FDM Solaris Release 2 Software
User Manual

ALS 53324 b-en

First issue: 02-2000
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	**01-2001**	Use of the CC141 board.

Revisions

1. PURPOSE OF MANUAL AND DOCUMENTED VERSION

This manual describes the FIP DEVICE MANAGER Solaris driver. It presents the software as well as its parameters. The system and FDM functions of the programming interface (API) are listed and described.

2. CONTENT OF THIS MANUAL

Chapter 1: **Presentation**: this chapter gives a description of the software and its architecture.

Chapter 2: **Installation**: this chapter includes information about the package files and about the set up. It also introduces the main parameters.

Chapter 3: **Use**: this chapter presents the API FDM compared with the usual FDM. Time management, performance trace files and errors are also evoked.

Chapter 4: **Interface functions**: this chapter presents the programming interface functions. It begins with the system functions and then describes the FDM functions.

3. RELATED PUBLICATIONS

The documents mentioned in this manual are presented between square brackets in the text. They are listed in this section.

ALS 50278 FIP DEVICE MANAGER Software User Reference Manual (R4). **[1]**.

ALS 53316 CC138 - CC139 - CC140 - CC141 Board User Manual. **[2]**.

ALS 50249 FIP Network General Introduction. **[3]**.

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 53324 b-en	*FDM Solaris Release 2 Software User 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

Figures

Tables

Chapter 1 Presentation

1. DESCRIPTION OF THE SOFTWARE

This software package is part of the FIP DEVICE MANAGER (FDM) Release 4 (R4) functions library. It contains a Solaris driver on a Sparc workstation for WorldFIP boards CC138, CC139, CC140 and CC141 on a PCI bus. These boards are PCI boards fitted with the chipset comprising FULLFIP2, FIELDUAL, FIELDRIVE, FIELDTR, etc.

This solution was designed to fulfil a two-fold need:

- the need for the simplified implementation of the FDM R4 library in a Solaris environment, requiring only the use of standard driver handling functions (installation, definition of parameters, start-up, stop),
- the need for a performance level very close to the level that would be obtained with an equivalent processor in the absence of any operating system.

Note

The driver can manage several boards.

The software has certain limitations. such as the quantity of communications objects (150 VCOMs, 48 AELEs, 2 Macrocycles, 40 Messaging contexts, 9 messaging system queues, etc.).

2. DEVICE DRIVER ARCHITECTURE

The software is made up of a driver and a programming interface in C/ANSI which provides access to the driver.

The driver integrates the FDM R4 library which provides access to the WorldFIP network for target hardware based on a FULLFIP2 communication coprocessor. Moreover, it manages the range of FDM services, whether for initialisation, timers or the processing of interrupts.

The programming interface (API) which communicates with the driver via the standard functions provided by Solaris (I/OCTRL, READ/WRITE FILE) provides a means for accessing the driver and, via the driver, the various FDM R4 functions.

Chapter 2 Installation

1. SUPPLY

The following files are supplied:

- **install**: installation of the */usr/fipdrv* directory and the driver on the station.
- **FipDrv-tar.Z**: file containing driver and files required by the driver.

2. INSTALLATION

To install the driver and the user files, you must be a super-user (root). An installation file is supplied with the driver (**install**).

Execute the **install** command. This command executes the following operations:

- Adds the */usr/fipdrv* directory.
- Deletes driver from the driver list (*rem_drv* command).
- Destroys any existing special files.
- Deletes driver files in */kernel/drv*.
- Copies driver fipdrv to */kernel/drv*.
- Adds driver to driver list (*add_drv* command).

You can visualize the anchorage of the driver to the core in a console window.

Any errors are displayed in the console window.

3. CONFIGURATION

To configure peripheral access to the FIP network, position the parameters in file fipdrv*X*.conf located in the */usr/fipdrv/cfg* directory. The letter X indicates the slot number where the board is located (file *fipdrv2.conf* means the board is located in slot 2).

If there is no file, the application cannot access the peripheral.

Parameters not specified in this file will have the following default values:

Identifier	Type	Possible values	Default value
k_phyadr	int	[0..256]	8
segment_number	char	[0..256]	0
model_name	char[20]		CC13x FDM SOLARIS Driver
vendor_name	char[8]		ALSTOM
vendor_field	char[32]		Copyright 1999 ALSTOM (Paris, France)
revision	char[3]		0
time_slot	int		250
mode	enum		WORLD_FIP_1000
time_desc	int	[0,1,2,3]	0
nr_repeat	short		1
def_medium	short		5
timer_cnt_register	short		0
mode_register	short		0
mode_start	int	[0,1]	0

Table 2.1 – Mode_start associated parameters

Vendor Name:

Character string containing the name of the product manufacturer.

Model Name:

Character string containing the name of the equipment designated by the manufacturer.

Revision:

Character string containing the revision index of the manufacturer.

Vendor field:

Character string containing additional information relating to the product manufacturer.

Subscriber number:

Number of the subscriber on the WorldFIP network.

Segment number:

Number of the network segment on which the subscriber is located. This number appears in the messaging system address.

Time slot:

Unit of time considered by FULLFIP2 for time management (arbitrator, refreshing, promptness, time).

Note

This parameter is called 'Tslot' in the FDM software.

Mode:

Combined choice of the speed, frame delimiter, turnaround time and protocol silence time. (This parameter is called *FULLFIP2_Mode* in the FDM software).

For details see [1] Chapter 3, Subsection "fdm_initialize_network()", mode parameter of the FDM_CONFIGURATION_SOFT structure.

Time distribution:

Selecting the *Time produced* option offers the possibility of producing the time (POSIX format, 9802 Identifier, 500 ms refreshing period, Tslot precision). In addition, the mechanism for managing the redundancy of the time producer may be selected using the *Producer election* option.

Mode_start:

If Mode_start is equal to 1, FDM is initialized automatically (without calling the *kfdm_init_network* function) using the parameters provided in the *fipdrvX.cfg* file.

If Mode_start is equal to 0, call the *kfdm_init_network* function to initialize FDM. In this case, the parameters provided in the *fipdrvX.cfg* file are not used.

To interpret the configuration parameters correctly, the *fipdrvX.conf* file must use the following syntax, with one line per identifier:

Identifier[space]=[space]**value**

```
k_phyadr = 5
segment_number = 1
revision = 0
time_slot = 62
time_desc = 0
vendor_name = ALSTOM
....
```

Figure 2.1 - Example of a configuration file

An example of a *fipdrv.cfg* configuration file is provided in the */usr/fipdrv/cfg* directory.

4. START-UP / STOP

The driver is operational as soon as installation is completed.

To stop the driver, use the following command (administrator only): */usr/sbin/rem_drv fipdrv*.

Chapter 3 Use

1. PROGRAMMING INTERFACE

The package is supplied with an API which must be included in the application. This application is compiled using Workshop C 5.0.

To include the API, the files *kfdm.c, evt.c* and *fdm_drv.c* must be included in the project and the following lines must be written in the application files which are using this API: `# include ''kufdm.h''`

An example of an application with a makefile is provided under the */usr/fipdrv/example* directory.

The interface which is then available mainly differs from the basic FDM R4 interface in terms of:

- management by the driver of initialization functions (hardware, software, network, medium) and management of communication (ticks, interrupts, etc.),
- services for the exchange of variables, messages and bus arbitration, limited to standard services,
- time management service, which is limited to broadcast the time of the PC and to manag the redundancy of the time producer,
- additional functions, used to create the contexts required for Solaris applications and to manage error processing.

Comparison table:

The following table presents the differences between the FDM R4 and the API FDM R4 for Solaris.

