ADAS

NEXEYA Products Division

9, rue Georges Besse
BP 47
78330 FONTENAY-LE-FLEURY
France
Tél. : 33(0)1 30 58 90 09
Fax : 33(0)1 30 58 21 33
Email : adas@nexeya.com
www.adas.fr

PMC-DigiCool

Bus Controller - Remote terminal – Spy for Digibus GAM-T-101 bus

User's Manual

Réf 2015.006.24E

GROUPE NEXEYA

Document revision history:

Index	Date	File name	Description of changes
24	11/09/14	201500624E_manual.doc	Add EI_DIGICOOL for SACHA Add WF630 and WF631
23	19/07/13	201500623E_manual.doc	Logo update
22	30/01/12	201500622E_manual.doc	Librairy version 2.4 Add DIGIBUSLN_DESCRIPTOR_ACKM in S_DIGIBUSLN
21	05/01/12	201500621F_ manual.doc	Library version 2.3 Add errors descriptor in S_DIGIBUSLN Modify digiCreateCycle (), doc DIGI_DGCSPY
12	08/07/11	201500612E_ manual.doc	For 64 bit OSs
09	20/01/10	201500609E_manuel.doc	Add Linux dynamic libraries (.so)
08	19/11/09	201500608E_manuel.doc	Add DIGI_EV_CMDOVFL, DIGI_EV_BCIT, digiCreateInterruptOp() Create operators function argument corrected
07	29/06/09	201500607E _manual.doc	Linux version Change digiDriverVersion(), digiLibVersion(), PMC connectors add spy fifo words Other minor changes
06	04/05/09	201500606E _manual.doc	The card manage 32 RT
05	21/04/09	201500605E _manual.doc	Public release
03	02/10/08	201500603F_manuel.doc	Add Honda connector
02	28/07/08	201500602F_manuel.doc	Minor corrections
01	26/05/08	201500601F_manuel.doc	First issue

Approbations:

Date	Title	Name	Signature
11/09/14	Author	A. Chebrou	
	Review		

Contents

1 WARNING

This chapter is intended for the users of the digilib library with version equal or lower than 2.2.

The users of the current library described in this manual can ignore it

1.1 64 bits OSs.

The version 1.8 (and superiors) of the digilib library is compatible with 64 bits operating systems Window XP/Seven/8, as well as Linux. It remains compatible with the 32 bits versions of these systems.

This evolution introduces the following changes for developers Using versions previous to 1.8:

- A new handle type is introduced for the handle supplied by *digiOpen()*. It is about HDIGIC instead of HDIGI.

  The handle HDIGI continues to be used for messages and operators.

  The main functions concerned by this change of handle are:

  digiOpen()

  digiSpyReaderOpen ()

  The user callback.

- The libraries are specific for every 32 bits or 64 bits operating system. It is thus advisable to install the adequate libraries on every system. The CD of delivery supplies one directory for the 32 bits systems and an other one for the 64 bits systems.

The card is also supported in its complete version by VxWorks 6.8.2 on X86 platform.

1.2 Library version 2.3.

The digilib library version 2.3 is intended to be used with the microcode of the card PMC-DigiCool version 3.1. This chapter addresses the developers which use the version 2.0 to 2.2 to indicate them quickly the evolutions brought by the new version of the library.

The evolutions concern the following points:

- Change of the prototype of *digiCreateCycle()*, to add it an identifier, and obtain the same features as *digiCreateCycleOp()*.
- Addition of indicators of errors in the field *descriptor* of S_DIGIBUSLN. The previous versions indicated only the presence of an error of transmission on one of the elements of the exchange without indicating which one, what forced the user to inspect all the qualifiers of the message.

1.3 Library version 2.0.

The digilib library version 2.0 is intended to be used with the microcode of the card PMC-DigiCool version 3.0. This chapter addresses the developers which use the digilib version 1.8 to indicate them quickly the evolutions brought by the new version of the library.

The main evolutions concern the following points:

- Detection and management of the errors: the detection of errors was very widely improved, and the erroneous messages can be indicated to the callback.
- Frame events: it is possible to introduce operators into the frame (cycleOp, eventOp) who are indicated to the callback, what allows to follow the progress of the frame.
- Multi-Frame Management improved: it is possible to store several frames in the card and to ask to the BC to commute quickly of the one in the other one.
- The V bit of the command acknowledge is automatically managed according to the standard GAM-T-101.
- Possibility of pace of the cycles Digibus by software, besides the internal timer or besides the external signal.
- Optional recognition of the commands CV255 and CLV255 to activate / inhibit emission by the subscribers (specific feature not being a part of the standard).

The design of digilib V2.0 was made to facilitate the porting of applications written for digilib v1.x However it is not possible to use the DLL V2.0 without recompiling the application.

1.3.1 Changes in the API.

The differences in function prototypes between version 1.x and 2.x libraries concern the following functions:

callback	It has one more argument which is used primarily to identify any errors on an exchange when called by the events DIGI_EV_MESSAGE and DIGI_EV_MSGERROR.
digiCreateFrame	The fourth parameter has meaning extended to allow sotfware cycle trig. Use one of the constants DIGI_CLKINT DIGI_CLKEXT or DIGI_CLKSOFT, instead of 0 or 1.
digiCreateCycleOp	Added parameter is the identifier of the cycle that is passed to the callback with the event DIGI_EV_CYCLEEND.
digiSpyReaderOpen	The second parameter is meaningful extent.

1.3.2 New Features.

digiGlobalErrorMask	Specifies errors that will generate an event in the callback.
digiMessageAttr	Specifies additional attributes of an exchange: Length of delay between message, message specific error mask, generating errors ...
digiReadBufferErr	Request message data and the error status word associated. Can replace digiReadBuffer.
digiCreateEventOp	Creates an operator that generates a call to the callback, passing the argument supplied when it was created.

digiTrigCycle	Software triggering of cycle in BC.
digiDeleteFrame	Lets free the memory used by the frames descriptions no longer needed.
digiLoadFrame	Allows loading several frames descriptions in the card memory.
digiRestart	Allows fast frame switching from the callback.

2 Introduction

This document describes the DIGICOOL cards made by ADAS Electronique.

The cards, along with its set-up and operating software, are discussed briefly in this first section and described in further detail in the remaining sections, which cover

- The PCI driver and implementation library.
- The card programming interface.
- Appendices: Connections.

2.1 Reference documents

Installation of Windows XP / Seven / 8 PCIG driver, ADAS 876.012.03

Installation of Linux PCIG Driver, ADAS 2015.018.01F

Digibus standard GAM-T-101, CELAR-ICEV, September, 1982

2.2 Characteristics of PMC-DigiCool cards

The main characteristics of these cards are as follows:

2.2.1 General characteristics

Card format	"Conduction Cooled" PCI mezzanine card format (CCPMC). Uses the primary and secondary thermal interfaces. The card is shorter than a Standard PMC and has no front pannel. Inputs and outputs use the Pn4 connector. The card is delivered with a mezzanine with front connector, this allows to use it in a standard case (without thermal drain).
Compatibility with standard PCI	32-bit, 33MHz interface, in compliance with PCI standard 2.1. The PCI card can be placed in a 5V or 3.3V PCI slot. It can be placed in a 64-bit, 66MHz PCI slot but bus exchanges will be restricted to 32-bit, 33MHz.
Climatic range	In use: −25°C to 75°C, max. humidity 90% without condensation. In storage: −40 to +85°C.
Electrical supply requirement	5V: 1. A, +12V: 0.3 A, -12V: 0.1 A max.
External interface	Pn4 connector. An optional interface circuit enables inputs-outputs to be used via a male 50-way Honda Connectors HDR-E50 on the front pannel.
Configuration	No selector on the card: the logic is uploaded by the application; the configuration is software defined.

2.2.2 Digibus Characteristics

Standard	GAM-T-101
Electrical interface	Direct coupled stub for one redundant Digibus. There is no bus termination on the card.
Functions	The card’s three functions can be used simultaneously: Remote terminal Bus controller Integral spy

2.2.3 Ancillary functions or characteristics:

Date stamping	The card incorporates an IRIG-B122 receiver (1 KHz sine wave, BCD coded). The messages received are date stamped to a resolution of 1µs, based on the date supplied by the IRIG-B receiver.

	If no external time-date signal is available, the card can generate its own IRIG-B122 signal.
Synchronisation	The card can be configured to recognise specific exchanges. Recognition of any such exchange generates an impulse on a logic output.
Supervisor clock rate	The supervisor clock tick can be taken from an internal clock or from a logic input.
Additional inputs-outputs	The card incorporates three TTL inputs and three TTL outputs. These can be used by the application and are shared by the Bus Controller synchronisation and trigger and the spy trigger
Versatility	The card logic is uploadable code executed by the on-board FPGA. Specific versions can be produced without modifying the card hardware. A PowerPC 405 processor is fully available for use in a special application.

The card comes in two versions:

PCM_DIGICOOL-S	Integral spy. This card is unable to transmit on the Digibus bus; consequently it provides neither Bus Controller nor Remote Terminal features.
PCM_DIGICOOL	Complete. This card provides all three features; Integral Spy, Bus Controller, Remote Terminal.

-

2.3 Available software

ADAS card drivers exist for the following operating systems:

- Microsoft Windows XP 32 bits
- Microsoft Windows Seven, 32 or 64 bits.
- Linux Ubuntu kernel 3.4.0 32 and 64 bits.

- VxWorks (processor type and VxWorks version to be defined).

The card comes with basic applications written in C to give an idea of the implementation of the card’s three functions. These applications are based on the **digilib.dll** library described later.

3 Overview of the DigiCool cards

This card family is intended for applications that implement a Digibus GAM-T-101 bus, redundant or otherwise.

Theses cards provides the three functions of a bus interface:

- Traffic spying. The full traffic is transmitted to the application: commands, data, preselection acknowledgements (echoes). Parity and coding errors are signalled. Each exchange is date stamped.
- Bus Controller.(BC) A frame, ie an execution sequence of exchange, can be specified whose circulation is handled by the BC.
- Remote terminal (TR). The card can be used to manage one or more equipments.

Useful testing tool features are included:

- Management of the data line’s V bit.
- Ability to generate a parity error on each command or data word.
- Ability to specify the preselection acknowledge value from each managed RT.

The available physical formats are:

PMC_DIGICOOL	Condiction cooled PCI Mezzanine Card.
EI_DIGICOOL	Card for SACHA systems.
PCI_BUS_DIGICOOL	PCI short.

3.1 The Conduction Cooled PMC format

The PMC format (PCI Mezzanine Card) is a mezzanine circuit which is intended to be placed on a carrier board. The PMC, as every electronic circuit has a certain amount of heat which must be evacuated. To remove this heat PMC standard provides two variants:

- Standard PMC. It has a front panel that supports most of the time the connection of inputs and outputs. To dissipate the heat the host system must have ventilation.

- Conduction Cooled PMC. In this format the card has printed circuit board surfaces to be plated on thermal drains to evacuate the heat generated by the card to the chassis of the system. This requires suitable carrier cards. In addition, the contact surfaces prevent the PMC to have a front pannel and I/O are made by the P4 connector on the back of the card.

A common shortcut is to say that a conduction cooled board does not need ventilation. This is true only if it is placed on a suitable carrier, and that itself can dissipate the heat that is transmitted. In conclusion Conducton Cooled PMC placed on a standard chassis must be ventilated.

The features of this card:

- Digibus short stub only.
- No internal terminationg load..
- IRIG-B Dating (receiver and generator).

- Has to use a WF609 or WF611 cable to connect in Digibus.

3.2 Le format EI

The EI_DIGICOOL card use the "Enhanced Interface" format for SACHA systems. This card performs the three functions of a Digibus interface, but is also available in spy only version.

The features of this card:

- Software selection of Digibus short or long stub. The choice is preserved when power is switched off.
- Software selection of Digibus line load of 75Ω. The choice is preserved when power is switched off.
- Message dating with RTC (Real Time Clock), a 10 MHz clock SACHA common to all the slots of the SACHA system. There is neither IRIG receiver nor generator.
- The Digibus front connector is a DSub 9 pin. The pinout is identical to that of the WF611 cable.

3.3 Le format PCI

The PCI_BUS_DIGICOOL card use the standard PCI short format. This card performs the three functions of a Digibus interface, but is also available in spy only version.

The features of this card:

- Software selection of Digibus short or long stub. The choice is preserved when power is switched off.
- Software selection of Digibus line load of 75Ω. The choice is preserved when power is switched off.
- IRIG-B Dating identical to that of the PMC_DIGICOOL.
- The Digibus front connector is a DSub 9 pin. The pinout is identical to that of the WF611 cable.

3.4 PMC_DIGICOOLO front connector

The signals used by the PMC_DIGICOOL card are available either by the mezzanine connector, or by the connector P4. These uses are exclusive

The signals of the connector P4 must not be used if the mezzanine in position. Certain exceptions are possible, if necessary contact ADA electronics

3.5 Integral spy

The integral spying function produces a data stream covering the whole of the traffic on the procedure and data lines. This stream is placed in a FIFO. The digilib.dll library uses a DMA to transfer the contents of the FIFO to main memory without putting any load on the processor.

Receipt of a message by the card is translated by the insertion of a certain number of words in the Spy FIFO, namely, two start-of-message date words followed by words containing the characters transmitted by the procedure and data lines.

Each character is qualified by a number of attribute bits – procedure, data, parity, etc.

Characters are inserted in the FIFO in their order of arrival on the bus; for example, the echo from the first command can come before or after the first character of the second command.

The digilib.dll library handles usage of the data flow and constructs a `S_DIGIBUSLN` type structure for each message.

Example of programme using the spy feature. We assume the Digibus card is already open.

```
HDIGIC          hReader;
u32             result;
S_DIGIBUSLN     * pDataBus;

// Start spy function
if (! digiSpyStart (hCard, 0, DIGI_BUSA))
{
     printf ("digiSpyStart error\n");
     return (0);
}

// Create a reader
hReader = digiSpyReaderOpen (hCard, 0);
if (! hReader)
{
     digiSpyStop (hCard);
     printf ("digiSpyReaderOpen error\n");
     return (0);
}

while (1)
{
     // Try to obtain message
     pDataBus = digiSpyReaderMessage (hReader, & result);
     if (! pDataBus)
     {
          if (result == SPY_NOTRECEIVED)
          {
               // Nothing to read
               Sleep (10);
               continue;
          }

          if (result == SPY_WRITEOVER)
          {
               // Time out
               digiSpyReaderReset (hReader);
               continue;
          }

          if (result == SPY_ERROR)
          {
               // Try again
               continue;
          }
     }

     // Use the message
     displayDigibusMessage (pDataBus);
}

digiSpyReaderClose (hReader);
digiSpyStop (hCard);
```

3.5.1 Filtering

The filtering is a supplementary feature of espionage. It can generate a pulse on one of the pin of the output port when a command is traveling on the bus

The filtering is done by comparing the first or the first two command words of messages. The filtering allows to report the passing of a particular message.

The filter works independently of the spy feature: there is no need to configure nor start the bus spy to use filtering

The port interface that generates the filtering signal is specified with *digiOutPortBind()* and *digiOutPortConfig()* (Default is signal 1 on output port).

Exemple:

```
// Start filtering with 1 command word
digiSpySetFilter (hCard,
                  DIGI_CMD (5, DIGI_CE, 0x12),
                  0,
                  DIGI_BUSA | DIGI_FILTER1| DIGI_ENABLE);


// Stop filtering
digiSpySetFilter (hCard, 0, 0, DIGI_BUSA | DIGI_DISABLE
```

3.6 Remote terminal

Remote terminal (RT) management is the most complex task performed by the DigiCool card and its associated library.

Tasks to be performed are

- Listening on the procedure line and recognising commands
- Issuing an echo for each command if necessary
- Analysing each command descriptor for
  - transmission and/or reception of a data buffer
  - recognising messages and notifying the application if so requested (This is a software task of the library)

For each command in an exchange, the library constructs a descriptor that will be used by the DigiCool card. The descriptor contains flags and pointers to the transmission and reception data buffers associated with this command (which may be null if there is no data).

Enough on-board card memory is available to simulate up to 32 Remote Terminals.

If the supervisor function is used at the same time as RT management, RT 0 must be declared (standardised supervisor address).

3.6.1 Recognising the exchange

For the application to be warned of an exchange circulating, an indicator is set in the descriptor of one of its commands.

During circulation of an exchange, the card constructs a table with the first four commands of the message. If a notification signal is requested, this table is placed in the command FIFO and an interruption is generated.

The library, thus warned, reads the FIFO contents, recognises the messages from the procedure words, and invokes the callback supplied by the application with DIGI_EV_MESSAGE event.

Problem of the X field of complementary commands.

To recognize an exchange and return its handle the library compares the commands of the exchange created with `digiCreataMessage()`, up to 4 commands, complementary command included. But for some exchanges, BC varies the X field value of the complementary command (with `digiMessageSetCC()`), which prevents recognition of the exchange.

If the number of X field values is limited it is possible to declare so many exchanges as there is of different complementary commands, in order to get the right handle to the callback.

With such messages, there are rules to be respected:

- For `digiMessageSetCC()` and the access functions to the data it is necessary to use the handle of the message which was used during the creation of the frame. Other handles is only alias which are of use only to the recognition of messages.
- For `digiMessageGetCC()` it is possible to use the handle of whatever of the exchanges, because all these exchanges use the same command descriptor.

