Claudio Francese

Flexible Arduino-Based board - Base Firmware R1.0

T.R. 24/2017 13/12/2017

I.N.RI.M. TECHNICAL REPORT

Contents

Abstract

This report describes the design of the firmware for an Arduino board which is intended to drive some generic ICs (e.g. DDS generators, A/D or D/A converters , etc) which support experiments.

The idea behind the design of the firmware it the possibility to support a generic board with a set of base functionalities which can be later extended according to the user needs. This approach allows the developer of the application firmware to focus only of the new parts of the firmware, while the common part remains the same.

This project is part of a wider study on the development of flexible instrumentation and distributed control systems.

Arduino Nano board overview and interconnections capabilities

The Arduino Nano board is based on an Atmel ATmega328 microcontroller.

This board has been chosen because of its small footprint (18 x 45 mm, 7 g weight), low power consumption (19 mA) and good versatility which make it a good choice for low-medium complexity applications.

The main specifications of the board are

- 16 MHz clock speed
- 6 PWM outputs
- 22 General purpose I/O pins
- 8 Analog I/O pins
- SPI bus connection
- I^{2}C bus connection
- 32 kB Flash memory for firmware code
- 2 kB RAM
- 1 kB EEPROM for the storage of user parameters

FIGURE 1 - ARDUINO NANO

It should be noted that some signals / peripherals available in Arduino are mutually exclusive. In addition, those resources are anyway a subset of the ATmega328 because of the board reduced pin count and the constraints given by the ATmega328 hardware. For a detailed description and more specifications, please see (1) (2).

The last constrain to the allocation of the Arduino pins is given by the application specific connections between the Arduino board and the Application Board. Figure 2 shows the pin allocation for an AD9959-based DDS board. While the orange-marked terminals assignment is restricted because given by the microcontroller, the green-marked terminals can be used for the application – in this case the connection to the DDS chip leaving only two GPI/O auxiliary pins and two analogic input terminals.

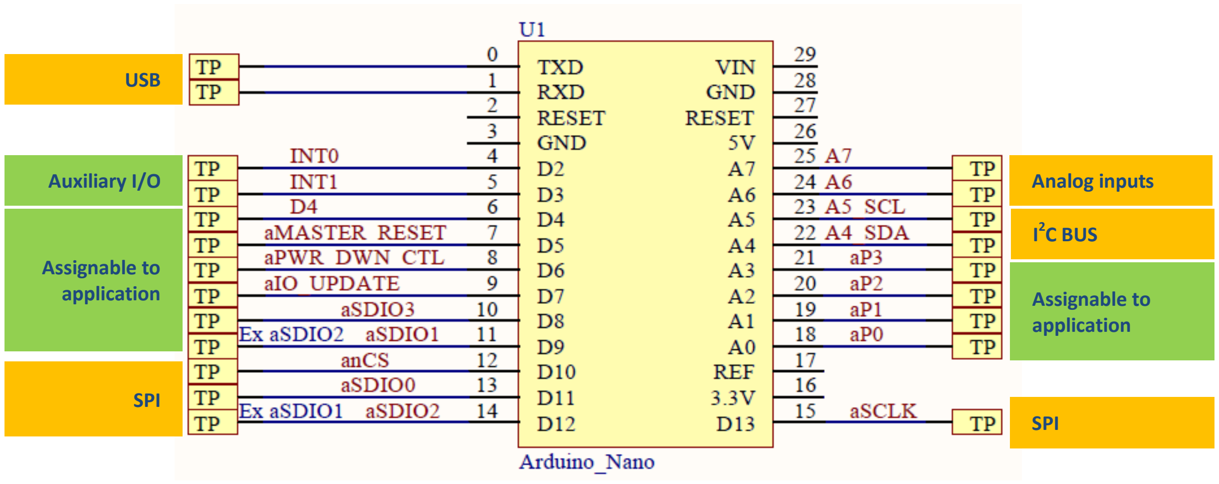


FIGURE 2 - ARDUINO BOARD PIN ALLOCATION

Design of the Firmware

Some considerations should be made before starting the design of the firmware.

1. Trade-off between
   - complexity, flexibility and reliability of the code
   - performance and development time
2. Typical use of the board
   - frequent reconfigurations of the board for new experiments
   - requirements not fully specified at design time
3. Distributing and installing different software versions on many instruments can be confusing and can lead to errors. In this scenario, tracking the configuration of an instrument shared among many people can be difficult
4. Interfacing the board to a user-application
   Usually the end user of the software is not interested in the technical details of the components of the application, so the software should provide the available resources hiding the details.