FIP DEVICE MANAGER R4	API FDM R4 for Solaris
fdm_initialize	*Carried out upon installation of the driver*
fdm_get_version	*Information given in the configuration splash screen*
fdm_ticks_counter	*Run periodically by the driver, every 500 ms*
fdm_change_test_medium_ticks	*No such procedure exists*
fdm_initialize_network	*kfdm_initialze_network*
fdm_stop_network	*kfdm-stop_network*
fdm_valid_medium	*Carried out by the driver*
fdm_online_test	*No such procedure exists*
fdm_process_its_fip	*Management of IRQ and EOC ensured by the driver.*
fdm_process_it-eoc	
fdm_process_it_irq	
	kfdm_create_contexte
fdm_ae_le_create	kfdm_ae_le_create
fdm_ae_le_delete	kfdm_ae_le_delete
fdm_ae_le_start	kfdm_ae_le_start
fdm_ae_le_stop	kfdm_ae_le_stop
fdm_ae_le_get_state	kfdm_ae_le_get_state
fdm_mps_var_create	kfdm_mps_var_create
fdm_mps_var_change_id	
fdm_mps_var_change_period	
fdm_mps_var_change_Rqa	*On-line modification of parameters cannot be used.*
fdm_mps_var_change_MSGa	
fdm_mps_var_change_priority	
fdm_mps_var_change_prod_cons	
fdm_mps_var_write_loc	kfdm_mps_var_write_loc
fdm_mps_var_read_loc	kfdm_mps_var_read_loc
fdm_mps_var_time_write_loc	*No such procedure exists*
fdm_mps_var_time_read_loc	*No such procedure exists*
fdm_mps_var_write_universal	kfdm_mps_var_write_far
fdm_mps_var_read_universal	kfdm_mps_var_read_far
fdm_mps_fifo_empty	*Not used*

Table 3.1 - FDM R4 vs API FDM R4 for Solaris

FIP DEVICE MANAGER R4	API FDM R4 for Solaris
fdm_generic_time_initialize	*kfdm_generic_time_initialize*
fdm_generic_time_read_loc	*No time consumption possible*
fdm_generic_time_write_loc	*Automatic writing by the driver*
fdm_generic_time_set_priority	kfdm_generic_time_set_priority
fdm_generic_time_set_candidate_for_election	kfdm_generic_time_set_candidate_for_election
fdm_generic_time_delete	kfdm_generic_time_delete
	kfdm_generic_time_get_election_status
fdm_channel_create	kfdm_channel_create
fdm_channel_delete	kfdm_channel_delete
fdm_change_channel_nr	*No such procedure exists*
fdm_messaging_fullduplex_create	kfdm_messaging_fullduplex_create
fdm_messaging_to_send_create	kfdm_messaging_to_send_create
fdm_messaging_to_rec_create	kfdm_messaging_to_rec_create
fdm_messaging_delete	kfdm_messaging_delete
fdm_send_message	kfdm_send_message
fdm_msg_ref_buffer_free	*Not used*
fdm_msg_data_buffer_free	*Not used*
fdm_msg_rec_fifo_empty	*Not used*
fdm_msg_send_fifo_empty	*Not used*
	kfdm_receive_message
fdm_ba_load_macrocycle_fipconfb	kfdm_ba_load_macrocycle_fipconfb
fdm_ba_load_macrocycle_manual	kfdm_ba_load_macrocycle_manual
fdm_ba_delete_macrocycle	kfdm_ba_delete_macrocycle
fdm_ba_start	kfdm_ba_start
fdm_ba_external_resync	*No such procedure exists*
fdm_ba_commute_macrocycle	kfdm_ba_commute_macrocycle
fdm_ba_set_priority	kfdm_ba_set_priority
fdm_ba_set_parameters	kfdm_ba_set_parameters
fdm_ba_status	kfdm_ba_status
fdm_ba_stop	kfdm_ba_stop
fdm_ba_loaded	*No such procedure exists*
fdm_read_report	kfdm_read_report
fdm_read_present_list	kfdm_read_present_list
fdm_read_identification	kfdm_read_identification
fdm_read_presence	kfdm_read_presence
fdm_read_ba_synchronize	kfdm_read_ba_synchronize
fdm_get_local_report	*No such procedure exists*
fdm_switch_image	kfdm_switch_image
fdm_get_image	kfdm_get_image
fdm_smmps_fifo_empty	*Not used*

Table 3.1 - FDM R4 vs API FDM R4 for Solaris (cont'd)

FIP DEVICE MANAGER R4	API FDM R4 for Solaris
	kudm_get_subscriber_number kudm_format_message kufdm_delete_evt

Table 3.1 - FDM R4 vs API FDM R4 for Solaris (cont'd)

Note

A driver is an independent program. If, for example, the application program creates an AELE (with *kdfm_ae_le_create ()*) in the driver, the AELE still exists even when the user application is stopped.

If the user program is relaunched, and an AELE is created with the same row, the driver returns a warning.

This also applies for other objects (BA, messaging context).

2. TIME MANAGEMENT

Depending on the settings made, the station may choose to broadcast its time over the network via the ID9802 variable using specific FDM R4 mechanisms. The time is displayed in POSIX format.

The variable produced is updated automatically every 500 ms, with a guaranteed precision of 1s for the recopy operation, the time being broadcast over the network using a level of precision fixed by *Time_slot* (selected during configuration).

Note

Therefore, under general conditions of use at 1 Mbit/s, the difference between the time of the station and the time broadcast will be less than 62.5 s if *Time_slot* has been selected with this value.

The management of time producer election may be activated on initialisation and is used to select a time producer from amongst various devices that are capable of time production on the same network.

Channel 1 must be configured before the election mechanism will work.

3. PERFORMANCE

Performance in terms of device exchange capacity is defined in the following table:

Parameter	Performance
Medium redundancy	TWO_BUS_MODE
Data base	TWO_IMAGE_MODE
Medium test period	500 ms
Value of Time-out	1 s
Value of Time-out message	500 ms
Period of on-line tests	not selected (there are no on-line tests)
BA_DIM dimension	0x2000
Maximum number of VCOMs	150
Maximum number of AELEs	10
Maximum number of macrocycles	2
Maximum number of msg contexts	40
Available messaging channels	0 for aperiodic 1..8 for periodic
access time for writing/reading	80 s + 1 s/byte

Table 3.2 - Parameter performance

4. TRACE FILES

Two trace files are provided to inform users of any problems or events. They are located in the */usr/fipdrv/log* directory:

- Messages concerning driver operation are traced in the fipdrv*slot_cpt.*log file.
- Events concerning WorldFIP management are fed back to the fipdrvEvtLog*slot_cpt*.log file.

Each file can contain up to 1,000 messages events. If there are more than 1,000 messages events, the oldest messages events are overwritten.

Events concerning WorldFIP management are fed back using the *GetEvtLog* function.

5. ERRORS

If errors occur during driver installation, you can display errors involving the driver by booting in verbose mode: *boot -v*.

Errors (code on 32 bits) retrieved by the driver can be accessed using the *GetLastError* command. They are presented in the following table.

Error code	Identifier	Cause
0x00001001	FDM_ERR_CONTEND_ALTERA	The altera component does not allow correct use of FREE_ACCES mode.
0x00001002	FDM_ERR_CONFIG_ALTERA	The configuration procedure for the altera component has been aborted.
0x00001003	ALTERA_ALREADY_CONFIG	The altera component is already initialized.
0x00001004	ALTERA_NOT_CONFIG	The altera component is not initialized.
0x00001005	FDM_ERR_INIT_NETWORK	The network initialization procedure has been aborted.
0x00001006	FDM_ERR_INIT_TIME	The initialization procedure for the MPS variable has been aborted.
0x00001007	NETWORK_ALREADY_INIT	The network is already initialized.
0x00001008	ERR_NETWORK_NOT_INIT	The network is not initialized.
0x00001009	ERR_TIME_NOT_INIT	The MPS variable is not created.
0x0000100A	ERR_TIME_ALREADY_INIT	The MPS variable is already created.
0x0000100B	ERR_SYNC_ALREADY_GO	A synchronization request has already been sent.
0x0000100C	ERR_NOT_SLEEP_CV	No variables are available to send an asynchronous request.

