Customer	:	Nexeya
Subject	:	PMC-AR429

Reference	:	MUA_2088
Issue	:	02
Edition date	:	04/02/2019

USER MANUAL
PMC-AR429

Author	Controller
Name : Marc NICOLAS Function : Product manager Date : 19/02/2019 Visa :	Name : Alain CHEBROU Function : Design manager Date : 19/02/2019 Visa :
Quality	
Name : Sabine LASSERRE Function : Quality manager Date : 19/02/2019 Visa :	

MODIFICATIONS TABLE

Issue	Date	Modification
01	26/03/2018	Creation
02	19/02/2019	Update of the translation

TABLE OF CONTENT

1 INTRODUCTION

1.1 Context

This document describes the ARINC 429 board developed and marketed by Nexeya France.

This board and its software (API and configuration) are described in a general way in the second chapter and in more detail in the following chapters

1.2 Object

This document describes the installation and use of the PMC-AR429 board.

1.3 Relevant and reference documents

1.3.1 Relevant documents

N°	Document	Reference	Issue	Date
DA1	ARINC SPECIFICATION 429 PART 1-17	2001351459		17/05/2004

2 BOARD OVERVIEW

2.1 General characteristics

Standard	Format PCI Mezzanine Card (PMC) standard.
PCI standard conformity	32 bits Interface 33 MHz, conforms to standard PCI 2.1. The board can be placed in a slot PCI 5V or 3.3V The board can be placed in a 66 MHz 64-bit PCI slot but with restrictions of the bus 33MHz, 32-bit.
Temperature range	Operating : -20°C à +75°C, Humidity : 90% without condensation. Storage : [-40 ; +85°C].
Electrical consumption	5V : 3.0 A, max 16 channels in transmission.
Interface	1x Connector Nicomatic ARINC channels, trigger and clock. 2x Lemo 2 points for IRIG-B (IN and OUT). All signals are available on the PMC P4 connector.
Configuration	No switch on the board : only software configuration
Maintenability	Firmware update can be done directly on the system without disassembling the board.

2.2 ARINC 429 characteristics

Norm	ARINC SPECIFICATION 429 PART 1-17, PUBLISHED: May 17, 2004
Channels	16 channels, independent, high and low speed.
Functions	Tx and Rx for each channel, configuration by software. SDI codes management. Cyclic and / or random emissions (Random emission of labels in addition to the cyclic pattern). Full or selective spy of all channels Real time FIFO. FIFO for each channel with filtering Synchronous update of channels data in transmission to ensure consistency of labels.

2.3 Annex functions

Dating	The board has an IRIG-B122 receiver. Monitored messages are dated with IRIG-B receiver. Resolution of 1 µs. In case of unavailability of an external dating signal the board can generate its own IRIG-B122 signal.
Synchronization	In reception, the board can be configured to recognize specific labels running on a specified channel. The recognition of a label generates of a pulse on a logic output (trigger out TTL). In transmission, it is possible to generate a pulse at a pre-determined location of the transmission frame.
Transmission clock	The words transmission pace can be clocked with an internal clock (0.1 to 2000 Hz) or with an external IO (clock in TTL).
Extension	The board has a downloadable firmware in its FPGA. Specific versions can be made without modification of the hardware.

The board has a 32-bit microcontroller whose firmware can be modified to add specific features.

2.4 Operating system

Libraries for the ARINC 429 board exist for the following operating systems:

- Windows 7 and 10, 32/64 bits.
- Linux 2.6, 3.x and 4.x, 32/64bits.

Examples of applications written in C are provided with the board. They show the implementation of the main functions of the board. These applications are based on the **a429lib library** described later.

3 PMC-AR429 DESCRIPTION

This board is designed for applications using many ARINC 429 channels, mainly for test benches application. The board provides for each channel the three functions of an ARINC bus:

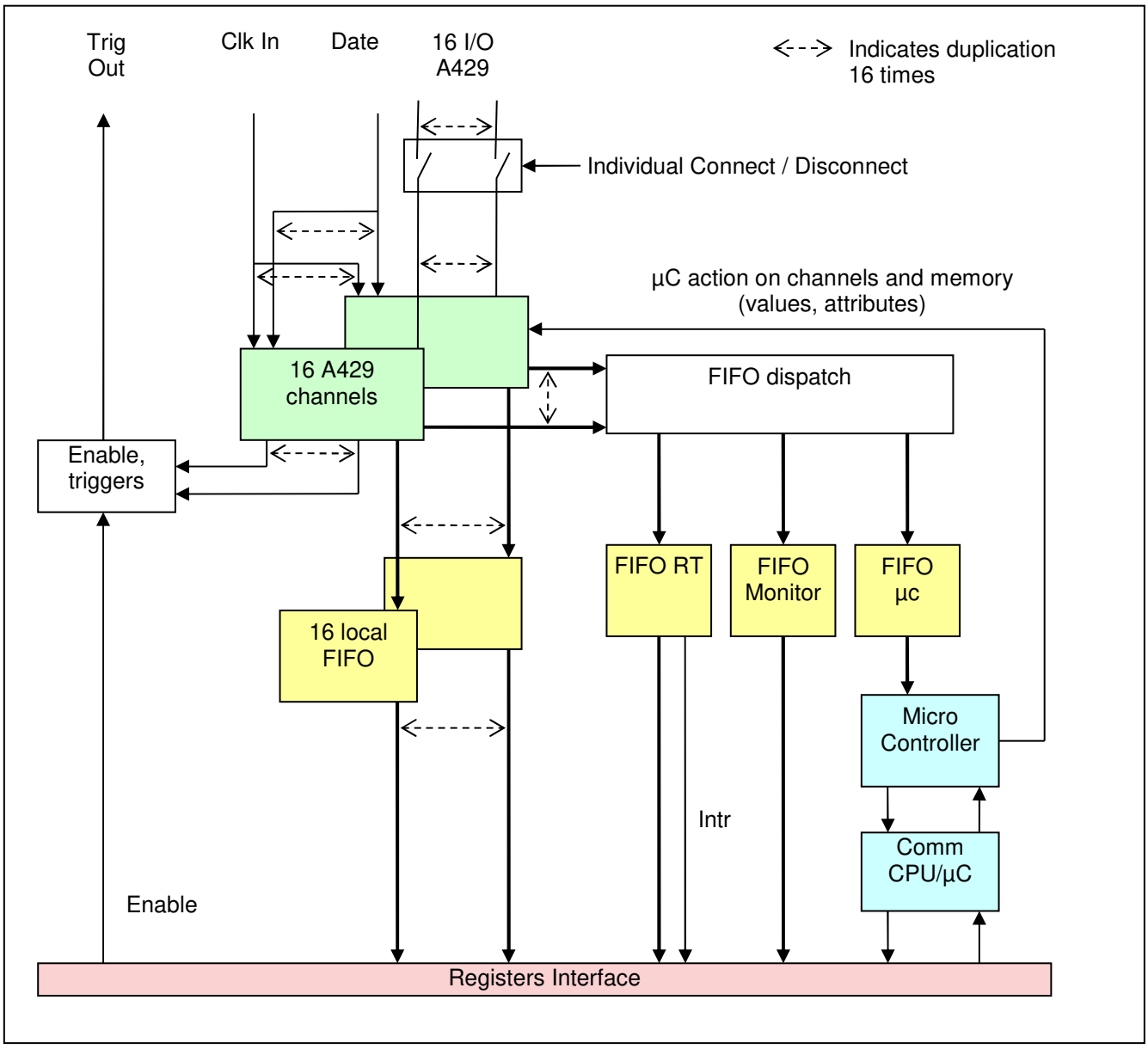
- Transmission clocked on an external or internal signal
- Reception with recognition and filtering of labels
- Data monitor : All bus traffic can be transmitted to the application: parity or coding errors are reported, each message is dated.

Functions that makes test tools easier to design:

- Configuration flexibility: the number of channels in transmission or reception is not fixed, since all the channels can be configured in transmission or reception (but not both at the same time). The transmission / reception channel ratio can be adapted to each system.
- Monitoring on transmit and receive channels to be able to replay the sequences.
- Transmitted data mastery: synchronous update of data, deterministic timings.
- Generation of external synchronization signals: By detection of a label in reception, or at a known time of the transmission frame.
- Insertion on demand of random data in a cyclic flow.
- Generation of various types of permanent or fugitive errors.
- In addition to the monitoring function, the board has a FIFO, common to all channels, in which can be inserted filtered labels, or events from the channels in transmission or from the microcontroller. This FIFO generates an interrupt when it is non-empty to allow real-time feedback.
- Each channel has a FIFO in which can be inserted filtered labels of this channel.
- The microcontroller of the board can perform some basic operations, which require real-time feedback, in order to help the host system.

3.1 Synoptic

The following diagram shows the general architecture of the board:



In the diagram, each green box represents the logic associated with an ARINC channel and includes:

- ARINC429 transmitter and receiver, which cannot be used simultaneously
- Dedicated memory to each channel: 1K values, space for the transmission frame, synchronous update blocks and random transmission.
- Attribute management logic (error generation, FIFO setting ...), timing of emissions, etc.

In the diagram, each yellow box represents the FIFOs

- Monitoring common FIFO
- Real time common FIFO with its “Not Empty” interruption
- Local FIFO for each channel
- Microcontroller FIFO for internal board use.

In the diagram, each blue box represents the resources of the micro controller. Access to the microcontroller is hidden by software.

3.2 ARINC frame definition

This paragraph introduces some notions on bit numbering of a 32-bit ARINC 429 word.

Generalized BNR Word Format

TABLE 6-2-1

P	31 30 SSM	29 ... 11 PAD	SDI	8 7 6 5 4 3 2 1 LABEL
		1/2 1/4 1/8 1/16 1/32 1/64 1/128 etc		
0	1 0	0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	0 0	0 1 1 0 1 1 1 1
Example:		512 Knots (i.e., 1/8 x 4096 where 4096 is entry in range column of Table 2, Att. 2)		N-S VELOCITY (366)

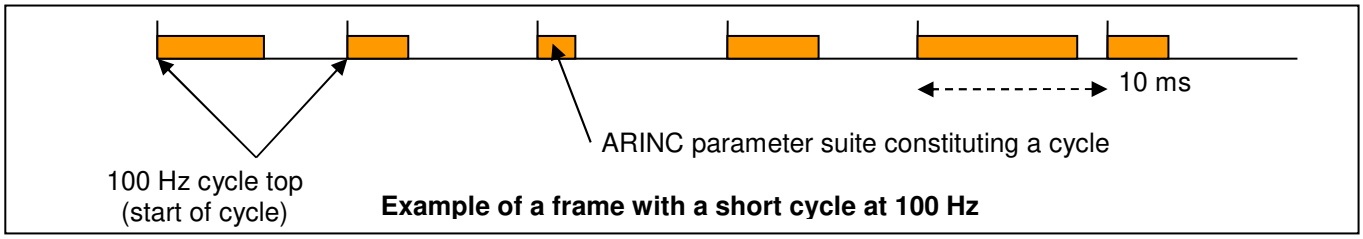
The less significant bit is number 1 and the most significant bit (parity) is number 32.

The description of the cyclic transmission order of ARINC words and their timing is called frame, or long cycle. ARINC words are also called parameters or messages.

The cyclic frame is composed of N short cycles clocked at a defined frequency (for example 100 Hz).

A short cycle is the set of parameters issued following the reception of a start of cycle. The start of the cycle can be an event generated by the board (timer) or an external event (clock in).

The parameters described in the frame are cyclic parameters. There are also random parameters.



Example of a frame with a short cycle at 100 Hz

3.3 Transmission

The board can manage 16 channels, which can be configured individually for transmission or reception.

For the transmission, the software sends 2 parameters:

- A table of 1024 words of 32 bits which contains the values of the messages
- A 32-bit word table that describes the transmission frame with a series of operations to do. The frame describes words to transmit cyclically (short cycles) and the cycle timing. In a short cycle, by default words are sent consecutively with a nominal 4-bit word inter delay

The timing of blocks uses an internal programmable or external divisible clock.

The transmission is completely managed by the board, which ensures precise and deterministic timings.

Each ARINC channel is independent: it own its own frame descriptor and its own timing. The frame is built by the application, and then transmitted to the board, which places it in its internal memory.

3.3.1 Cyclic frame

The cyclic frame is an array of operators built by using functions of the *a429TxOpXx()* family.

Example of a description of a two-cycle frame that transmits the labels 0312 and 0205 at 2 different frequencies.

Opérateur	Argument	Commentaire
OPCYCLE		Wait for the cycle top
OPDATA	0312	
OPUPDATE	0	Validate the synchronous update from block 0
OPCYCLE		Wait for the cycle top
OPDATA	0312	
OPDATA	0205	
OPUPDATE	0	Validate the synchronous update from block 0
OPEND		End of long cycle (added automatically).