The above considerations, in particular 1, 2 and 4, took the author to implement the firmware using the OOP (Object Oriented Programming) paradigm which better describes and helps to efficiently implement the behaviour of the instrument under development. The extra efforts introduced by the abstraction of the functionalities are convenient because

- it allows to develop a generic board object, based only on a standalone Arduino, even when the application hardware is not available yet
- the development of a new application is more flexible because it can be derived by extending the base object

Point 3 shows that using the same firmware on all the boards is convenient (unless some modules require special functions). This suggests that the version tracking mechanism should be included in the base board capabilities. Mixing different firmware releases in a system is possible as well, but the protocols must be compatible with the standard.

The resulting keywords for the development of the firmware are: *Abstraction* and *Object Oriented Programming.* C++ has been selected as a programming language for the board. At the first development step the firmware will be able to manage a generic board with fixed resources (orange terminals in Figure 2) by means of a base class. In the second step, the base class is extended to handle the final board with the remaining resources.

The reliability of the developed software can be further improved by using automatic code generation techniques and tools. This leads to describe parts of the expected code in a human-readable format which is the converted in the appropriate coding language (C, C++, C#, Python (3), other). The immediate result is the ability to correct a great number of sections of the code by correcting only a small portion of the code-generator software. The requirement for this approach is the separation of the resulting firmware sources into two groups of files. The first group is composed of code automatically generated according to the specifications. The second group is composed by manually written files which implement in detail the parts of the logic of the firmware which need to be customized.

The software has been developed in C++ language. As the Arduino IDE lacks of flexibility and refactoring capabilities, Eclipse CDT has been used.

As of now, Eclipse Oxygen 4.7.0 with Arduino C++ Tools (9.3.0) provide a good development environment.

Abstraction of the board

The board is accessed by an external user software (e.g. a graphical user interface) as described in (4). This requirement can be achieved in many ways but the trade-off between software complexity and flexibility took to presenting the board as a set of registers at standard addresses. Thus the external software has access to the contents of the registers by means of a set of standard commands exchanged with the board as shown in Figure 3.

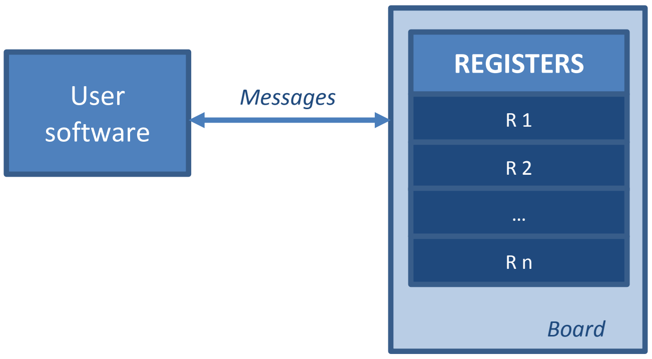
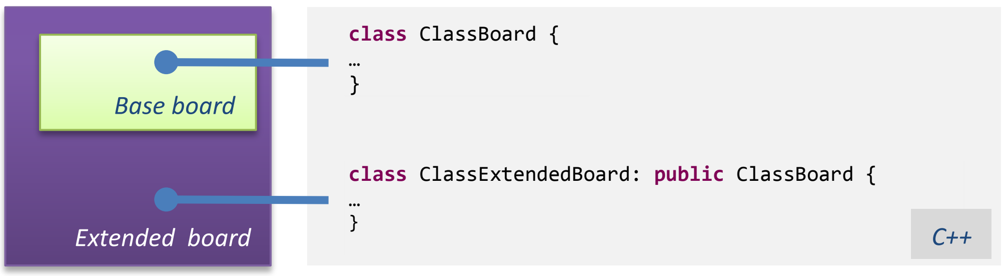


FIGURE 3 –ABSTRACTION OF THE BOARD AS A SET OF REGISTERS

In addition to the resources access requirement, the implementation must allow future expansion. Figure 4 shows the relation between a generic base board and the extended board which accesses the full set of board resources. The C++ implementation is based on the *inheritance* from the base class.



```
class ClassBoard {
…
}


class ClassExtendedBoard: public ClassBoard {
…
}
```

FIGURE 4 - BOARD OBJECTS HIERARCHY

The implementation of the base board class requires the definition of a set of operating configurations of the board, the definition of the communication protocol and the definition of a standard set of registers.