3.6.2 Error generation

A parity error can be generated on subscriber-issued data words by placing the appropriate indicator in the message data word.

Similarly, a format error can be generated on these words (no alternation on the fourth bit of the word, bit 0x10).

3.6.3 Error Detection

The card realizes controls during the exchanges and builds an error staus word. This staus word can be got back in several ways:

- If the message signaling was asked during its creation, the error status word is passed on in the callback at the same time as the event DIGI_EV_MESSAGE.
- If the message signaling was not asked during its creation, it is possible to configure the message error mask as to generate the event DIGI_EV_MSGERROR for the specified errors.
- If the message signaling was asked, but the message is not recognized (for example the X field of CC changed), the event DIGI_EV_MSGERROR is generated with a special message handle value: DIGI_UNKNOWNMSG.
- If the exchange has data in reception, it is possible to read the error status word at the same time as the data by using the function `digiReadBufferErr()`.

The meaning of the bits of the error status word is indicated by constants DIGI_ERR_xx, refer in `digiGlobalErrorMask()`.

The events generation on error is managed by two errors masks.

- The global mask, defined with the `digiGlobalErrorMask`() function. This mask applies to all exchanges. By default all errors are masked, there is thus no generated event on error.
- The exchange specific mask. By default all the errors are unmasked, what authorizes the generation of error events.

So that an event is generated for an error the corespondants error bitsr have to be present in 2 masks.

For an event to be generated for an error, bits for this error must be present in both masks

Parity and format errors of the commands have a special treatment: The standard requires Digibus detect these errors and to stop the exchange. In this case the exchange can't be recognized and the event DIGI_EV_MSGERROR is generated with the DIGI_UNKNOWNMSG handle.

3.6.4 Preselection acknowledge generation

A Remote Terminal sends acknowledge in response to a Bus Controller command. There is a unique acknowledge value for each managed subscriber.

If sending of an acknowledge is enabled for a Remote Terminal, a command always returns an acknowledge unless

- the Terminal number for the command is 0, i.e., the command is from the Bus Controller, who never sends an acknowledge
- the command is a supplementary command (not intended for any specific RT)

The interface software allows the acknowledge byte value to be modified for each RT. This enables generation of an invalid acknowledge for testing purposes.

The library allows you various ways to specify acknowledge values:

- automatic construction of a valid acknowledge for an equipment (RT)
- generation of a parity error on the acknowledge
- generation of a wrong RT address (different from that of the equipment)
- inhibition of acknowledge generation for a managed RT
- V bit set permanently to 0.

The value of the V bit of acknowledges is normally 1. The standard provides that the coupler automatically manages the V bit of acknowledge to indicate an error on the previous exchange

3.6.5 Data buffers.

Each exchange can involve one subscriber sending data and another (or more) receiving it. The card assigns buffers to exchanges when this is necessary, not to RT.

The application can write data from a send buffer and read data from a receive buffer any time it wants.

Two buffers arranged in FLIP/FLOP are used by the data associated with an exchange, so that the card’s RT management and the application can both access a buffer without getting in each other’s way. A mutual exclusion (Mutex) mechanism ensures buffer switching takes place at the right time.

3.6.6 Sample application

Example of an application creating a RT and a message.

```
//---------------------------------------------------------------------
//   Exchange signalling function

void cb (HDIGIC hCard, u32 event, HDIGI hMessage, void * arg, u32 attr)
{
          .
          .
          .
}

//---------------------------------------------------------------------

u32  test ()
{
     S_DIGI_MESSAGE      mess ;
     HDIGIC              hCard ;
     HDIGI               hMess ;
     u32                 equip1 = 0x0A ;
     u32                 uVal ;

     uVal = digiOpen (& hCard, 0, cb, 0, 0, NULL) ;
     if (uVal != DIGI_ENONE)
     {
          printf ("digiOpen error: %d <%s>\n", uVal, digiErrorMessage (uVal)) ;
          return (0) ;
     }

     // Manage RT 0
     if (! digiCreateEquipment (hCard, 0, 0))
     {
          printf ("digiCreateEquipment error\n") ;
          digiClose (hCard) ;
          return (0) ;
     }

     // Manage another equipment
     if (! digiCreateEquipment (hCard, equip1, 0))
     {
          printf ("digiCreateEquipment error\n") ;
          digiClose (hCard) ;
          return (0) ;
     }

     // Isolated CR, sender is BC
     memset (& mess, 0, sizeof (S_DIGI_MESSAGE)) ;
     mess.commands[0]     = DIGI_CMD(equip1, DIGI_CR, 0) ;
     mess.commands[1]     = DIGI_CMD(0, DIGI_CC, 1) ;
     mess.dataSize        = 8 ;     // Bytes
     mess.interrupt       = 1 ;     // Signaling requested
     mess.nbTempo         = 0 ;     // No delay
     mess.bus             = DIGI_BUSA ;

     hMess = digiCreateMessage (hCard, & mess) ;
     if (! hMess)
     {
          printf ("digiCreateMessage error\n") ;
          digiClose (hCard) ;
          return (0) ;
     }

     // Configuration finished, initialise card
     if (! digiInitCard (hCard))
     {
          printf ("digiInitCard error\n") ;

          digiClose (hCard);
          return (0);
     }

     // Start RT management
     digiStart (hCard, NULL, 0);
       .
       .
       .
     digiStop (hCard);

     digiClose (hCard);
     return (1);
```

3.7 Bus Controller

The purpose of the Bus Controller (BC) function is to control the bus procedure line based on a template description.

The template specification is based on exchanges and on special factors such as delays or triggers.

A frame consists of a succession of cycles at a specified clock rate. A cycle is a series of exchanges build with `digiCreateCycle()`. The cycles are then assembled to create a frame with `digiCreateFrame()`. This function automatically inserts, in front of each cycle, a wait until the cycle start signal (Start of cycle operator).

Another way to build a frame is to describe it completely in a single HDIGI array using operators handles (`digiCreateCycleOp()`, `digireateDelayOp()`, `digiCreateTriggerOp()` ...), and messages handles. It must then pass this array to the function `digiCreateFrame()`, (recall `digiCreateFrameRaw ()` is deprecated, all frames can be performed with `digiCreateFrame()`)

Cycle timing can be controlled by one of two methods:

- An internal card clock
- A tick on one of the card's logic inputs
- By software

The clocking characteristics are specified by `digiCreateFrame()`during creation of the frame.

If the cycle contents have a longer duration than that of the cycle, the next cycle is immediately invoked so as to absorb the backlog without any cycle shift. Consequently a cycle overflow is not an error. But an event can be generated in case of cycle overload, refer in `digiGlobalErrorMask()`.

Remark: Supplying a frame to the card and starting the RT does not mean that the management of RT corresponding to the BC (at address 0) is enabled. To do this we must also use `digiCreateEquipment()` with the equipment number 0.

3.7.1 Message validation and inhibition.

In some applications it is useful that some messages do not travel at all times. Sometimes it is necessary that they move only once on request.

`DigiEnableMessage()` function allows this feature. Indeed, it admits a list of handle messages that we must act on and a parameter indicating whether to validate or invalidate the flow of these messages.

To move a message only once on request we can use the following procedure:

- Open the card and specify a callback function with `digiOpen()`.
- Create the message with `digiCreateMessage()`, asking to be notified of its passage (interrupt setting).
- Build the frame including the message with `digiCreateFrame()`.
- Inhibit this message with `digiEnableMessage(…,FALSE)`.
- Start RT an BC management with `digiStart()`.
- At the time of making the message circulate, validate it with `digiEnableMessage(, TRUE)`, having written possibly the data which he has to emit with `digiWrite()`.
- When the message circulated the callback is called, then we can inhibit this message again with `digiEnableMessage(, FALSE)`, and possibly read the data that it received.

.

Remark: The treatment in the callback should be as short as possible.

3.7.2 Frame operators.

The BC frame is a list of identifiers that allows you to specify the sequence and timing of exchanges on the bus. This list relate to exhange, but also allow operators to insert specific BC operations at specific locations of the frame.

Operators are created with functions of the library:

digiCreateCycleOp()	Create a start of cycle operator. When it encounters this operator in the frame, the BC will wait for the next begin of cycle signal (internal timer, external top…) to continue.
digiCreateDelay()	Create an operator who ask the manager to insert a delay between two messages. The function parameter specify the delay.
digiCreateTriggerOp	Created an operator generating a trigger signal on the output port
digiCreateLabelOp()	Creates a label in the frame, ie memorize the current address witch may be used by the jump operator
digiCreateJumpOp()	Creates a jump operator that uses an already established label number

digiCreateEventOp	Creates an operator that generates an event in the callback when it is run. A number associated with each event allows to identify each event.

The label and jump operators have been introduced to help create a frame that has two distinct parts:

- The first part shall be operated only once, at the beginning of the frame (initialization frame)
- A second part is cyclical. This second part begins with a label operator, and ends with a jump operator to that label (operating frame).

The library automatically put a label operator at the beginning of the frame, and a jump operator at end of frame, to loop the frame at beginning. In this way to declare a purely cyclical frame, the programmer does not have to use the label and jump operators

3.7.3 Frame construction.

The use of the Bus controller (BC) supposes the use of a frame describing the pace of the exchanges in time. There are mainly three cases of use of frame, these cases are briefly explained here.

1) Single frame.

The most current case uses only a single frame. This one is created by `digiCreateFrame()`, and passed to the function `digiStart()` to start the BC with this frame.

2) Initialisation frame.

Another very current case of use is represented by the applications which require a phase of initialization of the bus executed only once, then a cyclic phase executed several times or in a permanent way.

The use of the operators label and jump allows to cut the frame in an initialization part and a cyclic part situated between a label and a jump.

Example of frame with initialization phase:

```
u32 nb = 0 ;

frame [nb++] = digiCreateCycleOp () ;
frame [nb++] = hMess1 ;
frame [nb++] = hMess2 ;                          initialization phase
frame [nb++] = digiCreateCycleOp () ;            executed only once
frame [nb++] = hMess3 ;
frame [nb++] = hMess4 ;

frame [nb++] = digiCreateLabelOp (1) ;

frame [nb++] = digiCreateCycleOp () ;
frame [nb++] = hMess5 ;
frame [nb++] = hMess6 ;                          Permanent cyclic phase
frame [nb++] = digiCreateCycleOp () ;
frame [nb++] = hMess5 ;
frame [nb++] = hMess7 ;

frame [nb++] = digiCreateJumpOp (1) ;

digiCreateFrame (hCard, frame, nb,
                  DIGI_CLKINT, 20000, 0) ;
```

3) Multi frame.

The most complex applications require several frames. In that case frames must be created by `digiCreateFrame()`, then loaded in the card by `digiLoadFrame()` before being able to be used.

The fact of loading frames in the memory of the card allows `digiStart()` to run faster and thus to decrease the deadline between the end of a frame and the starting up of the following one.

Frames which are executed a finished number of time can indicate their end of execution with the callback event DIGI_EV_FRMEND, if this event is validated by `digiGlobalErrorMask()`.

The number of frame which it is possible to load in the card is limited by the memory size and the complexity of the traffic on the bus. To consider if planned frame can live in the card we can use the following approximate calculation:

- The card has 16384 words to accommodate the description of the exchanges and the frame.
- Every exchange consumes approximately 12 words.
- Every item of a frame (message or operator) consumes 1 word.
- The busy memory corresponds approximately to the formula:

  (12 * exchange_number) + sum of_frames_sizes.

If the size memory is insufficient, `digiLoadFrame()` or `digiStart()` indicate it.

3.8 The output port.

The card has a port of three-bit output. These bits can be used for internal signals of the card, or be released for the application.

The signals are named SIG0, SIG1 and SIG2. Signals that can generate the card are:

- SIG0: The signal generated by the trigger operator in the BC frame.
- SIG1: The trigger signal generated by the bus A spy
- SIG2: The trigger signal generated by the bus B spy

Note: In its current version the card manages a single redundant bus, therefore only the trigger signal of bus A (SIG1) is used.

If part or all of the signals generated by the card are not relevant to the application, it may request that these signals are inhibited and replaced by signals from a user register that can be written by the application.

On the other hand the output port has a mixing matrix that guides any signal to any of the three physical interfaces. This can be useful for example with the standard mezzanine that has 2 TTL outputs and an RS422 output

To configure the output signals, use successively the two following functions:

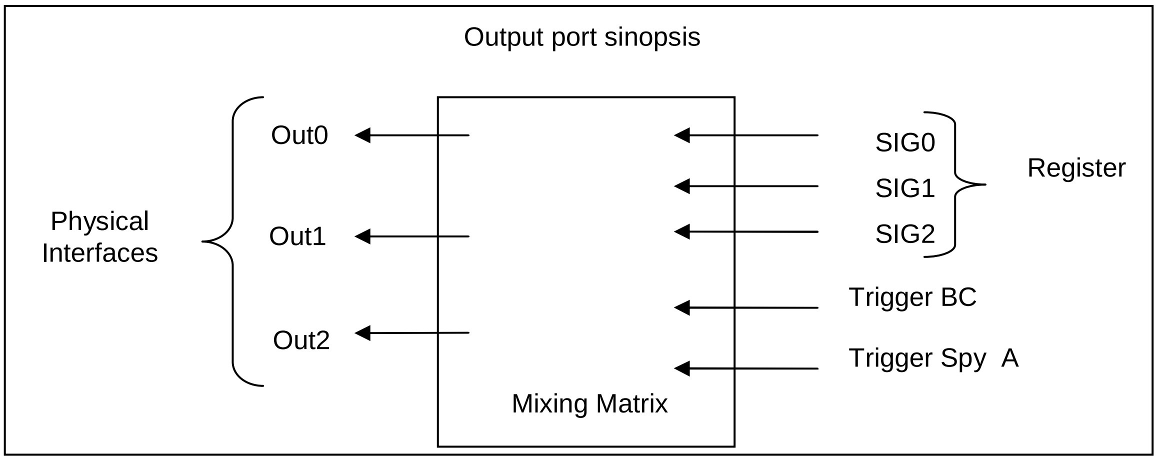
`digiOutPortBind()`	Allow you to specify for each physical interface which signal is affected. This is the configuration of the output multiplexer
`digiOutPortConfig()`	Allow you to specify for each signal whether the signal of the card or the user signal from the register. This function also allows you to specify the polarity of the signals on the output port (optional insertion of an inverter

The state of user signals can be changed using the function `digiOutPortSet()`

At the opening of the card output port is configured as follows:

```
digiOutPortBind (pEnv,
                 DIGI_SIG0,                    // For TTL 0
                 DIGI_SIG1,                    // For TTL 1
                 DIGI_SIG0) ;                  // For RS422 2

digiOutPortConfig (pEnv,
                 DIGI_SIG0_BCTRIG | DIGI_POLUP,   // For if Out 0
                 DIGI_SIG1_TRIGA  | DIGI_POLUP,   // For if Out 1
                 DIGI_SIG2_USR    | DIGI_POLUP);  // For if Out 2
```



3.9 The input port

The card has a port of three-bit input. These bits can be used for internal signals of the card, or be released for the application.

The signals are named SIG0, SIG1 and SIG2. Signals used by the card are:

- SIG0: Sart of cycle sync signal (Bus controller).
- SIG1: Unused
- SIG2: Unused

If part or all of the signals used by the card are not relevant to the application, it may request that these signals are inhibited and replaced by user-readable signals on a register.