The frame is built by the application software and must be sent to the channel when configured with *a429TxSetFrame()* and before the channel starts

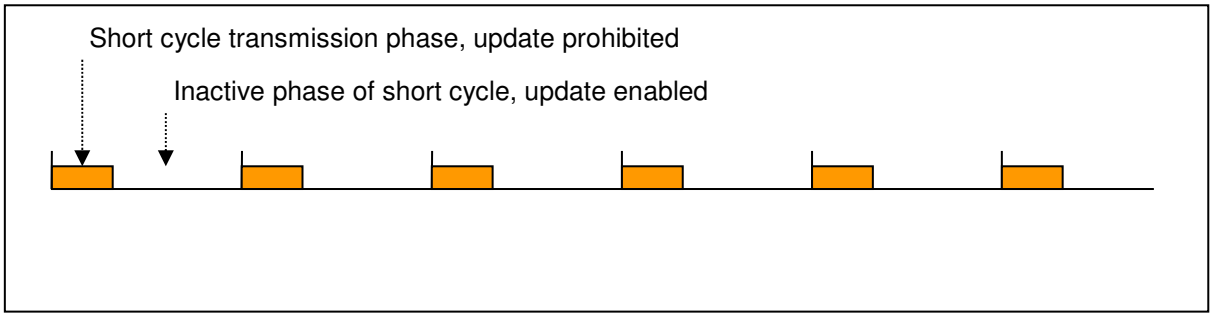
Note: The frame has a size limited to 7168 operators, which allows to build a frame of 10 seconds.

3.3.2 Synchronous update

Message values can be updated by the application by two means:

- By direct access to the table of values, before or after starting the program with *a429TxWrite()* or *a429TxWrites()*.
- By an update synchronized by the emission cycles, after starting the transmission, as explained here.

Some applications require that words update is impossible while their treatment is in progress that is to say, during the active phase of a short cycle. This is to ensure the consistency of a set of parameters.

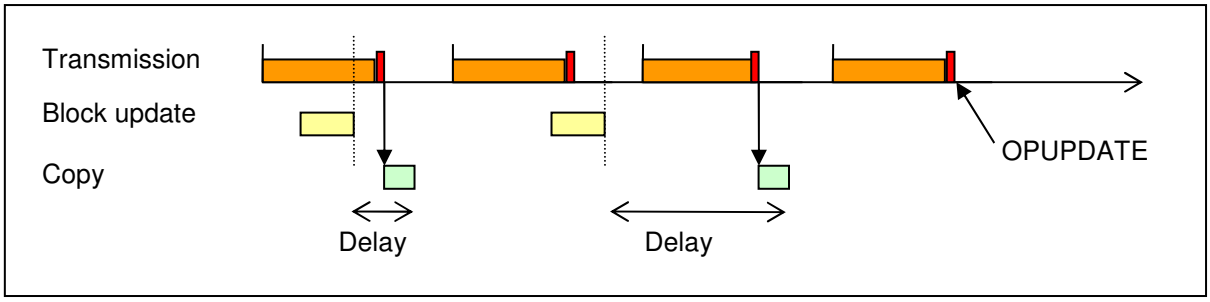


For this, two mechanisms are used:

- The data is written to an intermediate update buffer instead of being directly written to the word table. There are several intermediate buffers to allow multiple independent update cycles
- The frame has operators that validate the update from a particular update block at a specific location in the frame.

When the transmitter finds an operator OPUPDATE in the frame, it validates the update from the corresponding block: it looks if this update block is signaled full, and if YES, it copies the contents of this block in the word table.

The copying takes place only during the execution of the OPUPDATE operator, which guarantees that there is no parameter being broadcast at this time. But this can cause a delay on the update.



The application provides a block of data to the library, which copied this block into the board, and which signals the availability of this block to the board. This report can only be canceled by the board that makes the copy: the application cannot reuse this block until it has been copied. However, after stopping a channel, it is necessary to release any blocks that the channel did not have time to consume with a429TxBlockReset ().

Each update block can contain up to 255 messages to update. It is however possible to update more using consecutive blocks, and providing the number of the first block used. It is the responsibility of the application not to reuse additional blocks until the first is available.

Note: Each channel has eight memory blocks. Each block can be use by the synchronous update or the random broadcast.

Example of synchronous update:

```
HA429  hCard ;
uint32_t         result ;
uint32_t         channel = 2 ;
uint32_t         updBlocNum = 4 ;   // Number of the block to use

uint32_t         updBloc[255] ;

// Buid an array of values
updBloc [0] = 0x600000CA ;
updBloc [1] = 0x20000085;

// Before write
// Avant ecriture: verify that the previous update is complete
while (A429_ENONE != a429TxCheckUpdate (hCard, channel, updBlocNum))
       . . . wait a while

// send the block
result = a429TxUpdate (hCard, channel, updBloc, 2, updBlocNum) ;

// Check result
if (result != A429_ENONE)
{
       // There was an error
       if (result == A429_EBUSY)
       {
                // The block was not free.
                // Impossible here since we checked before,
                // unless another thread does the same thing...
       }
       Else
       {
                // Another error. The channel is sending, started...
       }
}
```

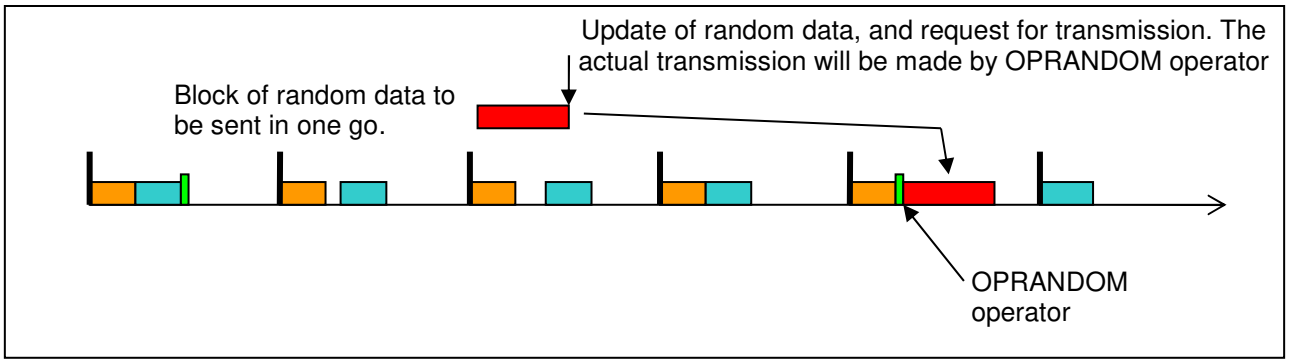
3.3.3 Random transmission

Some applications need to be able to emit words that are not in the frame, hence their random name (as opposed to cyclic)..

A mechanism makes it possible to fill a descriptor block then to ask the board to transmit words.

The emission descriptor block is constructed in the same way as a frame description: it contains OPDATA, OPDELAY operators or others. Emitted values are taken from the word value table, and must be previously updated with *a429TxWrite()*. The block is sent to the library with *a429TxRandom()*. The emission itself is triggered by the execution of an operator OPRANDOM in the frame, which makes it possible to control the moment of emission and not to disturb the timing of the cyclic parameters.

The random data block is issued at one go, so it is the responsibility of the application to place the OPRANDOM operators only in the cycles with the necessary space for the transmission of the complete block. If the time at the end of the cycle is not sufficient, the beginning of the next cycle is shifted to the end of the random transmission.



The application provides a transmission descriptor (array of operators) and a memory block number to use. The library copies the descriptor into the block, and then reports the availability of this block to the board. Only the board can cancel this report once the transmission is done, and the application cannot reuse this block until it has been transmitted.

Note: Each channel has eight memory blocks that are trivialized and used by both synchronous update and random broadcast.

Example of use of the random broadcast:
We suppose the transmit channel is configured and started, most error handling is not present for text clarity

```
HA429  hCard ;
uint32_t         result, ii ;
uint32_t         channel = 2 ;
uint32_t         randBlocNum = 0 ;  // Number of the block to use

uint32_t         randBloc[255] ;

// Build the frame. Delays are for the example.
ii = 0 ;
randBloc[ii++] = a429TxOpData  (0xCA) ;
randBloc[ii++] = a429TxOpDelay (8) ;         // 12 bits time (4+8) between words
randBloc[ii++] = a429TxOpData  (0x85) ;
randBloc[ii++] = a429TxOpDelay (8) ;         // 12 bits time (4+8) between words

// Update the table of values
A429TxWrite (hCard, channel, 0x600000CA) ;
A429TxWrite (hCard, channel, 0x20000085) ;


// Before writing: check that the previous show is finished
While (A429_ENONE != a429TxCheckRandom (hCard, channel, randBlocNum))
       . . . Wait a while

// Send the bloc
kresult = a429TxRandom (hCard, channel, randBloc, ii, randBlocNum) ;

// check the result
if (result != A429_ENONE)
{
       // There was an error
       if (result == A429_EBUSY)
       {
                // The block was not free.
                // Impossible here since we checked before,
                // unless another thread does the same thing...
```

```
        }
        Else
        {
                // Another error. The channel is sending, started...
        }
}
```

3.3.4 SDI Field

In transmission, it is possible to transmit several words with the same label, but different SDI fields values. These words must be located at different places in the word value table, which is the responsibility of the management library.

For a cyclic emission, the operator OPDATA must be built by the application by specifying the SDI fields values and labels values. In addition, the SDI indicator table must have been provided before, during the configuration of the channel..

3.3.5 Transmission exemple

Example of a program that configures a channel in cyclic transmission and starts it. The channel uses a 2-cycle frame that emits the 0312 label at 50Hz, and the 0205 label at 25Hz. It is planned to use the synchronous update from block number 0. The SDI fields are not used. Most error handling is omitted for text clarity

```
HA429  hCard ;
uint32_t         result ;
uint32_t         channel = 2 ;
uint32_t         frame [1024] ;
uint32_t         ii ;

// Open the board
result = a429Open (& hCard, 0, 0) ;
if (result != A429_ENONE)
{
       // Opening error, displaying the text of the error message
       printf ("a429Open : %s\n", a429ErrorMessage (result)) ;
       return ;
}

// Configure the channel for emission
result = a429TxConfig (hCard, channel, NULL,
                           a429BaudRate (100000),     // bit rate 100 kHz
                           A429_SFIFOEN | A429_HIGH | A429_CLKINT,
                           a429CycleDiv (50)) ;       // Short cycle 50 Hz

// Build the fram
ii = 0 ;
// 1er cycle
frame [ii++] = a429TxOpCycle        () ;
frame [ii++] = a429TxOpData         (0312) ;
frame [ii++] = a429TxOpUpdate       (0) ;

// 2nd cycle
frame [ii++] = a429TxOpCycle        () ;
frame [ii++] = a429TxOpData         (0312) ;
frame [ii++] = a429TxOpData         (0205) ;
frame [ii++] = a429TxOpUpdate       (0) ;
```

```
result = a429TxSetFrame (hCard, channel, frame, ii) ;

// Initialize data words
result = a429TxWrite (hCard, channel, 0x600000CA) ;
result = a429TxWrite (hCard, channel, 0x20000085) ;

// Start continuous cyclic emission
result = a429TxStart (hCard, channel, 0) ;
.
.
.
```

3.4 Reception

For each channel in reception, the received words can go to several destinations:

- A memory specific to each channel.
- A monitor FIFO common to all channels.
- A "real time" FIFO common to all channels.
- A "local" FIFO specific to each channel

3.4.1 Memory mode

The utility of this mode is to allow the application that uses the receiver to have access at any time to the last value received from each label.

When reading an indicator lets you know if the value read is the same as during the previous reading, or if this label has been received in the meantime, see *a429RxRead()*.

The received labels are systematically placed in this memory, unless they have an error

3.4.2 Monitor FIFO

When receiving the first bit of a word, the date and the receiver number are stored. When the word is received, a 4 words frame is available to be put in FIFO. See the monitoring chapter.

The interest of this mode is to obtain all the dated traffic, including errors (full monitor). The monitoring starts on request.

The monitor FIFO setting can define filtering at the channel or label level. By default, there is no filtering label: all labels are put in monitor FIFO.

The monitor function API consists of the functions *a429SpyXxx()*.

3.4.3 Real time FIFO

It is possible to send some labels to a FIFO common to all channels, called real-time FIFO. This FIFO is designed to receive a reduced flow compared to the monitor FIFO. This flow is sometime necessary for the immediate processing of the application.

The labels that must go in this FIFO must be specified with *a429SetLabelAttr()*. This FIFO has a depth of 2048 labels.

Use of the Real-Time FIFO must be acquired using the *a429RtFifoEnable()* function before you can read its contents.

There are three possible methods to use this real-time FIFO :

1) **By Polling**

In this case the application periodically calls the function *a429RtFifoRead()*, which reads the contents of the FIFO. The call must be frequent enough to avoid an overflow of the FIFO.

2) By Interruption

This method uses the FIFO-generated interrupt when it becomes "non-empty", which makes it possible to wake up a thread as soon as there is something to read in the FIFO. The thread is created by the application.

The command sequence to use by the application thread is:

- a429RtFifoEnable() to acquire the use of the FIFO and initialize it.
- a429RtWaitForInt() to wait until there is something to read in the FIFO.
- a429RtFifoRead() to read the contents of the FIFO until this function returns A429_NOTRECEIVED, then return to a429RtWaitForInt().

Before the end of the thread, release the RT FIFO with *a429RtFifoEnable (hCard, 0)*.

3) By Callback

This method uses the FIFO-generated interrupt when it becomes "non empty", which makes it possible to wake up a thread as soon as there is something to read in the FIFO. The thread is created and managed by a429lib.

The application must use *a429RtSetIntCallback()* to register the function that must be called when a label is available in the FIFO and provide it with this label. Calling *a429RtSetIntCallback()* with a non-NULL callback results in the creation of the FIFO management thread and FIFO monitoring. The NULL callback call ends the management thread of the RT FIFO.

