_create	W	No buffer associated with this channel
0xA05	_NOT_CREATED_CHANNEL_ON_THIS_VARIABLE	fdm_channel_create	W	Creation of channel impossible on this ID
0xA06	_MSG_PROG_MISSING	fdm_initialize_network	W	User procedure indicating message transmission or reception has not been defined
0xA07	_EXCEEDED_SEGMENT_NUMBER	fdm_messaging_fullduplex_create fdm_messaging_to_send_create fdm_messaging_to_rec_create	W W W	Creation of messaging context impossible as concerns segment number possibilities in standard
0xA08	_MISSING_MEMORY_DELETED_MESSAGE	fdm_msg_rec_fifo_empty	W	Memory pool for message reception empty
0xA09	_IMAGE_NOT_CONFIGURED	fdm_messaging_to_rec_create fdm_messaging_fullduplex_create fdm_messaging_to_send_create	W W W	Image field missing
0xB01	_GEN_MESSAGING_USER_ACK_EMIS	fdm_messaging_fullduplex_create fdm_messaging_to_send_create fdm_messaging_to_rec_create	W W W	User procedure indicating transmission acknowledgement not defined
0xB02	_GEN_MESSAGING_USER_ACK_RECEP	fdm_messaging_fullduplex_create fdm_messaging_to_send_create fdm_messaging_to_rec_create	W W W	User procedure indicating reception not defined

VALUE	MNEMONIC	CONTEXT	E/W	PROBABLE CAUSE
0xB03	_GEN_MESSAGING_BLOCK_DESC_ERROR	`fdm_send_message`	W	Attempt to transmit message with l = 0
0xB04	_MESSAGING_CONTEXT_NOT_TO_SEND	`fdm_channel_nr_modif`	W	Context used not of transmission type
0xB05	_MESSAGING_CONTEXT_NOT_ALLOWED	`fdm_msg_send_fifo_empty`	W	Message transmission on non-existent context

0xB06	_GEN_MESSAGING_ALREADY_CONFIGURED	`fdm_messaging_fullduplex_create`
		`fdm_messaging_to_send_create`
		`fdm_messaging_to_rec_create`
			W	LSAP already exists

0xB07	_GEN_MESSAGING_ILLEGAL_SEGMENT	`fdm_messaging_to_rec_create` `fdm_messaging_fullduplex_create`	W W	Reception request of bridge type on its own segment.
0xC01	_TEST_RAM_FAIL	`fdm_initialize_network`	E	Private FULLFIP2 RAM failure
0xC02	_DIAG_FAIL_ON_BUSY_PIN	`fdm_initialize_network`	E	NOK pin test
0xC03	_DIAG_FAIL_ON_EOC_PIN	`fdm_initialize_network`	E	NOK pin test
0xC04	_DIAG_FAIL_ON_IRQ_PIN	`fdm_initialize_network`	E	NOK test of interface register bit
0xC05	_DIAG_FAIL_ON_SV_PIN	`fdm_initialize_network`	E	NOK test of interface register bit
0xC06	_DIAG_FAIL_ON_FR_PIN	`fdm_initialize_network`	E	NOK test of interface register bit
0xC07	_DIAG_FAIL_ON_AE_PIN	`fdm_initialize_network`	E	NOK test of interface register bit
0xC08	_DIAG_FAIL_ON_FE_PIN	`fdm_initialize_network`	E	NOK test of interface register bit
0xC09	_DIAG_FAIL_ON_IC_PIN	`fdm_initialize_network`	E	NOK test of interface register bit
0xC0A	_DIAG_FAIL_ON_INTERNAL_FIFO	`fdm_initialize_network`	E	Internal test of NOK component
0xC0B	_DIAG_FAIL_ON_INTERNAL_REGISTERS	`fdm_initialize_network`	E	Internal test of NOK component
0xC0C	_DIAG_FAIL_ON_INTERNAL_TIMERS	`fdm_initialize_network`	E	Internal test of NOK component
0xC0D	_ON_LINE_DIAG_FIPCODE_FAILED	`fdm_online_test`	E	Internal test of NOK component
0xC0E	_ON_LINE_COMPONENT_ACCES_FAILED	`fdm_online_test`	E	NOK component access test
0xC0F	_ON_LINE_TEST_RAM_FAILED	`fdm_online_test`	E	RAM test
0xC10	_TEST_ON_TICKS_ON_LINE	`fdm_online_test`	W	Automatic self-tests in use therefore manual tests NOK
0xD01	_ERR_FIP_DOWNLOAD_FILE	`fdm_initialize_network`	E	FIPCODE.DAT file cannot be opened

