RODOS HAL

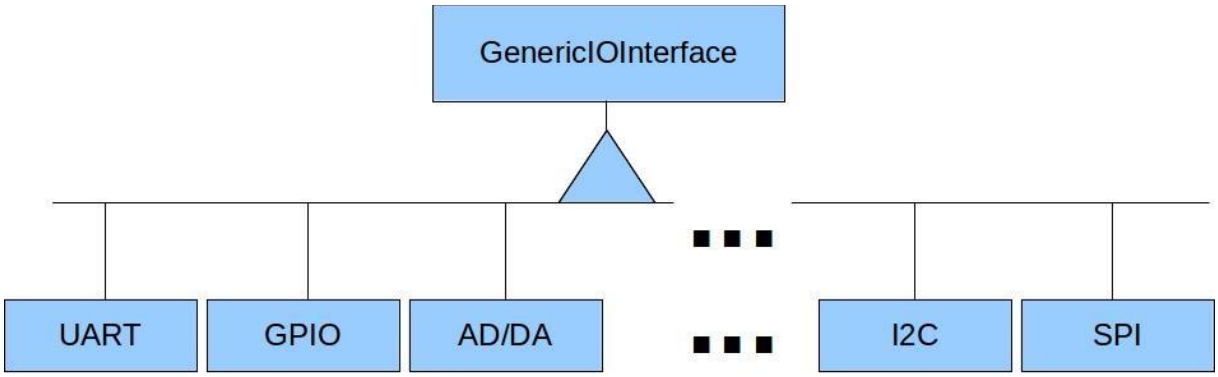
Hardware Abstraction Layer

Version: 1.0
Document Id: Vorlage-1
Date: 16.09.2013
Author: Johannes Freitag

The RODOS HAL (Hardware Abstraction Layer) was developed to provide standardized access to buses like I²C and other functionalities like PWM provided by several platforms. The user can call the api-functions and does not have to think about the underlying implementation for the specific hardware.

1. API definition

The API to send/receive messages from several kinds of buses is almost the same (write, read, writeRead), but the initialization/configuration of different kind of buses differs considerably. Therefore, even if the classes seem very similar, there are different classes for different hardware ports. These classes all inherit from one generic IO interface that defines the basic methods for accessing the buses or hardware functionalities.



1.1. Generic IO Interface

```
class GenericIOInterface {
private:
    IOEventReceiver* ioEventReceiver;
protected:
    void* context;
public:
    GenericIOInterface() {ioEventReceiver=0; }

    /*** Implemented by each interface ****/
    virtual int init(...)   {return 0;}
    virtual void reset() { }
    virtual int config(...) {return 0;}

    virtual int status(...) {return 0;}
    virtual bool readyToSend()  {return false; }
    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;}

    virtual void upCallSendReady() {if(ioEventReceiver) ioEventReceiver->SendFinished();}
    virtual void upCallDataReady() {if(ioEventReceiver) ioEventReceiver->DataReady();}
    void setIoEventReceiver(IOEventReceiver* receiver) {ioEventReceiver=receiver;}
};
```

The basic methods for accessing a bus can be divided into four groups: Initialization, current status, write/read and events after an action has finished.

For initialization there are three methods. The **init**() and **config**() method will be found in each interface, but they all differ in parameter and execution. They are defines as virtual, just to signalize 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 but to use the init or config function. Many configs have as parameter the desired 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). The **reset**() method is used to reset the state of the bus.

Additionally, three methods can be used to retrieve the current status. The **status**() method returns the status for a given status type dependent of the bus implementation. The **readyToSend**() and **dataReady**(), per default return false, but each implementation shall return true if following calls to read or write would succeed.

The methods **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 by byte triggered by interrupts. 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. The method **writeRead**() is a 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 intermediately after the last transited bit. After this, data will arrive (from external device) and a buffer has to be ready to store the data. This is the second part of the parameter of writeRead().