On the other hand, the input port has a mixing matrix that guides the signal from any of the three physical interfaces to any input signal. This can be useful for example with the mezzanine standard that has 2 inputs and a TTL input RS422

To configure the input signals, use successively the two following functions:

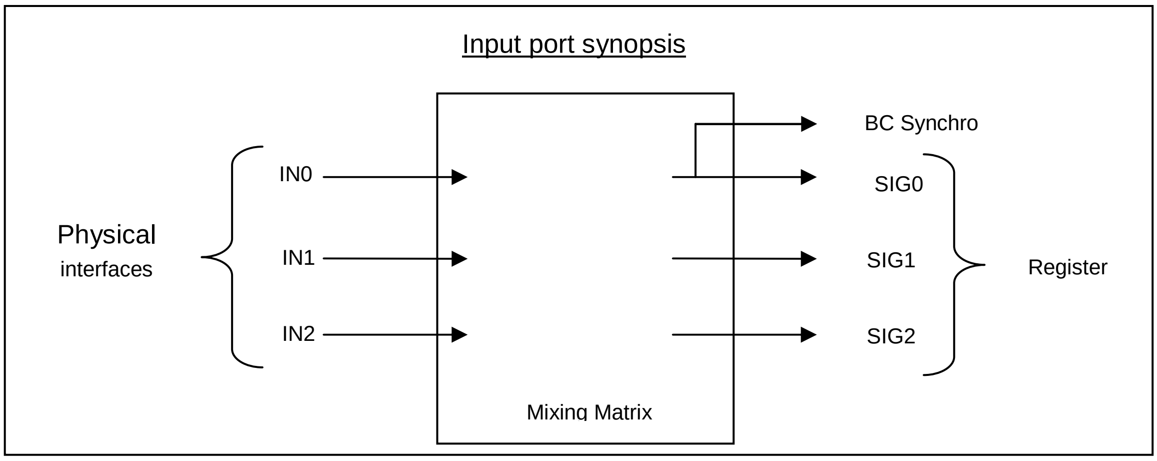
`digiInPortBind()`	Allows you to specify which physical interface is assigned to each signal. This is the configuration of the input multiplexer
`digiInPortConfig()`	Allows you to specify for each signal whether it should use the card signal or the user signal. This function also allows you to specify the polarity ot the signal (insertion of an optional inverter)

The state of user signals can be read using the function `digiOutPortGet()`

At the opening of the card input port is configured as follows:

```
digiInPortBind   (pEnv,
                  DIGI_IF0,
                  DIGI_IF1,
                  DIGI_IF2) ;

digiInPortConfig (pEnv,
                  DIGI_SIG0_BCSYNC  | DIGI_POLUP,   // For Signal In 0
                  DIGI_SIG1_USR     | DIGI_POLUP,   // For Signal In 1
                  DIGI_SIG2_USR     | DIGI_POLUP) ; // For Signal In 2
```



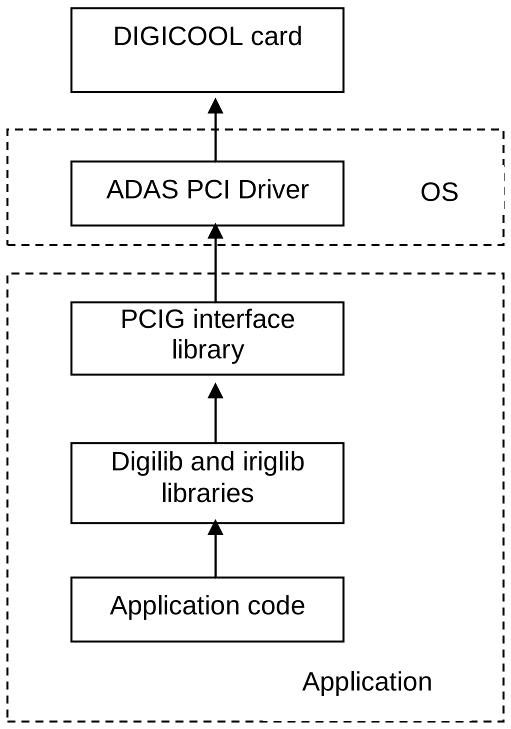
3.10 Driver and library

ADAS has developed PCI drivers common to all ADAS cards. This driver allows use of these cards under the Linux and Windows operating systems. These drivers are very similar but present some differences due to the specifics of each operating system.

In order to facilitate applications development, ADAS has developed a software module called "PCIG Interface" which hides any OS-related differences between drivers. Concretely, an application does not use the low-level driver functions but instead uses the high level functions of the PCIG Interface.

The digilib library uses this PCIG interface, thereby making it independent of the operating system. Using this library, the programmer can ignore the other components (PCI interface and PCIG driver).

All the software components of an application that uses a DIGICOOL card are represented in the block diagram below for guidance. You can see that the application communicates only with the digilib and iriglib library.



The PCI driver is automatically loaded on system start-up. Installation of this driver is described in a specific document quoted in the references.

The PCI driver version can be read by the function `digiGetDriverVersion()`; for example,

```
version = digiDriverVersion() ;
printf ("Driver PCIG version : %d.%d\n",
     (version >> 8) & 0xFF, version & 0xFF);
```

3.11 Managing several cards

The system can contain one or more DIGICOOL cards. The library is adapted to such cases, in which case a card is initialised by stating its rank. The first DIGICOOL card bears the number 0; the next, the number 1, etc.

Remark 1: the card number supplied must be that used in the DIGICOOL function and not the PCI card number supplied by PCI numbering.

Remark 2: The order of all the PCI cards in a system is determined when the mainboard BIOS assigns numbers to PCI components. Consequently, the order of the DIGICOOL cards is also dictated by this numbering.

To find out the number of DIGICOOL cards recognised by the system, use the `digiGetCardCount`() function. For example:

```
u32 nbCard = digiGetCardCount (0);
if (nbCard == 0)
{
              printf ("No PMC-DIGICOOL card found\n");
              return (0);
}
else
              printf ("Found %d PMC-DIGICOOL cards, \n",  nbCard);

nbCard = digiGetCardCount (DIGI_DGCSPY);
if (nbCard == 0)
{
              printf ("No PMC-DIGICOOL-S card found\n");
              return (0);
}
else
              printf ("Found %d PMC-DIGICOOL-S cards, \n",
              nbCard);
```

4 IRIG-B dating.

The card has an IRIG-B feature which is used to date messages acquired by the spy.

4.1 Start dating.

At start up the card the receiver initializes with date by default 00:00:00 on January 1st, and makes live the date at the rate of his local clock.

In a second phase it detects the IRIG-B signal, discipline its local clock then commutes on the successful date. This mechanism takes several seconds.

If there is no available external IRIG-B signal and if the date by default does not agree, there are two possibilities:

- The application can initialize the date of the receiver with `irigSetDate()` or `irigSetLocalDate()`.

- The application can initialize the IRIG-B transmitter date with *irigSetTxDate()*. In this case loopback the IRIG-B output of the card on the IRIG-B input

4.2 Loss of the IRIG-B signal.

When the IRIG-B signal disappears, or its quality is no longer sufficient (detection of parasites in the signal), the receiver switches to freewheel, and then sets status bits of the receiver to indicate this mode.

In this mode the reference clock is not slaved to external signals, and continues to run with its current settings. At each counter overflow of micro seconds the date is incremented by 1 second by the receiver.

4.3 Return of the IRIG-B signal.

Upon return of the IRIG-B signal, the receiver starts with the quality of the received signal. Then he re-enslaves the local clock on the external signal, then switches on the date received. This is the mechanism for automatic reconnection.

This mechanism creates two side effects that can be drawn.

- The re-enslavement of the clock leads to fluctuations in the frequency of the clock, and therefore the quality of dating. This quality is indicated by a bit of the status register of IRIG-B receiver.
- The passage on the date received can cause a "break time" in the past or the future.

While it is important to avoid these disturbances to the return signal IRIG-B, it is possible to inhibit the automatic reconnection using the *irigSetEnableConnect()* of the library.

Once the automatic reconnection is inhibited, the clock go freewheel in the event of loss of signal and stay in this state. Reconnection will occur during the validation of the reconnection, if the IRIG-B signal is present.

4.4 Compensation for delays

The IRIG-B receiver can insert a small offset in the enslavement of the internal clock in relation to external signal.

This delay can be used to compensate for cable lengths or delays introduced by treatment (filters) or a particular interface.

The delay step is 20ns.

The algorithm used by the receiver leads to multiplying the value by 32. So to shift the signal of 20 ns, we must provide the value 32 (one step * 32).

The offset can be positive or negative in the interval -50µs / 50 µs.

4.5 Calibration.

The card generates an internal clock slaved on the received signal. However, transmission delays are dependent on the installation and electrical interfaces, so that the slave clock can be shifted relative to the true signal.

This calibration is useful to ensure that two devices receiving the same signal IRIG-B will affect the same date to an event.

The calibration of the card is to ensure that the beginning of a seconds of the clock arrives at the same time as the beginning of a second of the generator. This setting uses the compensation for delay.

Comparing the sinusoidal signal IRIG itself is difficult because it can be noisy. The most effective way to calibrate a receiver is to use the PPS signal (Pulse Per Second) of the transmitter and receiver.

Remarks

- PPS signal of some generators may be offset against their IRIG AM signal.
- We must calibrate all receivers

To calibrate the receiver, connect the two channels of an oscilloscope on the PPS outputs of the card and the sync generator. Set the oscilloscope trigger on the generator PPS. On the card you have to switch the PPS output to the receiver PPS (default).

Measuring the gap between the two fronts in ns, which corresponds to the value to compensate.

The application **irigutil** can transmit this value to the microcontroller of the card and store this value in EEPROM so that it is used at each power up.

4.6 The IRIG-B generator.

The card has an IRIG-B amplitude modulated signal generator. This generator is completely independent of the receiver, and is clocked directly by the oscillator accuracy of the card.

At the start of the card the generator initialize to the default date 00:00:00 on January 1st. Thereafter the application can change the date issued with the *irigSetTxDate()*.

This generator can be connected to multiple receivers, and particularly that of the card dating looping external.

The PPS output of the card can be set to send the PPS of the transmitter with *irigSetTxPps(1)*.

4.7 Date words format.

The date given by *irigDate()* function or by the structure S_DATABUSLN of intefral spy, has the following type:

```
typedef struct
{
    u32    w1 ;     // First date word
    u32    w2 ;     // Second date word

} S_LIRIGDATE;
```

First word:

Field	Description	Value
31	Date indicator. This bit qualifies the word as a date.	0
30	Date: first word indicator	0
29: 23	Reserved (7 bits)	x_H
22: 17	Year (6 bits)	x_H
16: 0	Seconds into the day (17 bits)	x_H

Second word:

Field	Description	Value
31	Date indicator. This bit qualifies the word as a date.	0
30	Date: second word indicator	1
29	IRIG synchronisation: 0: non synchronised 1: synchronised	x_H
28: 20	Days into the year (9 bits)	x_H
19: 0	Microseconds into the second (20 bits)	x_H

Bits 30 and 31 are useful in certain special IRIG core usage cases, such as the Digibus card (for recognising a message start in the DMA stream).

5 Implementation library

To enable application software to be written, the implementation library exposes an API in the "C" programming language. This language was chosen to make the library easy to port (VxWorks, LabView, etc.).

Library objects are prefixed by "digi" and constants by "DIGI_".

The library comprises the following files:

- digilib.h	Library interface file to be included when compiling the application
- digilib.lib	File to be linked with the application, mandatory
- digilib.dll	DLL runtime called by the application during execution (Windows only), , mandatory
- libdigilib.a	Digibus static library linked to the application for Linux, mandatory if dynamic libraries are not used.
- iriglib.h	Library interface file to be included when compiling the application, if IRIG-B functions are used.
- iriglib.lib	File to be linked with the application, mandatory.
- iriglib.dll	DLL runtime called by the application during execution (Windows only), mandatory.
- libiriglib.a	Irig-B static library linked to the application for Linux, mandatory if dynamic libraries are not used.
- libdigilib.so libiriglib.so	Linux dynamic libraries

Note: Under Linux it is possible to use static (.a) or dynamic (.so) libraries. The static libraries must be linked to the application. The dynamic libraries must be copied in the directory /lib, where they will be used during the link phase and during the application execution. It may be necessary to use the command "ldconfig-n / lib"

To use the card, the generic ADAS driver must be installed on the system.

The minimum driver version is 1.8 for Windows, and 1.2.0 for Linux, for 32 bits.

The minimum driver version is 2.0 for Windows, and 2.1.1 for Linux, for 64 bits.

5.1 Digilib library specification

The library uses the notions of exchange, frame, and equipment item. These notions are used in configuring the card's functions.

An exchange (or message) comprises commands and data. This is the Digibus protocol's "base packet". The library allows these exchanges to be defined for reception or transmission, along with access to associated data. The command notion is useful only for defining exchanges.

A redundant bus can be managed: the bus on which the BC circulates an exchange can be specified and the RT will reply on the same bus.

The application can be warned of certain messages circulating, which is essential for RT management to work. In this case the library calls a function supplied by the application and sends it the handle of the message concerned (callback notion).

Known limitations are as follows:

- An exchange comprises at most 5 commands besides the complementary command. This choice has not proved restrictive in any system we have yet encountered. If needs be, it can be increased simply by modifying a constant in the library
- No sub-bus management as described in the standard
- When circulation of an exchange is detected, response time is governed by the processor and the OS; under Windows this time is a few hundred µs
- message circulation response time is limited by what the application does in the callback

5.1.1 Error management

Most library functions return a value of 0 (FALSE) or 1 (TRUE) to indicate the result of the requested operation. In the event of an error, the function returns FALSE and further information can be obtained with function `digiLastError()`, which gives the last error number. An explanatory error message can be obtained with the function `digiErrorMessage()`.

In the event of a `digiOpen()` function error, the `digiLastError()` function cannot be used as there is no context for storing the last error. The `digiOpen()` function directly returns an error number.

5.1.2 Library states

The library is a state machine, in which state changes occur when certain functions are called. No change of state occurs on the library's own initiative; the library is slave to the application.

The three library states correspond to stages in the use of the Digibus. Library functions are associated with certain states; other functions cause a change from one state to another.

DIGI_OPENED state

This state is entered by `digiOpen()` and exited by `digiClose()`. In this state, we define the Digibus elements: equipment managed, exchanges, short cycles and frames.

The current definition can be cancelled with `digiClear()`.

DIGI_INITIALIZED state

This state is entered by calling `digiInitCard()` from the DIGI_OPENED state or by calling `digiStop()` from the DIGI_RUNNING state.

It can be exited by `digiStart()`, which switches to the DIGI_RUNNING state, or by *`digiClear()`* to revert to DIGI_OPENED.

Switching from DIGI_OPENED to DIGI_INITIALIZED via `digiInitCard()` indicates that all Digibus parameters are now defined and so the configuration can be sent to the card. From this moment on, data transmitted from exchanges can be initialised.

Switching from DIGI_INITIALIZED to DIGI_OPENED via `digiClear()` indicates that the current configuration is no longer of any use. The card RAM is erased along with all definitions made in the DIGI_OPENED state.

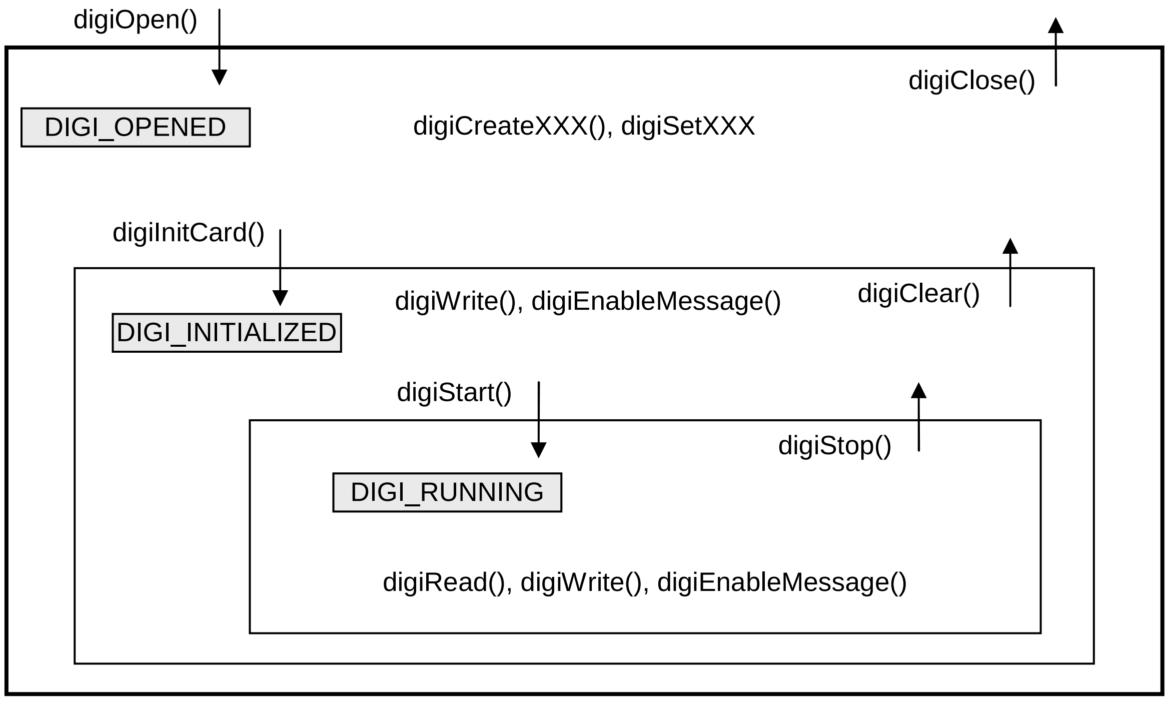
DIGI_RUNNING state

This state is entered by calling `digiStart()` from the DIGI_INITIALIZED state.

It exits to the DIGI_INITIALIZED state after a call to `digiStop()`.

In this state, the card RT and/or RT management are started and `digiRead()` and `digiWrite()` can be used to access managed RT data.

The diagram below indicates which functions cause a state change and which functions can be used in each state.



5.2 The callback function

The application can be warned when an exchange transits for which a request has been made. To do so, the library invokes the callback function supplied by the application when it called the `digiInitCard()` function.

The callback function prototype is as follows:

```
void callback (HDIGIC    hCard,
               u32       event,
               HDIGI     hMessage,
               void      * arg,
               u32       attr);
```

hCard	The handle of the card that raised a signal on the notified event. Value returned by *digiOpen()*.

event	The type of event signalled:
hMessage	The handle of the signalled message.
arg	The user argument supplied when calling *digiOpen()*.
attr	A attribute value which depends on the event.