Notes on using the RT FIFO

- To enable a process to use RT FIFO on a particular board, it must have specified the flag A429_OPENRTFIFOEN when opening this board with *a429Open()*. Only one active process can specify this flag: if a process has already used this flag *a429Open()* returns the A429_EINUSE error.
- The use of the three management methods of the FIFO is exclusive: you should not use 2 methods simultaneously, but it is possible to change the method in the process (in this case it is possible that some labels are lost during change).
- You have to start the management of the RT FIFO (*a429RtFifoEnable()* or *a429RtSetIntCallback()*) before starting the channels so as not to lose the first FIFO labels.

3.4.4 Local FIFO

For each channel, it is possible to send some labels to a FIFO specific to the channel, called local FIFO. This FIFO is intended to receive a more reduced flow than the real time FIFO. The stream is selected by channel.

These FIFOs must be exploited by polling, and have a depth of 128 labels.

The labels that must go in this FIFO must be specified with *a429SetLabelAttr()*.

3.5 FIFO frame format

When a word is received or sent, a 32-bit 4-words of 32-bits frame is built for FIFOs. The FIFO frame is defined as follows:

```
typedef struct
{
        uint32_t   flags ;          // Header word
        uint32_t   data ;           // Arinc / event word
        uint32_t   date1 ;          // Irig date word 1
        uint32_t   date2 ;          // Irig date word 2

} S_A429_MESSAGE ;
```

Two types of frames are used, differentiated by a bit in the header word:

- The frame for the ARINC words, the header word contains the bit A429_SPYLABEL. In this case the data word contains the ARINC word.
- The frame to signal an event, the header word contains the bit A429_SPYEVENT. In this case the data word contains the identifier of the event on the 16 low-order bits. Events are dated like messages.

The format of the header word is as follows:

Bits	Name	Description
3-0		Channel number
5-4		
7-6	A429_SPYSILENCEM	Silence too short before this word if different from 0
8	A429_SPYBITLONG	Bit too much
9	A429_SPYBITSHORT	Bit missing
10	A429_SPYFRAME	Framing error (bad bit)
11	A429_SPYPARITY	Parity error
12	A429_SPYEVENTUC	Microcontroller event
13	A429_SPYFILL	Filling (Monitor FIFO only)
14	A429_SPYEVENT	Event (exclusive with label)
15	A429_SPYLABEL	Label (exclusive with event)
31-16		Ident 0xA429

Note: The A429_SPYERRORMASK value is made from all the errors of the header word and facilitates the detection of erroneous words:

```
#define A429_SPYERRORMASK (A429_SPYPARITY | A429_SPYFRAME | \
                           A429_SPYBITSHORT | A429_SPYBITLONG)
```

The value A429_SPYSILENCEM is not considered as an error because the word is exploitable in spite of the violation of the timing imposed by the standard.

3.6 Recognition of a label

In a test bench it is useful to have an external signal corresponding to a particular label (scope synchro for example). A label can be marked with *a429SetLabelAttr()* so that a pulse is generated on a logical output (called trigOut) of the board at the end of its reception or emission.

3.7 SDI field

In reception, it is possible to receive several words with the same label, but with different SDI fields values and identify them. For that:

- A space of 1 K word is reserved to be able to memorize all the combinations of label and SDI.
- It is necessary to indicate for which label the SDI field is to be taken.into account when configuring the channel by *a429RxConfig()*.

When receiving a label, if the corresponding SDI flag is 0, the word is sorted using only the label field as an index, otherwise the index is built from SDI + label (SDI = heavy weight, label = low weight).

3.8 Errors

If a word is received with an error, it is placed in the spy FIFO with the corresponding status indicators. However, it is not stored in the word table (we do not know if the label is correct, so we do not know where to write it), nor in the local FIFOs and real time.

Errors are counted for each channel, and this count is readable with *a429RxErrorCount()*.

3.9 Full Monitor

The full monitor produces a stream with all the traffic on the requested channels, whether these channels are transmitting or receiving. This stream is placed in a FIFO.

The reception or the emission of an ARINC word by the board results in the insertion of a block of 4 words (state + value + date) in the monitor FIFO at the end of the reception or emission. For a particular channel, the chronological order of the data is assured, but the inter-channel chronological order is not.

The a429lib library is responsible for exploiting the data flow and for constructing an S_A429_MESSAGE structure for each message, see the functions *a429SpyRead()* and *a429SpyReaderMessages()*.

Monitoring is integral because it is possible to monitor all the labels of all channels, including errors. However, filtering possibilities exist:

- On channel level with the A429_SFIFOEN flag during configuration.
- On label level with the flag A429_ATTR_SPYFIFO in the attributes, see *a429SetLabelAttr()*.

The attribute A429_ATTR_SPYFIFO is positioned by default, it is thus enough to position the flag A429_SFIFOEN during the configuration of the channel so that it is fully monitored.

If a process wants to use the monitor, it must have specified the flag A429_OPENSPYEN when opening this board with *a429Open()*. Multiple threads can specify this flag simultaneously, but only one can use the monitor at any given time.

Example of a program using monitor, channels are opened, the error handling is simplified:

```
uint32_t                result, count ;
S_A429_MESSAGE  dataBus ;

// Start the monitor
if (A429_ENONE != a429SpyStart (hCard, 0, A429_SPYDIRECT))
{
       printf ("a429SpyStart error\n") ;
       return (0) ;
```

```
}

while (1)
{
	// Try to get a message
	result = a429SpyRead (hCard, & dataBus, 1, & count) ;
	if (result != A429_ENONE)
	{
		if (result == A429_NOTRECEIVED)
		{
			// Nothing to read
			Sleep (10) ;
			continue ;
		}

		if (result == A429_WRITEOVER)
		{
			// Too late, some data are lost
			continue ;
		}
	}

	// Exploit the message
	if (count)
		displayA429Message (& dataBus) ;
}
a429SpyStop (hCard) ;
```

Note: This example uses direct access to the board monitor FIFO to process the stream in real time. If it is not necessary it is recommended to use the DMA engine (do not use the flag A429_SPYDIRECT).

3.10 Automatic operations

The microcontroller of the board can be responsible for performing basic operations, but with a real time timing.

These operations concern the values and the attributes of the labels of the channels in emission.

The communication between the application and the µC uses the following mechanisms:

- A descriptor structure of the operation to be performed.
- Possibly a block of data associated with the operation.

The structure has general fields followed by a structure union related to each type of operation:

```
typedef struct
{
	uint32_t	label ;		// The concerned label
	uint32_t	flags ;		// Operator attributes
	uint32_t	dataCount ;	// Number of occurrences
	uint32_t	value ;		// Forced value

	uint32_t  res [9] ;		// Reserved - fill

} S_A429_UCFORCE ;			// For A429_UCOP_SETATTR and A429_UCOP_FORCE

//-----------------------------------------------------
typedef struct
```

```
{
       uint32_t    label ;          // The concerned label
       uint32_t    flags ;          // Operator attributes
       uint32_t    dataCount ;      // Data bloc size (count of label values)
       uint32_t    idEvent ;        // Event Id at end of buffer
       uint32_t    iBuffer ;        // Communication data index

       uint32_t    res [8] ;        // Reserved - fill

} S_A429_UCBUFFER ;

//-------------------------------------------------------
typedef struct
{
       uint32_t    label ;          // The concerned label
       uint32_t    flags ;          // Operator attributes
       uint32_t    mask ;           // Counter mask
       uint32_t    incr ;           // Counter increment
       uint32_t    value ;          // Counter initial value

       uint32_t    res [8] ;        // Reserved - fill

} S_A429_UCCOUNTER ;

//-------------------------------------------------------
typedef struct
{
       uint32_t    idCks ;          // Capsule ID number
       uint32_t    flags ;          //
       uint32_t    idEvent ;        // Event Id in RT FIFO for receive channel
                                    // Event Id in TX frame for transmit channel
       uint32_t    idLabel ;        // Label of capsule ID
       uint32_t    msbCksLabel ;    // Checksum MSB (transmit and receive)
       uint32_t    lsbCksLabel ;    // Checksum LSB (transmit and receive)
       uint32_t    itemCount ;      // Count of labels of checksumm
       uint32_t    iBuffer ;        // Label list

       uint32_t    lastLabel ;      // last label for receive
       uint32_t    counterLabel ;   // Counter label for transmit
       uint32_t    counterIncr ;    // Counter increment for transmit
                                    // Counter range tolerated for receive

       uint32_t    res [2] ;        // Reserved - fill

} S_A429_UCCKS1760 ;                // For A429_UCOP_CKS1760

//-------------------------------------------------------
// Request structure
typedef struct
{
       uint32_t  operation ;        // Operator id (A429_UCOP_xx)
       uint32_t  result ;           // Return value: A429_ENONE or error number
       uint32_t  channel ;          // The concerned label

       union
       {
            S_A429_UCFORCE    force ;
            S_A429_UCCOUNTER  counter ;
            S_A429_UCBUFFER   buffer ;
            S_A429_UCCKS1760  cks1760 ;
       } ;
```

```
} S_A429_UCOP ;
```

The operations are sent to the library using *a429UcOp()*.

The available operations are:

- A429_UCOP_SETATTR Modification of label attributes to generate permanent or temporary errors.
- A429_UCOP_COUNTER Lets you specify how the value of a label changes as a counter.
- A429_UCOP_FORCE Allows you to force the value and attributes of a label permanently or temporarily.
- A429_UCOP_BUFFER Provides a buffer of values that the label will use consecutively.
- A429_UCOP_CKS1760 Define a 1760 type checksum capsule.

Operations recorded for one channel are all released when the channel is stopped by *a429TxStop()*, or automatically at the end of a one shot emission.

3.10.1 A429_UCOP_SETATTR

Allows you to modify some of the attributes of a label, in order to generate temporary errors, on an emission channel.

label The label concerned

flags the attribute word constructed in the same way as the function *a429SetLabelAttr()*. One of the two indicators A429_ATTR_ADD or A429_ATTR_REMOVE must be present.

dataCount the number of occurrences of the label during which these changes are to take place. This value must be greater than or equal to 1.

The indicators are:

A429_ATTR_ADD	Add the attributes provided to the attributes of the label.
A429_ATTR_REMOVE	Remove the attributes provided to the attributes of the label.
A429_ATTR_DELETE	Cancel an operation A429_UCOP_FORCE or A429_UCOP_BUFFER. Used alone.
A429_ATTR_DISABLE	Inhibition of the label transmission. Exclusive with A429_ATTR_DISABLET
A429_ATTR_DISABLET	Inhibition preserving the emission timing. Exclusive with A429_ATTR_DISABLE.
A429_ATTR_EPARITY	Forcing a parity error
A429_ATTR_EFRAME	Forcing a framing error: bit 11, second half of the bit is at level 0.
A429_ATTR_ESHORT	Forcing message too short: cancel the last bit
A429_ATTR_ELONG	Forcing message too long: the last bit is duplicated
A429_ATTR_SEP1 A429_ATTR_SEP2 A429_ATTR_SEP3 A429_ATTR_SEP4	Number of silent bit between messages: 1 to 4 bits. Default value A429_ATTR_SEP4. Action A429_ATTR_REMOVE on this attributes restores the value A429_ATTR_SEP4

Initial attributes are restored at the end of the occurrence count, before completing the operation.

Only one operation A429_UCOP_SETATTR can be active for a label: A new request will be ignored.

A429_UCOP_SETATTR operation and the A429_ATTR_DELETE attribute allows to cancel a forcing of attributes.

There are several ways to change the attributes of a label:

- *a429SetLabelAttr()*.
- a429_UCOP_SETATTR().
- a429_UCOP_FORCE()

It is not recommended to use these operations simultaneously to modify attributes of a label. Indeed in this case the behavior of the attributes is unpredictable.

3.10.2 A429_UCOP_COUNTER

Allows you to specify how the value of a label changes as a counter. The counter is defined as a contiguous bit area defined by a mask, to which an increment is added to each occurrence of the label on the bus.

Useful fields:

label	The label concerned
mask	The mask of the bits forming the counter
incr	The counter increment
value	The initial value of the label
cycleDiv	Number of instances of the label between each increment. A value of 1 indicates an increment at each occurrence, a value of 2 an increment of one occurrence out of 2, and so on.

If the label is inhibited during the operation request, the initial value is immediately placed in the channel value table. In this way, when validating the label, the first value to circulate will be the initial value of the counter. If the label is valid during the operation request, the change will be made after the next label pass on the bus, to ensure the sequence of values. To guarantee a controlled evolution of the counter at startup, it is necessary to initialize the label before starting the emission with the first value, and to give "value" the following value.

Example: For 0212 label of channel 2, counter on bits 29-22, increment of 2 for each occurrence, initial value of 0x80, SSM = valid data:

```
S_A429_UCOP op ;

op.operation             = A429_UCOP_COUNTER ;
op.channel               = 2 ;
op.counter.flags         = 0 ;
op.counter.label         = 0212 ;
op.counter.mask          = 0xFF << 21 ;
op.counter.incr          = 0x02 << 21 ;
op.counter.value         = (0x03 << 29) | (0x80 << 21) |  0212 ;
op.counter.cycleDiv      = 1 ;      // Increment on every cycle

a429UcOp (hCard, & op, NULL) ;
if (op->result != A429_ENONE)
       printf ("erreur a429UcOp\n") ;
```

There is only one active A429_UCOP_COUNTER operation for a label. A new counter replaces the old one.

To suspend a counter, use the A429_ATTR_DISABLE flag, the *mask* and *incr*, fields are ignored.

To resume a count, use the flag A429_ATTR_ENABLE, the fields *mask* and *incr* fileds are ignored.