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.

The method **setIoEventReceiver**() is used to define an event receiver that will be called when an action has completed. In order to raise an event, the methods **upCallSendReady**() and **upCallDataReady**() have to be used.

All IO interfaces contain a pointer in their platform independent header file to a platform dependent class (e.g HAL_UART contains HW_HAL_UART* context). This class is only forward declared in the header file. The system programmer defines this class in his platform dependent implementation and puts hist data structures and methods there. The memory for this class should be allocated with xmalloc. If a proper initialization and constructor call is required the placement new operator can be used to construct an object on previously xmalloced memory. The context pointer in Genericiointerface is deprecated and shoudn't be used anymore.

1.2. Specific interfaces API

All the IO interfaces that inherit from GenericIOInterface are located in RODOSROOT/api/hal. In these interfaces only the needed functions are implemented and sometimes additional methods have been added. Aside from the abstract interface for the specific port there is a generic class defined. This class is used as the type of the context and has no relevance for the end-user. This class is only relevant for the system developer to define this class for the specific needs of the corresponding hardware. Besides that, there are indexes defined to use multiple interface of the same type, for instance two I²C interfaces, on one controller.

1.2.1. GPIO (General Purpose IO) Pins

```
// Generic class for hardware-specific context
class HW_HAL_GPIO;

// GPIO HAL
class HAL_GPIO : public GenericIOInterface {
private:
    HW_HAL_GPIO *context;
public:
    HAL_GPIO(GPIO_PIN pinIdx = GPIO_000);
    int32_t init(bool isOutput = false, uint32_t numOfPins = 1, uint32_t initVal = 0x00);
    void reset();
    int32_t config(GPIO_CFG_TYPE type, uint32_t paramVal);

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

    void setPins(uint32_t);
    uint32_t readPins();
    void interruptEnable(bool enable = true,
        GPIO_IRQ_SENSITIVITY mode = GPIO_IRQ_SENS_BOTH, void (*isr)() = NULL);
```

```
};
```

The GPIO HAL is designed to access one or more pins simultaneously. But be careful when defining pin-groups using "numOfPins" in init(). The use of all functions is limited to port boundaries! When port boundaries are exceeded only pins up to the most significant pin of the port will be set/read. Via the interruptEnable function it is possible to react on changes to the pin. A proper service routine can be defined via this function. Instead of the read and write functions defined in GenericIOInterface the GPIO HAL uses the specific functions setPins() and readPins().

1.2.2. PWM (Pulse Width Modulation)

```
// Generic class for hardware-specific context
class HW_HAL_PWM;

// PWM HAL
class HAL_PWM : public GenericIOInterface {
    HW_HAL_PWM* context;
public:
    HAL_PWM(PWM_IDX idx);
    int init(int frequency = 1000, int resolution = 8);
    int config(PWM_PARAMETER_TYPE type, int paramVal = 0);
    void reset();

    /*** only for PWM **/
    int write(unsigned int pulseWidth);
};
```

The PWM HAL can be used to generate a PWM signal on a pin defined with the PWM_IDX. This HAL does not use the read/write functions but has a specific write() function.

1.2.3. I²C

```
// Generic class for hardware-specific context
class HW_HAL_I2C;

// I²C HAL
class HAL_I2C : public GenericIOInterface {
    HW_HAL_I2C *context;
public:
    HAL_I2C(I2C_IDX idx);

    int32_t init(uint32_t speed = 400000);
    void reset();

    bool readyToSend();
    bool dataReady();

    int32_t write(const uint8_t addr, const uint8_t *txBuf, uint32_t txBufSize);
    int32_t read(const uint8_t addr, uint8_t *rxBuf, uint32_t rxBufSize);
    //
    int32_t writeRead(const uint8_t addr, const uint8_t *txBuf,
        uint32_t txBufSize, uint8_t *rxBuf, uint32_t rxBufSize);
};
```

The I²C is developed to send and receive data vi the I²C bus. To send and receive data without a STOP condition between the actions the writeRead() function can be used.