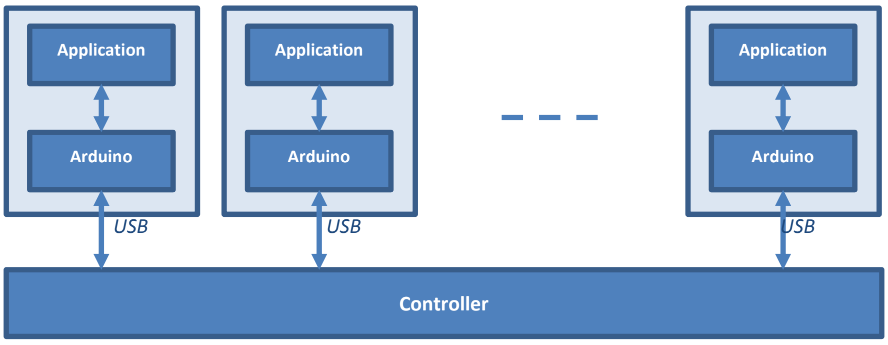
Board operating modes

The board is intended to be operated in three modes according to the user needs and the firmware has been designed to simplify the operation of the board in all the conditions. For this reason, one firmware version must be able to automatically support all the operating conditions without the user intervention.

Single board Configuration (Mode 1)

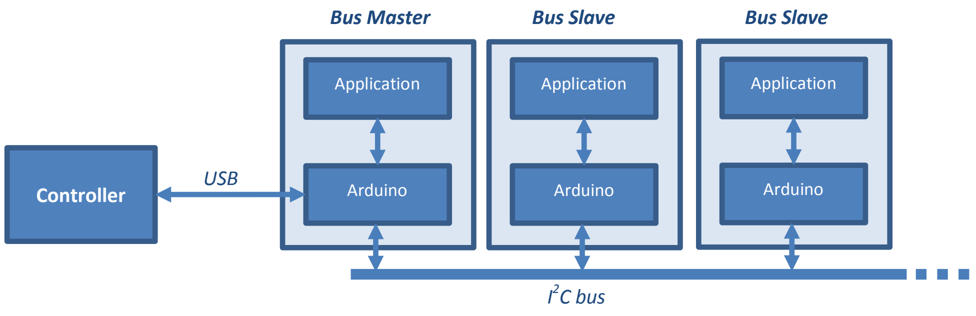
In this configuration, the board is connected to an external PC which provides the User Interface. The connection via Serial over USB through the Arduino Nano on-board conversion module.

This configuration is suitable for simple setups, where the mess of USB cables can be tolerated. In this setup, indeed, every board needs a separate USB connection.



Single Point Controlled System with Bus Connected Boards through I2C (Mode 2)

In order to reduce the controller-to-boards connections, a bus topology can be implemented as well. All the boards thus are installed into a rack and only a single connection to the controller is needed.

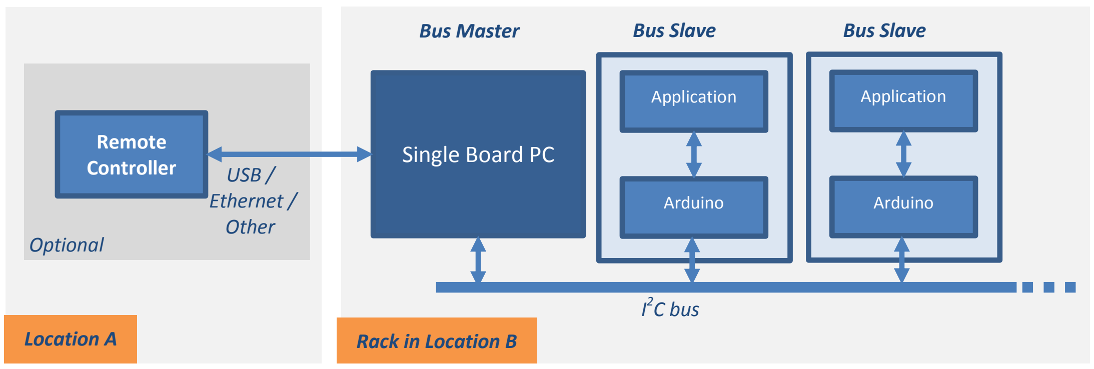


In this configuration all the boards can share the same firmware. The board connected via USB becomes the master of the I^{2}C bus, while the remaining boards remain slaves. It should be noted that although the I^{2}C bus is multi-master capable, this feature is not worth to be implemented in this firmware release because of the added complexity which would increase the firmware size. **Thus only one master a time is guaranteed to work correctly with the firmware described in this report.**

Standalone System with Bus Connected Boards through I2C (Mode 3)

In this configuration all the boards act as I^{2}C slaves. Strictly speaking, this configuration is not that much different from Mode2 as it only lacks the Arduino Board acting as the I^{2}C bus master. Thus the master function is assigned to an external dedicated hardware, for example a Single Board Computer (e.g. Raspberry Pi, Arduino with Ethernet or Wi-Fi Shield, Beaglebone, Industrial PC, etc).