Table 3.3 – Error code and causes

Chapter 4 Interface Functions

1. SYSTEM FUNCTIONS

1.1 ModifyKey()

Corresponds to FDM function:

None.

Prototype:

```
#include            « fdm_drv.h »
       BOOL         ModifyKey(void);
```

Description:

The *modifyKey* function must be called once in each thread (main or secondary) before sending a request or feeding back events concerning WorldFIP management.

Report:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

1.2 GetEvtLog()

Call the *modifyKey* function before calling the *getEvtLog* function.

Corresponds to FDM function:

None.

Prototype:

```
#include            « fdm_drv.h »
       BOOL        GetEvtLog(
       Handle      hdevice);
```

Description:

The *GetEvtLog* function allows you to use an asynchronous request to feed back events or warnings created by the FDM module and copied in file */usr/fipdrv/log/fipdrvevtLog*. The size of this file is limited (see Section 3).

Input parameters:

`hdevice:` obtained when the describer is opened on the special file (open).

Report:

- **TRUE** the messages are fed back correctly.
- **FALSE** the thread allowing event feedback is not created.

Note

If the executable stops, you must close the application properly using the *closeHandle* command.

If the executable freezes, launch the */usr/fipdrv/tools/free_drv* command.

Since the size of the driver message FIFO is not dynamic, you may lose messages if this function is launched too late.

1.3 GetLastError()

Corresponds to FDM function:

None.

Prototype:

```
#include          « fdm_drv.h »
BOOL              GetLastError(VOID);
```

Description:

This function returns the value of the last error code to the calling Thread. The FDM driver, however, updates the error code when a fault occurs.

Description:

```
DWORD  GetLastError(VOID);
```

Input parameters:

None.

Report:

This function returns the error code *dwErrCode*, which is updated during the call for help to the VOID *SetLastError(DWORD dwErrCode)* function.

1.4 kudm_format_message()

Corresponds to FDM function:

None.

Prototype:

```
VOID    kudm_format_message(
        HINSTANCE hlib );
```

Description:

Utility for displaying the last error.

Report:

Not applicable.

1.5 CreateEvent()

Corresponds to FDM function:

None.

Prototype:

```
HANDLE CreateEvent(
       LPSECURITY_ATTRIBUTES     lpEventAttributes,
       BOOL                      bManualReset,
       BOOL                      bInitialState,
       LPCTSTR                   lpName);
```

Description:

The *CreateEvent* function creates a named or unnamed event object.

Input parameters:

`lpEventAttributes:` pointer to a security attribute (LPSECURITY_ATTRIBUTES**)** structure that determines whether the returned handle can be inherited by child processes. If *lpEventAttributes* is NULL, the handle cannot be inherited. In this implementation it should be NULL.

`bManualReset:` flag for manual-reset event. It specifies whether a manual-reset or auto-reset event object is created. If FALSE, the system automatically resets the state to non-signaled after a single waiting thread has been released. In this implementation only the FALSE value is accepted.

`bInitialState:` flag for initial state. It specifies the initial state of the event object. If TRUE, the initial state is signaled. Otherwise, it is non-signaled.

`lpName:` pointer to event-object name, i.e. a null-terminated string specifying the name of the event object. The name is limited to MAX_PATH characters and can contain any character except the backslash path-separator character (\). Name comparison is case sensitive. If *lpName* is NULL, the event object is created without a name.

Return Values:

If the function succeeds, the return value is a handle to the event object. If the named event object existed before the function call, the function returns a handle to the existing object and *GetLastError* returns ERROR_ALREADY_EXISTS. If the function fails, the return value is NULL. To get extended error information, call *GetLastError*.

1.6 SetEvent()

Corresponds to FDM function:

None.

Prototype:

```
BOOL    SetEvent(
        HANDLE      hEvent);
```

Description:

The *SetEvent* function sets the state of the specified event object to signaled.

Input parameters:

`hEvent:` handle to the event object. The CreateEvent function returns this handle.

Return Values:

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call *GetLastError*.

Note

The state of an auto-reset event object remains signaled until a single waiting thread is released. Then the system automatically sets the state to non-signaled. If no threads are waiting, the event object's state remains signaled.

1.7 WaitForSingleObject()

Corresponds to FDM function:

None.

Prototype:

```
DWORD  WaitForSingleObject(
       HANDLE      hHandle,
       DWORD       dwMilliseconds);
```

Description:

The *WaitForSingleObject* function returns when one of the following situations occurs:

- the specified object is in the signaled state.
- the time-out interval elapses.

Input parameters:

`hHandle :` handle to the object.

`dwMilliseconds:` specifies the time-out interval, in milliseconds. The function returns if the interval elapses, even if the object's state is non-signaled. If *dwMilliseconds* is zero, the function tests the object's state and returns immediately. If *dwMilliseconds* is INFINITE, the function's time-out interval never elapses.

Return Values:

If the function succeeds, the return value indicates the event that caused the function to return.

If the function fails, the return value is WAIT_FAILED. To get extended error information, call *GetLastError*.

The return value on success is one of the values indicated in the folowing table:

Value	Meaning
WAIT_FAILED	System error; the function fails.
WAIT_OBJECT_0	The state of the specified object is signaled.
WAIT_TIMEOUT	The time-out interval elapsed, and the object's state is non-signaled.

Table 4.1 – Table of value meaning for the WaitForSingleObject function

1.8 WaitForMultipleObjects()

Corresponds to FDM function:

None.

Prototype:

```
DWORD  WaitForMultipleObjects(
       DWORD            nCount,
       CONST HANDLE     *lpHandles,
       BOOL             bWaitAll,
       DWORD            dwMilliseconds);
```

Description:

The *WaitForMultipleObjects* function returns when one of the following situations occurs:

- any one or all of the specified objects are in the signaled state.
- the time-out interval elapses.

Input parameters:

`nCount:`	specifies the number of event handles in the array pointed to by *lpHandles*.
`lpHandles:`	pointer to an array of event handles. The array can contain handles of events.
`bWaitAll:`	wait flag. This parameter is not used.
`dwMilliseconds:`	specifies the time-out interval, in milliseconds. The function returns if the interval elapses, even if the conditions specified by the *bWaitAll* parameter are not met. If *dwMilliseconds* is zero, the function tests the states of the specified objects and returns immediately. If *dwMilliseconds* is INFINITE, the function's time-out interval never elapses.

Return Values:

If the function succeeds, the return value indicates the event that caused the function to return.

If the function fails, the return value is WAIT_FAILED. To get extended error information, call *GetLastError*.

If the function succeeds, the return value is one of the values indicated in the following table:

Value	Meaning
WAIT_OBJECT_0 to (WAIT_OBJECT_0 + *nCount* – 1)	• If *bWaitAll* is TRUE, the return value indicates that the state of all specified objects is signaled. • If *bWaitAll* is FALSE, the return value minus WAIT_OBJECT_0 indicates the *lpHandles* array index of the object that satisfied the wait. If more than one object was signalled during the call, this is the array index of the signalled object with the smallest index value of all the signalled objects.
WAIT_ABANDONED_0 to (WAIT_ABANDONED_0 + *nCount* – 1)	• If *bWaitAll* is TRUE, the return value indicates that the state of all specified objects is signaled and at least one of the objects is an abandoned mutex object. • If *bWaitAll* is FALSE, the return value minus WAIT_ABANDONED_0 indicates the *lpHandles* array index of an abandoned mutex object that satisfied the wait.
WAIT_TIMEOUT	The time-out interval elapsed and the conditions specified by the *bWaitAll* parameter are not satisfied.