To change the value of the increment of an existing counter, use the flag A429_ATTR_COUNTERINCR, the *mask* and *value* fields are ignored.

A counter has a lower priority than the value forcing: if a value is forced for the same label, the counter is suspended.

3.10.3 A429_UCOP_FORCE

Force to set the value and the attribute of a label temporally.

Useful field:

label	The label concerned
flags	The attribute word constructed in the same way as the function a429SetLabelAttr (). One of the two indicators A429_ATTR_ADD or A429_ATTR_REMOVE may be present. If none are present the attributes are not used.
dataCount	The number of occurrences of the label during which these changes are to take place. This value must be greater than or equal to 1.
value	The value to force

Example : Force a value with a parity error for 2 occurrences:

```
S_A429_UCOP op ;

op.operation            = A429_UCOP_FORCE ;
op.channel              = 2 ;
op.force.flags          = A429_ATTR_ADD | A429_ATTR_EPARITY ;
op.force.label          = 0212 ;
op.force.value          = (0x03 << 29) | (0x18 << 21) | 0212 ;
op.force.dataCount      = 2 ;

a429UcOp (hCard, & op, NULL) ;
if (op->result != A429_ENONE)
       printf ("erreur a429UcOp\n") ;
```

Only one active A429_UCOP_FORCE operation for one label. A new request is ignored.

Notes:

- Forcing has a higher priority than A429_UCOP_BUFFER. During forcing, the emission of the buffer values is suspended, it will resume at the end of the forcing operation.
- At the end of the forcing operation, the attributes are restored to their previous value, but not the value.

Priority can be overridden with operation A429_UCOP_FORCE and attribute A429_ATTR_DELETE.

3.10.4 A429_UCOP_BUFFER

Provides a buffer of values that the label will use consecutively.

It is possible to loop back automatically on the buffer, or to go to the next buffer if there is one available.

The limitations of the operation are as follows:

- For each channel up to 16 labels can simultaneously be associated with buffers.
- Each label can accumulate up to 8 buffers, beyond which the operation will be refused.
- Each buffer can contain up to 256 values.

In order to keep the application informed of the progress of buffer consumption, it is possible to request that an event be inserted in the real-time FIFO or the local FIFO of the channel at the end of the transmission of the buffer.

Useful fields:

Label	The label concerned.
flags	Indicators for configuring the emission.
dataCount	The number of data (uint32_t) in the data buffer.
idEvent	Number of the event to be placed in a FIFO when A429_ATTR_BUFFEREVENT is used. It is required that this identifier be on 16 bits (between 0x0 and 0xFFFF). By default or if the flag A429_EVENTUC_RTFIFO is added to the event it will be inserted into the RT FIFO. Using the flag A429_EVENTUC_CHANFIFO it will be inserted in the local FIFO of the channel.

The configuration indicators:

A429_ATTR_LOOP	Request loopback. The same buffer is issued continuously.
A429_ATTR_ BUFFEREVENT	FIFO event request when the last value is consumed.
A429_ATTR_DELETE	All buffers on this label are canceled. Exclusive with other flags, but you still need to provide a non-empty data buffer that will be ignored.

Example: Alternately, emit the values 0x55AA and 0xAA55 alternately:

```
S_A429_UCOP         op ;
uint32_t            data [2] ;

data [0] = 0x6AA55000 | 0212 ;
data [1] = 0x655AA000 | 0212 ;

op.operation                = A429_UCOP_BUFFER ;
op.channel                  = 2 ;
op.buffer.flags             = A429_ATTR_LOOP ;
op.buffer.label     = 0212 ;
op.buffer.dataCount         = 2 ;

a429UcOp (hCard, & op, data) ;
if (op->result != A429_ENONE)
       printf ("erreur a429UcOp\n") ;
```

A buffer operator has a lower priority than the value forcing operator: if a value is forced for the same label, the buffer is suspended. However, if only the attributes are forced, the value of the buffer is used.

3.10.5 A429_UCOP_ CKS1760

This operator is used to manage 1760 checksum operations on an ARINC 429 bus.

The set of labels involved in a checksum calculation constitutes a capsule. The elements of the capsule are in order:

- The identifier of the capsule,
- A counter incremented by 1 at each transmission of the capsule, and loopback from 0xFFFF to 0.. The incrementation of the counter before each calculation of the cheksum, and the verification of the increment of the counter by the receiver are made by the card.
- Data (bits 10 to 30 of the ARINC 429 words, including SSM),
- The 32-bit checksum spread over 2 labels, MSB then LSB.

The 16-bit values of the identifier, the counter and the 2 words of the checksum are placed in the bits 10 to 25 of the ARINC 429 words. The other bits have the value initialized by the application.

Physically a capsule is made as follows :

P	0 2 bits	Reserved 3 bits	Capsule Identifier 16 bits	SDI 2 bits	Label b 8 bits
P	0 2 bits	Reserved 3 bits	Capsule Counter 16 bits	SDI 2 bits	Label a 8 bits
P	Data 1 : Payload 21 bits (2 bits SSM + 19 bits data)			SDI 2 bits	Label c 8 bits
P	Data 1 : Payload 21 bits (2 bits SSM + 19 bits data)			SDI 2 bits	Label d 8 bits
P	Data 1 : Payload 21 bits (2 bits SSM + 19 bits data)			SDI 2 bits	Label w 8 bits
P	Data 1 : Payload 21 bits (2 bits SSM + 19 bits data)			SDI 2 bits	Label x 8 bits
P	0 2 bits	Reserved 3 bits	Capsule Checksum MSW 16 bits	SDI 2 bits	Label y 8 bits
P	0 2 bits	Reserved 3 bits	Capsule Checksum LSW 16 bits	SDI 2 bits	Label z 8 bits

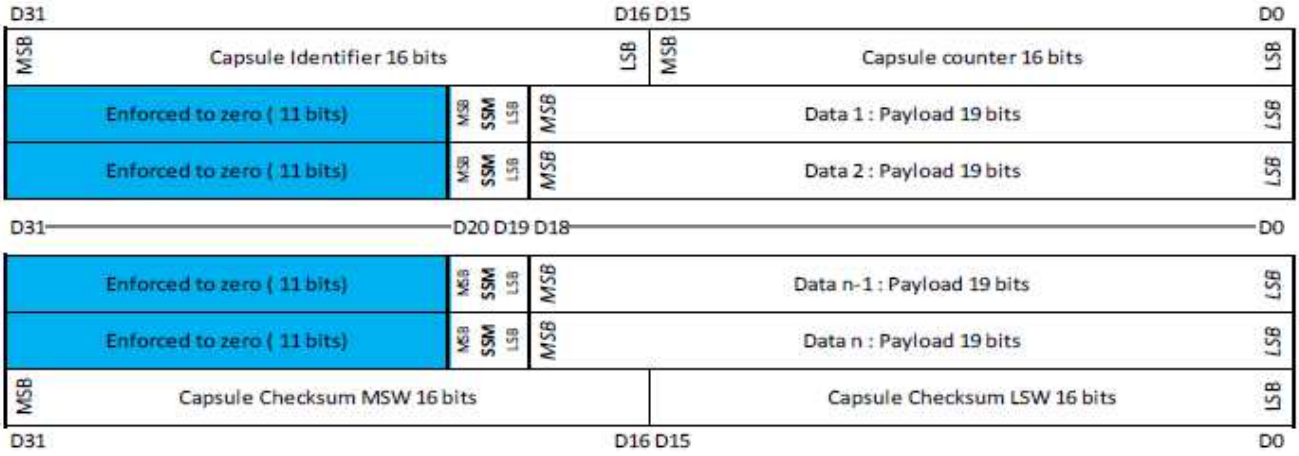
The capsules must not overlap (all words of a capsule are issued before starting the transmission of the next capsule with the same id).

The bit values SSM or unused bits by the values are not managed by the library or board. They must be initialized or modified by the application (SSM will be set at zero for the words of the identifier, the counter and the MSB and LSB control checksum by the application).

The checksum is calculated by including the identifier, the counter, and data. To calculate the payload bits are shifted toward the low weight, and unnecessary 2 bits are set to 0.

Note that ARINC protocol data (label, SDI and parity) are not covered by the checksum calculation as they do not belong to the functional data.

From a logical point of view the checksum is calculated as follows:



You can define up to 8 capsules per ARINC channel.

A capsule can contain up to 256 words of data excluding id, counter and checksum. The order of the words is important for the calculation: it is necessary to provide the list of the data labels in the order necessary for the calculation (which is not necessarily that of circulation on the bus).

The same capsule definition is used in transmission or reception, this definition should be made after the channel configuration.

Checksum Algorithm:

```
/*****************************************************************************
 * This function computes the 32 bits checksum 1760 of data.
 * #Parameters:
 *  1] IN : The address of the first data.
 *  2] IN : The number of 32 bits data.
 * #Returned value:
 *  The computed checksum 1760.
 *****************************************************************************/

unsigned int Calc_Chk1760 (const unsigned int *Data_Adr, unsigned int Nb_Data)
{
    unsigned int Csum;
    unsigned int Data;
    unsigned int I;
    unsigned int J;
    register const unsigned int *Pt_Data_Adr;

    Csum = 0;
    Pt_Data_Adr = Data_Adr;


    /* Loop on number of bytes */

    for ( I = 0 ; I < Nb_Data ; I++ )
    {
         J    = I & (unsigned int) 31;
         Data = Pt_Data_Adr [ I ];
         Csum = Csum ^ ( ( Data >> J ) | ( Data << ( (unsigned int) 32 - J ) ) );
    }

    J = I & (unsigned int) 31;
    Csum = ( Csum << J ) | ( Csum >> ( (unsigned int) 32 - J ) );
    return ( Csum );
}
```

Transmission:

Use checksum capsules imposes constraints on the application: the transmission frame is to be built using the operators and OPUPDATE and OPEVENTUC right after OPDATA operators belonging to the capsule.

The data of the capsule should not be altered during transmission of this capsule, this is what requires the use of a synchronous operator update OPUPDATE.

The microcontroller is warned that he must calculate the checksum, after the update data through the OPEVENTUC operator associated to the capsule (usually OPEVENTUC operator will be placed just after the OPUPDATE operator). The value of the OPEVENTUC operator shall be that used by the field *IdEvent* when creating the capsule. The increment counter and the checksum calculation is made at each occurrence of OPEVENTUC operator, and therefore even if the data have not been updated by the application.

When defining an emission capsule the checksum of the capsule is immediately calculated and updated in the RAM of the values of the labels so that the first capsule has issued a correct checksum (and while there not had OPEVENTUC operator execution). We must therefore define the capsules after initializing the values of labels, including counter.

Note: the value of the counter used by the microcontroller is one that is in the RAM of the values of the labels. The application can therefore modify the current value of the counter at the same time as updating the data.

Reception :

The microcontroller identifies the last label of the capsule when it is received (recall the circulation order of the labels is not necessarily the order of calculation), which triggers the calculation of the checksum with words previously received. After each calculation a pre determined event is inserted in the RT FIFO or the local FIFO the way to indicate the valid / invalid result of the capsule.

There are several cases to consider (considering the value of the "data" field of S_A429_MESSAGE structure):

- Event OK: the received checksum corresponds to the checksum calculated, and the counter that is incremented by a value between 1 and 2.
- Event not OK flag A429_EVENTUC_ERROR1 present in the value of the event: the received checksum is different from the calculated checksum.
- Event not OK, flag A429_EVENTUC_ERROR2 present in the value of the event: the counter is not equal to the previous counter + 1 or 2 +.
- No event: the microcontroller is not able to detect the last label of the capsule: no traffic, reception errors ...

Note: The counter value is checked only if the checksum of the capsule is correct. Therefore, it is not possible to have both errors simultaneously. The first correct capsule received after a429RxStart () or after capsule with a checksum error cannot have counter error, and the counter value of the capsule is taken as the initial value counter for the following.

Definition / modification of a capsule :

The capsule is defined by the S_A429_UCCKS1760 structure:

idCks	Identifier of the capsule for the microcontroller, so that this description can be modified later. This identifier is independent of the capsule identifier circulating on the bus.
IdEvent	number of the event used in transmission in the frame, or placed in a FIFO in reception. It is required that this identifier is of 16 bits (between 0x0 and 0xFFFF). In reception, default or if the flag A429_EVENTUC_RTFIFO is added to the event it will be inserted in the RT FIFO. Using the flag A429_EVENTUC_CHANFIFO it will be inserted in the local FIFO of the channel.
idLabel	The label containing the capsule identifier.
msbCksLabel	The checksum high-value label.
lsbCksLabel	The checksum low value label.
itemCount	The number of data labels involved in the calculation of the checksum.
iBuffer	Not used by the application.
lastLabel	In reception, the detection of this label triggers the calculation and verification of the checksum. This should be the last label of the capsule, so the checksum LSB label.
counterLabel	The label that contains the counter of the capsule.
counterIncr	The increment of the counter. In transmission this increment is normally 1, but can be modified to generate protocol errors without generating a checksum error.
flags	0 for the creation of an emission capsule To create a reception capsule, it is possible to use the flag A429_ATTR_CKSNOEVENTOK. It allows not generate events when a capsule without error is received, but keep the normal behavior when there is a checksum error or counter error, which is to generate an event. In transmission it is possible to modify an existing capsule by designating *idCks*, and using the flag A429_ATTR_ADD accompanied by one of the following flags:

A429_ATTR_CKSINCR: Changes the value of the counter increment, which can generate a sequence of invalid values.