The type of event signalled is:

`DIGI_EV_MESSAGE`	signals transiting of a message, whose handle is hMessage parameter. The signalling is requested with `digiCreateMessage()`.
`DIGI_EV_MSGERROR`	signals the passage of a message whose handle is hMessage. The signaling has not been requested by `digiCreateMessage()`, but the combination of global and message masks resulted in the generation of this event. Attr parameter contains the error status word of this message.
`DIGI_EV_CYCLEOVFL`	A cycle has spilled over into the next. The BC generates this event when it encounters the operator opCycle, and the period of the cycle is exceeded. The identifier of the operator opCycle met is provided in the parameter attr. This event is sent before the event DIGI_EV_CYCLEEND
`DIGI_EV_CYCLEEND`	The BC has just met an operator opCycle. This indicates that the exchange of a cycle just ended and the BC awaits the start of the next cycle. `Attr` parameter contains the value supplied when creating the opCycle operator
`DIGI_EV_BCEVENT`	The BC has just met an operator opEvent. `Attr` parameter contains the value supplied when creating the event operator.
`DIGI_EV_FRMEND`	The BC finished the frame to circulate the specified number of times (instead of using infinite).
`DIGI_EV_ITBC`	Signaled when the Bus Controller execute an interrupt operator created by `digiCreateInterruptOp()`.
`DIGI_EV_CMDOVFL`	Signaled when the card command FIFO overflow: the message signaling is stopped. This is a serious error, who can be reseted only by closing the card.

The callback is executed by the thread that manages the card messages FIFO. The execution of the callback should be as fast as possible so as not to disrupt the processing of the FIFO. Similarly the number of messages which request signalling should be reasonable, eg if the reporting of all messages is requested, it is likely that the FIFO overflow, and that the callback receives the event DIGI_EV_CMDOVFL.

5.3 The S_DIGIBUSLN structure

This structure contains all the information acquired by the spy concerning a Digibus message.

```
typedef struct
{
  u32  date1;                              /*!< Irig first date word */
  u32  date2;                              /*!< Irig second date word */
  u16  descriptor;                         /*!< Message identifier */
  u16  data_count;                         /*!< Number of data bytes */
  u16  cmd_count;                          /*!< Number of command bytes: 1 cmd = 2 bytes */
  u8   cmd        [DIGIBUSLN_MAX_CMD*2];   /*!< Table of commands */
  u8   cmd_qual   [DIGIBUSLN_MAX_CMD*2];   /*!< Table of command qualifiers */
  u8   echo       [DIGIBUSLN_MAX_CMD];     /*!< Table of echoes */
  u8   echo_qual  [DIGIBUSLN_MAX_CMD];     /*!< Table of echo qualifiers */
  u8   data       [DIGIBUSLN_MAX_DATA];    /*!< Data table */
  u8   data_qual  [DIGIBUSLN_MAX_DATA];    /*!< Data qualifier table */
  u16  echo_offset [DIGIBUSLN_MAX_CMD];    /*!< Time offset of echo in µs */
  u16  data_offset;                        /*!< Time offset of first databyte in µs */

} S_DIGIBUSLN;
```

Field descriptions:

- **date1 and date2** are the IRIG date words used to date stamp the 1st message bit. A description of these words is shown in the IRIG-B section.

- **descriptor** is a set of message state indicator bits, comprising,
  - `DIGIBUSLN_DESCRIPTOR_BUSB:` if this bit is set, the message circulated on bus B, otherwise it circulated on bus A.
  - `DIGIBUSLN_DESCRIPTOR_ERR:` at least one character in the message contains an error, the following indicators help determine the type of error. The character qualifiers can be examined to reveal which character is concerned.
  - DIGIBUSLN_DESCRIPTOR_CMDP: parity error on a command character.
  - DIGIBUSLN_DESCRIPTOR_CMDF: format error on a command character.
  - DIGIBUSLN_DESCRIPTOR_DATAP: parity error on a data character.
  - DIGIBUSLN_DESCRIPTOR_DATAF: format error on a data character.
  - DIGIBUSLN_DESCRIPTOR_ACKA: The address of an acqknowledge does not match the command.
  - DIGIBUSLN_DESCRIPTOR_ACKQ: Q field of an acqknowledge is not zero.
  - DIGIBUSLN_DESCRIPTOR_ACKV: V Bit of an acqknowledge is 0.
  - DIGIBUSLN_DESCRIPTOR_ACKP: parity error on acqknowledge.
  - DIGIBUSLN_DESCRIPTOR_ACKM: missing acqknowledge.

- **data_count**: the number of data characters received and placed in the data table, value in the range 0 to `DIGIBUSLN_MAX_DATA`.

- **cmd_count**: the number of procedure characters received and placed in the cmd table, value in the range 0 to `DIGIBUSLN_MAX_CMD*2`.

`DIGIBUSLN_MAX_CMD` actually corresponds to a quantity of Digibus commands, each having 2 characters. The value cmd_count/2 is the number of available values in the tables "echo" and "echo_qual".

- **cmd**: the table of procedure characters, stored in their order of circulation on the bus. The order of the characters in a command (which invariably contains two characters) might not match the endianness of the system. The use of function *`ntohs()`* is advised when using command words.

- **cmd_qual**: procedure character qualifiers, stored in their order of circulation on the bus. These qualifiers comprise the following bits:

- `DIGIBUSLB_BITV`	If this bit is set, the character's bit V has the value 1
- `DIGIBUSLB_FRAME_ERR`	The character has at least 1 malformed bit
- `DIGIBUSLN_BY_ERR`	The character has a parity error
- `DIGIBUSLB_COMPLETE_ERR`	The character has less than 10 bits

  `DIGIBUSLN_ERR_MASK` is a mask for use in testing for the presence of the three errors in a qualification character.

- **echo**: the table of preselection acknowledge characters , stored in their order of circulation on the bus. If an echo is absent, it is replaced by the value 0 and is signalled by a qualifier in "echo_qual" (see below). There are therefore as many acknowledge in this table as there are commands in cmd (echo_count = cmd_count/2)**.**

- **echo_qual**: the preselection acknowledge characters qualifiers, stored in the order of their circulation on the bus. These qualifiers use the same bits as "'cmd_qual" (see above), plus one more bit:

- `DIGIBUSLN_NO_ECHO`	This bit indicates that the acknowledge for this command has not circulated and has been detected as missing.

- **data**: the table of data characters, stored in their order of circulation on the bus. The order of characters making up 16-bit words might not match the endianness of the system. The use of function *`ntohs()`* is advised when constructing data words.

- **data_qual**: the data character qualifiers, stored in their order of circulation on the bus. These qualifiers are the same as for "cmd_qual" (sea above).

- **echo_offset**: a table giving the time in microseconds between receiving the 1st bit of the exchange and the last bit of each echo. To obtain the echo start time, subtract 10 from this value.

- **data_offset**: the time in microseconds between receiving the 1st bit of the exchange and the end of the last bit in the first data character. To obtain the start time of the first data character, subtract 10 from this value.

5.4 Error codes

The error codes used by the library are

`DIGI_ENONE`	No error
`DIGI_EOPENDEVICE`	Device opening error
`DIGI_EDOWNLOAD`	FPGA initialisation error
`DIGI_EARG`	A function argument has an invalid value
`DIGI_EMEMORY`	Memory allocation error
`DIGI_ESTATE`	Function disallowed in the library's current state
`DIGI_EBADCMD`	Invalid command or number of commands in a Digibus message
`DIGI_EDATASIZE`	Inconsistent data size between two exchanges containing the same command.
`DIGI_EDATAOVFL`	Too much data for a Digibus message
`DIGI_ETOODATA`	Too much data to fit in the card RAM
`DIGI_ETOODESC`	Too many descriptors to fit in the card RAM
`DIGI_ETOOTABLE`	Too many command tables to fit in the card RAM
`DIGI_EDUPLICATED`	Hash table creation error, duplicate message
`DIGI_ETHREAD`	Thread creation error
`DIGI_ALREADYEXIST`	Spy already running
`DIGI_ECARDNOTFOUND`	The Digibus card has not been found for one of the following reasons: - There is no card in the system or the number supplied is too high - The function number in the PLX PROM is invalid (Full vs Spy digicool) - The FPGA identification register value is invalid (error of .hex file)
`DIGI_MAXFRAME`	Maximum number of supervisor frames reached
`DIGI_REGINTR`	PcigRegisterUserInt error
`DIGI_INITDMA`	PcigInitializeLoopedDMAChannel error
`DIGI_STARTDMA`	PcigStartDMAChannel error
`DIGI_SPYNOTREADY`	The spy function is not in the expected state
`DIGI_FRAMETOOBIG`	The BC frame is too big to fit in the card RAM
`DIGI_ELABELOP`	Invalid label in for label operator.
`DIGI_EJUMPOP`	too many jump, or undefined label
`DIGI_SPYONLY`	Not available in spy only card
`DIGI_EDRIVERVER`	Library/driver or 32/64 bits incompatibility.

5.5 Digibus Reference manual

digiDriverVersion () digiDriverVersion ()

Syntaxe :

```
u32 digiDriverVersion (void)
```

Description :

Returned value contain the version number of the PCIG driver. The version number can be up to four components in each byte of the return value. The major component is the most significant byte and the minor component is in the low byte..

Windows driver has 2 components, example: 1.8
Linux driver has 3 components, example: 1.2.0

Exemple of Windows driver version:

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

digiLibVersion () digiLibVersion ()

Syntaxe :

```
u32 digiLibVersion (void)
```

Description :

Returned value contain version and revision numbers of the digilib library in the two upper bytes.

Exemple:

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

digiGetCardCount () digiGetCardCount ()

Syntax:

```
u32 digiGetCardCount (u32 flags)
```

Description:

Returns the number of DigiCool cards recognised by the system.

flags	Indicates the type of card to search:	
	0	Full card (PMC-DIGICOOL)
	DIGI_DGCSPY	Spy only card (PMC-DIGICOOL-S)

digiGetCardType () digiGetCardType ()

Syntaxe :

```
u32 digiGetCardType (u32    pCardType,
                     u32    cardNumber,
                     u32    flags)
```

Description :

Return a card descriptor word.

pCardType	A pointer to the address where to copy the result word.
cardNumber	Number of which Digibus card to get the descriptor. This number must lie between 0 and the number returned by *digiGetCardCount()*.
flags	Card selector, combination of bits from:

DIGI_DGCSPY Spy card only (eg PMC_DIGICOOL-S). A spy card can not be used for BC or RT features.

DIGI_DGCPCINUM The number is a PCI slot number, not a card number (not implemented).

The returned descriptor word values are: :

DIGI_CARD_PMC_DIGICOOL

DIGI_CARD_EI_DIGICOOL

DIGI_CARD_PCI_DIGICOOL

Each of these values is a combination(overall) of the following bits::

DIGI_CARD_FPGA_DOWNLOAD	The card request firmware downloading (PMC_DIGICOOL).
DIGI_CARD_IRIG	The card have an IRIG interface.
DIGI_CARD_RTC	The card have a 10 MHZ RTC clock.
DIGI_CARD_STUB	The card can select long/short Digibus stub.

This function can be called before card opening with digiOpen().

The function returns DIGI_ENONE if successful, else `DIGI_ECARDNOTFOUND`.

digiLastError () digiLastError ()

Syntax:

```
u32 digiLastError (HDIGIC hCard)
```

Description:

This function returns the number of the last error on the card whose handle was given as the argument.

digiErrorMessage () digiErrorMessage ()

Syntax:

```
char * digiErrorMessage (u32 error)
```

Description:

This function returns a pointer to a character string explaining the error whose number was given as the argument. This error number is returned by digiOpen() or digiLastError().

digiOpen () digiOpen ()

Syntax:

```
u32 digiOpen ( HDIGIC         * phCard,
               u32            cardNumber)
               DIGI_CALLBACK  *pCallback,
               void           * arg,
               u32            flags,
               char           * designName)
```

Description:

Opens the specified Digibus device whose number is given as an argument.

phCard	A pointer to a HDIGIC object, set by the function.
cardNumber	Number of which Digibus card to open. This number must lie between 0 and the number returned by *digiGetCardCount()*.
pCallback	Name of the function to be called when an event is signalled to the application.
arg	Application argument transmitted to the callback when invoked.
reserved	Reserved for future use, must be set to 0.
flags	flags indicates the type of card to search:
DIGI_DGCSPY	Spy only card. A spy card can not be used for BC or RT features.
	DIGI_DGCPCINUM cardNumber is a PCI slot number, not a card number (not implemented).
designName	The filename of FPGA firmware.

The PMC_DIGICOOL card requires the load of its firmware by `digiOpen()`. By default the library une a FPGA firmware named "pmc_digicool.hex", which musb be placed in the application's current directory. In this case the parameter `designName` must be NULL. If the firmware file has another name or is not in the current directory the application must supply the path of this file through `designName`.
For other cards `designName` has to be NULL.

Any return value different from DIGI_ENONE is an error number, in which case the device is not opened.

A return value of DIGI_ENONE indicates the device is open and the value pointed to by phCard is a handle to be given as first argument in most other Digibus library functions.

Function `digiClose()` must be called when the device is no longer in use.

digiClose () digiClose ()

Syntax:

```
          void digiClose (HDIGIC hCard)
```

Description:

Closes the Digibus device and frees up all used resources.

digiCardVersion () digiCardVersion ()

Syntaxe :

```
          u32 digiCardVersion (HDIGIC hCard)
```

Description :

The return value give version/revision values:

- général FPGA firmware in high word,
- Digibus IP il low word

Exemple:

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

digiIrigHandle () digiIrigHandle ()

Syntaxe :

```
HANDLE digiIrigHandle (HDIGIC hCard)
```

Description :

This function give the Irig hange to be used with irig functions o the card

digiSetOption () digiSetOption ()

Syntaxe :

```
HANDLE digiSetOption (HDIGIC   hCard,
                      u32      option,
                      u32      arg1,
                      u32      arg2)
```

Description :

Allows to configure particular characteristics of certain cards. All the cards do not allow the same options.

option	The option to set.
arg1	The first parameter of the option
arg2	The second parameter of the option

The available options are:

`DIGI_OPT_SHORTSTUB`	Select short Digibus stub. No parameter.
`DIGI_OPT_LONGSTUB`	Select long Digibus stub. No parameter.
`DIGI_OPT_SHORTSTUBL`	Select short Digibus stub with lines loads. No parameter.
`DIGI_OPT_RXTHRA`	Reception threshold for the bus A. Arg1 indicates the threshold for long stub, arg2 indicates the threshold for the short stub, between 0 and 1023.
`DIGI_OPT_RXTHRB`	Reception threshold for the bus B. Arg1 indicates the threshold for long stub, arg2 indicates the threshold for the short stub, between 0 and 1023..
`DIGI_OPT_TXVOLT`	Signal evel of emission, common to both buses. Arg1 indicates the level for the long stub, arg2 the level for the short stub, the values between 0 and 1023.
`DIGI_OPT_RTCSET`	RTC counter initialization: arg1 are the 23 higher bits of the value, arg2 are tyhe 24 lower bits of the value.

`DIGI_OPT_RTCEN`	RTC counter commands:
	Arg1 = 1 : Start,
	Arg1 = 0 : Stop.

`DIGI_OPT_RTCCLKEXT`	Select RTC counter clock source: Arg1 = 1 : External (from the backplane), Arg1 = 0 : Local clock of the card.

Remark: reception thresholds and emission level are settled by factory, it is not recommended to change them

The function returns TRUE value if successful. Else return FALSE and lastError show the cause as:

- `DIGI_ENOTALLOWED` for not supported option.

- `DIGI_EARG` for unknown option.

digiGlobalErrorMask() digiGlobalErrorMask()

Syntaxe :

```
          void digiGlobalErrorMask (HDIGIC hCard, u32 errorMask)
```

Description :

This function allows you to configure events and errors on exchanges that should be reported to the callback. It is used after creating the message, and before using `digiInitcard()`.

errorMask	A combination of bits forming the mask.

By default all events and errors are masked.

The bits used for the reporting of events:

DIGI_EVM_EOFRAME	End of frame
DIGI_EVM_BCEVENT	Event Manager (eventOp
DIGI_EVM_CYEND	End of cycle
DIGI_EVM_CYOVFL	overflow cycle

Other events are not maskable.

The bits used for the reporting of errors:

DIGI_ERM_CT	test command error.
DIGI_ERM_TX_LENGTH	ccount error on transmit.
DIGI_ERM_RX_LENGTH	count data reception error.
DIGI_ERM_COD_LD	coding error on on data line.
DIGI_ERM_COD_SYC	coding error on procedure line, phase tempo or sync.
DIGI_ERM_ACK_QFIELD	acknowledge Q field non-zero.
DIGI_ERM_ACK_AFIELD	acknowledge with a wrong address.
DIGI_ERM_NO_ECHO	a command of the exchange has not sent acknowledge.
DIGI_ERM_BIT_V_ECHO	acknowledge with V bit to 0.
DIGI_ERM_BIT_V_DATA	data byte with V bit to 0.
DIGI_ERM_PAR_ECHO	parity error on an acknowledge.

DIGI_ERM_PAR_DATA	parity error on a data byte.
DIGI_ERM_PAR_SYNC	parity error on the sync phase (Procedure line).
DIGI_ERM_PAR_CMD	parity error or coding of a command word.

These errors bits can be used to decrypt the error status word of events DIGI_EV_MESSAGE and DIGI_EV_MSGERROR or function `digiReadBufferErr()`, or build specific exchange error mask for `digiMessageAttr()`.