Table 4.2 - Table of value meaning for the WaitForMulitipleObjects function

1.9 SleepEx()

Corresponds to FDM function:

None.

Prototype:

```
DWORD  SleepEx(
       DWORD       dwMilliseconds,
       BOOL        bAlertable);
```

Description:

The *SleepEx* function causes the current thread to enter a wait state until one of the following situations occurs:

- an I/O completion *CallBack* function is called.
- an asynchronous procedure call (APC) is queued to the thread.

Input parameters:

`dwMilliseconds:` time-out interval in milliseconds. This parameter is not used.

`bAlertable:` early completion flag. This parameter is not used.

Return Values:

`WAIT_ERROR:` System error

`WAIT_IO_COMPLETION:` The function returned due to one or more I/O completion *CallBack* functions.

2. FIP DEVICE MANAGER FUNCTIONS

2.1 kfdm_create_contexte()

Corresponds to FDM function:

None.

Prototype:

```
HANDLE kfdm_create_contexte(int slot);
```

Description:

Used to create a logical link between the user "world" and the kernel "world" of the FDM driver.

Input parameters:

`int slot:` this parameter is the slot where the board is located.

Note

The slot can be found with the */usr/fipdrv/tools/give_slot* command.

Report:

HANDLE reference on the open device.

If INVALID_HANDLE_VALUE is displayed, the opening has not been performed correctly.

If it is not produced, the reference of the open device appears.

2.2 kfdm_initialize_network()

Corresponds to FDM function:

`fdm_initialize_network.`

Prototype:

```
#include  « kufdm.h »
       BOOL                      kfdm_initialize_network(
       HANDLE                    hdevice,
       FDM_CONFIGURATION_SOFT    *user_soft_definition,
       FDM_CONFIGURATION_HARD    *user_hard_definition,
       FDM_IDENTIFICATION        *user_ident_param);
```

Description:

The *kfdm_initialize_network* function allows you to start and configure FDM/FIPCODE/FULLFIP2.

Input parameters:

`hdevice:` handle reference on open device.

`user_soft_definition:` pointer on FDM_CONFIGURATION_SOFT data structure. Only the following fields are acknowledged:

- Mode,
- Tslot,
- Nr_Of_Repeat,
- user_responsibility.

`user_hard_definition:` pointer on FDM_CONFIGURATION_HARD data structure. Only the following fields are acknowledged:

- K_PHYADR,
- MySegment.

`user_ident_param:` pointer on FDM_IDENTIFICATION data structure. Only the following fields are acknowledged:

- vendor_name,
- model_name,
- revision,
- vendor_field.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the primitive was correctly processed.
- **FALSE** if the primitive was not correctly processed.

Note

If the network has already been initialized, calling the function again will have no effect on the network and will produce an error.

2.3 kufdm_delete_evt

Corresponds to FDM function:

None.

Prototype:

```
#include  « kufdm.h »
       BOOL        kufdm_delete_evt(
       HANDLE      hdevice);
```

Description:

Used to flush the waiting requests. This function have to be called before the *kfdm_stop_network* function.

Input parameters:

`hdevice:` HANDLE reference on the open device.

Report:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.4 kfdm_stop_network()

Corresponds to FDM function:

`fdm_stop_network.`

Prototype:

```
#include  « kufdm.h »
       BOOL        kfdm_stop_network(
       HANDLE      hdevice);
```

Description:

The *kfdm_stop_network* function stops the network.

Input parameters:

`hdevice:` HANDLE reference on the open device.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the primitive was correctly processed.
- **FALSE** if the primitive was not correctly processed.

Note

This function stops the network. You cannot stop an uninitialized network.

The time variable is deleted when this function is called.

2.5 kfdm_ae_le_create()

Corresponds to FDM function:

`fdm_ae_le_create().`

Prototype:

```
BOOL   kfdm_ae_le_create(
       HANDLE      hdevice,
       int         AE_LE_RANG,
       int         AE_LE_DIM);
```

Description:

Creation of an AE_LE.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`AE_LE_RANG:` AE_LE row. The value range is [0..9].

`AE_LE_DIM:` maximum number of variables which may make up this AE_LE.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.6 kfdm_mps_var_create()

Corresponds to FDM function:

`fdm_mps_var_create().`

Prototype:

```
BOOL                kfdm_mps_var_create(
       HANDLE       hdevice,
       int          AE_LE_RANG,
       FDM_XAE *    Var_Param);
```

Description:

Defines the parameters of a variable in an AE_LE.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`AE_LE_RANG:` AE_LE row. The value range is [0..9].

`Var_Param:` pointer to a structure describing the parameters of the variable.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.7 kfdm_ae_le_start()

Corresponds to FDM function:

`fdm_ae_le_start().`

Prototype:

```
BOOL        kfdm_ae_le_start(
HANDLE      hdevice,
int         AE_LE_RANG);
```

Description:

Start-up of an AE_LE.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`AE_LE_RANG:` AE_LE row. The value range is [0..9].

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.

2.8 kfdm_ae_le_stop()

Corresponds to FDM function:

`fdm_ae_le_stop().`

Prototype:

```
BOOL   kfdm_ae_le_stop(
       HANDLE      hdevice,
       int         AE_LE_RANG);
```

Description:

Stopping an AE_LE.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`AE_LE_RANG:` AE_LE row. The value range is [0..9].

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.9 kfdm_ae_le_delete()

Corresponds to FDM function:

`fdm_ae_le_delete().`

Prototype:

```
BOOL   kfdm_ae_le_delete(
       HANDLE      hdevice,
       int         AE_LE_RANG);
```

Description:

Deletion/destruction of an AE_LE.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`AE_LE_RANG:` AE_LE row. The value range is [0..9].

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.10 kfdm_mps_var_write_loc()

Corresponds to FDM function:

`fdm_ae_le_delete().`

Prototype:

```
BOOL    kfdm_mps_var_write_loc(
        HANDLE                  hdevice,
        int                     AE_LE_RANG,
        int                     VAR_RANG,
        int                     Lg,
        USER_BUFFER_TO_READ     Data_Buffer);
```

Description:

Writing of an MPS variable.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`AE_LE_RANG:` AE_LE row. The value range is [0..9].

`VAR_RANG:` row of the variable in the AE_LE.

`Lg:` number of variable bytes.

`Data_Buffer:` pointer to a USER_BUFFER_TO_READ structure containing the data to be written. Refer to **[1]** for the description of this structure.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.11 kfdm_mps_var_read_loc()

Corresponds to FDM function:

`fdm_ae_le_delete().`

Prototype:

```
BOOL    kfdm_mps_var_read_loc(
        HANDLE      hdevice,
        int         AE_LE_RANG,
        int         VAR_RANG,
        K_VAR_DATA  *Data_Buffer);
```

Description:

Reads a variable.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`AE_LE_RANG:` AE_LE row. The value range is [0..9].

`VAR_RANG:` row of the variable in the AE_LE.

`Lg:` number of variable bytes.

`Data_Buffer:` pointer to a K_VAR_DATA buffer described below which will contain the value of the variable read.

```
typedef struct {
            int   Last_Error;
            FDM_MPS_READ_STATUS Status;
            FDM_MPS_VAR_DATA Datas;
} K_VAR_DATA;
```

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.12 kfdm_mps_var_synchronize()

Corresponds to FDM function:

None.

Prototype:

```
BOOL    kfdm_mps_var_synchronize(
        HANDLE                                  hdevice,
        LPKFDM_ABSTRACT_SYNCHRO_TYPE            InternalInfos,
        LPOVERLAPPED_COMPLETION_ROUTINE         CallBackRoutine);
```

Description:

Enables synchronisation of the application on receipt of the Sync variable declared in one of the AE_LEs which have been started up.