Chapter 6 Installation procedure

1. SOFTWARE SUPPLY

The software is supplied on a CD-ROM. Under **Programs\Fdm_R4.X** you can find the following files and directories:

- FIP DEVICE MANAGER files,
- the following sub-directories:
  - **ST1**: directory containing FIPCODE files that have to be used in your *makefile* if you need the following services (see Chapter 4, Subsection 2.1):
    - Periodic MPS.
  - **ST2**: directory containing FIPCODE files that have to be used in your *makefile* if you need the following services (see Chapter 4, Subsection 2.1):
    - periodic MPS,
    - periodic DLL messaging.
  - **ST3**: directory containing FIPCODE files that have to be used in your *makefile* if you need the following services (see Chapter 4, Subsection 2.1):
    - periodic MPS,
    - aperiodic MPS,
    - periodic and aperiodic DLL messaging.
  - **ST4**: directory containing FIPCODE files that have to be used in your *makefile* if you needs the following services (see Chapter 4, Subsection 2.1) :
    - periodic MPS,
    - aperiodic MPS,
    - periodic and aperiodic DLL messaging,
    - Bus Arbitrator.
  - **Demos\CC116**: directory containing one example using an ALSTOM Technology target CC116/117/118

Note

Under this sub-directory you can find an example of the *user_opt.h* file.

2. IMPLEMENTATION

To implement the software you have to:

- first write the user procedures likely to be used,
- fill in the macros for the interface with the operating system (*fdm_os.c* file),
- customise the *user_opt.h* file according to the requirements of the application in question,
- compile all xx.c files using the required compiler and the appropriate compilation options,
- build the FIPMAN.LIB library (this name is given for reference only: any other name can be chosen) using the appropriate library manager.
- connect the created library to the rest of the software.

Note

If a 16-bit, 80x86 microprocessor is used along with free access operation, the compilation model to be used is HUGE.

3. COMPILATION PARAMETERS

The user can set compilation parameters as configuration options in the file *user.opt.h* (see example below).