For an error generates an event, the error bit must be present in the global mask and the message mask, see `digiMessageAttr()`.

digiInitCard () digiInitCard ()

Syntax:

```
u32 digiInitCard (HDIGIC hCard)
```

Description:

This function signals that the exchange configuration is complete and can be sent to the card RAM.

Is to be used after creating the messages, and before creating the BC frame if necessary. From this point it is possible to use `digiWrite()` to initialize the data buffers messages, then use `digiStart()`.

The function returns TRUE value if successful.

digiClear () digiClear ()

Syntax:

```
u32 digiClear (HDIGIC hCard)
```

Description:

This function releases all the items used by the interface: exchanges, cycles, frames, etc.

It cannot be used in the DIGI_RUNNING state; it is for use after the interface has been stopped by `digiStop()` and before any new elements are defined.

digiCreateEquipment () digiCreateEquipment ()

Syntax:

```
u32 digiCreateEquipment ( HDIGIC   hCard,
                          u32      number,
                          u32      flags)
```

Description:

A managed equipement is a remote terminal responding to commands issued by the supervisor, by a preselection acknowledgement (echo), and possibly by data.

The managed remote terminal must be created before creating any exchanges that use it. No equipment must be created that is not managed by the card.

number	The equipment number, which must lie between 0 and 31.
flags	Configuration indicators

`DIGI_ATTR_CVEN`	Enable equipment management by CV255 and CLV255 commands.
`DIGI_ATTR_DISBUSA`	inhibits RT on bus A.
`DIGI_ATTR_DISBUSB`	inhibits RT on bus B.

Indicators DIGI_ATTR_DISBUSA and and DIGI_ATTR_DISBUSB are taken into account if DIGI_ATTR_CVEN is present. By default the RT is not sensitive to commands CLV255 or CV255 and is validated on two buses.

A default (standard) acknowledge value is associated with the equipment. To change the acknowledge value, use `digiSetEcho()` after `digiCreateEquipment()`.

A return value of TRUE indicates a successful declaration.

digiEnableEquipment() digiEnableEquipment()

Syntaxe :

```
u32 digiEnableEquipment (HDIGIC  hCard,
                         u32     number,
                         u32     flags)
```

Description :

This function allows you to change the sensitivity of the RT to messages CLV255 and CV255, and its ability to transmit on the bus.

Number	The address of equipment that must be between 1 and 31.
Flags	Indicators of configuration flags:

`DIGI_ATTR_CVEN`	Enable equipment management by CV255 and CLV255 commands.
`DIGI_ATTR_DISBUSA`	inhibits RT on bus A.
`DIGI_ATTR_DISBUSB`	inhibits RT on bus B.

Indicators DIGI_ATTR_DISBUSA and DIGI_ATTR_DISBUSB are taken into account if DIGI_ATTR_CVEN is present.

A return value of TRUE indicates a successful declaration.

Syntax:

```
HDIGI digiCreateMessage (HDIGIC         hCard,
                         S_DIGI_MESSAGE * pMessage)
```

Description:

This function defines all the parameters of a Digibus exchange using an S_DIGI_MESSAGE structure.

The exchanges must be defined after creating the equipment and before creating the frames.

After creating all the messages it is necessary to call `digiInitCard()` before creating the BC frames or start the RT simulation by `digiStart()`.

The return value is non NULL if successful; this value is used as the exchange identifier.

The S_DIGI_MES structure:

```
typedef struct
{
   u32     commands [DIGI_MAXCMD];  // The commands, including CC
   u16     dataSize;                // Number of data bytes
   u16     nbTempo;                 // Number of delay characters
   u16     bus;                     // Bus ID: DIGI_BUSA or DIGI_BUSB
   u16     interrupt;               // Signal request (callback)

} S_DIGI_MESSAGE;
```

commands

Commands are made up of two parts:

- The command itself which is assembled from the equipment number, the Digibus command, and field X. A command can be constructed using the DIGI_CMD macro.
- The command attributes can be either of the following:
  - DIGI_GENERRPARITY: generates a parity error on this command
  - DIGI_GENERRFORMAT: generates a format error on this command

The DIGI_CMD macro constructs a command word from its components and has the following syntax:

```
u32 DIGI_CMD (u32 equipment, u32 cmd, u32 X)
```

"equipment" is the RT address, a value between 0 and 31.

Cmd is one of the following:

DIGI_CC	Complementary command
DIGI_CF	Function command

DIGI_CR	Reception command
DIGI_Cxx	Reserved in the standard
DIGI_THIS	Transmission command
DIGI_CV	Reserved in the standard, sometimes called channel command.
DIGI_CL	Label command.
DIGI_CT	Test command.

X is a value between 0 and 255.

dataSize

This is the number of data bytes associated with the exchange. It must be null or less than 257.

nbTempo

The number of delay characters the BC inserts between the complementary command and the data for this exchange. This number is limited to 128 by the library.

bus

The bus number on which the exchange will take place if the card is in redundant mode (default). This value must be DIGI_BUSA or DIGI_BUSB.

interrupt

A non-null value requests the end of this exchange to be signalled to the application, calling the callback specified by `digiOpen()`.

digiMessageAttr() digiMessageAttr()

Syntaxe :

```
u32 digiMessageAttr (HDIGIC   hCard,
                     HDIGI    hMessage,
                     u32      attr,
                     u32      value)
```

Description :

Set additional attributes to describe an exchange, to use after the creation of the exchange, and before to transmit the configuration to the card with `digiInitcard()`.

hMessage	The handle of the exchange which must change an attribute.
attr	The name of the attribute to change
value	The value of the attribute

Attribute names:

DIGI_ATTR_DELAY	value is intermessage delay following this message in microseconds, between 20 and 4095. The default time is 20 microseconds.
DIGI_ATTR_ERRORS	Value is the mask of errors which are going to trigger a callback event if they are detected during the exchange. By default all errors trigger an event.
DIGI_ATTR_ERRSYNC	Request to generates a coding or parity error on procedure line during the sync phase (tempo or data).

Value consists of bits DIGI_GENERRPARITY and/or DIGI_GENERRFORMAT and the rank of the byte which generate the error.

The return value is TRUE if there is no error.

digiMessageSetCC() digiMessageSetCC()

Syntaxe :

```
u32 digiMessageSetCC (HDIGIC  hCard,
                      HDIGI   hMessage,
                      u32     command)
```

Description :

Changes the value of X field of the complementary command of the exchange specified.

To be used after the creation of frame, can be used during BC operation.

hMessage	The handle of the exchange that must change the X field of CC.
command	The X field value (0-255) plus any generating errors attributes DIGI_GENERRPARITY or DIGI_GENERRFORMAT

The return value is TRUE if there is no error.

digiMessageGetCC() digiMessageGetCC()

Syntaxe :

```
u32 digiMessageGetCC (HDIGIC  hCard,
                      HDIGI   hMessage,
                      u32     * pXField)
```

Description :

Retrieves the X field of the last occurrence of the exchange that circulated on the bus.

hMessage	The handle of exchange that must read the CC X field.
pXFiels	A pointer to a word that contains the X field value of the CC at function return.

The return value is TRUE if there is no error.

digiEnableMessage () digiEnableMessage ()

Syntaxe :

```
u32 digiEnableMessage ( HDIGIC hCard,
                        HDIGI  * pList,
                        u32    nb,
                        u32    enable)
```

Description :

With this function, in BC mode, the circulation of messages can be enabled or disabled.

pList	A pointer to an array containing the handles of the messages concerned.
nb	The count of handles dans pList.
Enable	TRUE to enable the exchanges, FALSE to inhibit the exchanges.

The return value is TRUE if there is no error.

digiCreateDelayOp () digiCreateDelayOp ()

Syntax:

```
          HDIGI digiCreateDelayOp (u32 delay)
```

Description:

This function defines a delay operator between messages that can be used for the creation of cycles or frames.

The value "delay" is the time lag in microseconds and must be less than 65536.

The library automatically inserts a standard delay of 20µs between two exchanges. Any explicit delay is therefore added to the standard delay.

A delay can be created at the moment of its insertion into a cycle or frame, because this function doesn't allocate resources.

The return value is a handle for use when constructing cycles or frame.

digiCreateInterruptOp () digiCreateInterruptOp ()

Syntaxe :

```
          HDIGI digiCreateInterruptOp ()
```

Description :

The use of this function is no longer recommended. It can advantageously be replaced by digiCreateEventOp ().

This function defines an interrupt operator that can be used for the creation of cycles or frames. This operator is designed to allow the application to be warned at specific locations of the frame executed by the Bus Controller.

An interrupt operator is placed in the frame used by the manager. When the manager execute this operator, it generates an interrupt to the CPU. This interruption is managed by the library who calls the callback, specified by digiOpen (), with the event DIGI_EV_ITBC to alert the user. This operator is

A delay can be created at the moment of its insertion into a cycle or frame, because this function don't allocate resources.

The return value is a handle for use when constructing cycles or frame.

digiCreateTriggerOp () digiCreateTriggerOp ()

Syntax:

```
HDIGI digiCreateTrigger (void)
```

Description:

This function defines a trigger operator, i.e., a request to generate an impulse on the card’s trigger output at the moment the supervisor encounters this request. This operator must be used in the creation of cycles.

A trigger can be created a the moment of its insertion into a cycle or frame, because this function don't allocate resources.

The return value is a handle for use when constructing cycles or frame.

digiCreateLabelOp () digiCreateLabelOp ()

Syntaxe :

```
HDIGI digiCreateLabelOp (u32 label)
```

Description :

This feature allows you to define a label operator, which allows to memorize the address where it is placed in the frame

`label`	The number to be used by the operator to jump to this label. The number 0 is reserved to identify the beginning of the frame. The numbers available are 1 to 7.

A label can be created at the moment of its insertion into a cycle or frame, because this function don't allocate resources.

The return value is a handle for use when constructing cycles or frame.

digiCreateJumpOp () digiCreateJumpOp ()

Syntaxe :

```
HDIGI digiCreateJumpOp (u32 label)
```

Description :

This feature allows you to define a jump operator, to divert the execution of the frame to the label indicated.

`label`	The label number which must have been previously defined

A jump can be created a the moment of its insertion into a cycle or frame, because this function don't allocate resources.

The return value is a handle for use when constructing cycles or frame.

The use of the label and jump operators can cut the frame into an initialization and a cyclic parts

digiCreateEventOp() digiCreateEventOp()

Syntaxe :

```
        HDIGI digiCreateEventOp (u32 eventId)
```

Description :

This feature allows to define an event operator, designed to allow the application to be signaled at specific locations of the frame used by the BC. This operator is used when a BC frame is built with `digiCreateFrame()`.

`eventId` Value of the operator 0 to 255, this value is passed to the callback.

During the execution of this operator, the BC causes the call to the callback with the event DIGI_EV_BCEVENT, and the value `eventId` in the parameter attr.

This operator can be created at the time of its inclusion in a cycle/frame since its creation does not allocate resource other than its return value.

The return value is a handle for use when constructing cycles or frame.

digiCreateCycleOp () digiCreateCycleOp ()

Syntaxe :

```
        HDIGI digiCreateCycleOp (u32 cycleId)
```

Description :

This function create a start of cycle operator. This operator allow thez BC to wait for the next start of cycle signal. This operator is used when buiding frame with `digiCreateFrame ()`.

A cycle operator can be created a the moment of its insertion into a frame, because this function don't allocate resources.

`cycleId` Identifier of the operator 0 to 255. This identifier is passed to the callback event with DIGI_EV_CYCLEEND if requested by errors masks.

The return value is a handle for use when constructing cycles or frame.

digiCreateCycle () digiCreateCycle ()

Syntax:

```
        HDIGI digiCreateCycle ( HDIGIC  hCard,
                                HDIGI   * pList,
                                u32     nb,
                                u32     cycleId)
```

Description:

This function defines a short cycle used in constructing a frame. The library automatically add a cycle operator at beginning of every cycle, when calling *digiCreateFrame()*.

pList	A pointer to an arraycontaining the handles of messages and operators (delays, trigger…) constituting the short cycle.
nb	The number of handles in pList.
`cycleId`	Identifier of the operator 0 to 255. This identifier is passed to the callback event with DIGI_EV_CYCLEEND if requested by errors masks.

The return value is null in the event of failure; otherwise it is a handle for use when constructing a frame.

digiCreateFrame () digiCreateFrame ()

Syntax:

```
HDIGI digiCreateFrame (HDIGIC    hCard,
                       HDIGI     * pList,
                       u32       nb,
                       u32       clockInt,
                       u32       period,
                       u32       frameCount)
```

Description:

This function defines a frame for the BC, along with the conditions in which it will be used. The frame if made from a list of cycles.

pList	A pointer to a table containing the handles of the cycles in the frame.
nb	The number of handles in pList.
clockInt	Defines whether the clock tick for short cycles is internal (`DIGI_CLKINT`), external (`DIGI_CLKEXT`), or sorfware (`DIGI_CLKSOFT`, see `digiTrigCycle()`)..
period	Internal clock period in µs. If an external clock is used, this defines the divider to be applied to the signal (use 1 for no division).
frameCount	The number of iterations of the frame to be executed, or 0 for continuous execution.

The frames should be constructed after calling `digiInitCard()`.

The return value is null in the event of failure; otherwise it is a frame handle for use with `digiStart()`.

digiCreateFrameRaw () digiCreateFrameRaw ()

Syntaxe :

```
HDIGI digiCreateFrameRaw (HDIGIC   hCard,
                          HDIGI    * pList,
                          u32      nb,
                          u32      clockInt,
                          u32      period,
                          u32      frameCount)
```

Description :

Obsolete, use `digiCreateFrame ()`.

digiDeleteFrame() digiDeleteFrame()

Syntaxe :

```
u32 digiDeleteFrame (HDIGIC hCard, HDIGI hFrame)
```

Description :

This function frees the resources held by the description of a frame.

hFrame	The frame handle.

Note: If the frame is loaded into the card memory, this memory is not released, only the memory used by the library is released (see `digiLoadFrame()`).

The return value is zero if it fails, either because the frame handle is invalid, either because the frame is in use.

digiLoadFrame() digiLoadFrame()

Syntaxe :

```
u32 digiLoadFrame(HDIGIC hCard, HDIGI hFrame)
```

Description :

This feature allows you to load a frame in the card memory. The loading is cumulative: you can load multiple frames in the card by calling this function for each frame.

If hFrame is 0, all frames are erased from the card, which allows to load new.

The return value is zero on failure

digiStart () digiStart ()

Syntax:

```
u32 digiStart (HDIGIC hCard, HDIGI hFrame, u32 reserved)
```

Description:

This function starts subscriber management and/or the card supervisor.

hFrame	A frame handle. If this value is null, the supervisor is not started and only RT management is enabled. If hFrame is a valid frame handle, this frame is used by the BC.
reserved	Reserved for future use; must be set to 0.

If the frame hFrame was not loaded into the card previously by `digiLoadFrame()`, it is automatically loaded in the wake of frames already in the card.

`digiStart()` should not be called from the callback.

The return value is null in the event of failure.

digiRestart () digiRestart ()

Syntaxe :

```
u32 digiRestart (HDIGIC hCard, HDIGI hFrame)
```

Description :

This feature allows you to change the active frame of BC. The BC must have been started earlier by `digiStart()`.

hFrame	A frame handle must already have been loaded into the card by `digiLoadFrame()`.

`digiRestart()` can only be called from the callback. To change the frame outside the callback, use `digiSop()` / `digiStart()`.

The time of silence on the bus for frame switching is a few milliseconds.

The return value is zero on failure (invalid frame, frame unloaded, or BC not running)

digiStop () digiStop ()

Syntax:

```
u32 digiStop (HDIGIC hCard)
```

Description:

This function stops subscriber management and the bus supervisor.

It must be used before changing frames or closing the card.

`digiStop()` should not be called from the callback.

The return value is null in the event of failure.

digiTrigCycle() digiTrigCycle()

Syntaxe :

```
u32 digiTrigCycle (HDIGIC hCard)
```

Description :

If the frame in use was defined for software triggering, this function trigger the next cycle.

The return value is zero on failure

digiState () digiState ()

Syntax:

```
u32 digiState (HDIGIC hCard)
```

Description:

This function queries the current state of the library: DIGI_OPENED, DIGI_INITIALIZED, or DIGI_RUNNING.

digiSetChar () digiSetChar ()

Syntax:

```
void digiSetChar (HDIGIC  hCard,
                  u8      data,
                  u8      tempo,
                  u8      ee,
                  u8      fd)
```

Description:

This function specifies the value of characters used in certain Digibus exchange phases.

data	Value of the character used by the supervisor on the procedure line during the data phase (service characters). Default value DIGI_DATAVALUE.
tempo	Value of the character used by the supervisor on the procedure line during the time-out phase. Default value DIGI_TEMPOVALUE.
ee	Value of the end of exchange character on the procedure line. Default value DIGI_EEVALUE.
fd	Value of the end of exchange character on the data line (final data). Default value DIGI_FDVALUE.

digiSetEcho () digiSetEcho ()

Syntax:

```
u32 digiSetEcho ( HDIGIC  hCard,
                  u32     number,
                  u32     echoValue,
                  u32     flags)
```

Description:

This function changes the value of the acknowledge sent by RT management when a command concerning a managed RT circulates.

