



# **NetworkCentric Core Avionics**



## **RODOS IO and Interrupts API**

**Version:** 07  
**Date:** 25.02.2012

Sergio Montenegro  
University Würzburg  
[sergio.montenegro@uni-wuerzburg.de](mailto:sergio.montenegro@uni-wuerzburg.de)



# RODOS IO and Interrupts API

*Sergio Montenegro*  
*Univerity Würzburg*  
*sergio.montenegro@uni-wuerzburg.de*

## 1. Scope

This document contains recommendations:

1. How to implement IO drivers and interrupt management when porting RODOS.
2. How to encapsulate IO Devices (eg. GPS, StarSensors, Wheels, RadioComs, etc) which are attached directly to the computer (not for network devices). The encapsulation my provide a middleware interface to other applications, but internally it has to treat low level hardware interfaces.

This document is not intended to be used (not necessary) by hardware independent application developers.

The include files which correspond to this specification may be found under

hw\_interrupts.h  
hw\_io.h

## 2. Why is it so difficult to define such standard

Applications have an interface to sensors and actuators, like for example reaction wheels, accelerometer, gyros and so on. These sensors and actuators provide some services which shall be distributed using the RODOS middleware. We can create an standard to distribute the services (see RODOS Middleware), but we can not define standards about data formats, commands, addresses an so on, because there is no limit to the variability of such parameter times variability of hardware. We have then many<sup>4</sup> implementations:

1. (CPUs): We may have many different CPU Architectures (and corresponding compiler tool chains), for example ARM, Cortex, PPC, x86, Leon, AVR, etc.
2. (Micro controller): We have many possible micro controllers. The same CPU Architecture can be used for many different IO structures in many different micro controllers. For example ARM is used by micro controllers of TI, AMD, Actel, Analog Devices, Philips, Samsunf, STM, etc. Each has its own (different) IO and memory Structure.
3. (Development Board): We may have many different development boards for the same micro controller. Normally a micro controller has less pins than required for all IO units inside. We have to configure which IO units will be used and to which pins they will be attached. The development board attach specific converters to some pins, so this restriction has to be considered.



4. (Systems): We can attach different (countless) devices to the different ports of the development board. For example, if we have 4 UARTS, there is no restriction where to attach which device, and even to change it. For example we can attach the GPS to UART-2 or to UART-3, another GPS to SPI-2 and so on.

= Many \* many \* many \* countless implementations! See Figure 1:

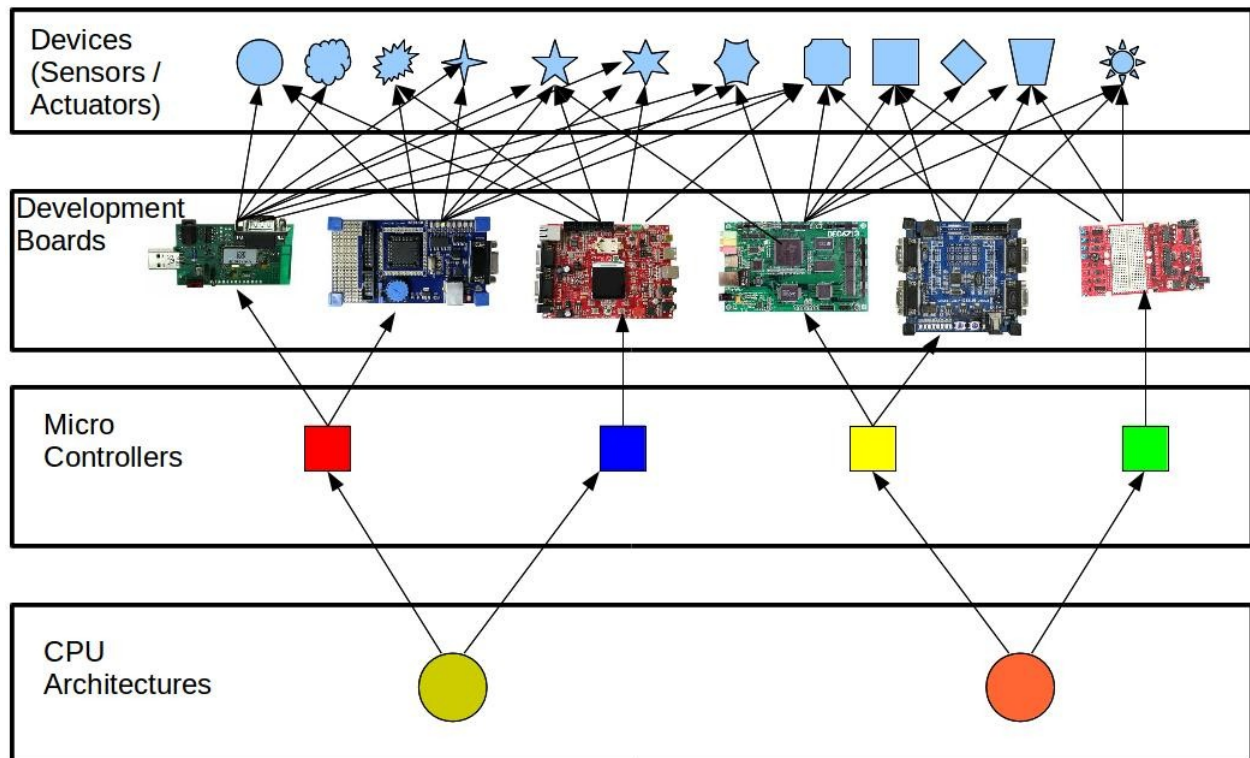


Figure 1: Hardware variability

### 3. RODOS Middleware Interface (Application Interface)

Using RODOS we attempt to compose (complex) software applications by interconnecting simple components totally independent from the underlying hardware. Some (software) components may be pure computations elements, some can be store (mass memory) elements and some may have access to IO devices. For the (software) interconnection network there shall be no difference what is inside of a component or if it represents a IO device attached to the system. All communication between applications have the same form, see figure 2.