Contents of *user_opt.h* file

```
#define                     YES                                 1
#define                     NO                                  0
#define                     infini                              0
#define                     5_millisecondes                     5000
#define                     FDM_WITH_FIPCODE_31_25              NO
#define                     FDM_WITH_FIPCODE_1000               YES
#define                     FDM_WITH_FIPCODE_5000               YES
#define                     FDM_WITH_APER                       YES
#if (FDM_WITH_APER ==YES)
#define                     FDM_WITH_SM_MPS                     YES
#endif
#define                     FDM_WITH_MESSAGING                  YES
#define                     FDM_WITH_BA                         YES
#define                     FDM_WITH_DIAG                       YES
#if (FDM_WITH_BA == YES)
#define SUSPEND_DELAI       _5_millisecondes                    5000
#define                     FDM_WITH_FIPIO                      YES
#define                     FDM_WITH_OPTIMIZED_BA               YES
#endif
#define                     FDM_WITH_CONTROLS                   YES
#define                     FDM_WITH_GT                         YES
#define                     FDM_WITH_GT_ONLY_PRODUCED           YES
#define                     FDM_WITH_GT_ONLY_CONSUMED           YES
#define                     FDM_WITH_SUSPEND_BA_ON_DEFAULT      NO
#define                     FDM_WITH_BI_MEDIUM                  YES
#define                     FDM_WITH_REDUNDANCY_MGNT            YES
#define                     FDM_WITH_FREE_ACCESS                YES
#define                     FDM_WITH_CHAMP_IO                   YES
#define                     FDM_WITH_DTACK                      YES
#define                     FDM_WITH_FIPCODE_LINKED             YES
#define                     FDM_WITH_IRQ_EOC_ON_SAME_INTERRUPT  YES
#define                     FDM_WITH_I960                       YES
#define                     FDM_WITH_68XXX                      YES
#define                     FDM_WITH_DSP                        YES
#define                     FDM_WITH_X86                        YES
#define                     FDM_WITH_PSOS                       YES
#define                     FDM_WITH_NT                         NO
#define                     FDM_WITH_VXWORKS                    NO
#define                     FDM_WITH_NDIS                       NO
#define                     FDM_WITH_SOLARIS                    NO

#if (FDM_WITH_PSOS == YES)
 #define FDM_WITH_DELETE_RN_OVERRIDE                             YES
```

```
 #if (FDM_WITH_I960 == YES)
 #define SPLX_VALUE_FOR_MASK_ALL                                  31
#endif
 #if (FDM_WITH_68XXX == YES)
#define                     SPLX_VALUE_FOR_MASK_ALL            7
#endif
#endif
#define                     FDM_WITH_SEMAPHORE                 YES
#define                     FDM_WITH_LITTLE_INDIAN             YES
#if (FDM_WITH_LITTLE_INDIAN ==YES)
 #define FDM_WITH_WIN32                                          NO
 #define FDM_WITH_SCOUNIX                                        NO
 #define FDM_WITH_MICROSOFT                                      NO
 #define FDM_WITH_BORLAND                                        NO
 #define FDM_WITH_METAWARE                                       NO
#endif
#define                     FDM_WITH_MICROTEC                  YES
#define                     FDM_WITH_GNU                       NO
#define                     FDM_WITH_SPECIAL_USAGE             NO
```

Note

If FDM_WITH_FIPECODE_31_25 is set to YES, then FDM can drive a FULLFIP2 component at 31.25 kbits/s.

If FDM_WITH_FIPECODE_1000 is set to YES, then FDM can drive a FULLFIP2 component at 1 Mbit/s and 2.5 Mbits/s.

If FDM_WITH_FIPECODE_5000 is set to YES, then FDM can drive a FULLFIP2 component at 5 Mbits/s.

If you have an multi-speed application, it is possible to set all the above options to YES.

Use

You should customise the FIP DEVICE MANAGER library by associating the value **YES** or **NO** to each parameter constant described in the *user_opt.h* file.

The homogeneity of the choices made by the user is systematically checked during compilation.

OPTIONS	EFFECT (IF VALUE ”YES” IS SELECTED)
FDM_WITH_BA	Use of bus arbitrator function.
FDM_WITH_OPTIMIZED_BA	Used for the calculation of election and start-up time-outs of the startup of the BA (see Chapter 2, Subsection 1.4.3).
FDM_WITH_MESSAGING	Use of messaging services at WorldFIP data link layer level.
FDM_WITH_FIPCODE_31_25	Use of specific microcode for 31.25 Kbits/s.
FDM_WITH_FIPCODE_1000	Use of specific microcode for 1 Mbyte/s
FDM_WITH_FIPCODE_5000	Use of specific microcode for 5 Mbytes/s
FDM_WITH_APER	Use of aperiodic messaging services or MPS or SM_MPS even if local report variable is read only.
FDM_WITH_SM_MPS	Use of SM_MPS functions (e.g. remote subscriber report variable read) other than production of its own variables, which is obligatory.
FDM_WITH_FIPIO	Use of specific FIPIO variable (i.e. to build a FIPIO manager to manage FIPIO agent).
FDM_WITH_DIAG	Use of “on-line” and “off-line” self-diagnosis.
FDM_WITH_CONTROLS	Tests to check validity of all primitive input parameters have been deleted (saves space in code).
FDM_WITH_GT	Use of entire time producer redundancy management procedure. Production or consumption of time.
FDM_WITH_GT_ONLY_PRODUCED	Use of time in production mode only.
FDM_WITH_GT_ONLY_CONSUMED	Use of time in consumption mode only.
FDM_WITH_SUSPEND_BA_ON_DEFAULT	In the case of a redundant medium, if no activity is detected on a medium then setting this parameter to : • NO means that the BA remains active, • YES means that the BA priority is set to the lowest value. In the case that this BA is the active BA then the BA is stopped and then started; in this case it is possible that another BA will be elected (see the Chapter 2, Subsection 1.4).
FDM_WITH_BI_MEDIUM	Medium redundancy management. Mandatory if card is double medium.
FDM_WITH_REDUNDANCY_MGNT	FIP DEVICE MANAGER (therefore host processor involved) manages medium redundancy. Otherwise, it is managed entirely by FIPCODE.
FDM_WITH_FIPCODE_LINKED	FIPCODE is link-edited with FIP DEVICE MANAGER (as opposed to FIPCODE that is contained in a disk file).