Note

This procedure must be MONO THREAD. It must be followed by SleepEx.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`InternalInfos:` internal LPKFDM_ABSTRACT_SYNCHRO_TYPE structure. The structure must not be modified during execution of the function.

`CallBackRoutine:` procedure called on receipt of the sync. The prototype of the *CallBackRoutine* function is as follows:

```
VOID WINAPI CallBackRoutine (
        DWORD               dwErrorCode,
        DWORD               Compte_rendu,
        LPOVERLAPPED        lpOverlapped);
```

`dwErrorCode:` equal to 0 if the request has been processed correctly.

`Compte_rendu:` equal to 2 if Sync correct and equal to 3 if Time-out or invalid.

`LpOverlapped:` pointer on the InternalInfos structure provided during the call. *InternalInfos.gOverlapped.IntervalHigh* is equal to 2 if sync event is received and equal to 3 if sync event is not received (the time_out is 5 seconds).

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

Example of usage:

```
VOID WINAPI         IOCompletionRoutine1(
       DWORD             dwErrorCode,
       DWORD             Compte_rendu,
       LPOVERLAPPED      lpOverlapped;

{

if (Compte_rendu == 2) SetEvent(Hevent);

}

main() {

for ( ; ;)

{

KFDM_ABSTRACT_SYNCHRO_TYPE Tmp;

DWORD Dw;
kfdm_mps_var_synchronize(

                      hdevice,

                      &Tmp,

                      IOCompletionRoutine);

Dw = SleepEx (6000, THRUE);

if (Dw == WAIT_IO_COMPLETION) {------➜ ok} else {------➜Synchro Id not
received.}

}
```

2.13 kfdm_mps_var_read_far()

Corresponds to FDM function:

`fdm_mps_var_read_universal().`

Prototype:

```
BOOL    kfdm_mps_var_read_far(
        HANDLE                              hdevice,
        int                                 AE_LE_RANG,
        int                                 VAR_RANG,
        K_FDM_MPS_VAR_DATA_FAR              *Data_Buffer,
        LPOVERLAPPED                        InternalInfos,
        LPOVERLAPPED_COMPLETION_ROUTINE     CallBackRoutine);
```

Description:

Enables remote reading of an MPS variable.

Note

It must be followed by *SleepEx*.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`AE_LE_RANG:` AE_LE row. The value range is [0..9].

`VAR_RANG:` row of the variable in the AE_LE.

`Data_Buffer:` pointer to a K_FDM_MPS_VAR_DATA_FAR data structure described below:

```
typedef struct {
        int                     Last_Error;
        FDM_MPS_READ_STATUS     Status;
        FDM_MPS_VAR_DATA        Datas;
} K_FDM_MPS_VAR_DATA_FAR;
```

`InternalInfos:` internal structure. The structure must not be modified during the execution of the function.

`CallBackRoutine:` procedure called on receipt of the variable or when time delay expires. The prototype of the *CallBack* function is as follows:

```
VOID WINAPI CallBackRoutine (
      DWORD             dwErrorCode,
      DWORD             FdmReport,
      LPOVERLAPPED      lpOverlapped);
```

`dwErrorCode:` equal to 0 if the request has been processed correctly.

`FdmReport:` equal to 0 if request is processed by OK FDM.

`lpOverlapped:` pointer to an OVERLAPPED structure. The OVERLAPPED structure contains information used in asynchronous input and output (I/O).

```
typedef struct _OVERLAPPED { // o
      DWORD       Internal;
      DWORD       InternalHigh;
      DWORD       Offset;
      DWORD       OffsetHigh;
      HANDLE      hEvent;
} OVERLAPPED;
```

`hEvent:` contains `( AE_LE_RANG << 16) | VAR_RANG`

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.14 kfdm_mps_var_write_far()

Corresponds to FDM function:

`fdm_mps_var_write_universal().`

Prototype:

```
BOOL    kfdm_mps_var_write_far(
        HANDLE                              hdevice,
        int                                 AE_LE_RANG,
        int                                 VAR_RANG,
        int                                 DataLength,
        USER_BUFFER_TO_READ                 DataBuffer,
        LPOVERLAPPED                        InternalInfos,
        LPOVERLAPPED_COMPLETION_ROUTINE     CallBackRoutine);
```

Description:

Enables remote writing of an MPS variable.

Note

It must be followed by *SleepEx*.

Input parameters:

`hdevice:`	HANDLE reference on the open device.
`AE_LE_RANG:`	AE_LE row. The value range is [0..9].
`VAR_RANG:`	row of the variable in the AE_LE.
`DataLength:`	number of bytes to be transferred.
`DataBuffer:`	pointer to a USER_BUFFER_TO_READ data structure of the data to be written. Refer to **[1]** for the Description of this structure.
`InternalInfos:`	internal structure. The structure must not be modified during the execution of the function.
`CallBackRoutine:`	procedure called on receipt of the variable or when time delay expires. The Prototype of the *CallBack* function is the same as for the *kfdm_mps_var_read_far()* function.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.15 kfdm_ba_load_macrocycle_manual()

Corresponds to FDM function:

`fdm_ba_load_macrocycle_manual().`

Prototype:

```
BOOL   kfdm_ba_load_macrocycle_manual(
       HANDLE      hdevice,
       int         BA_RANG,
       int         Nb_of_Liste,
       int         Nb_of_Instruction,
       unsigned    short Label,
       const       PTR_LISTS*,
       const       PTR_INSTRUCTIONS *);
```

The macro below enables a simplified call to be made:

```
BOOL Bus_Arbitrator__CREATE(
       hdevice,
       BA_RANG,
       Label,
       Listes,
       Programme);
```

Description:

Used to create and load a macrocycle.

Input parameters:

`hdevice:`	HANDLE reference on the open device.
`BA_RANG:`	row of the macrocycle [0..1].
`Nb_of_Liste:`	Refer to **[1]**.
`Nb_of_Instruction:`	Refer to **[1]**.
`Label:`	Refer to **[1]**.
`PTR_LISTS*:`	Refer to **[1]**.
`PTR_INSTRUCTIONS *:`	Refer to **[1]**.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

Example of usage:

```
const LIST_ELEMENT L0[] = {
                        0xE100, ID_DAT};
const LIST_ELEMENT L1[] = {
                        0x0001, ID_DAT,
                        0x0002, ID_DAT,
                        0x0002, ID_DAT,
                        0x0003, ID_DAT,
                        0x0016, ID_DAT,
                        0x0017, ID_DAT,
                        0x0018, ID_DAT,
                        0x0019, ID_DAT,
                        0x0021, ID_DAT,
                        0x0022, ID_DAT,
                        0x0023, ID_DAT,
                        0x0024, ID_DAT,
                        0x0025, ID_DAT};
PTR_LISTS Listes_BA[] = {
(Ushort)sizeof( L0 ) ,(LIST_ELEMENT*) L0,
(Ushort)sizeof( L1 ) ,(LIST_ELEMENT*) L1,};
      #define milliseconds *1000
const PTR_INSTRUCTIONS Prg_BA[] = {
```

```
                        SEND_LIST,        0,

                        BA_WAIT,          5000,

                        SEND_LIST,        1,

                        TEST_P,           0,

                        TEST_P,           0,

                        SEND_APER,        200 milliseconds,

                        SEND_MSG,         200 milliseconds,

                        BA_WAIT,          200 milliseconds,

                        NEXT_MACRO, 0};
B = Bus_Arbitrator__CREATE(
       hdevice ,
       0,                row [0..1].
       0x300,            Label.
       Listes_BA,        Name of the table which defines the lists.
       Prg_BA            Name of the table which defines the macrocycle);
```

2.16 kfdm_ba_load_macrocycle_fipconfb()

Corresponds to FDM function:

`fdm_ba_load_macrocycle_fipconfb().`

Prototype:

```
BOOL   kfdm_ba_load_macrocycle_fipconfb(
       HANDLE                    hdevice,
       int                       BA_RANG,
       const unsigned short      *ba_fipconfb);
```

Description:

Used to create and load a macrocycle.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`BA_RANG:` row of the macrocycle [0..1].