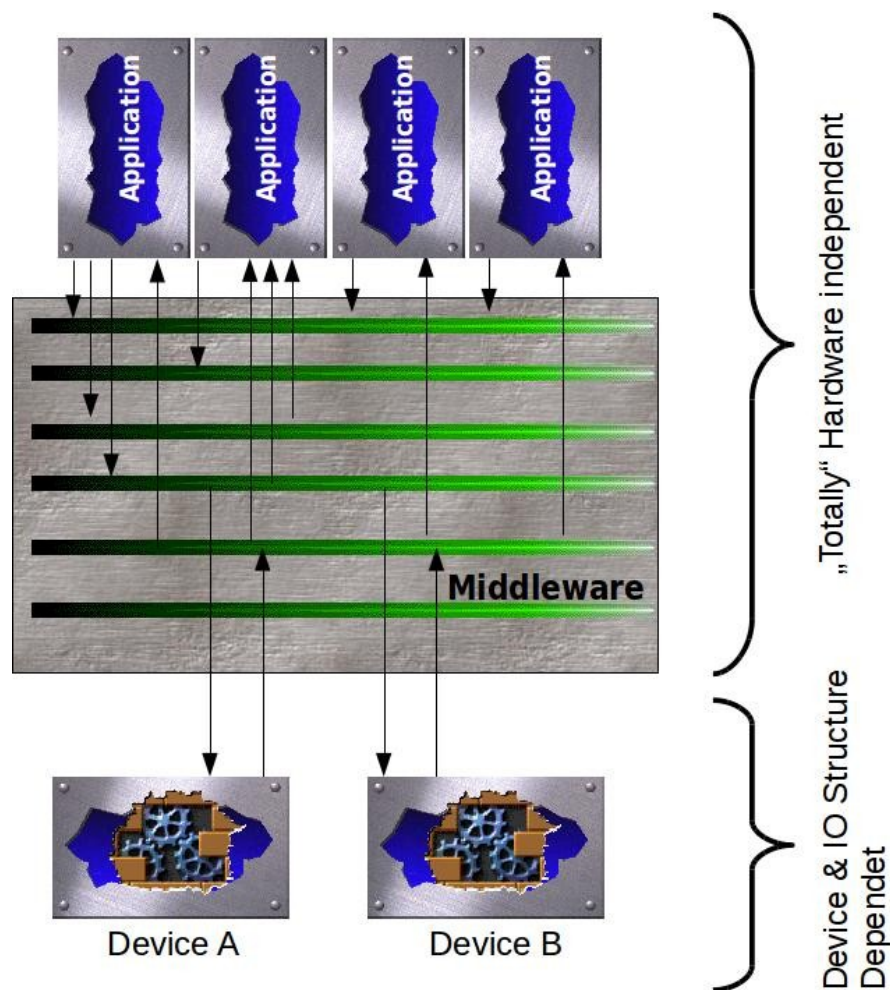


Figure 2: Networking of Applications including IO Devices

## 4. Layers

Applications containing an interface to IO Devices can not be totally hardware independent. We try to separate between hardware independent parts and hardware dependent parts. We can identify three different dependencies when accessing IO Devices (see Figure 3):

1. Device dependent, but independent from the micro controller structure. This part handles messages from/to devices, for example GPS messages, Starsensor messages, Motor controller messages. So it depends on which kind of device we are using, but it is independent of the CPU Architecture and IO Structure of the micro controller. This part may be portable. To keep it portable we have to have an "standard" IO API (Input/Output Application Interface).

2. Given a standard IO API which shall be the same for all UART, I<sup>2</sup>C, SPI, AD, DA, PWM etc, ports we have to implement this API on the IO Structure of each (used) micro controller. This layer is CPU-architecture and device independent, it only depends on the IO Structure of the



micro controller. Once implemented it can be used for many different Devices and applications on the same micro controller.

3. The management of the IO-Structure of micro controllers relays on interrupts. The Interrupt implementation is IO-Structure independent but it is CPU Core architecture dependent. This layer offers an API to the IO Drivers implementation. This API could be used from the over layers from Operating system too, but we would like to recommend to be very carefully using and disabling interrupts. RODOS does not disable interrupts!