OPTIONS	EFFECT (IF VALUE ”YES” IS SELECTED)
FDM_WITH_PSOS FDM_WITH_NT FDM_WITH_VXWORKS FDM_WITH_NDIS FDM_WITH_SCOUNIX FDM_WITH_SOLARIS FDM_WITH_RTX	For options specific to various operating system or execution contexts.
FDM_WITH_DELETERN_OVERRIDE	Restoration of resources after shutdown caused by pSOS.
FDM_WITH_VXWORKS	Use with real time monitor VxWORKS.
FDM_WITH_NDIS	Use with ALSTOM TECHNOLOGY driver PC NDIS.
FDM_WITH_FREE_ACCESS	Use of free accesses function on system user interface: database is in shared memory between the host processor and FULLFIP2 and transactions with the system are reduced to a minimum.
FDM_WITH_CHAMP_IO	Can be used with INTEL type microprocessors when the FULLFIP2 system is located in the I/O field, as opposed to conventional location in the memory field.
FDM_WITH_DTACK	Dtack ”hard” signal can be used. As opposed to activation by the software of FULLFIP2 Ustate register FR bit test each time the system FILE register is accessed.
FDM_WITH_IRQ_EOC_ON_SAME_INTERRUPT	Use of interrupts EOC et IRQ, by host microprocessor, at the same interrupt level.
FDM_WITH_LITTLE_ENDIAN	Data representation in conventional INTEL internal memory.
FDM_WITH_BORLAND FDM_WITH_MICROSOFT FDM_WITH_METAWARE FDM_WITH_GNU FDM_WITH_MICROTEC FDM_WITH_MSDOS FDM_WITH_CAD_UL	For options specific to various compilers.
FDM_WITH_DSP FDM_WITH_X86 FDM_WITH_PLX9050	For options specific to various microprocessors.
FDM_WITH_SPECIAL_USAGE	If the option is set to YES then the user can specify some elements for his application. These elements can be defined between a #ifdef FDM_WITH_SPECIAL_USAGE and #endif couple.
FDM_WITH_WIN32	For use with Win 32.

Table 6.1 – FIP DEVICE MANAGER library options

There are three constants:

SUSPEND_DELAI:	Time-out on execution of TEST_P by an active BA. If this timeout elapses, the BA is stopped. If the value equals INFINITY, there is no active timeout.
_5_millisecondes	
SPLX_VALUE_FOR_MASKALL:	Max. operating level for an interrupt to be impossible (must be greater than max. level used) Used with certain processors (68xxx mask = 7, i960 mask = 31).

Note

A certain number of development contexts are taken into account as standard by FIP DEVICE MANAGER (see options above). For all special cases not covered by these options, the user must add its own options.

If they prove to be of general interest and are sent to ALSTOM Technology, they could be included in the library as standard.

Appendix A Compatibility

The following table gives the functional compatibility between Version 2 and 4 primitives.