`ba_fipconfb:` pointer to a table generated in the FIPCONFB tool format. Refer to **[1]**.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.17 kfdm_ba_delete_macrocycle()

Corresponds to FDM function:

`fdm_ba_delete_macrocycle().`

Prototype:

```
BOOL   kfdm_ba_delete_macrocycle(
       HANDLE      hdevice,
       int         BA_RANG);
```

Description:

Used to delete a macrocycle.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`BA_RANG:` row of the macrocycle [0..1].

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.18 kfdm_ba_start()

Corresponds to FDM function:

`fdm_ba_start().`

Prototype:

```
BOOL    kfdm_ba_start(
        HANDLE      hdevice,
        int         BA_RANG);
```

Description:

Used to start up a macrocycle.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`BA_RANG:` row of the macrocycle [0..1].

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.19 kfdm_ba_stop()

Corresponds to FDM function:

`fdm_ba_stop().`

Prototype:

```
BOOL    kfdm_ba_stop(
        HANDLE      hdevice);
```

Description:

Used to stop the current macrocycle.

Input parameters:

`hdevice:` HANDLE reference on the open device.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.20 kfdm_ba_status()

Corresponds to FDM function:

`fdm_ba_status().`

Prototype:

```
BOOL    kfdm_ba_status(
        HANDLE           hdevice,
        K_BA_INF_STATUS  Resultats);
```

Description:

Used to read the status of the station BA function.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`Resultats:` pointer to a K_BA_INF_STATUS structure described below and containing the status of the BA.

```
typedef struct {

        int              Last_Error;

        BA_INF_STATUS    BA_Infos;

} K_BA_INF_STATUS
```

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.21 kfdm_ba_commute_macrocycle()

Corresponds to FDM function:

`fdm_ba_commute_macrocycle().`

Prototype:

```
BOOL        kfdm_ba_commute_macrocycle(
HANDLE      hdevice,
int BA_     RANG);
```

Description:

Used to switch a macrocycle.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`BA_RANG:` row of the macrocycle [0..1].

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.22 kfdm_ba_set_priority()

Corresponds to FDM function:

`fdm_ba_set_priority().`

Prototype:

```
BOOL    kfdm_ba_set_priority(
        HANDLE          hdevice ,
        unsigned char   Priority_Level);
```

Description:

Used to modify the priority of the local BA function.

Input parameters:

`hdevice:` HANDLE reference on the open devices.

`Priority_Level:` desired level of priority [0..15], with 0 being the highest priority.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.23 kfdm_ba_set_parameters()

Corresponds to FDM function:

`fdm_ba_set_parameters().`

Prototype:

```
BOOL    kfdm_ba_set_parameters(
        HANDLE                  hdevice,
        enum _BA_SET_MODE       BA_Mode,
        unsigned char           MAX_Subscriber,
        unsigned char           MAX_Priority);
```

Description:

Used to modify the parameters used in the time delay calculations of the station BA function.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`BA_Mode:` Refer to **[1]**.

`MAX_Subscriber:` Refer to **[1]**.

`MAX_Priority:` Refer to **[1]**.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.24 kfdm_read_presence()

Corresponds to FDM function:

`fdm_read_presence().`

Prototype:

```
BOOL    kfdm_read_presence(
        HANDLE                              hdevice,
        int                                 subscriber,
        K_FDM_PRESENCE_VAR                  *Data_Buffer,
        LPOVERLAPPED                        InternalInfos,
        LPOVERLAPPED_COMPLETION_ROUTINE     CallBackRoutine);
```

Description:

Used to read a "subscriber present" variable.

Note

It must be followed by *SleepEx*.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`subscriber:` physical address of the subscriber whose presence variable is to be read.

`Data_Buffer:` pointer to a K_FDM_PRESENCE_VAR data structure described below:

```
typedef struct {
        int                     Last_Error;
        FDM_PRESENCE_VAR        Result
} K_FDM_PRESENCE_VAR;
```

`InternalInfos:` refer to *fdm_mps_var_read_far()*.

`CallBackRoutine:` refer to *fdm_mps_var_read_far()*.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.25 kfdm_read_present_list()

Corresponds to FDM function:

`fdm_read_present_list()`.

Prototype:

```
BOOL    kfdm_read_present_list(
        HANDLE                              hdevice,
        K_FDM_PRESENT_LIST                  *Data_Buffer,
        LPOVERLAPPED                        InternalInfos,
        LPOVERLAPPED_COMPLETION_ROUTINE     CallBackRoutine);
```

Description:

Used to read a "subscriber present list" variable.

Note

It must be followed by *SleepEx*.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`Data_Buffer:` pointer to a K_FDM_PRESENT_LIST data structure described below containing the value read:

```
typedef struct {

      int               Last_Error;

      FDM_PRESENT_LIST  Result

} K_FDM_PRESENT_LIST;
```

`InternalInfos:` refer to *fdm_mps_var_read_far()*.

`CallBackRoutine:` refer to *fdm_mps_var_read_far()*.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.26 kfdm_read_report()

Corresponds to FDM function:

`fdm_read_Report().`

Prototype:

```
BOOL    kfdm_read_Report(
        HANDLE                                  hdevice,
        int                                     subscriber,
        K_FDM_REPORT_VAR                        *Data_Buffer,
        LPOVERLAPPED                            InternalInfos,
        LPOVERLAPPED_COMPLETION_ROUTINE         CallBackRoutine);
```

Description:

Used to read a subscriber Report variable.

Note

It must be followed by *SleepEx*.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`subscriber:` physical address of the subscriber whose Report variable is to be read.

`Data_Buffer:` pointer to a K_FDM_REPORT_VAR data structure described below containing the value read:

```
typedef struct {
        int             Last_Error;
        FDM_REPORT_VAR  Result
} K_FDM_REPORT_VAR;
```

`InternalInfos:` refer to *fdm_mps_var_read_far()*.

`CallBackRoutine:` refer to *fdm_mps_var_read_far()*.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.27 kfdm_read_identification()

Corresponds to FDM function:

`fdm_read_identification().`

Prototype:

```
BOOL    kfdm_read_identification(
        HANDLE                              hdevice,
        int                                 subscriber,
        K_FDM_IDENT_VAR                     *Data_Buffer,
        LPOVERLAPPED                        InternalInfos,
        LPOVERLAPPED_COMPLETION_ROUTINE     CallBackRoutine);
```

Description:

Used to read the subscriber identification variable.

Note

It must be followed by *SleepEx*.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`subscriber:` physical address of the subscriber whose identification variable is to be read.

`Data_Buffer:` pointer to a K_FDM_IDENT_VAR data structure described below containing the value read:

```
typedef struct {
        int             Last_Error;
        FDM_IDENT_VAR   Result
} K_FDM_IDENT_VAR;
```

`InternalInfos:` refer to *fdm_mps_var_read_far()*.

`CallBackRoutine:` refer to *fdm_mps_var_read_far()*.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

Note

The station identification variable (in comparison with the one used in FDM R4) is:

`Vendor_Name`	Field entered during configuration.
`Model_Name`	Field entered during configuration.
`Revision`	Field entered during configuration.
`Tag_Name`	No such field.
`SM_MPS_Conform`	0x10.
`SMS_Conform`	No such field.
`PMDP_Conform`	No such field.
`Vendor_Field`	Field entered during configuration.