This value can be changed while the card is running.

number	The equipment item number which must lie between 0 and 31.
echoValue	The acknowledge value that must be returned by RT when a command concerning this RT is received. If a standard value has to be returned, use the value DIGI_ECHO.
flags	Combination of bits:

`DIGI_ECHONONE`	No echo for this RT.
`DIGI_ECHOBITV0`	The V bit is set to 0, else it is managed by the card.

The echoValue field consists of the value itself and the following indicators:

`DIGI_ECHONONE`	Echoing is inhibited for this RT.
`DIGI_ECHOBITV0`	The V bit is forced to 0, otherwise it is managed by the card.
`DIGI_ECHODELAY`	The acknowledge is sent after a delay.
`DIGI_ECHOERRP`	The acknowledge is sent with a parity error.

This function must be used after setting up the RT account with `digiCreateEquipment()`.

She must not be used between *digiInitCard()* and *digiStart()*, because then the value would not be passed on the card. In other words it is necessary to position the echo during the configuration and before the transmission of this configuration to the card, either when the bus activity is started.

The return value is non null in the event of success.

digiSetEchoDelay () digiSetEchoDelay ()

Syntaxe :

```
u32 digiSetEchoDelay (HDIGIC hCard, u32 delay)
```

Description :

This function allow to specify the delay before RT send his command acknowledge. The delay is enables with *digiSetEcho()*, and the delay value is given by *digiSetEchoDelay()*. The default value is no delay.

This value can be changed while the car dis running.

delay	The delay value in card clock period, ie 20ns for PMC-DigiCool. The value must be between 1 and 255, which allows you to specify a max delay of 5.1µs.

digiReadBuffer () digiReadBuffer ()

Syntax:

```
u32 digiReadBuffer  (HDIGIC   hCard,
                     HDIGI    hMessage,
                     u8       * pData,
                     u8       * pQual)
```

Description:

This function reads the reception data associated with an exchange. The reception buffers are switched at the end of the exchange and the copied data are the last received. The same data are read as long as the same exchange, or another exchange with the same reception command, has not yet circulated.

Only data from managed equipment or from broadcast exchanges can be accessed. If the exchange does not involve a RT who is managed for reception, or if the message is not broadcast reception, reading will fail.

An exchange transmission buffer can not be read.

The "pData" and "pQual" buffers must be big enough to contain at least as many bytes as the number of data bytes declared when this exchange was created.

The "pData" buffer receives the Digibus data characters in the order they circulated on the bus: use *ntohs()* to construct 16-bit words where relevant.

The "pQual" buffer receives the qualification characters applying to the data characters. See the S_DIGIBUSLN structure description for the meaning of these qualification characters.
The "pQual" buffer is optional. If it is not used, supply a value of NULL when calling the function.

The return value is TRUE in the event of success.

digiReadBufferErr() digiReadBufferErr()

Syntaxe :

```
u32 digiReadBufferErr ( HDIGIC hCard,
                        HDIGI  hMessage,
                        u8     * pData,
                        u32    * pError)
```

Description :

This function reads the reception data associated with an exchange. The reception buffers are switched at the end of the exchange and the copied data are the last received. The same data are read as long as the same exchange, or another exchange with the same reception command, has not yet circulated.

Only data from managed equipment or from broadcast exchanges can be accessed. If the exchange does not involve a RT who is managed for reception, or if the message is not broadcast reception, reading will fail.

An exchange transmission buffer can not be read ange.

The "pData" buffer must be big enough to contain at least as many bytes as the number of data bytes declared when this exchange was created.

The "pData" buffer receives the Digibus data characters in the order they circulated on the bus: use *ntohs()* to construct 16-bit words where relevant.

`pError` is a pointer to an area that will contain the error status word of the message back from the function. This allows the application to know if she can use the acquired data. See `digiSetGlobalErrorMask()` for the meaning of the bits of error.

The return value is TRUE in the event of success.

digiWriteBuffer () digiWriteBuffer ()

Syntax:

```
u32 digiWriteBuffer (HDIGIC  hCard,
                     HDIGI   hMessage,
                     u8      * pData,
                     u8      * pQual)
```

Description:

This function writes the data to be associated with an exchange. When writing ends, the transmission buffers are switched to make the data immediately available to RT management. The same data continue being transmitted by RT as long as the application has not yet supplied any new data.

Write access is limited to data for managed equipment. If the exchange does not involve transmission, writing will fail.

An exchange reception buffer cannot be written to.

The "pData" buffer must be big enough to contain at least as many bytes as the number of data bytes declared when this exchange was created.

The "pQual" buffer indicates the attributes of the data characters to be transmitted. This buffer is optional. If it is not used, a NULL value must be supplied when calling the function. Available attributes are

DIGI_DATABITV0	Forces the V bit to 0 on this byte. The V bit in data bytes is normally 1 by default.
DIGI_DATAERRP	Generates a parity error.
DIGI_DATAERRF	Generates a format error.

The return value is TRUE in the event of success.

digiSpyStart () digiSpyStart ()

Syntax:

```
u32 digiSpyStart ( HDIGIC   hCard,
                   u32 bufferSize,
                   u32 flags)
```

Description:

The spy function can be used independently from the other features. When spying is started, a buffer is allocated in which a thread copies the contents of the card FIFO. The size of the buffer is established to suit the backlog that the application (i.e., the readers) can acquire with respect to the FIFO throughput.

hCard	The handle supplied by *digiOpen()*.	
bufferSize	The size of the FIFO buffer in 32-bit words. If this value is null, a default size is used (preferred method).	
flags	Use following values:	
	DIGI_BUSA.	Mandatory, use bus A or redundant bus.
	DIGI_SPY_POLL	Don't use DMA for spy data. The card FIFI is read at every spy read function call.

Remark: bufferSize is not used, and set to 65536 by the library.

The return value is TRUE in the event of success.

digiSpyStop () digiSpyStop ()

Syntax:

```
u32 digiSpyStop (HDIGIC hCard)
```

Description:

This function stops the spy feature and frees up resources (buffer memory and thread).

The readers must be closed with `digiSpyReaderClose()` before spying is ended.

hCard The handle supplied by digiOpen().

The return value is TRUE in the event of success.

digiSpyReaderOpen () digiSpyReaderOpen ()

Syntax:

```
HDIGIC digiSpyReaderOpen (HDIGIC hCard, u32 flags)
```

Description:

Several consumers (readers) can use the spy feature at the same time. Each consumer possesses its own environment: its position in the spy buffer, its S_DIGIBUSLN message presentation structure.

A reader is created specifying which card bus to use. As things stand, the card can manage only one redundant bus, so the bus argument is always 0.

During creation, the reader is initialised to read the next Digibus messages received: it can not access any messages circulating before it was created.

The readers must be created after the spy function has been started using `digiSpyStart()`. They must be closed with `digiSpyReaderClose()` before spying is ended.

hCard	The handle supplied by `digiOpen()`.
flags	Configuration indicators

`DIGI_BUSA`	spy of the redundant bus, or bus A or in the case of two independent buses.
`DIGI_BUSB`	spy bus B in the case of two independent buses.

Note: Currently only the redundant bus is supported (`DIGI_BUSA`).

Spy data can be retrived with *digiSpyReaderMessage()* or *digiSpyReaderRaw()*.

The function returns a value of null in the event of error. Otherwise this value is a handle for use in other functions involving a spy reader.

digiSpyReaderReset () digiSpyReaderReset ()

Syntax:

```
void digiSpyReset (HDIGIC hReader)
```

Description:

The reader reception index is reset with the current index of the reception thread. The next message the Digibus receives will be the first the reader can access using function `digiSpyReaderMessage()`.

This function is used in the event of the reader building up excessive backlog with respect to the receiver, in which case the `digiSpyReaderMessage()` function returns SPY_WRITEOVER.

hReader	The handle supplied by `digiSpyReaderOpen()`.

digiSpyReaderClose () digiSpyReaderClose ()

Syntax:

```
void digiSpyReaderClose (HDIGIC hReader)
```

Description:

Reader resources are freed up and the reader must no longer be used.

The readers must be released before spying is ended by `digiSpyStop()`.

hReader	The handle supplied by `digiSpyReaderOpen()`.

Syntax:

```
S_DIGIBUSLN * digiSpyReaderMessage ( HDIGIC   hReader,
                                     u32      * pResult)
```

Description:

This function retrieves the next message on the bus the reader was created for.

The message is constructed in a S_DIGIBUSLN type structure whose address is returned to the application. The contents of the structure are available until the next call to `digiSpyReaderMessage()` with the same reader. These contents must not be modified.

hReader	The handle supplied by `digiSpyReaderOpen ()`.
pResult	A pointer to a variable in which the function stores its result.

The following function result indicators are placed in the variable pointed to by pResult:

`DIGI_ITEMOK`	This value is returned when a message was readable, in which case the function return value is non NULL. This does not imply valid message contents: the message is presented exactly as it circulated on the bus.
`DIGI_NOTRECEIVED`	There are no more messages left for reading in the reception FIFO for now. Try later.
`DIGI_WRITEOVER`	The reader was not quick enough; words were lost in the reception FIFO. The reader must be reset with *digiSpyReset()* before reading can continue.
`DIGI_RETRY`	The reader encountered an inconsistency in the reception FIFO and abandoned reception of this message. Retry to get the next message.
`DIGI_SPY_OVFL`	When the spy is used without DMA, (polling mode), thi bit is set when the card FIFO overflows.

The S_DIGIBUSLN structure is described in file "digibusln.h"

The return value is non NULL when a message is available. This value is the message structure address. If the return value is NULL, the variable pointed to by pReturn gives the reason.

digiSpyReaderRaw() digiSpyReaderRaw()

Syntaxe :

```
u32 S_DIGIBUSLN * digiSpyReaderRaw ( HDIGIC  hReader,
                                     u32     * pResult,
                                     u32     * pData,
                                     u32     maxLength)
```

Description :

This feature allows you to obtain the raw data words from the spy of the card.

hReader	The handle provided by `digiSpyReaderOpen()`.
pResult	A pointer to a variable that indicates the function's result, see `digiSpyReaderMessage()`.
pData	Result array

maxLength The number of words available in pData.

The return value is the number of words read if pResult indicates DIGI_ITEMOK.

The format of date words is indicated in the chapter on the IRIG-B.

The remaining data words have the following format:

rmat of characters from the procedure line

Champ	Description	Valeur
31	DIGIBUS character indicator. This bit qualifies the word indicates that it represents a DIGIBUS character, not a date	1
30 : 16	Dating the appearance of the character on the DIGIBUS procedure line. This field indicates the relative time of receipt of this character since the beginning of the message expressed in microseconds. This is actually the detection time of the receiving end of character, to know the relative time of the first bit of the character subtract 10	x_H
15	This is the first character from procedure line.	x_H
14	This character come from procedure line	0
13	If bit 14 is 0 and bit 13 is 1, then the acknowledge of the previous command is missing.	x_H
12	Coding error on bits of this character	x_H
11	Indicator of incompleteness. This bit indicates that the received character does not have 10 bits, as expected.	x_H
10	Reserved	x_H
9	V Bit of character	x_H
8	Parity error	x_H
0 : 7	The character value, without parity and V bits.	x_H

Format of characters from the data line:

Champ	Description	Valeur
31	DIGIBUS character indicator. This bit qualifies the word indicates that it represents a DIGIBUS character, not a date.	1
30 : 16	Dating the appearance of the character on the DIGIBUS procedure line. This field indicates the relative time of receipt of this character since the beginning of the message expressed in microseconds. This is actually the detection time of the receiving end of character, to know the relative time of the first bit of the character subtract 10	x
15	This is the first data character of the message	x_H
14	This character come from data line.	1
13	When bit 14 is 1 (data line character) 0 : This is a data character 1 : This is an acknowledge	x_H
12	Coding error on bits of this character	x_H
11	Indicator of incompleteness. This bit indicates that the received character does not have 10 bits, as expected.	x_H
10	Reserved	x_H
9	V Bit of character	x_H
8	Parity error	x_H
0 : 7	The character value, without parity and V bits.	x_H

digiSpySetFilter () digiSpySetFilter ()

Syntaxe :

```
void  digiSpySetFilter (HDIGIC hCard,
                        u16 word1, u16 word2,
                        u32 flags) ;
```

Description :

This feature allows you to configure the filter of the spy. This filter will generate a pulse on the output port when the combination of specified message commands runs on the bus.

hCard	The handle returned by *digiOpen()*.
word1	First command word of the message
word2	Second command word of the message, if two words are required.
flags	Supplenetary configuration.

The flags field is a combination of bits

DIGI_BUSA ou DIGIBUSB	Indicates on which bus filter. When in redundant mode, use DIGI_BUSA
DIGI_FILTER1	Use only one command word
DIGI_FILTER2	Use tho command words
DIGI_ENABLE ou DIGI_DISABLE	Start or stop filtering.

digiMessageDataSize () digiMessageDataSize ()

Syntax:

```
u32 digiMessageDataSize (HDIGIC hCard, HDIGI hMessage)
```

Description:

This function retrieves the size in bytes of the message whose handle was given as an argument.

digiFindMessage8/32() digiFindMessage8/32 ()

Syntax:

```
HDIGI digiFindMessage8  (HDIGIC hCard, u8 * pCmd)

HDIGI digiFindMessage32 (HDIGIC hCard, u32 * pCmd)
```

Description:

These functions allow retrieval of the handle of a message whose list of commands is known. Theses functions are used internally by the library, and `can be used to recognize a message from the spy.`

The pCmd table must contain the message commands as provided by *digiCreateMessage()*. The function uses at most 4 commands to recognise a message.

In the *digiFindMessage8()* function, pCmd is a table of u8s. This function allows direct use of the "cmd" field of an S_DIGIBUSLN structure used by the spy.

In the *digiFindMessage32()*function, pCmd is a table of u32s, the commands being placed in the low order of the words; this is compatible with the "commands" field of the `S_DIGI_MESSAGE` structure used by *digiCreateMessage()*.

The value returned is the message handle when the list of commands is found to correspond to a recorded message, otherwise it is NULL.

digiOutPortBind () digiOutPortBind ()

.

Syntaxe :

```
void digiOutPortBind (HDIGIC   hCard,
                      u32      sigif0,
                      u32      sigif1,
                      u32      sigif2)
```

Description :

This function allows you to configure the matrix mixing of the ouput port, ie to assign a signal to each physical interface.

Each `sigifX` parameter corresponds to the physical interface `X`, and indicates the signal that it is connected to. For example to assign the signal number 2 to the interface number 0, the signal 1 at the interface 1 and signal 0 to the interface 2:

digiOutPortBind `(hCard, DIGI_SIG2, DIGI_SIG1, DIGI_SIG0)`

Remark:You can use the same signal on multiple output:

digiOutPortBind `(hCard, DIGI_SIG0, DIGI_SIG0, DIGI_SIG1)`

digiOutPortConfig () digiOutPortConfig ()

Syntaxe :

```
void digiOutPortConfig ( HDIGIC  hCard,
                         u32     conf0,
                         u32     conf1,
                         u32     conf2)
```

Description :

This fonction allows you to configure the source signals and their polarity.
Each `confx` parameter corresponds to a signal and the value of each signal is to choose from two:

- conf0	DIGI_SIG0_USR:	Use the user register value
	DIGI_SIG0_BCTRIGGER:	Use the BC trigger output
- conf1	DIGI_SIG1_USR:	Use the user register value
	DIGI_SIG1_TRIGGERA:	Use the bus A spy trigger output
- conf2	DIGI_SIG2_USR:	Use the user register value
	DIGI_SIG2_TRIGGERB:	Use the bus B spy trigger output

You can choose the polarity of the signals by adding to `confx` the value:

- DIGI_SIG_POLUP	Default configuration.
- DIGI_SIG_POLDOWN	Invert the signal

Default output port configuration:

```
digiOutPortBind   (hCard, DIGI_SIG0, DIGI_SIG1, DIGI_SIG2) ;
digiOutPortConfig (hCard,
              DIGI_SIG0_BCTRIG  | DIGI_POLUP,       // For if Out 0
              DIGI_SIG1_TRIGA   | DIGI_POLUP,       // For if Out 1
              DIGI_SIG2_USR     | DIGI_POLUP) ;     // For if Out 2
```

digiOutPortSet () digiOutPortSet ()

Syntaxe :

```
void digiOutPortSet (HDIGIC hCard, u32 bits)
```

Description :

This function allows you to assign a signal to the output port according to the value of the bits. The correspondence is as follows:

Bit 0	SIG0
Bit1	SIG1
BIT2	SIG2

The physical interface on which the signal are output depends on the configuration made by *digiOutPortBind(),* and the significance of the signal from each interface depends on the configuration made by *digiOutPortConfig()*.

digiInPortBind () digiInPortBind ()

Syntaxe :

```
void digiInPortBind (HDIGIC  hCard,
                     u32     ifsig0,
                     u32     ifsig1,
                     u32     ifsig2)
```

Description :

This function allows you to configure the matrix mixing of the input port, ie to assign a signal to each physical interface.