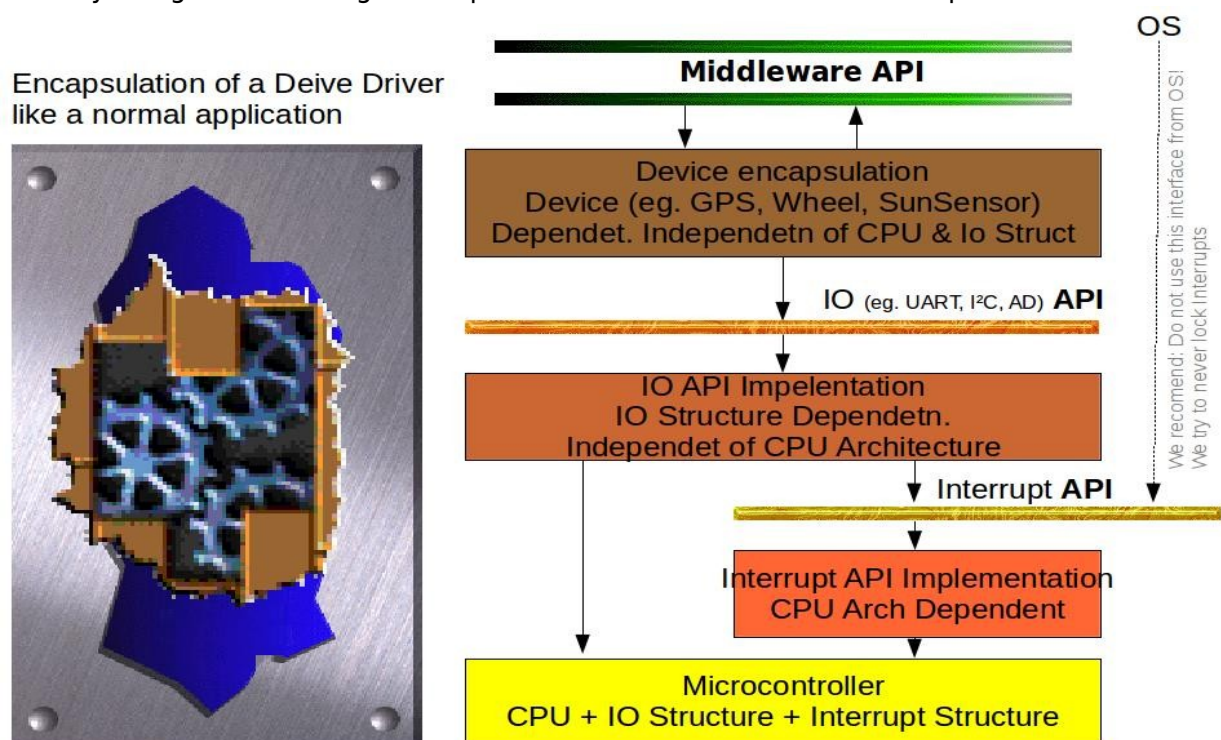



Figure 3: IO Layers

Some interrupts are not directly associated to IO devices, they are more internal, like for example

- Timer Interrupts
- Software Interrupt (Traps)
- Exceptions
- co-processor interrupts
- Mass memory Unit Interrupts
  - page fault
  - Segmentation fault
  - Protection
  - Descriptor fault

and some more.

The Interrupt API makes no difference between interrupts associated to IO Ports and such “internal” interrupts.

	<p style="text-align: center;"><b>RODOS</b> IO Drivers</p>	
--	--	--

It is possible to create virtual devices to these interrupts in order to have a common view from the IO API (handled in other document).

## 5. Interrupt API

### 5.1 Interrupt API constraints

There is no a common method to handle interrupts for all CPU Architectures. Here we try to define an API which can be implemented more or less on all architectures. The Behaviour of some calls may differ on different CPU-Achitectures.

The Interrupt API provides means for installing interrupt handlers and controlling when interrupts are accepted. This functionality is used primarily by device drivers and by very few applications which interacts directly with hardware (we do not recommend this). For IO-Applications we recommend to go trough the IO API and to not not access directly the Interrupt API.

Any code called from the interrupt handler will be executed on an undefined stack (some processors provide an extra Interrupt stack, but most processors continue on the same stack where the CPU was before the interrupt arrived). This code has to be as short (fast) as possible and may not access threads data, or semaphores, etc. It is possible to resume threads (Thread::resume()) from the interrupts server. So we recommend to have a thread which handles complex data associated to interrupts and the interrupt server just resumes the corresponding thread.

The interrupts enable/disable mechanisms may vary very strong from one CPU architecture to another. Some processors have several priorities (eg. 4 or 7 or even 128) for different interrupt sources, some other have only one priority. When an interrupt arrives the hardware disables all interrupts with the same or lower priority (for some processors this means “all interrupts will be disabled”) until the interrupt server routine terminates (Instruction “Return from Interrupt”). While handling an interrupt, another interrupt with higher priority may arrive. The same mechanisms can be accessed from the API: The user may disable or mask some interrupts. For some processors we can just enable/disable all interrupts, for some we can enable/disable all interrupts with a lower priority than the given (parameter) and for some other it is possible to disable directly some interrupt sources. Warning: Some processors are very dangerous (eg. LEON and SPARC) if the interrupts (all) are disabled and a software exception or software interrupt (trap) occurs, then the CPU goes in a “Error” mode and stops working! So please think it two times before you disable interrupts (by the way, the RODOS kernel never disables interrupts). The interrupt API provides functions to enable/disable all interrupts

**HW\_disableInterrupts()** and

**HW\_enableInterrupts()**.

This shall have the same behaviour in all (us known) processors. There are another functions to mask interrupts

**HW\_maskInterrupts**(unsigned int id) and

**HW\_unmaskInterrupts** (unsigned int id)).

These function may have different effect on different CPUs, for most of them the meaning is “disable/enable all interrupts with a lower priority than given ID” for some few the meaning is

“disable/enable only the interrupt index ID” and for some CPUs it will mean “disable/enable all interrupts”.

Most Processors provide Interrupt vectors to register different interrupt servers for different interrupt sources (eg. 128 Vectors), some provide vectors for different interrupt priorities (eg. 7) and some allow only one interrupt server and the software has to determine which interrupts server shall be called. The Api provides a function to register interrupt server for different vectors, but the user has to know how many vectors he may use and which vector is associated to which device. There is NO hardware abstraction at this point. The function **HW\_setInterruptHandler**(unsigned int index, void (\*interruptHandler)() ) registers the given interruptHandler at position index in the interrupt vector table. There is no global definition about the range or meaning of index.