2.28 kfdm_read_ba_synchronize()

Corresponds to FDM function:

`fdm_read_ba_synchronize().`

Prototype:

```
BOOL    kfdm_read_ba_synchronize(
        HANDLE                              hdevice,
        K_FDM_SYNCHRO_BA_VAR                *Data_Buffer,
        LPOVERLAPPED                        InternalInfos,
        LPOVERLAPPED_COMPLETION_ROUTINE     CallBackRoutine);
```

Description:

Used to read the network BA sync variable.

Note

It must be followed by *SleepEx*.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`Data_Buffer:` pointer to a K_FDM_SYNCHRO_BA_VAR data structure described below containing the value read:

```
typedef struct {

      int                     Last_Error;

      FDM_SYNCHRO_BA_VAR      Result

} K_FDM_SYNCHRO_BA_VAR;
```

`InternalInfos:` refer to *fdm_mps_var_read_far()*.

`CallBackRoutine:` refer to *fdm_mps_var_read_far()*.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.29 kfdm_messaging_fullduplex_create()

Corresponds to FDM function:

`fdm_messaging_fullduplex_create().`

Prototype:

```
BOOL    kfdm_messaging_fullduplex_create(
        HANDLE                      hdevice,
        int                         Rang,
        FDM_MESSAGING_FULLDUPLEX    *Data_Buffer);
```

Description:

Used to create a new FULLDUPLEX messaging system context.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`Rang:` context number for the driver.

`Data_Buffer:` pointer to an FDM_MESSAGING_FULLDUPLEX data structure describing the context. Refer to **[1]** for the description of this structure. The scored out fields of the structure below are not to be initialised.

```
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;
```

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.30 kfdm_messaging_to_send_create()

Corresponds to FDM function:

`fdm_messaging_to_send_create().`

Prototype:

```
BOOL    kfdm_messaging_to_send_create(
        HANDLE                  hdevice,
        int                     Rang,
        FDM_MESSAGING_TO_SEND   *Data_Buffer);
```

Description:

Used to create a messaging context for sending messages.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`Rang:` context number for the driver.

`Data_Buffer:` pointer to an FDM_MESSAGING_TO_SEND data structure describing the context. Refer to **[1]** for description of this structure. The scored out fields of the structure below are not to be initialised.

```
typedef struct {
        enum _FDM_MSG_IMAGE Position;
        struct{
            ~~void (*User_Msg_Ack)~~
            ~~(FDM_MESSAGING_REF* _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;
```

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.31 kfdm_messaging_to_rec_create()

Corresponds to FDM function:

`fdm_messaging_to_rec_create().`

Prototype:

```
BOOL    kfdm_messaging_to_rec_create(
        HANDLE                  hdevice,
        int                     Rang,
        FDM_MESSAGING_TO_REC    *Data_Buffer);
```

Description:

Used to create a messaging context for receiving messages.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`Rang:` context number for the driver.

`Data_Buffer:` pointer to an FDM_MESSAGING_TO_REC data structure describing the context. Refer to **[1]** for description of this structure. The scored out fields of the structure below are not to be initialised.

```
typedef struct {
     enum _FDM_MSG_IMAGE Position;
     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_TO_REC;
```

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.32 kfdm_messaging_delete()

Corresponds to FDM function:

`fdm_messaging_delete().`

Prototype:

```
BOOL    kfdm_messaging_delete(
        HANDLE      hdevice,
        int         Rang);
```

Description:

Used to delete a messaging context.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`Rang:` number of the context to be deleted.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.33 kfdm_channel_create()

Corresponds to FDM function:

`fdm_channel_create().`

Prototype:

```
BOOL    kfdm_channel_create(
        HANDLE                      hdevice,
        FDM_CHANNEL_PARAM *         Data_Buffer);
```

Description:

Used to create a messaging channel.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`Data_Buffer:` pointer to an FDM_CHANNEL_PARAM structure describing the context. Refer to **[1]** for the description of this structure.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.34 kfdm_channel_delete()

Corresponds to FDM function:

`fdm_channel_delete().`

Prototype:

```
BOOL    kfdm_channel_delete(
        HANDLE      hdevice,
        int         Channel_nr);
```

Description:

Used to delete a messaging channel.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`Channel_nr:` number of the channel to be deleted.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.35 kfdm_send_message()

Corresponds to FDM function:

`fdm_send_message()`.

Prototype:

```
BOOL    kfdm_send_message(
        HANDLE                              hdevice,
        int                                 RANG,
        K_FDM_MSG_TO_SEND *                 Data_Buffer,
        LPOVERLAPPED                        InternalInfos,
        LPOVERLAPPED_COMPLETION_ROUTINE     CallBackRoutine);
```

Description:

Used to request the sending of a message.

Note

It must be followed by *SleepEx*.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`RANG:` row of the messaging context to be used.

`Data_Buffer:` pointer to a K_FDM_MSG_TO_SEND data structure described below. The scored out fields are not to be initialised.

```
typedef struct {
            FDM_MSG_TO_SEND         MsgToSend;
            void                    *Irp;
            FDM_MSG_T_DESC          Block;
            unsigned char           Message[256];
} K_FDM_MSG_TO_SEND;

typedef struct _FDM_MSG_TO_SEND {
            struct_FDM_MSG_TO_SEND  *Next;
            struct_FDM_MSG_TO_SEND  *Prev;
```

```
                int                     Nr_Of_Blocks;
                FDM_MSG_T_DESC          *Ptr_Block;
                unsigned long           Local_DLL_Address;
                unsigned long           Remote_DLL_Address;
                FDM_MSG_SEND_SRERVICE_REPORT Service_Report;
                FDM_PRIVATE_DLI_T       Private;
        } FDM_MSG_TO_SEND;
        typedef struct _FDM_MSG_T_DESC {
                unsigned short          Nr_Of_Bytes;
                unsigned char           *Ptr_Data;
                struct_FDM_MSG_T_DESC   *Next_Block;
        } FDM_MSG_T_DESC;
```

`InternalInfos:` internal structure. The structure must not be modified during execution of the function.

`CallBackRoutine:` procedure called on receipt of the message send acknowledgement or when time delay expires. The prototype of the CallBack function is as follows:

```
VOID WINAPI CallBackRoutine (
     DWORD              dwErrorCode,
     DWORD              FdmReport,
     LPOVERLAPPED       lpOverlapped);
```

`dwErrorCode:` equal to 0 if the request has been processed correctly.

`FdmReport:` not used.

`lpOverlapped:` pointer to an OVERLAPPED structure. The OVERLAPPED structure contains information used in asynchronous input and output (I/O).

```
typedef struct _OVERLAPPED {
     DWORD       Internal;
```

```
                    DWORD       InternalHigh;
                    DWORD       Offset;
                    DWORD       OffsetHigh;
                    HANDLE      hEvent;
                } OVERLAPPED;
```

`hEvent:` contains user information which may be defined during the request to send.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

There are also a differed report update by FDM after sending the message. You have to read the following field:

`InternalInfos.InternalHigh:` is the FDM Report of type:

```
struct {
            enum_FDM_MSG_SND_CNF        //: bits [16..23]
            enum_FDM_MSG_USER_ERROR     //: bits [8..15]
            enum_FIP_MSG_SND_REP        //: bits [0..7]
};
```

This different report can be expored after the succesfull completition of the *SleepEx* function or in the *CallBack* Routine.

2.36 kfdm_received_message()

Corresponds to FDM function:

None.

Prototype:

```
BOOL    kfdm_received_message(
        HANDLE                              hdevice,
        int                                 RANG,
        K_FDM_MSG_RECEIVED                  * Data_Buffer,
        LPOVERLAPPED                        InternalInfos,
        LPOVERLAPPED_COMPLETION_ROUTINE     CallBackRoutine);
```

Description:

Used to signal the reception of a message.