FIP DEVICE MANAGER R2	FIP DEVICE MANAGER R4
Short INIT_FIP_DB_PRG()	deleted
Short INIT_FIP_DB_FILE()	deleted
Short INIT_FIP_DB_PROM()	deleted
Short INIT_FIP_DB_PRG_TST()	deleted
Short INIT_FIP_DB_FILE_TST()	deleted
Short INIT_FIP_DB_PROM_TST()	deleted
_FDM_REF *Initialize_FULLFIP_Network()	
void REMOVE_FIP_DB()	FDM_REF *fdm_initialize_network()
void TEST_MEDIUM()	unsigned short fdm_stop_network()
	void fdm_initialize()
short VALID_MEDIUM()	
void GET_COUNTER()	deleted
short FIP_ONLINE_TEST()	unsigned short fdm_valid_medium()
void SMAP_FIP_EVT()	unsigned short fdm_online_test()
	unsigned short fdm_process_its_fip()
	unsigned short fdm_process_it_eoc()
	unsigned short fdm_process_it_irq()
short PROCESS_MESSAGE()	const FDM_VERSION * fdm_get_version()
short PROCESS_MSG_SENT()	fdm_msg_rec_fifo_empty()
short PROCESS_UPDATE()	fdm_msg_send_fifo_empty()
	fdm_mps_fifo_empty()
short ACK_EVENT()	fdm_smmps_fifo_empty()
short ACK_EOC()	deleted
void IT_HTR_FIP()	deleted
	void fdm_ticks_counter()
	fdm_change_test_medium_ticks()
unsigned char *AE_LE_CREATE()	
unsigned char *AE_LE_DELETE()	FDM_AE_LE_REF *fdm_ae_le_create()
short AE_LE_START()	unsigned short fdm_ae_le_delete()
short AE_LE_STOP()	unsigned short fdm_ae_le_start()
short AE_LE_SWITCH()	unsigned short fdm_ae_le_stop()
short AE_LE_LOAD()	unsigned short fdm_switch_image()
unsigned short	FDM_MPS_VAR_REF *fdm_mps_var_create()
AE_LE_GET_STATE()	int fdm_ae_le_get_state()
Short GET_IMAGE()	int fdm_get_image()
Short AE_LE_DNLOAD()	deleted
Short AE_LE_MODIFICATION()	deleted
	unsigned short fdm_mps_var_change_id()
	unsigned short fdm_mps_var_change_periods()
	unsigned short fdm_mps_var_change_priority()
	unsigned short fdm_mps_var_change_prod_cons()
	unsigned short fdm_mps_var_change_RQa()
	unsigned short fdm_mps_var_change_MSGa()
	unsigned short fdm_mps_var_write_loc()
	unsigned short fdm_mps_var_write_universal()
Short WRITE_LOC()	unsigned short fdm_mps_var_read_loc()
Short WRITE_UNIVERSAL()	unsigned short fdm_mps_var_read_universal()
Short READ_LOC()	
Short READ_UNIVERSAL()	

FIP DEVICE MANAGER R2	FIP DEVICE MANAGER R4
Short GET_DATA()	deleted
	short fdm_mps_var_time_read_loc() ¡
	short fdm_mps_var_time_write_loc() ¡
	fdm_generic_time_initialize()
	fdm_generic_time_read_loc()
	fdm_generic_time_write_loc()
Short CREATE_MACRO_CYCLE()	FDM_BA_REF *fdm_ba_load_macrocycle_fipconfb()
Short DEL_MACRO_CYCLE()	FDM_BA_REF *fdm_ba_load_macrocycle_manual()
Short START_BA()	unsigned short fdm_ba_delete_macrocycle()
Short STOP_BA()	unsigned short fdm_ba_start()
Short FIP_BA_STATUS()	unsigned short fdm_ba_stop()
Short COMMUTE()	unsigned short fdm_ba_status()
void RESYNC()	unsigned short fdm_ba_commute_macrocycle()
short SET_BA_PARAMETERS()	unsigned short fdm_ba_external_resync()
short SET_BA_PRIORITY()	unsigned short fdm_ba_set_parameters()
long SHOW_MACRO_CYCLE()	unsigned short fdm_ba_set_priority()
	deleted
	int fdm_ba_loaded()
short READ_PRESENT()	unsigned short fdm_read_present_list()
short GET_PRESENT()	deleted
short READ_PRESENCE()	unsigned short fdm_read_presence()
short GET_PRESENCE()	deleted
short READ_IDENT()	unsigned short fdm_read_ident()
short GET_IDENT()	deleted
short READ_REPORT()	unsigned short fdm_read_report()
short GET_REPORT()	unsigned short fdm_get_local_report()
	unsigned short fdm_read_ba_synchronize()
short PRINT_IDENT()	deleted
void PRINT_MNGR_COPYRIGHT()	deleted
void PRINT_MNGR_ACK()	deleted
short ACCEPT_SEGMENT()	deleted
short DELETE_SEGMENT()	deleted
short LSAP_MODIFICATION()	deleted
	FDM_MESSAGING_REF*fdm_messaging_fullduplex_create()
	FDM_MESSAGING_REF*fdm_messaging_to_rec_create
	FDM_MESSAGING_REF*fdm_messaging_to_send_create
	unsigned short fdm_messaging_delete
short UNLOCK_MESSAGE()	deleted
short SET_CHANNEL_QUOTA()	deleted
short CHANNEL_MODIFICATION()	unsigned short fdm_channel_create()
	unsigned short fdm_channel_delete()
Short ACCEPT_MESSAGE()	deleted
Short CHANNEL_ALLOC()	deleted
Short CHANNEL_FREE()	deleted
Short SEND_MESSAGE()	unsigned short fdm_send_message()
	unsigned short fdm_msg_ref_buffer_free()
	unsigned short fdm_msg_data_buffer_free()
	unsigned short fdm_change_messaging_acknowledge_type()