## 5.2 Interrupt API

### Initialisation

At boot time, the booter will

1. disable interrupts
2. activate the Hardware Watchdog to approx 3 seconds (if applicable)
3. clear (set to 0) the interrupt vector table (if applicable)

after boot RODOS and its execution environment will:

1. initialize static variables
2. if linker-option possible: init the interrupt vector with predefined code or pointers
3. call all constructors of static objects
4. call the **hwInit()** function (you may define/modify this function)
5. Call initialisation for following static objects
  - Initiators
  - TimeEvents
  - Threads
6. Start Interrupt timer
7. Start execution of threads & events

You may set interrupt handlers from a constructor of static objects or from **hwInit()**.


We recommend to use a static objects for each interrupt server (associated to the corresponding IO-Driver), Its constructor shall register the interrupt handler. **hwInit()** shall set default interrupt handlers for all not initialized vectors (still containing 0) and then activate interrupts.

**hwInit()** it shall trigger the hardware watchdog for the last time, before RODOS takes control (RODOS will do it periodically).

Warning: Do not modify the vector table after the initialisation is concluded.

Interrupt-API Functions for the initialisation:

```
void HW_setInterruptHandler(
    unsigned char index, void (*interruptHandler)() )
```

	<p style="text-align: center;"><b>RODOS</b> <b>IO Drivers</b></p>	
--	---	--

Index defines the position in the interrupt vector table (if applicable). The range and meaning of index and hardware dependent.

for example:

```
void timerTick() { tickCnt++ } // warning: use return from interrupt __attribute__((interrupt))
HW_setInterruptHandler(8, timerTick);
```

## Enabling / Disabling interrupts

Almost every hardware can enable and disable all interrupts globally. This is done (controlled) by calling:

**void HW\_enableInterrupts()**

**void HW\_disableInterrupts()**

Warning: some CPUs don't like to get a software interrupt or MMU exception while interrupts are disabled. Some CPUs goes (eg. LEON, Sparc) to an ERROR state and stay there. Please be very carefully when disabling interrupts. RODOS never disables interrupts. To have a secure communication between threads and interrupt handlers you may use objects from classes **ComBuffer**, **Fifo**, and **Ringbuffer** which guaranty data consistency even for asynchronous concurrent processes without disabling interrupts or scheduling. Restriction: only one writer and only one reader per object.

Some CPUs allow to disable/enable determined interrupts while allowing all others. You can use

**void HW\_maskInterrupts(unsigned int interruptId)**

**void HW\_unmaskInterrupts(unsigned int interruptId).**

to disable/enable a determined interrupt, but the result may be different for different CPUs. The InterruptId will be the index in the interrupt vector. Some CPUs can not disable only one interrupt, but all interrupts with a priority equal or less then the given ID. In this case the interruptId will be this priority to be disabled/enabled and other CPUs can only enable / disable all interrupts. In the latter case all interrupts will be enabled/disabled.

In case unexpected interrupts arrive, or if there is an interrupt source which did not get an Interrupt handler from the user, the System will count

**unsigned long HW\_getUnhandledInterruptsCounter().**



### 5.3 Interrupt Propagation

It is important to keep the interrupt service routine as short (and fast) and local as possible. Whenever you can, you shall implement the whole management in a single place. But some times the interrupt handling is a matter of applications and not of the lower software. For this cases the RODOS middleware provides means to propagate interrupts to 0, one or many applications. (please be very carefully when doing so).

For this purposes you may create interrupt topics (class **Topic**) and use the method `Topic::publishFromInterrupt(void* data)`.

## 6. IO API and IO Ports

The Input Output Layer is Micro controller architecture dependent but CPU architecture independent. The Input Output layers may use the Interrupt management Layer, but it is not absolutely required. Many simple IO Drivers do not need interrupts. Interrupt management is CPU architecture dependent, but the Interrupt layer provides a “standard” interface which is an abstraction of CPUs so that the IO layer may remain independent from CPU. The IO Layer provides then an Interface to upper layers, which is independent from CPU and to Microcontroller architectures. On the top of this API the system programmer may implement Device management (eg. GPS, Wheels, StarSensors, etc) which is Device dependent but totally board computer independent.