A429_ATTR_CKSERRON: Specifies to generate a false checksum for this capsule.

A429_ATTR_CKSERROFF: Specifies to stop the generation of false checksum, and thus to generate valid checksum.

These modifications can be made while the channel is operating.

When creating a capsule, the *itemCount* parameter must be non-zero, and the *pData* parameter of *a429UcOp()* function should point to an array of *itemCount* data labels that will be used **in this order** to calculate the checksum.

When modifying a capsule (A429_ATTR_ADD present in flags) *itemCount* must be 0, and the parameter *pData* of *a429UcOp()* function to NULL.

3.10.6 A429_UCOP_CLEAR

Removes all registered operations for a channel.

Note: If a value or attribute forcing is in progress on a label, the label retains the value and / or the forced attribute.

Example:

```
S_A429_UCOP         op ;

op.operation        = A429_UCOP_CLEAR ;
op.channel = 2 ;

a429UcOp (hCard, & op, data) ;
if (op->result != A429_ENONE)
       printf ("erreur a429UcOp\n") ;
```

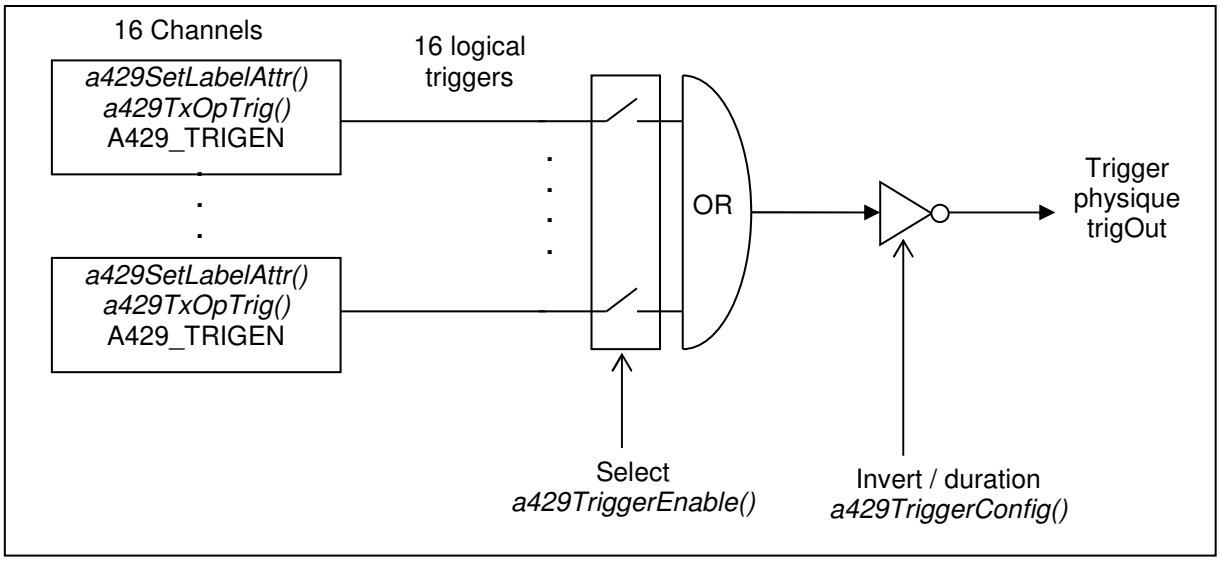
3.11 External signals

3.11.1 Trigger output

The board has a physical output called trigOut . This output is controlled by a logical OR which combines the trigger signals from all the transmit and receive paths.

As the logic output is shared between all the labels of all channels. We must use the trigger filtering capability to generate the desired signal

- The generation of the trigger for a label recognized in reception is configured by y *a429SetLabelAttr()*. This allows to specify which labels of each channel should generate a logic trigger.
- Generation of a trigger by an emission channel is obtained by inserting a *a429TxOpTrig()* operator in the frame.
- Trigger generation can be enabled or disabled at each channel during configuration using the A429_TRIGEN flag. This allows to specify the channels whose logical trigger will be taken into account to generate the external trigger.
- A centralized validation of triggers is handled by *a429TriggerEnable()*. This makes it possible to select, independently of the configurations of the channels and the labels, the channels whose logical triggers will participate in the elaboration of the physical trigger output trigOut..



The duration of the physical signal is programmable up to 255 microseconds.

The physical output signal can be inverted, which allows to choose the rest level of the line. If the line is at rest at 0 the trigger pulse will be positive if the rest level of the line is at level 1 the trigger pulse will be negative.

3.11.2 Clock input

The board has a clock input that can clock the cycles of channels in transmission.

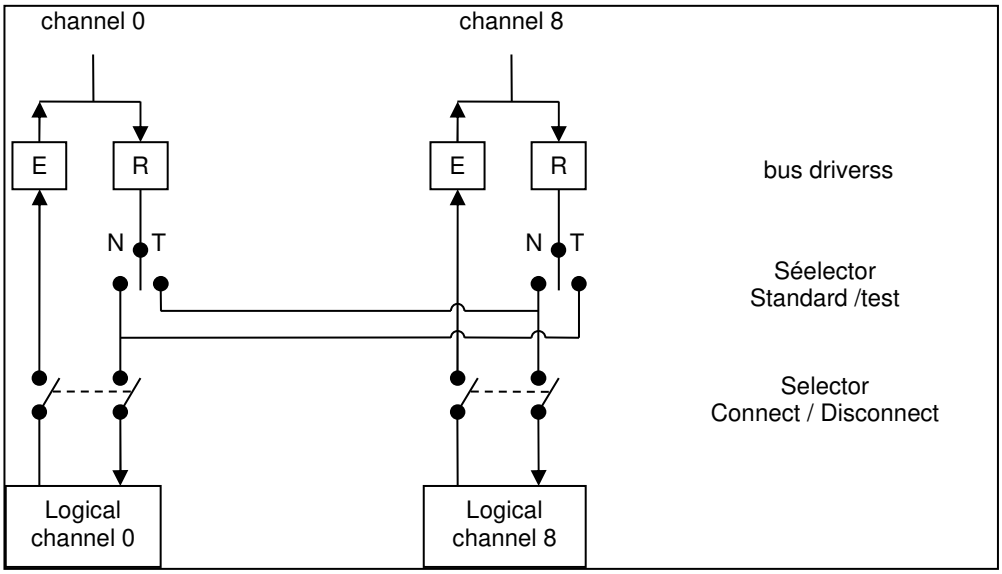
The clock signal can be inverted, which amounts to managing the active edge. Each transmitter chooses the clock that it uses: the internal timer or the external clock (a429TxConfig ()).

4 TESTS

The board offers the possibility of testing the entire operation to the connector, without the need for a loopback cable. The principle of the test is logically looping transmitters and receivers pathways with *a429SetTestMode()* function, and then to transmit on channels transmit and verify the traffic with the reception channels

Each channel has a transmitter and a receiver, but that cannot be used simultaneously. However at the connector the transmit and receive buses of the same channel are connected. The test mode takes advantage of this connection.

The a429SetTestMode () function requests the board to connect the emitters of the channels 0 to 7 to receptor of channels 8 to 15, and the emitters of the channels 8 to 15 to receptors on channels 0 to 7



This diagram shows the interconnections between channels 0 and 8:

In the Standard position, the receive bus is connected to its own channel.

On Test position the receiving bus is connected to the other channel. In this position it is possible to transmit on channel 0 and simultaneously receive the traffic on the channel 8 and vice versa.

In Test mode, the physical transmit and receive drivers of the transmit channel are used and therefore tested.

Principle of the test:

- Switch to test mode with *a429SetTestMode()*.
- Configure channels 0 to 7 in transmission and channels 8 to 15 in reception.
- Emit and check what is received.
- Configure channels 0 to 7 in reception and channels 8 to 15 in transmission.
- Emit and check what is received.
- Switch to normal mode with *a429SetTestMode()*

Note: During the test, there is no physical disconnection of the board to the outside: the emissions can be found on the connector.

If equipment in emissions are connected, or a bus is short-circuited the test may fail despite a serviceable card.

Note: Connecting / Disconnecting channel function can be used in the test mode.

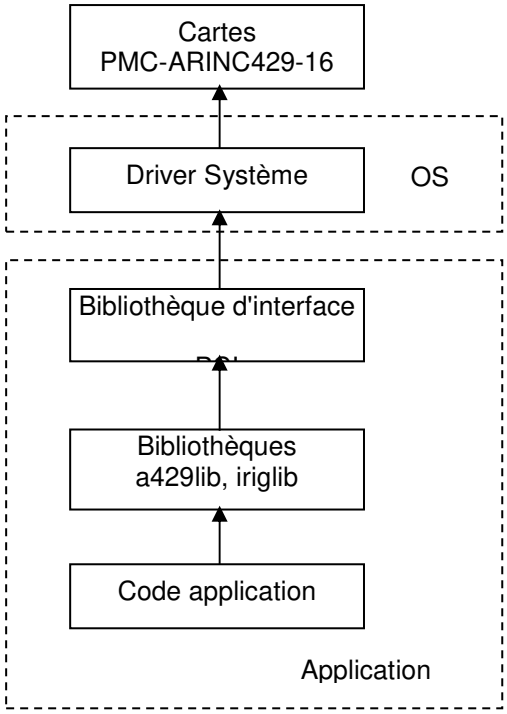
5 SOFTWARE LIBRARY

5.1 Driver and libraries

The implementation of the ADAS ARINC 429 board library is designed to hide the underlying system to applications.

Internally the library uses access libraries to the PCI driver.

For information all the software components of an application using PMC-ARINC429-16 boards is shown in the following diagram. One notices that the application communicates only with a429lib iriglib and libraries.



5.2 Multi boards management

The system can contain one or more PMC-ARINC429-16 cards, and the library is adapted to these cases. Cards must be addressed by rank: The first PMC-ARINC429-16 board is numbered 0, the next numbered 1, etc. See *a429Open()*.

5.3 Library implementation

For the realization of the application software, the implementation library has an API in C.

The objects in the library are prefixed with "A429" or "A429_" for constants.

The library consists of the following files:

a429lib.h	Interface file of the library to be included by the application.
a429lib.lib	File to bind with the application, required.
iriglib.h	Interface file of the IRIG library to be included by the application, if the IRIG-B features are used by the application.
iriglib.lib	File to bind with the application, required.
vclib.h	Declaration of types and utilities.
vclib.c	Some utilitarian functions.
a429lib	dll required at runtime.
iriglib.dll	Needed at runtime.

5.4 Library a429lib specification

5.4.1 Error management

Most functions of the library a429lib return an error code to indicate the result of the requested operation. A message explaining the error can be obtained with the *a429ErrorMessage()* function.

Most functions of the library iriglib return a 0 (FALSE) or 1 (TRUE) to indicate the result of the requested operation. In case of error, the function returns FALSE, and additional information can be obtained with the *irigLastError()* function which gives the last error number.

5.4.2 S_A429_MESSAGE structure

This structure contains all information about a message ARINC 429 acquired by the monitor, the FIFO common real time or local FIFO for each channel.

5.4.3 Error codes

The error codes used by the library are:

0.	a429_ENONE	No error.
1.	a429_ENOTINIT	Uninitialized library (VME only).
2.	a429_EOPENDEVICE	Error opening the device.
3.	a429_EARG	An argument of the function has an invalid value
4.	a429_EMEMORY	Memory allocation error.
5.	a429_ESTATE	Function prohibited in the current state of the channel (reception or transmission not started or example).
6.	a429_ETOODATA	Too much data for an update block or random messages.
7.	a429_ETHREAD	Error creating a thread.
8.	a429_EALREADYEXIST	Spy already started.
9.	a429_ECARDNOTFOUND	ARINC 429 board was not found for any of the following reasons: There is no board in the system or the number provided is too high. The function number in the PLX PROM is invalid. The value of the FPGA identification register is invalid.
10.	a429_EREGINTR	*PcigRegisterUserInt* error.
11.	a429_EINITDMA	*PcigInitDma* error.

12. a429_ESTARTDMA Error PcigIStartDma.
13. a429_ESPYNOTREADY The spy function is not in the expected state.
14. a429_EFRAMETOOBIG The frame it is asked to build is too large compared to the RAM of the board .
15. a429_EBUSY The specified block is busy.
16. a429_ESHAREMEM Shared memory allocation error.
17. a429_EBADHANDLE Invalid handle.
18. a429_EINUSE Function already acquired by another process.
19. a429_ESHAREMUTEX Mutex allocation error.
20. a429_EUNKNOWN Unknown error.
21. a429_EUCCOMM Communication error with the microcontroller.

5.5 a429lib user manual

5.5.1 a429LibVersion ()

Syntax:

```
uint32_t a429LibVersion (void)
```

Description:

Returns the version and revision number s of a429lib library, in the two most significant bytes of the return value.

Example of use:

```
uVal = a429LibVersion () ;
printf ("a429lib V%lu.%lu\n", uVal >> 24, (uVal >> 16) & 0xFF);
```

5.5.2 a429GetCardCount ()

Syntax:

```
uint32_t a429GetCardCount (void)
```

Description:

Returns the count of PMC AR429 board in the system.

5.5.3 a429ErrorMessage ()

Syntax:

```
char * a429ErrorMessage (uint32_t error)
```

Description:

This function returns a pointer to a string explaining the error whose number is passed as parameter.