Table A.1 – Compatibility between Version 2 and Version 4

List of modifications between Version 2 and Version 4:

- Possibility of managing several FULLFIP2s with the same software instance. The number of FULLFIP2s which can be managed is not limited by the software itself.
- Deletion of useless, obsolete primitives.
- Error processing: every time an error can be detected in the code, give an error code which is clearly identified in the user manual and detail as many different errors as possible.
- Whenever possible and significant, all functions on the user interface produce reports whose formats are identical.
- Deletion of direct dynamic memory allocation: use of a relay which gives the user complete control of memory allocation.
- Modification of FULLFIP2 system access protection management.
- Possibility for user of setting number of `IT_HTR_FIP()` calls to calculate time-outs (instead of systematically using 100).
- Deletion of possibility of configuring packing identifier value to be used: identifier 0x9080 is used systematically for all networks (cf. WorldFIP standard).
- Deletion of obligation to periodically call the `TEST_MEDIUM()` primitive: this will be done by the equivalent of `IT_HTR_FIP()` (which must still be called periodically), the frequency of which will be determined by the user at the time of configuration.
- Addition of a new type of AE/LE: the produced/consumed attribute of the variables they contain can be modified during operation.
- FIFO type interface for aperiodic management.
- The value of a universal consumed variable is supplied directly as a parameter of the user function called when it is received: deletion of primitive `GET_DATA()`.
- Possibility of creating BA program macrocycles in two formats: a FIPCONFB/BA_BUILDER compatible format (also compatible with previous versions of FIP DEVICE MANAGER) or a more user-friendly format for manual building.
- Possibility of setting maximum number of parameters for bus arbitrator start-up and election time-out.
- FIFO type interface for managing reception of values of variables requested.
- The various user functions to be called when the SM-MPS variable values arrive are no longer sent to FIP DEVICE MANAGER each time there is a user request, but once and for all at the time of configuration.
- The values of the variables requested are supplied directly as a parameter of the user function called when they are received: `GET_xxx` primitives deleted.
- Addition of possibility of not deleting the variable created each time the SM-MPS variable is requested. The user is responsible for deletion (saves time in request procedure if not created each time and in read procedure if not deleted each time).
- Use of mechanism and data structure resulting from work on upper layer integration (MCS, SubMMS).
- Switch to a FIFO type interface.