1.2.4. UART

```
// Generic class for hardware-specific context
class HW_HAL_UART;

// UART HAL
class HAL_UART : public GenericIOInterface {
private:
    HW_HAL_UART* context;
public:
    HAL_UART(UART_IDX uartIdx);

    /* Initialization of uart interface: mode 8N1, no HW flow control*/
    int init(unsigned int baudrate = 115200);
    void reset();
    int config(UART_PARAMETER_TYPE type, int paramVal);
    int status(UART_STATUS_TYPE type);

    bool readyToSend();
    bool dataReady();

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

    int getcharNoWait();             // returns character on success else -1
    int putcharNoWait(char c);       // returns c on success else -1
};
```

The UART HAL has two specific functions for sending and receiving a single character via the UART bus.

The config() function accepts the folowing parameters:

UART_PARAMETER_BAUDRATE,
UART_PARAMETER_HW_FLOW_CONTROL,
UART_ENABLE_DMA

When a specific feature is not supported by the hardware -1 is returned.

1.2.5. CAN

```
// Generic class for hardware-specific context
class HW_HAL_CAN;

// CAN HAL
class HAL_CAN : public GenericIOInterface {
private:
    HW_HAL_CAN* context;
public:
    HAL_CAN(CAN_IDX canIdx);

    int init(unsigned int baudrate);
    void reset();
    int config(CAN_PARAMETER_TYPE type, int paramVal);
    int status(CAN_STATUS_TYPE type);

    bool readyToSend();
    bool dataReady();

    /*** only for CAN **/
    // Add a filter for incoming Messages. Only Message which pass a filter are received.
    // Filters may be implemented in hardware an can be limited.
    bool addIncomingFilter(uint32_t ID, uint32_t IDMask=0, bool extID = true,
        bool rtr=false);
    int write(const char* sendBuf, int len,uint32_t canID,  bool extID = true,
        bool rtr=false);
    int read(char* recBuf, uint32_t* canID=0, bool* isExtID = 0,bool* rtr=0);
};
```

The CAN HAL does not use the standard read/write functions but has own functions for reading and writing data to the CAN bus. Every call to read/write receives/sends exactly one CAN frame. So there can be be maximal 8 bytes received/transmitted in one call. Besides that, there is a addIncomingFilter() function to specify which CAN Frames should be received. Frames not matching a filter may be already dropped by the controller hardware. It is possible to create multiple instances of HAL_CAN for the same CAN controller to distribute CAN messages to different applications/threads. If the identifier of an incoming frame matches the filter of two or more HAL_CAN objects the result is undefined.

2. Usage of a HAL-interface

The usage of the HAL is very simple as shown in the following minimal example. This will work fine if the I²C port is used in one thread only:

```
#include "rodos.h"

HAL_I2C i2cBus(I2C_IDX1); //static instance of HAL

class SensorReader: public Thread {
   void run() {
      i2cBus.init(); //one time initialisation of this bus

      uint8_t sensorValue[2];
      TIME_LOOP(0,1*SECONDS)
      {
         i2cBus.read(0x30, sensorValue, 2); //several times reading the bus
      }
   }
} sensorReader;
```

There are a few rules to follow to get everything working properly:

- It is recommended to allocate all drivers statically (as shown in the example above), no new and not on the stack.
- If several threads are using the same bus, e.g. I²C port 1, only one instance for all threads should be created. All the threads should use the instance, but protect their access with semaphores. (exception: HAL_CAN, there you can create multiple inteances for the same CAN controller and it is internally protected with semaphores) To initiate a bus that is used by several threads is important to use an **Initiator** as shown in the example below, otherwise it cannot be guaranteed that the thread, that runs the bus initialization, runs first. An initialization in every thread will cause even more problems.
- The config() method (when used) has to be executed after the init() method.
- Implement everything as simple as possible.