The API to send/receive messages from several kind of buses may be almost the same (write, read, writeRead), but the initialisation/configuration of different kind of buses differs considerably. Therefore even if the classes seem very similar, we have to have different clases for differnet hardware ports. See Figure 4.

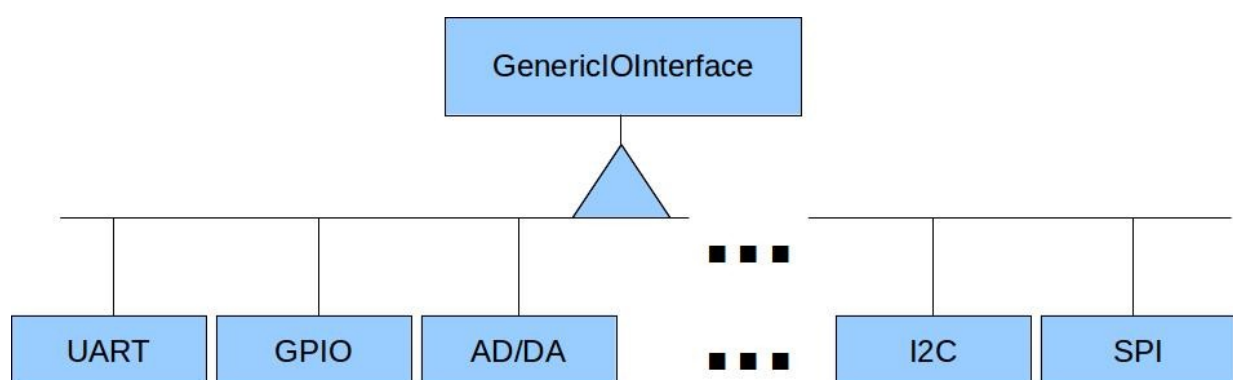


Figure 4: IO interaces

No low level software is able to suspend or resume threads. Each call shall return “intermediately”. Busy waiting may be implemented only if it will take *less than one millisecond* and if it implements a time out (one millisecond).



## 6.1 GenericIOInterface

```
class GenericIOInterface {
protected:
    void* context;          ///< HAL dependent, reserved for low level developers
public:
    GenericIOInterface() { } ///< create all drivers static but do not init there, init in config/reset

    /** Implemtned by each interface ****/
    virtual int config(...) { } ///< Virtual, but do not call using a pointer, each config differs
    virtual void reset() { }

    virtual bool readyToSend() {return true; }
    virtual bool dataReady() {return false;}

    virtual int write(const char* sendBuf, int len) {return 0;}
    virtual int read(char* recBuf, int maxlen) {return 0;}
    virtual int writeRead(const char* sendBuf, int len, char* recBuf, int maxlen) {return 0;}

    /******* may be implemented by the user, called by the corresponding driver *****/
    virtual void upCallSendReady() { } ///< last transmit order is concluded
    virtual void upCallDataReady() { } ///< data has arrived, shall call read(...)
};
```

All IO ports inherit from GenericIOInterface a pointer to a **context**, which may be different for each Micro controller and each IO port. The System programmer shall create his internal structure and store in context just a pointer to it. To allocate memory one can use xmalloc or if possible static data allocation.

The **config()** method will be found in each interface, but all they differ in parameter and execution. It is virtual, just to signalise the subclasses will redefine it. Do not call it as virtual (from a pointer)! We recommend to do **not** configure the devices using the **constructor**, which per default is left empty. We recommend to allocate all drivers statically, no new() and not on the stack. Therefore we do not have any control when each constructor will be executed and in which sequence, therefore it is not safe to initialize hardware using constructors of static objects. Many configs have as parameter the wished speed (baudrate). The driver will try to set the closest possible speed supported by hardware. Config returns a value < 0 if an error occurred, else return 0 or a positive value (implementation dependent).

**readyToSend()** and **dataReady()**, per default return true, but each implementation shall return true if following calls to read or write would succeed.