5.5.4 a429Open ()

Syntax:

```
uint32_t a429Open (   HA429       * phCard,
                      uint32_t    cardNumber,
                      uint32_t    flags) ;
```

Description:

Open the ARINC 429 board whose number is specified

phCard	A pointer to an HA429 object that will be filled in by the function.
cardNumber	The index of the ARINC 429 board to open, from 0 to N-1, where N is the number of cards in the system.
flags	Indicators on the use of the board by the process that opens it: A429_OPENRTFIFOEN if the process that opens the board is the one that will manage the interruption of the real-time FIFO. A429_OPENSPYEN if the process that opens the board is the one that will manage the monitoring of the board .

When you first open the board all RT FIFOs and monitor FIFO are empty and channels are inactive. But as it is possible for multiple processes to open the same board , and use the same channels, the following openings leave the board in the state.

Several processes can open the same board , and use the same channels, but only one can manage real time FIFO or monitor FIFO of this card.

A A429_ENONE return value indicates that the board is open, and the value pointed by *phCard* is a handle which must be specified as the first argument in most other functions of the a429lib library.

It is necessary to use *a429Close()* when the board is no longer needed.

A return value different from A429_ENONE is an error number, and the board is not open.

Notes:

If the IRIG-B generator of the board is used, or if there is no signal connected to the receiver, it is necessary to initialize them after powering the system.

To do that:

```
HANDLE hIrig = a429IrigHandle (hCard) ;
irigSetTxLocalDate (hIrig) ;
irigSetLocalDate (hIrig) ;
irigSetLocalYear (hIrig) ;
```

5.5.5 a429Close ()

Syntax:

```
void a429Close (HA429 hCard)
```

Description:

Close ARINC 429 board and releases the resources it uses. But the library is still usable

5.5.6 a429CardVersion ()

Syntax:

```
uint32_t a429CardVersion (HA429 hCard)
```

Description:

This function returns the version / release of FPGA design in its most significant word, and version / revision of the microcontroller software in its low less significant word..

Example of use:

```
uVal = a429CardVersion (hCard) ;
printf ("FPGA Design V%d.%d  IP V%d.%d\n\n",
            uVal >> 24, (uVal >> 16) & 0xFF,
            (uVal >> 8) & 0xFF, uVal & 0xFF) ;
```

5.5.7 a429Reset ()

Syntax:

```
Uint32_t a429Reset (HA429 hCard)
```

Description:

This puts the board in its original condition, as after a power-up: All non-configured channels, empty FIFOs, No update / random (all blocks are free).

Note: This also resets the microcontroller of the board , which takes about 1.5 seconds to configure.

The function returns A429_ENONE if successful, otherwise an error number.

5.5.8 a429IrigHandle ()

Syntax:

```
HANDLE a429IrigHandle (HA429 hCard)
```

Description:

This feature provides the necessary HANDLE to use IRIG-B features of the ARINC-429 board : all irigXXX() functions of the iriglib library.

A NULL handle indicates an error.

5.5.9 a429TriggerConfig ()

Syntax:

```
uint32_t a429TriggerConfig (HA429          hCard,
                            uint32_t       length,
                            uint32_t       flagsT0,
                            uint32_t       flagsT1) ;
```

Description:

Output trigger configuration:

hCard	Board Handle.
length	Pulses times from 1 to 255 microseconds.
flagsT0	Configuration of Output 0.
flagsT1	Configuration of Output 1.

The flags are a combination of the following values:

A429_INVERT Inversion of the output signal (default : "1").

Example. Configure a duration of 255 µs, trigger output 0 inverted, and trigger output 1 normal:

```
arincTriggerConfig (hCard, 255, A429_INVERT, 0) ;
```

The function returns A429_ENONE if successful, otherwise an error number.

5.5.10 a429TriggerEnable ()

Syntax:

```
uint32_t a429TriggerEnable (HA429          hCard,
                            uint32_t       enableT0,
                            uint32_t       enableT1)
```

Description:

Used to indicate the channels for which the trigger signal is transmitted to the physical outputs.

hCard	Board Handle.
enableT0	Output 1 mask.
enableT1	Output 2 mask.

In the masks each bit between 0 and 15 corresponds to one channel. It is possible to combine the triggers of several channels.

Example. Configure the trigger0 output so that only the trigger of channel 2 is transmitted (trigger output 1 is not used):

```
A429TriggerEnable (hCard, 1 << 2, 0) ;
```

The function returns A429_ENONE if successful, otherwise an error number.

5.5.11 a429ClockConfig ()

Syntax:

```
HANDLE a429ClockConfig (   HA429     hCard,
                           uint32_t  flagsC0,
                           uint32_t  flagsC1) ;
```

Description:

Trigger output configuration.

hCard	Board Handle.
flagsC0	Input 1 configuration
flagsC1	Input 2 configuration

The flags are a combination of the following values:

A429_INVERT input signal inversion, active on rising edge.

Example. Validate the inverted clock 0 and the normal clock 1:

```
a429ClockConfig (hCard, A429_INVERT, 0) ;
```

The function returns A429_ENONE if successful, otherwise an error number.

5.5.12 a429Connect ()

Syntax:

```
HANDLE a429Connect (   HA429      hCard,
                       uint32_t   channel,
                       uint32_t   bConnect) ;
```

Description:

Enable to connect the channel to the board.

hCard	Board Handle.
channel	Channel number from 0 to 15.
bConnect	1 to connect, 0 to disconnect.

This function must be called after the configuration of the channel, because it is necessary to know if it is necessary to connect / disconnect the emission or the reception.

The function returns A429_ENONE if successful, otherwise an error number.

5.5.13 a429SetTestMode ()

Syntax:

```
HANDLE a429SetTestMode (HA429 hCard, uint32_t bEnable) ;
```

Description:

This switches the board in test mode or normal mode.

hCard	Board Handle.
bEnable	1 pour le mode test, 0 pour le mode normal.

This switches the board into a mode that allows testing operation without a loopback cable (see § TESTS).

The function returns A429_ENONE if successful, otherwise an error number.

5.5.14 a429RtFifoEnable ()

Syntax:

```
uint32_t a429RtFifoEnable (HA429     hCard,
                           uint32_t  bEnable)
```

Description:

This feature allows acquiring the exclusive use of the RT FIFO for the calling process, and should be used before waiting for an interrupt or read the contents of the FIFO.

hCard	Board Handle. The flag A429_OPENRTFIFOEN must be specified when opening the board , otherwise the A429_ESTATE error will be returned.
bEnable	1 enable RT FIFO, 0 release it.

This function performs a reset of the RT FIFO, which is thus cleared and the overflow bit cleared.

The function returns A429_ENONE if successful, otherwise an error number.

5.5.15 a429RtWaitForInt ()

Syntax:

```
uint32_t a429RtWaitForInt (HA429          hCard,
                           uint32_t       timeout)
```

Description:

This function asleep the calling thread until the occurrence of an interrupt generated by the non-empty real time RT FIFO.

hCard	Board Handle.
timeout	The timeout value in milliseconds.

If there has been an interrupt use a429RtFifoRead () until the FIFO is empty, that is the function returns A429_NOTRECEIVED.

Note for VMAE users: interrupt handling depends strongly on the available VME library. In particular the management of the timeout will be implemented only if the VME library allows. On the other hand if this function is not feasible it will be replaced by a429RtSetIntCallback ().

You must have acquired the use of the RT FIFO with *a429RtFifoEnable()* before using this function.

The function returns A429_IT_OK if an interrupt occurred, A429_IT_TMO if there was a timeout, or an error number.

5.5.16 a429RtSetIntCallback ()

Syntax:

```
uint32_t a429RtSetIntCallback (HA429      hCard,
                               void       * callback,
                               uintptr_t  arg)
```

Description:

This function indicates the function that will be called after an interrupt generated by the not empty RT FIFO.

hCard	Board Handle.
callback	pointer of the function to call
arg	Argument passed to the callback function.

The called function has prototype:

```
void callback (
               uint32_t  event,      //A429_CBMESSAGE ou A429_CBOVFL
               S_A429_MESSAGE  * pData,    // Message
               uintptr_t arg)  // Parametre utilisateur
```

If event = A429_CBMESSAGE the callback is called for an ARINC message provided in pData.

If event = A429_CBOVFL, there was a FIFO overflow and the callback will no longer be called. This is a serious error: to re-initialize the RT FIFO you have to close the board and re-open it.

Arg is the third argument of *a429RtSetIntCallback()*.

To remove the callback and free the use of the FIFO provide callback = NULL.

Using *a429RtSetIntCallback()* and *a429RtWaitForInt()* is exclusive: only one of them must be used.

The function returns A429_ENONE if successful, otherwise an error number.

5.5.17 a429RtFifoRead ()

Syntax:

```
uint32_t a429RtFifoRead (    HA429              hCard,
                             S_A429_MESSAGE     * pData,
                             uint32_t           count,
                             uint32_t           * pReadCount)
```

Description:

This function reads the data available in the real-time FIFO:

hCard	Board Handle.
pData	A pointer to an array of S_A429_MESSAGE structures in which to read the data.
count	Maximum number of structures available in pData.
pReadCount	When returning from the function the pointed word contains the number of structures actually read and available in *pData*.

The return value is an execution report:

A429_ITEMOK	This value is returned when messages could be read, their number is provided by the word pointed to by *pReadCount*.
A429_NOTRECEIVED	There is no more message to read in the receiving FIFO at the moment, Re-try later.
A429_WRITEOVER	The read was not fast enough, words were lost in the receive FIFO. The FIFO was emptied and researched for sync for the following receptions. Many ARINC words may have been lost (all the depth of the FIFO).
A429_OUTOFSYNC	The message stream is desynchronized, the message count specified by *readCount* is exploitable. At the next call, there will be an attempt to resynchronize.

If the use of the RT FIFO was not previously acquired with *a429RtFifoEnable()* function returns the error A429_ESTATE.

5.5.18 a429ChanFifoRead ()

Syntax:

```
uint32_t a429ChanFifoRead (  HA429              hCard,
                             uint32_t           channel
                             S_A429_MESSAGE     * pData,
                             uint32_t           count,
                             uint32_t           * pReadCount)
```

Description:

This function reads the data available in the local FIFO of a channel:

hCard	Board Handle.
channel	Channel number, between 0 and 15.
pData	Pointer to an array of S_A429_MESSAGE structures in which to read the data.
count	Maximum number of structures available in pData.
pReadCount	When returning from the function the pointed word contains the number of structures actually read and available in *pData*.

The return value is one of the following values:

A429_ITEMOK	This value is returned when messages could be read, their number is provided by the word pointed to by pReadCount.
A429_NOTRECEIVED	There is no message to read in the receive FIFO for the moment, Re-try later.

A429_WRITEOVER	The read was not fast enough, words were lost in the receive FIFO. It is necessary to reset the FIFO before being able to obtain data again.
A429_OUTOFSYNC	The message stream is desynchronized, the message count specified by readCount is exploitable. At the next call, there will be an attempt to resynchronize

How to reset Local FIFO: If the function is called with pData = NULL, performs a reset of the local FIFO without reading data (count and pReadCount are ignored) and returns A429_NOTRECEIVED.

Example: a429ChanFifoRead (hCard, chan, NULL, 0, NULL) ;

5.5.19 a429SetLabelAttr ()

Syntax:

```
uint32_t a429SetLabelAttr (    HA429       hCard,
                               uint32_t    channel,
                               uint32_t    * pLabel,
                               uint32_t    * pAttributes,
                               uint32_t    count)
```

Description:

This function modify the attributes associated with labels. Theses attributes onfigure different behaviors.

The words of *pAttributes* array indicate the attributes to be modified for each corresponding label *pLabel* array. Each attribute word must contain one of the values that specifies the operation to perform:

A429_ATTR_ADD	to add an attribute.
A429_ATTR_REMOVE	to remove an attribute.

Usable attributes are a combination of the following bits (some are incompatible) :

RX	TX	Bits	Description
X	X	A429_ATTR_ADD	Add the attributes provided to the attributes of the label.
X	X	A429_ATTR_REMOVE	Remove the attributes provided to the attributes of the label.
X	X	A429_ATTR_CHANFIFO	This label must be placed in the local FIFO of the channel.
X	X	A429_ATTR_SPYFIFO	This label must be placed in the common monitor FIFO. This attribute is set by default when configuring the channel for all labels.
X	X	A429_ATTR_RTFIFO	This label must be placed in the common real-time FIFO.
X	X	A429_ATTR_RXTRIG	Reception of this label generates a trigger.
	X	A429_ATTR_DISABLE	Inhibition of the emission of this label. Exclusive with A429_ATTR_DISABLET
	X	A429_ATTR_DISABLET	Inhibition preserving the timing. Exclusive with A429_ATTR_DISABLE.
	X	A429_ATTR_EPARITY	Force the parity error
	X	A429_ATTR_EFRAME	Force the framing error : bit (11), Second part to 0.
	X	A429_ATTR_ESHORT	Force message too short: remove last bit

	X	A429_ATTR_ELONG	Force message too long: duplicate the last bit
	X	A429_ATTR_SEP1 A429_ATTR_SEP2 A429_ATTR_SEP3 A429_ATTR_SEP4	Number of inter-message silence bits: 1 to 4 bits Default value is A429_ATTR_SEP4. Action A429_ATTR_REMOVE on attribute resets the value to A429_ATTR_SEP4 Note : These attributes are encoded using 2 bits.

count number of useful values in *pLabel* and *pAttributes*.

An attribute change may be performed at any moment, and repeatedly, once the channel is configured.