Each `ifsigX` parameter corresponds to a signal `X` and indicates the assigned interface. For example to assign interface 2 to signal number 0, interface 1 to signal 1 and interface 0 to signal 2:

digiInPortBind `(hCard, DIGI_IF2, DIGI_IF1, DIGI_IF0)`

Remark:You can use the same interface fot two input signals:

digiInPortBind `(hCard, DIGI_IF0, DIGI_IF0, DIGI_IF1)`

digiInPortConfig () digiInPortConfig ()

Syntaxe :

```
void digiInPortConfig (HDIGIC   hCard,
                       u32      conf0,
                       u32      conf1,
                       u32      conf2)
```

Description :

This fonction allows you to configure the source signals and their polarity.
Each `confx` parameter corresponds to a signal and the value of each signal is to choose from two:

- conf0	DIGI_SIG0_USR:	User register value
	DIGI_SIG0_BCSYNC:	Used as start of cycle for BC
- conf1	DIGI_SIG1_USR:	User register value
- conf2	DIGI_SIG2_USR:	User register value

You can choose the polarity of the signals by adding to `confx` the value:

- DIGI_SIG_POLUP	Default configuration.
- DIGI_SIG_POLDOWN	Invert the signal

// Default input port configuration:

```
digiInPortBind (hCard, DIGI_IF0, DIGI_IF1, DIGI_IF2) ;
digiInPortConfig (hCard,
                  DIGI_SIG0_BCSYNC | DIGI_POLUP,      // For Signal In 0
                  DIGI_SIG1_USR    | DIGI_POLUP,      // For Signal In 1
                  DIGI_SIG2_USR    | DIGI_POLUP) ;    // For Signal In 2
```

digiInPortGet () digiInPortGet ()

Syntaxe :

```
u32 digiInPortGet (HDIGIC hCard)
```

Description :

This feature allows you to read the state signal input port. The return value contains the following information

Bit 0	SIG0
Bit1	SIG1
BIT2	SIG2

The physical interface that provides the signal and its meaning depends on the configuration made by *digiInPortBind ()* and *digiInPortConfig ()*.

5.6 IRIG-B reference manual.

irigDriverVersion () irigDriverVersion ()

Syntaxe :

```
u32 irigDriverVersion (void)
```

Description :

The return contain the version and revision number of the PCIG driver in the lower word.

Exemple

```
uVal = irigDriverVersion () ;
printf ("PCIG Driver V%d.%d\n", (uVal >> 8) & 0xFF, uVal & 0xFF);
```

irigLibVersion () irigLibVersion ()

Syntaxe :

```
u32 irigLibVersion (void)
```

Description :

The return contain the version and revision number of the IRIGB library.

Exemple

```
uVal = irigLibVersion () ;
printf ("iriglib V%d.%d\n", uVal >> 16, uVal & 0xFFFF);
```

irigInit () irigInit ()

Syntaxe :

```
HANDLE irigInit (HANDLE hDevice, u32 registerOffset)
```

Description :

The IRIG-B features of the card is managed by an independent library, this library don't open the PCI card. When the card is open by another library, you can initialise IRIG with this function.

hDevice	The return value of *PcigOpenDevice()*.
registerOffset	The offset of IRIG-B registers in card register map.

The return value is zero on failure, otherwise a handle to be used in all other functions irigXx.

Note: This function is normally used by the open function of the card, and will not be used directly by an application.

irigClose () irigClose ()

Syntaxe :

```
        void irigClose (HANDLE hdl)
```

Description :

Function to call before closing the card to free up resources allocated by *irigInit()*.

hdl	The return value of *irigInit()*.

Note: This function is normally used by the close function of the card, and will not be used directly by an application.

irigLastError () irigLastError ()

Syntaxe :

```
          u32 irigLastError (HANDLE hdl);
```

Description :

To retrieve the number of the error that caused a function of the library to return FALSE. Refer to file iriglib.h for the significance of these numbers

irigVersion () irigVersion ()

```
          u32 irigVersion (void);
```

Description :

To retrieve the version number of the IRIG-B receiver. This version has two components

High word	FPGA version
Low word	microcontroller version.

irigDate() irigDate ()

Syntaxe :

```
          u32 irigDate (HANDLE hdl, & LIRIGDATE date);
```

Description :

The current IRIG-B receiver date is copied in paramater `date`..

irigSetDate() irigSetDate ()

Syntaxe :

```
u32 irigSetDate (HANDLE hdl, struct tm * pTime);
```

Description :

Sets the IRIG receiver date with the date provided in the tm structure of the standard C library

If an external IRIG signal is received, the date of this signal is automatically selected. Function setDate is usefull only if the card does not receive IRIG signal

irigSetLocalDate() irigSetLocalDate ()

Syntaxe :

```
u32 irigSetLocalDate (HANDLE hd);
```

Description :

`setLocalDate` is a shortcut to initialize IRIG receiver date with tjhe suysten date.

If an external IRIG signal is received, the date of this signal is automatically selected. Function setDate is usefull only if the card does not receive IRIG signal

irigSetTxDate() setTxDate ()

Syntaxe :

```
u32 irigSetTxDate (HANDLE hdl, struct tm * pTime);
```

Description :

Sets the IRIG transmiter date with the date provided in the tm structure of the standard C library

.

irigSetTxLocalDate() irigSetTxLocalDate ()

Syntaxe :

```
u32 irigSetTxLocalDate (HANDLE hdl);
```

Description :

Sets the IRIG receiver date with the current date provided by the system

irigStatus() irigStatus ()

Syntaxe :

```
u32 irigStatus (HANDLE hdl);
```

Description :

`irigStatus` gives the status indicators of IRIG-B receiver. This is a field of bits which have the values

LIFD_STSI_AREN	Automatic connexion permitted. Represents the state of the last `setEnableConnect().`
LIFD_STSI_MLOCK	IRIG signal recognized, transitional state in the clock disciplining mechanism

.

LIFD_STSI_DTLOCK	Dates received and decoded.
LIFD_STSI_ASSLOCK	Locked loop. The presence of this bit indicates that the internal clock signal and the IRIG-B are in phase with a lag less than the value specified by *setOkRange()* for at least 2 seconds. This bit goes to 0 instantly when the phase exceeds the validity threshold

The stability of LIFD_STSI_ASSLOCK bit depends on the stability of the signal received. A threshold of 0.5 µs is severe, and commercial generators can not meet this criterion

irigSetEnableConnect() irigSetEnableConnect ()

Syntaxe :

```
u32 irigSetEnableConnect (HANDLE hdl, u32 enable);
```

Description :

setEnableConnect allow you to valid or inhibit automatic receiver reconnetion at the back of the IRIG-B signal after interruption.

irigSetOffset() irigSetOffset ()

Syntaxe :

```
u32 irigSetOffset (HANDLE hdl, i32 offset);
```

Description :

setOffset()lets you specify the offset that needs to introduce the IRIG receiver in the enslavement of its clock

`offset`	The offset value in nanosecond. The value should be ±50000 (±50µs).

The value sent is volatile. At the next power up, the default value will be used by default (see `eepromWrite()`).

irigGetOffset() irigGetOffset ()

Syntaxe :

```
u32 irigGetOffset (HANDLE hdl, i32 * pOffset);
```

Description :

`getOffset()` give the offset of the IRIG clock in nanoseconds.

irigSetTrim() irigSetTrim ()

Syntaxe :

```
u32 irigSetTrim (HANDLE hdl, u32 trim);
```

Description :

`setTrim()`	Allow you to specify the voltage sent to the card VCTCXO.
`trim`	The setting value from 0 to 65535 for -5 à +5 ppm. The oscillator frequency is 10MHz.

This setting is useful to adjust very precisely the IRIG-B date generator of the card.

The value sent is volatile. At the next power up, the default value will be used by default (see `eepromWrite()`).

irigGetTrim() irigGetTrim ()

Syntaxe :

```
u32 irigGetTrim (HANDLE hdl, u32 * pTrim);
```

Description :

`getTrim()`Allow you to know the voltage sent to the card VCTCXO..

The set ranges from 0 to 65535 for -5 to 5 ppm, The oscillator frequency is 10MHz.

irigSetMsThr() irigSetMsThr ()

Syntaxe :

```
u32 irigSetMsThr (HANDLE hdl, u32 value);
```

Description :

`setMsThr()` allow you to specify the receiver tolerance on IRIG-B millisecond jitter.

`value` The setting value from 0 à 255 µs. The default value is 60 µs, so you can accept milliseconds from 940 to 1060µs

The value sent is volatile. At the next power up, the default value will be used by default (see `eepromWrite()`).

irigGetMsThr() irigGetMsTh ()

Syntaxe :

```
u32 irigGetMsThr (HANDLE hdl, u32 * pMsThr);
```

Description :

`getMsThr()` Allow you to know the receiver tolerance for IRIG milliseconds.

irigSetOkRange() irigSetOkRange ()

Syntaxe :

```
u32 irigSetOkRange (HANDLE hdl, u32 value);
```

Description :

setOkRange() allow you to set the tolerance of the indicator of locked loop.

`value`	The setting value in nano seconds, from 0 to 32767 ns. The default value is 2000ns, which allows the indicator to be 1 if the phase shift of the internal clock and IRIG signal does not exceed ± 2µs

The value sent is volatile. At the next power up, the default value will be used by default (see `eepromWrite()`).

irigGetOkRange() irigGetOkRange ()

Syntaxe :

```
u32 irigGetOkRange (HANDLE hdl, u32 * pMsThr);
```

Description :

`getOkRange()` give you the the indicator of accuracy threshold of the clock locked loop.

irigEepromWrite() irigEepromWrite ()

Syntaxe :

```
u32 irigEepromWrite (HANDLE hdl);
```

Description :

`irigEepromWrite`() write the receiver current configuration in EEPROM. This configuration become the default settings, and are used at the next power-up

.

The parameter wrote in the EEPROM are:

- Offset : irigSetOffset().
- Oscillatorr setting: irigSetTrim().
- Receiver tolerance : irigSetMsThr().
- Indicator of accuracy threshold: irigSetOkRange().

irigSetYear() irigSetYear ()

Syntaxe :

```
u32 irigSetYear (HANDLE hdl, u32 year);
```

Description :

The IRIG-B signal does not pass the year. For the IRIG-B receiver calculates correctly the day of the year and can provide the year in dates, the application can provide this value with the function `setYear()`.

`year`	A value between 0 and 63, which corresponds to the years 2000 to 2063

irigSetLocalYear() irigSetLocalYear ()

Syntaxe :

```
u32 irigSetLocalYear (HANDLE hdl);
```

Description :

The IRIG-B signal does not pass the year. For the IRIG-B receiver calculates correctly the day of the year and can provide the year in dates, the application can provide the system date yeay with the function `setLocalYear()`.

irigSetTxPps() irigSetTxPps ()

Syntaxe :

```
u32 irigSetTxPps (HANDLE hdl, u32 enableTX);
```

Description :

Allows you to specify which pp sis output: receiver one or tranmiter one..

`value` 0 : receiver PPS,
1 : transmitter PPS.

The value sent is volatile. At the next power up, the default value will be used.

irigGetTxPps() irigGetTxPps ()

Syntaxe :

```
u32 irigGetTxPps (HANDLE hdl, u32 * pPps);
```

Description :

Allow you to know which PPS is output (receiver or transmitter)

Return value 0 : receiver PPS,
1 : transmitter PPS.

irigTmDate() irigTmDate ()

Syntaxe :

```
u32 irigTmDate (S_LIRIGTM    * pTm,
                S_LIRIGDATE  * pDate);
```

Description :

This feature allows you to break a date in native format (S_LIRIGDATE) into a structure that provides access to all components of the date

.

The output structure is:

```
typedef struct
{
       int tm_usec;       /* Microseconds after second [0,999999]     */
       int tm_sec;        /* Seconds after the minute - [0,59]        */
       int tm_min;        /* Minutes after the hour - [0,59]          */
       int tm_hour;       /* Hours since midnight - [0,23]            */
       int tm_mday;       /* Day of the month - [1,31]                */
       int tm_mon;        /* Months of the year - [1,12]              */
       int tm_year;       /* Year, eg 2009                            */
       int tm_yday;       /* Days of year - [1,365/366]               */
} S_LIRIGTM ;
```

irigTmTime()

Syntaxe :

```
          u32 irigTmTime (S_LIRIGTM    * pTm,
                          S_LIRIGDATE  * pDate);
```

Description :

This feature allows you to break a date in native format (S_LIRIGDATE) into a structure that provides access to time components

.

Only the following fields in the structure `S_LIRIGTM` (see *irigTmDate()*) are provided

:

tm_hour

tm_min

tm_sec

tm_usec

6 Choice of implementation, extensions.

The realization of the card and PMC-Digicool software library uses the standard document Digibus GAM-T-101, September 1982 CELAR-ICEVI. However, this standard is not precise on certain points, leaving some freedom of interpretation to the developer. This chapter describes the choices that were made on these points.

6.1 Data length errors.

The standard requires that during the data phase, it runs as many data characters as services character (§ 5.2.2.2).

Therefore if, for a particular exchange, the BC and the RT don't have the same amount of data there will be the following effects:

- The BC is the master of the procedure, it will flow always the amount of data specified by the B.
- If a RT has less data than requested by the BC, it will provide additional data, with unspecified values, and report an error in data length.
- If a RT has more character to transmit that request, it interrupts its transmission as indicated by the V bit-line procedure, and reports an error in data length
- The manager can not detect the data length error.

In the case of a RT not managed by the PMC-Digicool, who transmits less data than is expected by the PMC-Digicool BC, the BC reports an error in length of the message.

6.2 Acknowledge V bit.

The standard § 5.2.4.1 describes how RT manage the transmission and operates the V bit of the acknowledge:

"In general, any equipment, whether RT or BC, using a message must check its integrity at the transmission level."

The design of the PMC-Digicool considers that any RT uses the commands to determine whether it is affected by the exchange. Therefore any RT check all messages, whether addressed or not in the exchange, and stores an error until the end of command phase of the following exchange. If it is addressed explicitly in the following exchange it reports the error by placing the V bit = 0 in its acknowledge.

6.3 CV255 and CLV255 Extension.

A particular feature, that is not included in the standard, was added: management of the RT inhibition by issuing commands called CLV255 and CV255.

This feature is supported for each device with the function `digiEnableEquipment()` that specifies sensitivity to these commands, and inhibition of transmission on each bus.

A RT whose emission is inhibited continues to receive messages, but do not transmit on Data Line nor Procedure Line (so don't acknowledge).

The control of the RT by the BC is using an exchange of two commands such as:

CV (N, FF) CC (0, XX)	command to a subscriber N.
CL (0, FF) CC (0, XX)	broadcast command.

In these commands N is the RT number, and byte XX is as follows:

BUS		B		A	
Bits	7 - 4	3	2	1	0
Enable translmit on data and proc lines	0	1	0	1	0
Inhibit translmit on data and proc lines	0	0	1	0	1
NOP	0	0	0	0	0
Invert the transmit state	0	1	1	1	1

Exemples: :

Inhibit RT 0x18 for bus A and B: :
CV(18,FF) CC(00,05)

Enable RT 0x18 on bus A without change on bus B :
CV(18,FF) CC(00,02)

7 PMC P4 connections

J1 - 32 Bit PCI					J2 - 32 Bit PCI			
Pin #	Signal Name	Signal Name	Pin #		Pin #	Signal Name	Signal Name	Pin #
1	NC	-12V	2		1	+12V	NC	2
3	Ground	INTA#	4		3	NC	TDO	4
5	NC	NC	6		5	TDI	Ground	6
7	BUSMODE1#	+5V	8		7	Ground	NC	8
9	NC	NC	10		9	NC	NC	10
11	Ground	NC	12		11	NC	+3.3V	12
13	CLK	Ground	14		13	RST#	BUSMODE3#	14
15	Ground	GNT#	16		15	3.3V	BUSMODE4#	16
17	REQ#	+5V	18		17	NC	Ground	18
19	V(I/O)	AD[31]	20		19	AD[30]	AD[29]	20
21	AD[28]	AD[27]	22		21	Ground	AD[26]	22
23	AD[25]	Ground	24		23	AD[24]	+3.3V	24
25	Ground	C/BE[3]#	26		25	IDSEL	AD[23]	26
27	AD[22]	AD[21]	28		27	+3.3V	AD[20]	28
29	AD[19]	+5V	30		29	AD[18]	Ground	30
31	V(I/O)	AD[17]	32		31	AD[16]	C/BE[2]#	32
33	FRAME#	Ground	34		33	Ground	NC	34
35	Ground	IRDY#	36		35	TRDY#	+3.3V	36
37	DEVSEL#	+5V	38		37	Ground	STOP#	38
39	Ground	LOCK#	40		39	PERR#	Ground	40
41	NC	NC	42		41	+3.3V	SERR#	42
43	PAR	Ground	44		43	C/BE[1]#	Ground	44
45	V(I/O)	AD[15]	46		45	AD[14]	AD[13]	46
47	AD[12]	AD[11]	48		47	Ground	AD[10]	48
49	AD[09]	+5V	50		49	AD[08]	+3.3V	50
51	Ground	C/BE[0]#	52		51	AD[07]	NC	52
53	AD[06]	AD[05]	54		53	+3.3V	NC	54
55	AD[04]	Ground	56		55	NC	Ground	56
57	V(I/O)	AD[03]	58		57	NC	NC	58
59	AD[02]	AD[01]	60		59	Ground	NC	60
61	AD[00]	+5V	62		61	NC	+3.3V	62
63	Ground	NC	64		63	Ground	NC	64