**write()**, **read()** and **wirteRead()** transfer several bytes as a unit. Some implementations may do the transfer writing to FIFOs, other using a DMA and other by coping the data to an internal buffer and then send byte for byte triggered by interrupts. We can not fix here how they shall be implemented, just: take the most simple possible implementation. While a transfer is running (device busy), the corresponding function **readyToSend()** or **dataReady()** shall return false. When a transfer is concluded, the driver shall call the corresponding **upCallSendReady()** or **upCallDataReady()** function and **readyToSend()** or **dataReady()** may return true again.

**wirteRead()** is an special case for bidirectional buses which are controlled by a single master, like for example UART RS-485, I2C, OneWire (SPI?). For such buses the software has to signalize the hardware to toggle from write to read intermediatly after the last transited bit. After this data will arrive (from external device) and a buffer has to be ready to store this data. This is the second part of the parameter of **wirteRead()**.



write(), read() and writeRead() return in normal case the number of transferred bytes (writeRead() returns the sum of send plus received bytes). If the port is not ready and no transfer can be done they return 0. In error case they return negative values. The meaning of negative error numbers is not defined here and is implementation dependent,

If the user does not implements **upCallSendReady()** or **upCallDataReady()**, the basis class implements default no-operation methods.

## 6.2 Discrete IO Ports

Discrete IO ports are very simple interfaces, for example set or read a pin, or latch the value from an Analogue/Digital converter. Each IO Interface even if it is a single pin, or a group of pins, has to be managed from an object (instance) of the corresponding class.

### GPIO (General Purpose IO) Pins

Most of the cases we have an relationship one pin ↔ one object to access these pins. The meaning of a pin depends on the attached external devices. But some times we have to access several pins together, for example to control multiplexers (typically analogue muxes for the A/D converter, or Chip Select for SPI). Using different objects to control 2 pins, we can not go for example in a single step from “10” to “01”. therefore we shall be able to manage several pins together.

```
class GPIO : public GenericIOInterface { ///< one object per pin (group)
public:
    GPIO();
    int config(int pinIndex = 0, bool isOutput = true, int numOfPins = 1);
    void reset();

    bool readyToSend() { return true; }
    bool dataReady()   { return true; }

    int write(const char* sendBuf, int len) { return 0; }
    int read(char* recBuf, int maxlen)     { return 0; }
    int writeRead(const char* sendBuf, int len, char* recBuf, int maxlen) { return 0; }

    /** Extra only for GPIO */
    void setPins(unsigned int val); ///< the least significant bits will be used
    unsigned int readPins();        ///< the least significant bits will be set
};
```

For **config()** we numerate the pins from 0 to n, without making a difference where each internal register ends. The number of pins is used to configure group of pins, which have to be numerical adjacent. isOutput set to true means output pin which may be set, set to false means input pin which may be read.



## PWM (Pulse Width Modulation)

```
class PWM : public GenericIOInterface { ///one object per pwm (+ one GPIO to set direction)
public:
    PWM();
    int config(int hwPwmGeneratorIndex = 0, int frequency = 50, int numOfSteps = 256);
    void reset();

    bool readyToSend() { return true; }
    bool dataReady() { return true; }

    int write(const char* sendBuf, int len) { return 0; }
    int read(char* recBuf, int maxlen) { return 0; }
    int writeRead(const char* sendBuf, int len, char* recBuf, int maxlen) { return 0; }

    /** Extra only for PWM */
    void setDutyCycle(unsigned int c); ///0 .. +numOfSteps
};
```

For **config()** we have provide an index to the hardware PW generator, which determines which pin shall be used.

## Digital to Analogue and Analogue to Digital converters

```
class DigitalToAnalog : public GenericIOInterface { //one object per analong pin output
public:
    DigitalToAnalog();
    int config(int daIndex = 0);
    void reset();

    bool readyToSend();
    bool dataReady() { return true; }

    int write(const char* sendBuf, int len) { return 0; }
    int read(char* recBuf, int maxlen) { return 0; }
    int writeRead(const char* sendBuf, int len, char* recBuf, int maxlen) { return 0; }

    /** Extra only for DA */
    int getNumOfSteps(); ///= 2^numOfBits, hardware dependent
    void setAnalogVal(int analogVal); ///-numOfSteps .. +numOfSteps
};

class AnalogToDigital : public GenericIOInterface { ///one per converter, (+ GPIO-group multiplexer)
public:
    AnalogToDigital();
    int config(int adIndex = 0);
    void setAssociatedGPIOMultilexer(GPIO* muxControl); ///in case we have an external analog mux
    void reset();

    bool readyToSend() { return true; }
    bool dataReady(); ///depending on analog multiplexer

    int write(const char* sendBuf, int len) { return 0; }
    int read(char* recBuf, int maxlen) { return 0; }
    int writeRead(const char* sendBuf, int len, char* recBuf, int maxlen) { return 0; }

    /** Extra only for AD */
    void selectChannel(unsigned int chanIndex); ///eventually using the external multiplex and GPIO
    int readAnalogValue();
};
```