Note

It must be followed by *SleepEx*.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`RANG:` row of the messaging context to be used.

`Data_Buffer:` pointer to a K_FDM_MSG_RECEIVED data structure described below:

```
typedef struct {
    int                     Last_Error;
    int                     Nbr_Of_Purged_Message;
    FDM_MSG_RECEIVED        ref_buffer;
    FDM_MSG_R_DESC          data_buffer;
} K_FDM_MSG_RECEIVED;
```

`InternalInfos:` internal structure. The structure must not be modified during execution of the function.

`CallBackRoutine:` procedure called on receipt of the message send acknowledgement or when the time delay expires. The prototype of the *CallBack* function is as follows:

```
VOID WINAPI CallBackRoutine (
    DWORD               dwErrorCode,
    DWORD               FdmReport,
    LPOVERLAPPED        lpOverlapped);
```

`dwErrorCode:` equal to 0 if the request has been processed correctly.

`FdmReport:` not used.

`lpOverlapped:` not used.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

Usage example:

```
B = kfdm_received_message(
       hdevice,
       1,              i.e. row[0..29]
       &MsgR,
       &gOverLapped,
       IOCompletionRoutine);
```

if (B == FALSE) kufdm_format_message(hLib);

Dw = SleepEx(6000, TRUE);

This function indicates the time-out interval in milliseconds. When TRUE appears, it returns to execute I/O completion routine.

Here, the message is either read or there is a time-out. In the last case, a new read request must be made.

2.37 kfdm_switch_image()

Corresponds to FDM function:

`fdm_switch_image().`

Prototype:

```
BOOL    kfdm_switch_image(
        HANDLE           hdevice,
        K_FDM_IMAGE_NR   *Value);
```

Description:

Used to request the switching of the image the AE_LEs are operating with.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`Value:` pointer to a K_FDM_IMAGE_NR data structure described below:

```
        typedef struct {
                    int                     Last_Error;
                    enum IMAGE_NR           Image_Value;
        } K_FDM_IMAGE_NR;
```

`Image_Value:` IMAGE_1 or IMAGE_2.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.38 kfdm_get_image()

Corresponds to FDM function:

`fdm_get_image().`

Prototype:

```
BOOL    kfdm_get_image(
        HANDLE           hdevice ,
        K_FDM_IMAGE_NR   *Value);
```

Description:

Used to find out which image the AE_LEs are operating with.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`Value:` pointer to a K_FDM_IMAGE_NR data structure described below:

```
        typedef struct {
                    int                 Last_Error;
                    enum IMAGE_NR       Image_Value;
        } K_FDM_IMAGE_NR;
```

`Image_Value:` IMAGE_1 or IMAGE_2.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.39 kfdm_ae_le_get_state()

Corresponds to FDM function:

`fdm_ae_le_get_state().`

Prototype:

```
BOOL    kfdm_ae_le_get_state(
        HANDLE                  hdevice,
        int                     AE_LE_Rang,
        K_FDM_AE_LE_STATE       *Value);
```

Description:

Used to identify the operating state of an AE_LE.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`AE_LE_Rang:` row of the AE_LE.

`Value:` pointer to a K_FDM_AE_LE_STATE data structure described below:

```
typedef struct {
     int                    Last_Error;
     enum _FDM_AE_LE_STATE  AE_LE_State
} K_FDM_AE_LE_STATE;
```

Report:

This function returns a report with the Boolean value:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.40 kufdm_get_subscriber_number()

Corresponds to FDM function:

None.

Prototype:

```
unsigned char      kufdm_get_subscriber_number(
       HANDLE      hdevice;
       Ushort      *Subsriber);
```

Description:

Utility used to read the number of the subscriber.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`Subscriber:` address that will be filled with the subscriber number.

Report:

- **TRUE** if the request has been processed correctly.
- **FALSE** if it has not.

2.41 kfdm_generic_time_initialize()

Corresponds to FDM function:

`fdm_generic_time_initialize.`

Prototype:

```
#include  « kufdm.h »
BOOL   kfdm_generic_time_initialize(
       HANDLE                          hdevice,
       FDM_GENERIC_TIME_DEFINITION     *user_definition);
```

Description:

The *kfdm_generic_time_initialize* function creates the time MPS variable and messaging LSAPs required to run the time producer redundancy management protocol. It can also start the protocol.

Input parameters:

`hdevice:` handle reference on open device.

`user_definition:` pointer to FDM_GENERIC_TIME_DEFINITION data structure. Only the following fields are acknowledged:

- *with_choice_producer,*
- *with_MPS_Var_Produced.*

Report:

This function returns a report with the Boolean value:

- **TRUE** if the primitive was correctly processed.
- **FALSE** if the primitive was not correctly processed.

Note

This function initializes the time variable. If the time variable is already initialized, calling this function again does not affect the time variable and produces an error.

2.42 kfdm_generic_time_delete()

Corresponds to FDM function:

`fdm_generic_time_delete.`

Prototype:

```
#include  « kufdm.h »
BOOL   kfdm_generic_time_delete(
       HANDLE      hdevice);
```

Description:

The *kfdm_generic_time_delete* function deletes the time MPS variable and messaging LSAPs required to run the time producer redundancy management protocol. It can also start the protocol.

Input parameters:

`hdevice:` HANDLE reference on open device.

Report:

This function returns a report with the Boolean value:

- **TRUE** if the primitive was correctly processed.
- **FALSE** if the primitive was not correctly processed.

Note

If the *kfdm_generic_time_initialize* function has not yet been launched, calling this function will not affect the time variable and produces an error.

2.43 kdfm_generic_time_set_priority

Corresponds to FDM function:

`fdm_generic_time_set_priority.`

Prototype:

```
BOOL    kfdm_generic_time_priority (
        HANDLE          hdevice,
        GT-PRIORITY     priority );
```

Description:

Used to modify the subscriber priority that is involved in the election process of the time variable producer.

Input parameters:

`hdevice:` HANDLE reference on the open device.

`priority` value of the priority to set [0..15]. 0 is the greatest priority.

Report:

This function returns a report with the boolean value.

- **TRUE** if the request has been processed correctly
- **FALSE** if has not.

2.44 kfdm_generic_time_set_candidate_for_election()

Corresponds to FDM function:

`fdm_generic_time_set_candidate_for_election().`

Description:

Used to include or not the subscriber in the election process ot the time variable producer.

Input parameters:

`hdevice:` HANDLE type reference on the open device.

`val:` if **FDM_FALSE**: the subscriber will not be included.

if **FDM_TRUE**: the subscriber will be included.

Report:

This function returns a report with the boolean value:

- **TRUE** if the request has been processed correctly
- **FALSE** if it has not

2.45 kfdm_generic_time_get_election_status()

Corresponds to FDM function:

None.

Prototype:

```
BOOL    kfdm_generic_time_get_election_status(
        HANDLE          hdevice;
        Etat Abonnés*   result);
```

Description:

Used to know the status of the subscriber regarding the process of the election of the time variable.

Input parameters:

`hdevice:` HANDLE type reference on the open device.

`result:` pointer to an *Etat Abonnes* type data structure, described below, that contains the status.

```
type def struct{
                    unsigned                    abonnés: 8;
                    enum subscriber states      state: 4;
                    unsigned                    Prio: 4;
} Etat Abonnés;
```

Report:

This function returns a report with the boolean value.

TRUE if the request has been processed correctly.

FALSE if it has not.

Chapter 5 Example

An example is provided in the */usr/fipdrv/example* directory that uses event-related functions (*SetEvent*, *WaitForMultipleObject*) and the *SleepEx* function.