The card supports both signaling voltages: 3.3V and 5V

J3 not used by DigiCool card

J4 PMC connector : I/O

Pin P14 column corresponds to the passive PMC carrier Tews TPCI-270

Broche J4	**Broche P14**	**Signal**	**Commentaire**	**Broche P14**	**Broche J4**	**Signal**	**Commentaire**
1	C1	**GND**		A1	2	**GND**	
3	C2	PA+	Procedure Bus A +	A2	4	PA-	Procedure Bus A -
5	C3	**GND**		A3	6	**GND**	
7	C4	DA+	Data Bus A +	A4	8	DA-	Data Bus A -
9	C5	**GND**		A5	10	**GND**	
11	C6	PB+	Procedure Bus B +	A6	12	PB-	Procedure Bus B -
13	C7	**GND**		A7	14	**GND**	
15	C8	DB+	Data Bus B +	A8	16	DB-	Data Bus B -
17	C9	**GND**		A9	18	**GND**	
19	C10	IRIG_IN	IRIG-B AM	A10	20	**GND**	
21	C11	IRIG_OUT	IRIG-B AM	A11	22	**GND**	
23	C12	TTLIN_0		A12	24	TTLOUT_0	
25	C13	TTLIN_1		A13	26	TTLOUT_1	
27	C14	TTLIN_2		A14	28	TTLOUT_2	
29	C15	**GND**		A15	30	**GND**	
31	C16	TMS	JTAG	A16	32	TCK	JTAG
33	C17	TDI	JTAG	A17	34	TDO	JTAG
35	C18	12V	Total 1A max	A18	36	12V	
37	C19	**GND**		A19	38	**GND**	
39	C20	F_RXD	FPGA RXD 3.3V	A20	40	F_TXD	FPGA TXD 3.3V
41	C21	**GND**		A21	42	**GND**	
43	C22	LED_1	Sans résistance	A22	44	LED_2	Sans résistance
45	C23	**GND**		A23	46	**GND**	
47	C24	PPS_OUT	Sans résistance	A24	48	**GND**	
49	C25	Test_0	L1P_6	A25	50	Test_1	L8N_CC_LC_6
51	C26	Test_2	L1N_6	A26	52	Test_3	L8P_CC_LC_6
53	C27	Test_4	L2P_6	A27	54	Test_5	L14N_6
55	C28	Test_6	L4P_6	A28	56	Test_7	L15P_6
57	C29	UC_TXD	µC IRIG 5.0V	A29	58	UC_RXD	µC IRIG 5.0V
59	C30	**GND**		A30	60	**GND**	
61	C31	NC		A31	62	NC	
63	C32	NC		A32	64	NC	

8 Bracket connector Honda

Connector on the card : Honda Connectors HDR-E50

Mating solder connector : Honda Connectors HDR-E50 M S G1

Broche	**Signal**	**Commentaire**	**Broche**	**Signal**	**Commentaire**
1	DA-	Data Bus A -	26	PA+	Procedure Bus A +
2	PB+	Procedure Bus –B +	27	DA+	Data Bus A +
3	PA-	Procedure Bus A -	28	DB+	Data Bus B +
4	PB-	Procedure Bus B -	29	**GND**	
5	DB-	Data Bus B -	30	**GND**	
6	IRIG_IN	IRIG-B AM	31	IRIG_OUT	IRIG-B AM
7	TTLIN_0		32	TTLOUT_0	
8	TTLIN_1		33	TTLOUT_1	
9	TTLIN_2	Opt: RS422_in +	34	TTLOUT_2	Opt RS422 out+
10	RS422In **-**	Opt RS422_in -	35	RS422out	Opt RS422 out -
11	**GND**		36	**GND**	
12	TMS	JTAG	37	TCK	JTAG
13	TDI	JTAG	38	TDO	JTAG
14	3.3V	JTAG, 0.3A max	39	12V	0.3A max
15	**GND**		40	**GND**	
16	F_RXD	FPGA RXD RS232	41	F_TXD	FPGA TXD RS232
17	LED_1	R 1.1K	42	LED_2	R 1.1K
18	PPS-OUT	R 1.1K	43	NC	
19	**GND**		44	**GND**	
20	Test_0	L1P_6	45	Test_1	L8N_CC_LC_6
21	Test_2	L1N_6	46	Test_3	L8P_CC_LC_6
22	Test_4	L2P_6	47	Test_5	L14N_6
23	Test_6	L4P_6	48	Test_7	L15P_6
24	UC_TXD	µC RS232 TX	49	UC_RXD	µC RS232 RX
25	**GND**		50	**GND**	

La sortie TTLOUT_2 et l'entrée TTLIN_2 peuvent être indépendamment transformées en RS422.

L'équipement de la mezzanine pour ces options et le suivant ("On" signifie qu'une résistance 0Ω est placée à cet endroit):

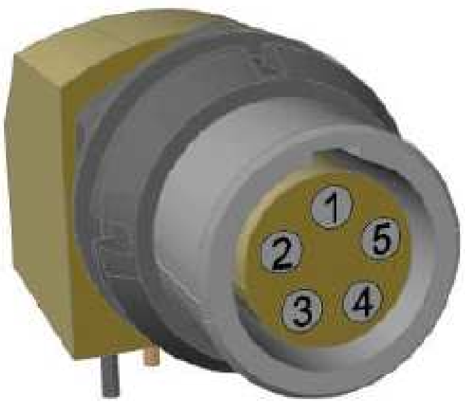
	R9	**R10**	**R14**	**R15**
TTLIN_2	nc	**On**	nc	nc
RS422_IN	**On**	nc	nc	nc
TTLOUT_2	nc	nc	**On**	nc
RS422_OUT	nc	nc	nc	**On**

9 EI_DIGICOOL connectors

Digibus connector:

SubD-9 Female	**Signal**
1	Procedure A **-**
2	Procedure B **-**
3	Procedure A **+**
4	Procedure B **+**
5	
6	Data A **-**
7	Data B **-**
8	Data A **+**
9	Data B **+**

Additional signals, PCB mount LEMO 0B 305:



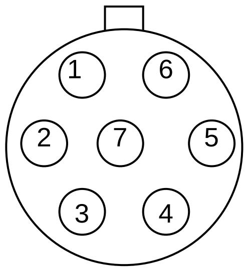
1	IO0 – TTLIN_0
2	GND
3	IO1 – TTLOUT_0
4	IO2 – TTLOUT_1
5	IO3 – TTLOUT_2

10 PCI_BUS _DIGICOOL connectors

Digibus connector:

SubD-9 Female	**Signal**
1	Procedure A **-**
2	Procedure B **-**
3	Procedure A **+**
4	Procedure B **+**
5	
6	Data A **-**
7	Data B **-**
8	Data A **+**
9	Data B **+**

Additional signals, PCB mount LEMO-EMB-1B-07C-CO:



1	IRIG In
2-7	Gnd
3	Irig Out
4	PPS Out
5	User In TTL (TTLIN_0)
6	User Out TTL (TTLOUT_0)

11 Cables and accessories (optional).

11.1 PMC_DIGICOOL bus A connector

HDR-E50 MSG1	SubD-9 Female	Color	Signal
3	1	Blue	Procedure A **-**
	2		
26	3	White	Procedure A **+**
	4		
	5		
1	6	Blue	Data A -
	7		
27	8	White	Data A **+**
	9		

This cable allows using bus A on Sub 9 pin female connector.

Digibus cable shield is connected to connectors shell.

Cable Trompeter TWC-78-2, length: 15 cm.

Commercial reference WF-609.

11.2 PMC_DIGICOOL bus A and B connector (Dual redondant)

HDR-E50 MSG1	SubD-9 Female	Color	Signal
3	1	Blue	Procedure A **-**
4	2	Blue	Procedure B **-**
26	3	White	Procedure A **+**
2	4	White	Procedure B **+**
	5		
1	6	Blue	Data A -
5	7	Blue	Data B -
27	8	White	Data A **+**
28	9	White	Data B **+**

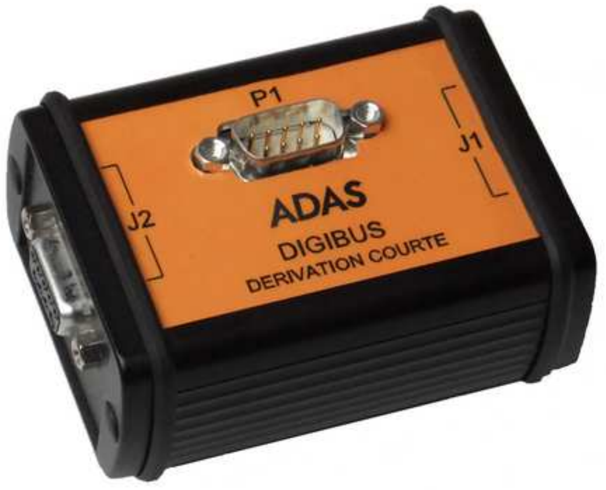
This cable allows using bus A and bus B on Sub 9 pin female connector.

Digibus cable shield is connected to connectors shell.

Cable Trompeter TWC-78-2, length: 15 cm.

Commercial reference WF-611.

11.3 Stub case WF631.



These cases allow to connect a Digicool card to a Digibus bus:

- WF631/DC : Short stub.
- WF631/DL : Long stub (EI_Digicool and PCI_BUS_Digicool only). This case contains transformers and resistances necessary for a long stub according to the GAM-T-101 standard.

The stub connector P1 allows to connect the case to:

- A PMC_DIGICOOL card with WF609 or WF611 cable.
- A EI_Digicool er PCI_BUS_Digicool card with an extension câble (<30cm for short stub and < 3m for long stub as per GAM-T-101 standart).

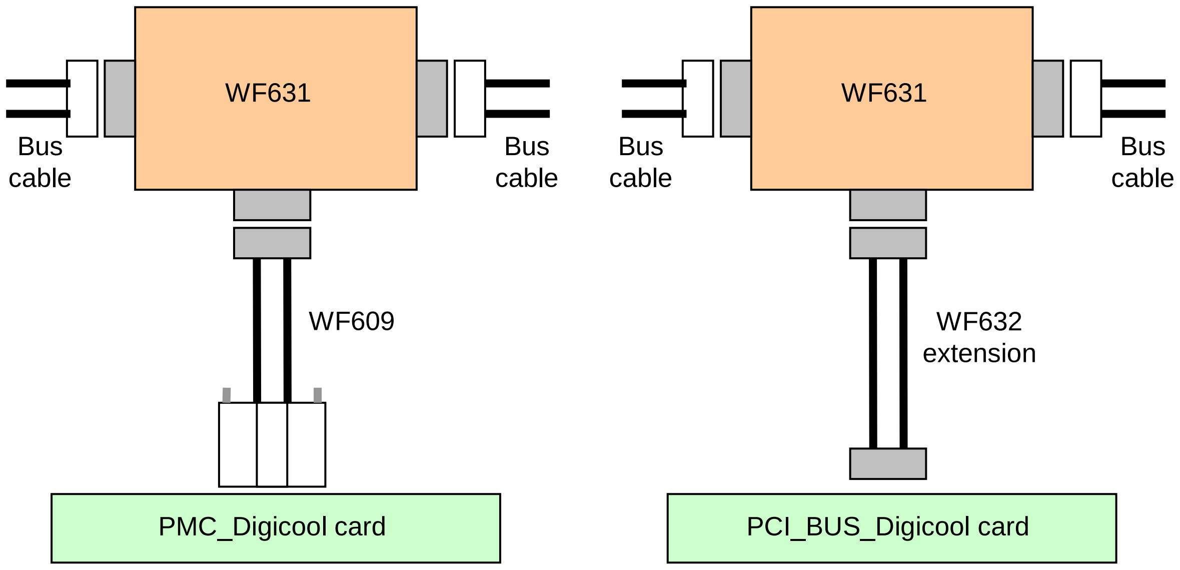
The J1 and J2 bus connectors allow:

- To insert the stub case into the cable of a bus.
- To insert the case instead of bus termination by connecting the cable to one bus connector of the case, and a terminating WF630 in the other bus connector

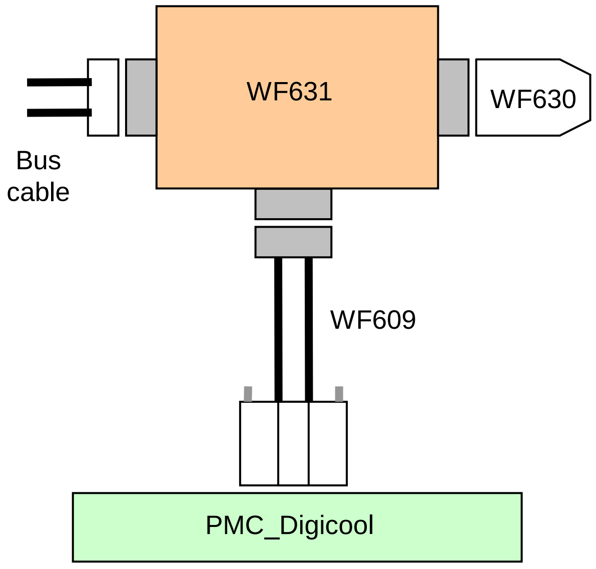
The connectors P1 (plug) and J1/J32 (socket) have the same pinout as the WF611 cable.

The connection between two stub cases can be realized with a WF632 cable (Its length must be specified with order).

Examples of insertion in a Digibus cable (not has its extremity):



Example of insertion in bus extremity, with terminating load:



11.4 Bus terminating load WF630.

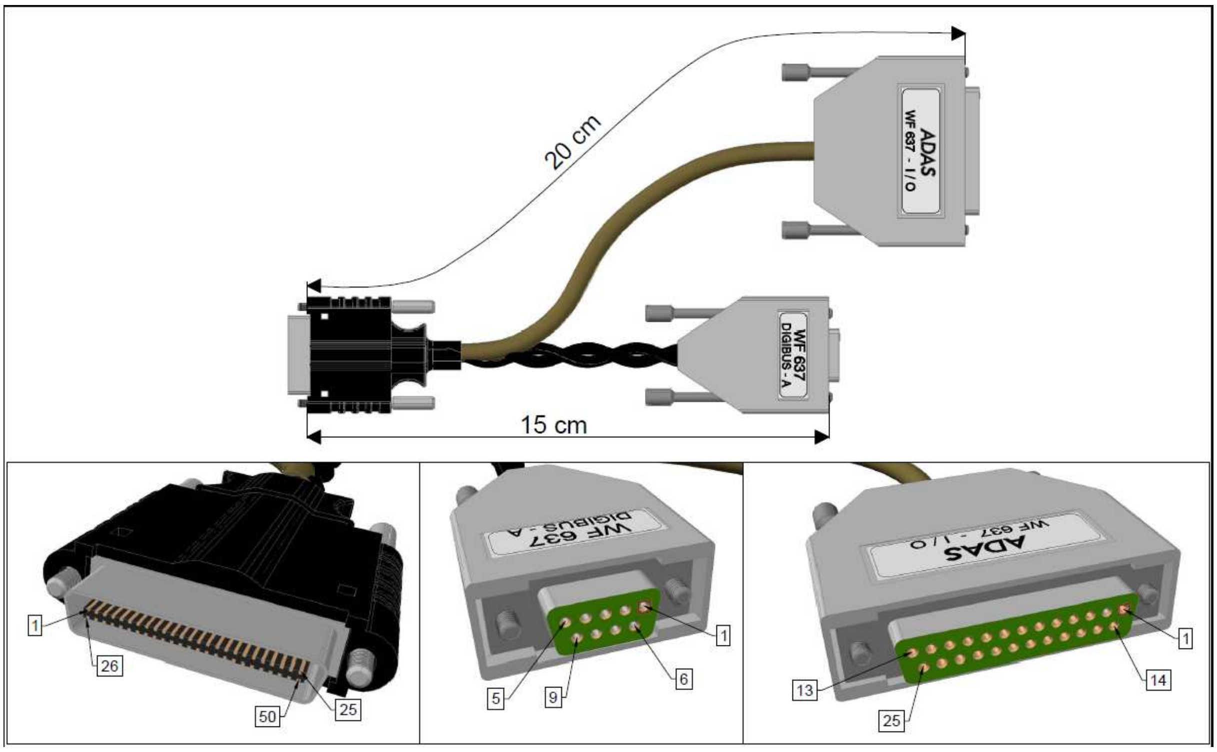
The WF630 module is a terminating load for dual redondant Digibus in compliance with the standard GAM-T-101:

- A 75 Ω resistor load on each of 4 lines.

- A 10 KΩ resistor between lines and cable shield.

It is directly compatible with the stub case WF631

11.5 PMC_DIGICOOL bus A et I/O connector.



Digibus A Connector

HDR-E50 MSG1	Sub-9 Female	Color	Signal
3	1	Blue	Procedure A **-**
26	3	White	Procedure A **+**
1	6	Blue	Data A -
27	8	White	Data A **+**

I/O Connector

HDR-E50 MSG1	Sub-25 Female	Signal
11	7	**GND**
7	20	TTLIN_0
32	21	TTLOUT_0
36	22	**GND**
6	11	IRIG_IN
31	12	IRIG_OUT
8	25	IRIG PPS_OUT
8	13	**GND**

Digibus and I/O cables shields are connected to connectors shell.

Cable Trompeter TWC-78-2, length: 15 cm.

Commercial reference WF-637.