The time taken into account is variable. Attribute management can be shared by the application and the local board microcontroller. As there must be coherence between these two actors:

- The application must not modify the same attributes as the microcontroller that has been asked to force attributes (A429_UCOP_FORCE) for example.
- The library delegates the modification of the attributes to the microcontroller using the operation A429_UCOP_SETATTR. The execution of this operation may take a variable time (not measured to date, but estimated less than 100 micro seconds).

The function returns A429_ENONE if successful, otherwise an error number.

5.5.20 a429BaudRate ()

Syntax:

```
uint32_t a429BaudRate (double frequency)
```

Description:

This function calculates the value of the *speed* parameter to supply the functions *a429RxConfig()* and *a429TxConfig()*.

The parameter is the desired frequency in hertz. To program the standard frequency of 12.5 KHz it is necessary to use *a429BaudRate(12500)*.

5.5.21 a429CycleDiv ()

Syntax:

```
uint32_t a429CycleDiv (double frequency)
```

Description:

This function calculates the value of the parameter cycleDiv to provide the functions *a429TxConfig()*, from the desired cycle frequency in Hz.

To program the cycle frequency to 100 Hz use *a429BaudRate(100)*.

The permissible cycle frequency range is 0.1 Hz to 2000 Hz

The internal clock that is divided to provide the channels cycles frequencies of is common to all channels. Therefore the cycles of the different channels are isochronous they do not arrive at the same time, but they do not shift against each other over time.

It is possible to introduce "slippage" between channels by altering the value provided by a429CycleDiv ().

The cycle generator resolution is 31.25 µs

Example 1: If the value is decreased by 1, the cycle frequency increases because the cycle period decreases of 31.5 microseconds. For a cycle frequency of 100Hz, the function a429CycleDiv () returns 320. If this value is changed to 319 before being passed to a429TxConfig (), the effective cycle frequency will be 100.31 Hz (period of 9968.75 µs instead of 10000µs, a difference of 0.31%).

Example 2: If the value is increased by 1, the cycle frequency decreases because the cycle period increases by 31.5 microseconds. For a cycle frequency of 10Hz, the function a429CycleDiv () returns 3200. If this value is changed to 3201 before being passed to a429TxConfig (), the effective cycle frequency will be 9.9969 Hz (100031.25 µs period instead of 100000 µs, a difference of 0.031%).

5.5.22 a429RxConfig ()

Syntax:

```
uint32_t a429RxConfig (   HA429       hCard,
                          uint32_t    channel,
                          uint8_t     * pSdi,
                          uint32_t    speed,
                          uint32_t    flags)
```

Description:

Configure a receiver before it starts

hCard	Board Handle.
channel	Channel number, between 0 and 15.
pSdi	An array of 256 characters, one per label, or NULL.
speed	Baud rate divider for bit frequency. To be entered using the function *a429BaudRate (double frequency)*, with frequency in Hertz.
flags	Combination of the following indicators:

A429_NOPARITY	Bit 32 is a data bit and not the parity bit of the word
A429_EVENPARITY	Use even parity
A429_ODDPARITY	Use odd parity
A429_SFIFOEN	Allow sending received words in the common monitoring FIFO.
A429_TRIGEN	Allow trigger generation of this channel on label recognition.

If *pSdi* is NULL, labels descriptors are initialized to a default value: No label uses SDI field, and all the labels are eligible for monitoring FIFO.

If *pSdi* in non-NULL, it points to a table in which each non-null character indicates that for the corresponding label (8-bit) you must use the SDI field to put the word in the table of values received or transmitted. In this case also all the labels (on 10 bits) are eligible for the monitoring FIFO.

After the configuration the local channel FIFO is empty, the channel is stopped and connected, the trigger is inhibited. Configuration is a good way to re-initialize a channel in a known state.

When used, parity is calculated by the firmware of the board , and replaces the 32nd bit of the ARINC words.

The function returns A429_ENONE if successful, otherwise an error number.

5.5.23 a429RxErrorCount ()

Syntax:

```
uint32_t a429RxErrorCount (HA429          hCard,
                           uint32_t       channel,
                           uint32_t       * pCount) ;
```

Description:

This function is used to retrieve the reception error count on the channel. The error count of the channel in the board is reset to 0 on each reading. So at each reading is obtained the error count since the previous reading.

Apart from the operation of the monitoring stream, this error counter is the only way to know that there are reception errors on a channel: the erroneous words are not put in the FIFOs, except the monitoring FIFO.

The counter has 16 bits

hCard	Board Handle.
channel	Channel number, between 0 and 15.
pCount	A pointer to where to place the error count.

The function returns A429_ENONE if successful, otherwise an error number.

5.5.24 a429RxStart ()

Syntax:

```
uint32_t a429RxStart (HA429 hCard, uint32_t channel)
```

Description:

This function starts reception on the specified channel.

hCard	Board Handle.
channel	Channel number, between 0 and 15.

It is possible to initialize the values of the ARINC words before starting the reception with *a429RxWrite()*.

The function returns A429_ENONE if successful, otherwise an error number.

5.5.25 a429RxStop ()

Syntax:

```
uint32_t a429RxStop (HA429 hCard, uint32_t channel)
```

Description:

This function is used to stop reception on the specified channel.

hCard	Board Handle.
channel	Channel number, between 0 and 15.

The function returns A429_ENONE if successful, otherwise an error number.

5.5.26 a429RxRead ()

Syntax:

```
uint32_t a429RxRead (HA429          hCard,
                     uint32_t       channel,
```

```
                                uint32_t    label,
                                uint32_t    * pData,
                                uint32_t    * pFlag) ;
```

Description:

This function retrieves the last received value for the specified label. If no value has been received for this label, the value 0 or the value initialized by *a429RxWrite()* is returned, with a null flag

hCard	Board Handle.
channel	Channel number, between 0 and 15.
label	Label value, including SDI if necessary.
pData	A pointer to where to place the received word.
pFlag	A pointer to a word to place the refresh indicator of the value. If at the end of the reading the indicator is zero, it means that there has been no reception of this label since the previous reading.

The function returns A429_ENONE if successful, otherwise an error number.

5.5.27 a429RxWrite ()

Syntax:

```
uint32_t a429RxWrite (   HA429          hCard,
                         uint32_t       channel,
                         uint32_t       data)
```

Description:

This function is used to initialize one word of the the table of received words once the channel has been configured for reception, and before starting reception.

hCard	Board Handle.
channel	Channel number, between 0 and 15.
data	ARINC word to initialize

The data word is stored in its place taking into account the label and possibly the value of the SDI field.

The function returns A429_ENONE if successful, otherwise an error number.

5.5.28 a429RxWrites ()

Syntax:

```
uint32_t a429RxWrites (   HA429       hCard,
                          uint32_t    channel,
                          uint32_t    *pData,
                          uint32_t    count)
```

Description:

This function is used to initialize the table of received words once the channel has been configured for reception, and before starting reception.

hCard	Board Handle.
channel	Channel number, between 0 and 15.
pData	ARINC 429 word table to initialize.
count	The number of word in *pData*

The data words are stored in their place considering the label and possibly the value of the SDI field each.

The function returns A429_ENONE if successful, otherwise an error number.

5.5.29 a429TxConfig ()

Syntax:

```
uint32_t a429TxConfig (   HA429          hCard,
                          uint32_t       channel,
                          uint8_t        * pSdi,
                          uint32_t       speed,
                          uint32_t       flags,
                          uint32_t       cycleDiv)
```

Description:

Configure a transmit channel.

hCard	Board Handle.
channel	Channel number, between 0 and 15.
pSdi	An array of 256 characters, one per label, or NULL.
speed	Baud rate divider (bit frequency). To be entered using the function *a429BaudRate (double frequency)*, with frequency in Hertz.
flags	Combination of the following indicators:

A429_NOPARITY	Bit 32 is a data bit and not the parity bit of the word.
A429_EVENPARITY	Use even parity
A429_ODDPARITY	Use odd parity
A429_HIGH A429_LOW	To select the slew rate.
A429_CLKINT A429_CLKEXT	To select the clock input of clocking cycles.
A429_SFIFOEN	For some test benches it is important to be able to reconstruct the interactions between emissions and receptions. For this the A429_SFIFOEN flag indicates to transmitter channel to issue the words in the monitoring FIFO, with the same frame as the receipt (date, error indicators, etc.).
A429_TRIGEN	Allow trigger generation by the *a429TxOptrig()* operators of this channel.

cycleDiv	Divider of the clock source for cycle tops, between 1 and 8191. If the internal clock is selected, one can use the function *a429CycleDiv (double frequency)*, with frequency in Hz, to calculate the divider.

If *pSdi* is NULL, labels descriptors are initialized to a default value: No label uses SDI field, and all the labels are eligible for monitoring FIFO.

If *pSdi* is non-NULL, it points to a table in which each character whose bit 0 is 1 indicates that for the corresponding label (of 8 bits) must be used SDI field for storing the word in the table of values received or issued . In this case also all the labels (on 8 or 10 bits) are eligible for the monitoring FIFO spy.

After the configuration the channel local FIFO is empty, the channel is stopped and connected, the trigger is invalidated. Configuration is a good way to re-initialize a channel in a known state.

When used, the parity is computed and checked by the firmware of the board at the time of issue. Words placed in FIFOs do not have parity.

The function returns A429_ENONE if successful, otherwise an error number.

5.5.30 a429TxRead ()

Syntax:

```
uint32_t a429TxRead (HA429          hCard,
 uint32_t               channel,
                        uint32_t    label,
                        uint32_t    * pData) ;
```

Description:

This function retrieves the current value of the specified label from the board memory of a transmitter channel.

hCard	Board Handle.
channel	Channel number, between 0 and 15.
label	Label value, including SDI if necessary.
pData	A pointer to where to place the received word.

The function returns A429_ENONE if successful, otherwise an error number.

5.5.31 a429TxWrite ()

Syntax:

```
uint32_t a429TxWrite (   HA429        hCard,
                         uint32_t     channel,
                         uint32_t     data)
```

Description:

This function is used to initialize the values of the words to be sent, once the channel has been configured for transmission, and before starting transmission. It can also be used for immediate updating of values during operation.

hCard	Board Handle.
channel	Channel number, between 0 and 15.
data	ARINC word to update.

The word data is stored in its place taking into account the label, and possibly the value of the SDI field.

The function returns A429_ENONE if successful, otherwise an error number.

5.5.32 a429TxWrites ()

Syntax:

```
uint32_t a429TxWrites (   HA429        hCard,
                          uint32_t     channel,
                          uint32_t     *pData,
                          uint32_t     count)
```

Description:

This function initializes the values of the words to be transmitted once the transmit channel is configured, and before starting transmission. It can also be used to immediately update values during operation.

hCard	Board Handle.
channel	Channel number, between 0 and 15.
pData	ARINC 429 word table for update.
count	The count of words in *pData*

The data words are stored in their place taking into account the label and possibly the value of the SDI field.

The function returns A429_ENONE if successful, otherwise an error number.

5.5.33 a429TxSetFrame ()

Syntax:

```
uint32_t a429TxSetFrame (HA429          hCard,
                         uint32_t       channel,
                         uint32_t       * pFrame,
                         uint32_t       count)
```

Description:

Provide a transmission channel with a description of the cyclic transmissions to be carried out, in the form of a frame.

hCard	Board Handle.
channel	Channel number, between 0 and 15.
pFrame	The array of operators constituting the frame.
count	The number of operators in the *pFrame* array. Must be less than 28*1024.

The function returns A429_ENONE if successful, otherwise an error number.

5.5.34 a429TxOpXx ()

Syntax:

```
uint32_t a429TxOpCycle     (void)

uint32_t a429TxOpDelay     (uint32_t delay)

uint32_t a429TxOpData      (uint32_t label)

uint32_t a429TxOpEvent     (uint32_t id)

uint32_t a429TxOpEventUc   (uint32_t id)

uint32_t a429TxOpRandom    (uint32_t block)

uint32_t a429TxOpTrig      (void)

uint32_t a429TxOpUpdate    (uint32_t block)
```

Description:

These functions create the operators to use for creating a transmission frame.

a429TxOpCycle()	This operator allows you to wait for the next cycle start signal (from the internal timer or the external clock).
a429TxOpDelay()	The argument is the number of bits constituting the time period of 1 to 16384. This gives a maximum delay of 163 ms at a bit rate of 100 kHz, and 1310 ms for a bit rate of 12.5 kHz. This delay is in addition to the standard inter message delay.
a429TxOpData()	This operator indicates which word in the array of values to emit, not the value itself. The argument consists of the label, possibly completed with SDI bits. Example : Send the label (octal) 0312 with SDI 01: *a429TxOpData (0x100 | 0312)*
a429TxOpEvent()	The execution of this operator causes the real-time FIFO setting of an event frame whose data is the identifier of the event (on 16 bits). This allows the application to be to be notified of the transition to a particular point in the frame..
a429TxOpEventUc()	The execution of this operator causes the microcontroller FIFO setting an event frame whose data is the identifier of the event (on 16 bits). This allows the application to be notified of the transition to a particular point of the frame.
a429TxOpRandom()	The argument is the block number (0 to 7) that must be used to trigger a random transmit, if this block is validated. If the block is not validated, the operator does nothing.
a429TxOpTrig()	The execution of this operator causes the generation of a pulse on the trigger output of the board . The pulse will actually be generated if it is allowed both at the channel level (by *a429TxConfig()*) and at the board level (by *a429TriggerEnable()*).
a429TxOpUpdate()	The argument is the block number (0 to 7) that should be used for a synchronous update, if the block is validated. If the block is not validated, the operator does n.