Following, there is another example showing the use of an Initiator. This Initiator will be executed before the threads start running.

```
#include "rodos.h"

HAL_I2C i2cBus(I2C_IDX1); //static variable

static class sensorInitiator : public Initiator {
	void init() {	//runs before the threads start running
		i2cBus.init(); //one time initialisation of this bus
	}
} sensorInitiator;

class AdxlReader1: public Thread {
	void run() {
		uint8_t adxlValue[2];
		TIME_LOOP(0,1*SECONDS)
		{
			i2cBus.read(0x30, adxlValue, 2); //several times reading the bus
		}
	}
} adxlReader1;

class GyroReader2: public Thread {
	void run() {
		uint8_t gyroValue[2];
		TIME_LOOP(0,2*SECONDS)
		{
			i2cBus.read(0x32, gyroValue, 2); //several times reading the bus
		}
	}
} gyroReader2;
```

3. Implementation of a HAL-interface for a new hardware platform

To create a HAL for a new hardware platform it is necessary to implement all the functions defined in the interface in api/hal. This implementation has to be placed in the source folder for the corresponding port in a separate hal folder, e.g. src/bare-metal/stm32f4/hal/. For saving the platform specific context of each I/O interface, the generic class defined in the api (e.g. HW_HAL_I2C) has to be implemented. In this class all hardware specific data and functions shall be defined.

The following shows an example of such a hardware specific context class implementation with attributes for the I/O pins and specific functions:

```
class HW_HAL_I2C {
public:
   HW_HAL_I2C(I2C_IDX i2cIdx){I2C_idx = i2cIdx;}
   HW_HAL_I2C(){};

   I2C_IDX I2C_idx;
   uint16_t I2C_SCL_PIN;
   uint16_t I2C_SDA_PIN;
   ...

   int32_t sendAddr(const uint8_t addr, uint8_t rwFlag){
          //send the adress implementation };
   int32_t start(){ //send start signal implementation };
   int32_t stop(){ //send the adress implementation };
};
```

A static array of type HW_HAL_I2C has to be created to allocate memory for the contexts for every I²C port:

```
HW_HAL_I2C I2C_contextArray[I2C_IDX_MAX-1];
```

If the HW_xxx class contains large buffers it may be a better idea to allocate the memory for it with xmalloc/placement new in the constructor of the corresponding HAL_xxx object.

If the definition of the HW_HAL_I2C is done, the API functions can be implemented. The next example is the constructor implementation. In this constructor the context shall be initialized. No hardware initialization should be done here because we do not have any control when each constructor will be executed and in which sequence.

```
HAL_I2C::HAL_I2C(I2C_IDX idx) {
   // placement new to avoid dynamic memory allocation
   context = new (&I2C_contextArray[idx - 1]) HW_HAL_I2C(idx);

   switch (idx) {
   case I2C_IDX1:
      context->I2Cx = I2C1;
      break;
   case I2C_IDX2:
   ...
   }
}
```

In the init() method the hardware should be initialized with the given parameters e.g. speed:

```
int32_t HAL_I2C::init(uint32_t speed) {
   context->I2C_SPEED = speed;

   /* enable peripheral clock for I2C module */
   RCC_APB1PeriphClockCmd(context->I2C_CLK, ENABLE);
   ...
}
```

To implement the write() and read() functions it is recommended to split the needed code in smaller functions in the context class and call them. For example:

```
int32_t HAL_I2C::write(uint8_t addr, uint8_t* txBuf, uint32_t txBufSize) {
   context->start();
   context->sendAddr(addr, WRITE);
   context->write(txBuf,txBufSize);
   context->stop();
   return SUCCESS;
}
```

For more examples have a look at the stm32f4 implementation. There are most of the HAL interfaces implemented. To ensure that the code is maintainable please keep it as simple as possible.