Very often we have only one analogue - Digital converter and an attached analogue multiplexer to select one out of several channels. Some micro controller have the multiplexer internally and some externally attached to GPIO signals. If we have an external Multiplexer, then we have to assign a GPIO Object to control the multiplexer: **setAssociatedGPIOMultilexer()**.



The **dataReady()** Method can be used to signalize if the AD converter is ready. It depends on external multiplexers too.

## 6.3 Message and Bus Oriented IO Ports

Each IO Interface has its own **config()** structure, but besides this, the access is the same for all of them. This gilts for UARTS, I2C, SPI, Ethernet, CAN, OnWire and if applicable SpaceWire.

```
class UART : public GenericIOInterface {
public:
    UART();
    int config(int uartIndex = 0, int baurate = 115200, int parityType = 0, int charLen = 8);
    void reset();

    bool readyToSend(); ///< at least one character
    bool dataReady();   ///< at least one character

    int write(const char* sendBuf, int len); ///< when returns is hw dependent
    int read(char* recBuf, int maxlen);      ///< when returns is hw dependent
    int writeRead(const char* sendBuf, int len, char* recBuf, int maxlen);

    /** Extra only for UARTS */
    int putcharNoWait(char c); ///< test with readToToSend(), returns 1, 0 or <0 in error case
    int getcharNoWait(char &c); ///< test with dataReady(), returns 1, 0 or <0 in error case
};

class I2C : public GenericIOInterface {
public:
    I2C();
    int config(int i2cIndex = 0, bool isMaster = true, int myAdrIfSlave = 0, int speed = 400000);
    void reset();

    bool readyToSend();
    bool dataReady();

    int write(const char chipAdr, const char* sendBuf, int len);
    int read(char* recBuf, int maxlen); ///< usefull only if we are slave
    int writeRead(char* chipAdr, const char* sendBuf, int len, char* recBuf, int maxlen);
};

class OneWire : public GenericIOInterface {
public:
    OneWire();
    int config(int onewIndex = 0, int speed = 115200, bool isMaster = true, int myAdrIfSlave = 0);
    void reset();

    bool readyToSend();
    bool dataReady();

    int write(const char* sendBuf, int len);
    int read(char* recBuf, int maxlen); ///< usefull only if we are slave
    int writeRead(const char* sendBuf, int len, char* recBuf, int maxlen);
};
```





```

class SPI : public GenericIOInterface {
public:
    SPI();
    int config(int spiIndex = 0, bool isMaster = true, int myAdrIfSlave = 0, int charLen = 16, int baudRate =
1000000);
    void setAssociatedGPIOSelect(GPIO* slaveSelect); ///< To select slaves
    void selectSlave(int slaveIndex = 0);          ///< using the associated GPIO
    void reset();

    bool readyToSend();
    bool dataReady();

    int write(const char* sendBuf, int len);
    int read(char* recBuf, int maxlen);          ///< usefull only if we are slave
    int writeRead(const char* sendBuf, int len, char* recBuf, int maxlen);
};

class Ethernet : public GenericIOInterface {
public:
    Ethernet();
    int config(int ethIndex = 0);
    void reset();

    bool readyToSend();
    bool dataReady();

    int write(const char* sendBuf, int len);
    int read(char* recBuf, int maxlen);
};

class CAN : public GenericIOInterface {
public:
    CAN();
    int config(int canIndex = 0, long inputObjectIdMask = 0xffffffff, int speed = 1000000);
    void reset();

    bool readyToSend();
    bool dataReady();

    int write(unsigned long objectId, const char* sendBuf, int len);
    int read(unsigned long* objectId, char* recBuf, int maxlen);
};

```