These functions return a word to insert in the frame.

5.5.35 a429TxStart ()

Syntax:

```
uint32_t a429TxStart ( HA429       hCard,
                       uint32_t    channel,
                       uint32_t    bOneShot)
```

Description:

This function starts the transmission on the specified channel.

hCard	Board Handle.
channel	Channel number, between 0 and 15.
bOneShot	0 for loop operation of the frame until stopped by the application. 1 to execute the frame only one time, see *a429TxCheckEnd()*.

It is possible to initialize the values of ARINC words before starting the transmit with *a429TxWrite()*.

The function returns A429_ENONE if successful, otherwise an error number.

5.5.36 a429TxStop ()

Syntax:

```
uint32_t a429TxStop (HA429 hCard, uint32_t channel)
```

Description:

This function stops the transmission on the specified channel.

hCard	Board Handle.
channel	Channel number, between 0 and 15.

The function returns A429_ENONE if successful, otherwise an error number.

5.5.37 a429TxCheckEnd ()

Syntax:

```
uint32_t a429TxCheckEnd (HA429 hCard, uint32_t channel)
```

Description:

Lets you know if a transmit channel has finished executing the frame that was started with the *bOneShot* parameter of *a429TxStart()* to 1.

hCard	Board Handle.
channel	Channel number, between 0 and 15.

The function returns A429_ENONE if the channel is stopped, otherwise A429_EBUSY, or another error number.

5.5.38 a429TxUpdate ()

Syntax:

```
uint32_t a429TxUpdate (HA429      hCard,
                       uint32_t   channel,
                       uint32_t   * pData,
                       uint32_t   count,
                       uint32_t   block)
```

Description:

Make a request to synchronously update the messages to be sent for the specified channel.

hCard	Board Handle.
channel	Channel number, between 0 and 15.
pData	The array of values to be transmitted on the bus (label, sdi, parameter …)
count	Word count in *pData*.
block	Number of the buffer to use, from 0 to 7.

The function is not blocking, and returns after writing the update request. If the buffer is not available the function fails. The *a429TxCheckUpdate()* function can be used to check the availability of a block used by the synchronous update.

If this function has not been used before the corresponding *OpUpdate* operator of the frame is executed, the operator remains inactive.

The function returns A429_ENONE on success, otherwise an error number.

5.5.39 a429TxCheckUpdate ()

Syntax:

```
uint32_t a429TxCheckUpdate (HA429      hCard,
                            uint32_t   channel,
                            uint32_t   block)
```

Description:

Lets you know if a synchronous update buffer is available.

block	Number of the buffer to use, from 0 to 7.

The function returns A429_ENONE if the block is free, otherwise A429_EBUSY, or another error number.

5.5.40 a429TxRandom ()

Syntax:

```
uint32_t a429TxRandom (HA429      hCard,
                       uint32_t   channel,
                       uint32_t   * pFrame,
                       uint32_t   count,
                       uint32_t   block)
```

Description:

Requests the random issue of a data buffer on the specified channel.

hCard	Board Handle.
channel	Channel number, between 0 and 15.

pFrame	The array of operators to use for the random transmit.
count	Count of operators in *pFrame*.
block	Number of the buffer to use, from 0 to 7.

The *pFrame* array must be constructed similarly to the cyclic frame using operators, and in particular *a429TxOpData()*. The words issued are those of the array of values, previously updated with *a429TxWrite()*, and indicated by the operators *a429TxOpData()*.

The allowed operators are: *a429TxOpData()*, *a429TxOpEvent()*, *a429TxOpDelay()*, and *a429TxOpTrig()*.

The function is not blocking and returns as soon as the request is registered by the board . A new request with the same block cannot be made until the random issue of this block is complete. In this case the function returns with A429_EBUSY, without having recorded the request, nor disturbed the emission in progress. See *a429TxCheckRandom()*.

The function returns A429_ENONE on success, otherwise an error number.

5.5.41 a429TxCheckRandom ()

Syntax:

```
uint32_t a429TxCheckRandom (HA429       hCard,
                            uint32_t    channel,
                            uint32_t    block)
```

Description:

Lets you know if a random transmit buffer is free.

hCard	Board Handle.
channel	Channel number, between 0 and 15.
block	Number of the buffer to use, from 0 to 7.

The function returns A429_ENONE if the block is free, otherwise A429_EBUSY, or another error number.

5.5.42 a429TxBlockReset()

Syntax:

```
uint32_t a429TxBlockReset (HA429        hCard,
                           uint32_t     channel,
                           uint32_t     iBlock)
```

Description:

Allows you to request the cancellation of an *a429TxRandom()* or *a429TxUpdate()* operation on a block.

Canceling an operation on an unused block has no effect.

hCard	Board Handle.
channel	Channel number, between 0 and 15.
block	Number of the buffer to use, from 0 to 7.

This function can for example be used in the following case:

An operation *a429TxRandom()* or *a429TxUpdate()* was requested and then the channel was stopped. It is possible that the operation was not carried out before the stop of the channel, and that the operation is thus still recorded. In this case it will be performed at the next start of the

channel, which may not be desirable. Therefore you must ask to cancel operations on the blocks used by the channel before starting or after having stopped.

The function returns A429_ENONE.

5.5.43 a429UcOp ()

Syntax:

```
uint32_t a429UcOp (HA429          hCard,
                   uint32_t       channel,
                   S_A429_UCOP    * pOp,
                   uint32_t       * pData)
```

Description:

Allows you to transfer an operation to the micro controller of the board .

hCard	Board Handle.
channel	Channel number, between 0 and 15.
pOp	The address of the structure that describes the operation to be performed.
pData	The address of a data array needed for the operation, otherwise NULL.

To use this function, refer to chapter “Automatic operations”.

The function returns the "result" field of the structure, which has been filled in by the microcontroller, and contains A429_ENONE on success, otherwise another error number.

5.5.44 a429SpyStart ()

Syntax:

```
uint32_t a429SpyStart ( HA429     hCard,
                        uint32_t  bufferSize,
                        uint32_t  flags)
```

Description:

The monitoring function can be used independently of others. It acquires the use of the monitor function (if it is already acquired by another process the error A429_ESTATE is returned) and validates the monitoring on the board .

hCard	The handle provided by *a429Open()*. The flag A429_OPENSPYEN must have been specified when opening the board , otherwise the A429_ESTATE error will be returned.
bufferSize	Must be 0.
flags	A429_SPYDIRECT. If this flag is indicated the application does not use a DMA engine to empty the monitoring FIFO. This must be done by the application that calls the functions *a429SpyRead()* or *a429SpyReaderMessage()*.

For VME implementation, bufferSize is not used, and A429_SPYDIRECT is required.

The function returns A429_ENONE on success, otherwise an error number.

5.5.45 a429SpyStop ()

Syntax:

```
uint32_t a429SpyStop (HA429 hCard)
```

Description:

The spy function is stopped by this function and the resources are released.

hCard The handle provided by *a429Open()*.

The function returns A429_ENONE on success, otherwise an error number.

5.5.46 a429SpyRead ()

Syntax:

```
uint32_t a429SpyRead ( HA429           hCard,
                       S_A429_MESSAGE  * pData,
                       uint32_t        count,
                       uint32_t        * pReadCount)
```

Description:

This function is used to read the data available in the monitoring FIFO, if the monitor was started with the flag A429_SPYDIRECT. If the monitor was started without this flag, you must use a reader (See *a429SpyReaderOpen()*).

hCard	Handle of the board provided by *a429Open()*.
pData	A pointer to an array of S_A429_MESSAGE structures in which to read the data.
Count	The maximum number of structures available in *pData*.
pReadCount	When the function returns, this pointed word contains the number of structures actually read and available in *pData*.

The return value is an indicator on the status of the reading:

A429_ITEMOK	This value is returned when messages could be read, their number is provided by the word pointed to by *pReadCount*
A429_NOTRECEIVED	There is no more message to read in the receiving FIFO for now, try again later.
A429_OVERFLOW	The reader was not fast enough, words were lost in the receiving FIFO.
A429_OUTOFSYNC	The message stream is desynchronized, the message count specified by readCount is exploitable. At the next call, there will be an attempt to resynchronize.

If the monitoring was not started, or started without A429_SPYDIRECT the function returns A429_ESTATE.

5.5.47 a429SpyReaderOpen ()

Syntax:

```
HA429 a429SpyReaderOpen ( HA429    hCard,
                          uint32_t channel,
                          HA429    * phReader)
```

Description:

Multiple consumers (readers) can simultaneously use the monitoring feature. Each consumer has his own environment (position in the monitoring buffer),

A reader is created by specifying which channel of the board he wants to use.

During creation, the reader is initialized to read the next received ARINC 429 messages: it cannot access messages received prior to its creation

Readers must be created after the monitoring feature has been started by *a429SpyStart()*. They must be closed with *a429SpyReaderClose()* before stopping monitoring.

hCard	Handle of the board provided by *a429Open()*.
channel	The channel you want to read the messages. If A429_ALLCHAN is used, all monitored messages are delivered regardless of the source channel.
phReader	A pointer to the reader handle created by the function. This handle is for use with other functions *a429SpyReaderXxx()*.

The monitored data can be read with *a429iSpyReaderMessage()*.

The function returns A429_ENONE or an error number.

Note 1:

To get the first messages of a transaction, you have to start the monitor and create the reader before starting the reception.

Note 2:

If the monitor is created with the flag A429_SPYDIRECT, the spy does not use the DMA engine, and the function *a429SpyReaderMessages()*, must be called often enough to avoid an overflow of the monitoring FIFO of the board .

If the spy uses the DMA a large buffer is used, which frees the constraint of calling *a429SpyReaderMessages()* often. However it must be called often enough that the DMA buffer does not overflow.

5.5.48 a429SpyReaderReset ()

Syntax:

```
void a429SpyReset (HA429 hReader)
```

Description:

The reader receive index is re-initialized with the current index of the receiving thread. The next received ARINC 429 message will be the first that the reader can get with *a429SpyReaderMessages()*.

This function is used when the player has fallen far behind the receiver, the a429SpyReaderMessages() function has returned SPY_WRITEOVER.

hReader	The handle provided by *a429SpyReaderOpen()*.

5.5.49 a429SpyReaderClose ()

Syntax:

```
void a429SpyReaderClose (HA429 hReader)
```

Description:

The reader's resources are released, and it should no longer be used.

It is necessary to free the readers before stopping the monitoring by *a429SpyStop()*.

hReader	The handle provided by *a429SpyReaderOpen()*.

5.5.50 a429SpyReaderMessages ()

Syntax:

```
uint32_t a429SpyReaderMessages (HA429            hReader,
                                S_A429_MESSAGE   * pData,
                                uint32_t         count,
                                uint32_t         * pReadCount)
```

Description:

This function is used to obtain the following messages on the channel for which the reader was created.

The messages are copied into an array of S_A429_MESSAGE type structures whose address is provided by the caller.

hReader	The handle provided by *a429SpyReaderOpen()*.
pData	A pointer to the structure array in which the messages are placed.
count	Maximum number of structures available in *pData*.
pReadCount	When returning from the function this pointed word contains the number of messages actually read and available in *pData*.

The return value is an indicator of the status of the reading:

A429_ITEMOK	This value is returned when a message could be read.
A429_NOTRECEIVED	There is no more message to read in the receive buffer for now, try again later.
A429_WRITEOVER	The reader was not fast enough, words were lost in the receive buffer. *A429SpyReaderReset()* must be used before continuing.
A429_OVERFLOW	The reader was not fast enough, words were lost in the monitoring FIFO (can only happen if A429_SPYDIRECT is used). We must stop and start the monitor before continuing.
A429_OUTOFSYNC	The message stream is desynchronized, the message count specified by *readCount* is exploitable. At the next call, there will be an attempt to resynchronize.
Other value	An error number.

5.5.51 a429BlockAlloc ()

Syntax:

```
uint32_t a429BlockAlloc ( HA429        hCard,
                          uint32_t     channel,
                          uint32_t     iBlock,
                          uint32_t_    * pBlock)
```

Description:

This function makes it possible to obtain the use of a memory block of the specified channel to use the synchronous update or the random transmission.

The systematic use of this function ensures that the same block cannot be used by multiple users.

hCard	The handle provided by *a429Open()*.
channel	The channel number from 0 to 15.
iBlock	Index of the requested block, from 0 to 7.
pBlock	The address where to put the index of the allocated block

If *iBlock* is a valid block number from 0 to 7, the function attempts to reserve that block if it is not already allocated. If successful, put this number in *pBlock*, otherwise it returns A429_EBUSY.

If *iBlock* is A429_BLOC_ANY, the function searches for a free block. If it finds a free block it allocates it and places its index at *pBlock* address, otherwise it returns A429_EBUSY.

If successful the function returns A429_ENONE and a valid block number (0 to 7) at *pBlock* address. Otherwise, it returns another error number.

5.5.52 a429BlockFree ()

Syntax:

```
uint32_t a429BlockFree ( HA429          hCard,
                         uint32_t       channel,
                         uint32_t       iBlock)
```

Description:

This function allows you to release a memory block used for synchronous update or random transmission.

Normally the memory block should have been allocated by *a429BlockAlloc()*, but this is not checked. Therefore, the function always releases the provided block.

hCard	The handle provided by *a429Open()*.
channel	The channel number from 0 to 15.
iBlock	Index of the requested block, from 0 to 7.

If successful the function returns A429_ENONE, otherwise another error number.