- Simplification of configuration interface of messaging in reception mode (LSAP).
- Coding of messaging addresses (LSAP numbers + segment numbers) systematically as ”long”.
- Increased modularity so that a user can include in his executable file only that which he really needs.
- Optimisation of code whenever possible (particularly by modifying code sequences resulting from successive additions in previous versions). Offers the user the possibility of optimising size or performance levels according to his requirements.
- In order to simplify use and make it more secure, possibility of using, in certain cases, FIP DEVICE MANAGER, a function prototype file (.h) which can be used with C++.
- All conditional compilation switches have been replaced by centralised configuration options in a .h file. They must have a name which is directly connected to their use.
- Supply of a file that contains all the functions to be written by the user with an empty procedure space.
- Adoption of rules for clear, accurate and conventional naming.
- Management of *time variables* (ALSTOM Technology terminology) i.e. variables for which the time spent with the transmitter and the receiver are sent to the consumer(s).
- Use of free access possibilities to FULLFIP2 database: memory shared between FIP DEVICE MANAGER and FIPCODE.
- Possibility of deleting processing linked to BA ”TEST_P” instruction (subscribers present test) since this is done internally by FIPCODE Version 6.
- Queues taken into account by type of event from FIPCODE/FULLFIP2.
- Possibility of exchanging messaging frames with a given field size, with the exception of the address field, which may be equal to 256 bytes (instead of 251 previously).
- Operation at 2.5 Mbits/s and 5 Mbits/s.
- Updating of software included with latest available versions, in particular FIPCODE Version 6 taken into account which provides significant improvements as regards processing performance levels on user interface and line (turnaround time).
- Management of time variable (ID9802H) and time multi-producers.

Appendix B Use with C++

FIP DEVICE MANAGER can be used in a C++ environment. The classes of FDM are the following:

- **Fdm_Initialize**: this class is used for the general initialisation of the FDM timers.
- **Fip_Device_Manager**; this is the root base class; it is used for the instantiation of a WorldFIP network connection.
- **Bus_Arbitrator**: this class is used for the instantiation of Bus Arbitrator objects; this class is derived from the **Fip_Device_Manager** class.
- **GenericTime**: this class is used for the instantiation of the MPS time variable object; this class is derived from the **Fip_Device_Manager** class.
- **MsgDataLinkFullduplex**: this class is used for the instantiation of a full-duplex message context object; this class is derived from the **Fip_Device_Manager** class.
- **MsgDataLinkToSend**: this class is used for the instantiation of a half-duplex transmitting message context object; this class is derived from the **Fip_Device_Manager** class.
- **MsgDataLinkToReceive**: this class is used for the instantiation of a half-duplex receiving message context object; this class is derived from the **Fip_Device_Manager** class.
- **AE_LE**: this class is used for the instantiation of AE_LE objects; this class is derived from the **Fip_Device_Manager** class.
- **MPS_Var_Prod**: this class is used for the instantiation of Producer MPS Variables objects; this class is derived from the **AE_LE** class.
- **MPS_Var_Cons**: this class is used for the instantiation of Consumer MPS Variables objects; this class is derived from the **AE_LE** class.
- **MPS_Var_Sync**: this class is used for the instantiation of Synchronisation MPS Variables objects; this class is derived from the **AE_LE** class.
- **MPS_Var_Cons_and_Prod**: this class is used for the instantiation of Producer/Consumer MPS Variables objects; this class is derived from the class **AE_LE.**
- **MPS_Areceive**: this class is used for the instantiation of periodic MPS Variables with indication objects; this class is derived from the **AE_LE** class.

The *fdm.h* file contains the class definition and implementation for each of the above classes. The body of the member functions contains calls to the C functions described in Chapter 3.

Note

Demos\CC116 sub-directory contains an C++ example.

AE/LE	**A**pplication **E**ntity/**L**ink **E**ntity
BA	**B**us **A**rbitrator (WorldFIP Bus Arbitrator)
DLL	**D**ata **L**ink **L**ayer
FDM	**F**IP **D**EVICE **M**ANAGER
FIPCODE	microcode executed by FULLFIP2
FULLFIP2	WorldFIP communication coprocessor
FIELDUAL	medium redundancy management system
kbits/s:	**k**ilo**b**its per **s**econd
KB	**k**ilo**bytes**
Mbits/s:	**m**ega**bits** per **s**econd
MHz	**m**ega**h**ert**z**
MPS:	**M**anufacturing **P**eriodic/aperiodic **S**ervices
SM_MPS	**S**ystem **M**anagement **M**anufacturing **P**eriodic/aperiodic **S**ervices

Glossary