This configuration can be used in experiments with complex topologies, for example in two remote locations A and B. The local Controller in location B coordinates the I^{2}C-connected boards with complex algorithms which otherwise could not be implemented on an Arduino board. Power consumption is kept low as well because of lower power absorption of single board computers with respect to conventional computers.



Another use case is a flexible control of the rack by a WiFi connection which grants access to mobile devices (tablets, PDAs, smartphones, etc) in the same Location B of the experiment.

A positive side effect of using a conventional I^{2}C bus to interconnect the Boards is that additional I^{2}C chips or third party boards can be integrated in a rack, for example to monitor the supply voltages and the temperature of the rack itself.

Firmware Architecture

The firmware is organized into layers and sublayers as depicted in Figure 5.

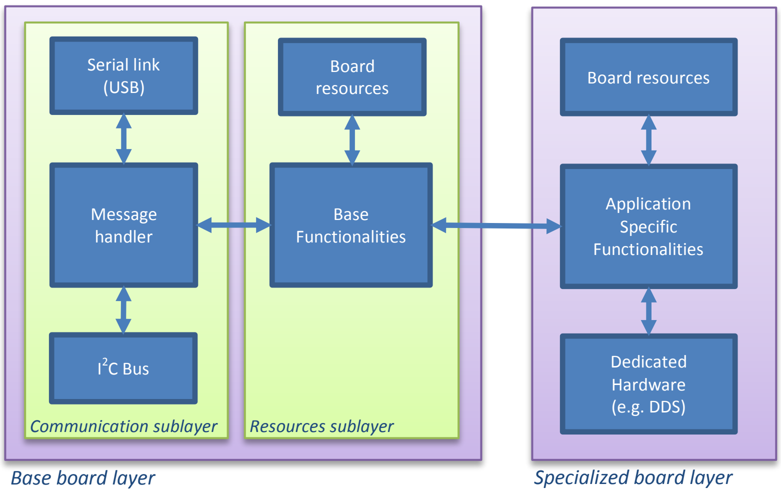


FIGURE 5 - FIRMWARE LAYERS

The resources of the board are handled by the *board* object, shown in Figure 6, which is the core of the firmware and has control of the internal and external resources.

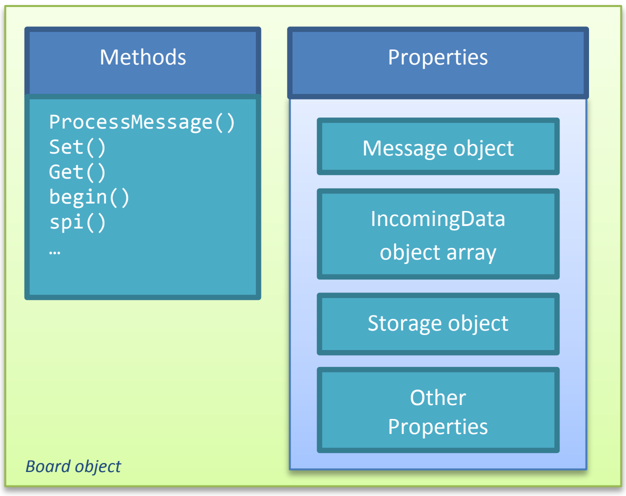


FIGURE 6 - BOARD OBJECT OVERVIEW

Figure 7 shows an example of the increasing abstraction given by the firmware when a user software sets the board name through a write access to register #20.

First the write register operation is turned into a call to the *Set()* method of the board. As the board name is stored in the non-volatile Arduino memory, the *Set()* method uses the board resource *Storage* which abstracts the EEPROM data transfer.

After finding the address at which the board name will be stored, the *Set()* method calls the method *store()* of the *Storage* object. Finally the *Storage* object calls the Arduino library function *EEPROM.write()* which performs the write operation.

Although the EEPROM write operation itself is quite simple, the organization of the firmware shows how to turn an abstract request into an access to an hardware resource.

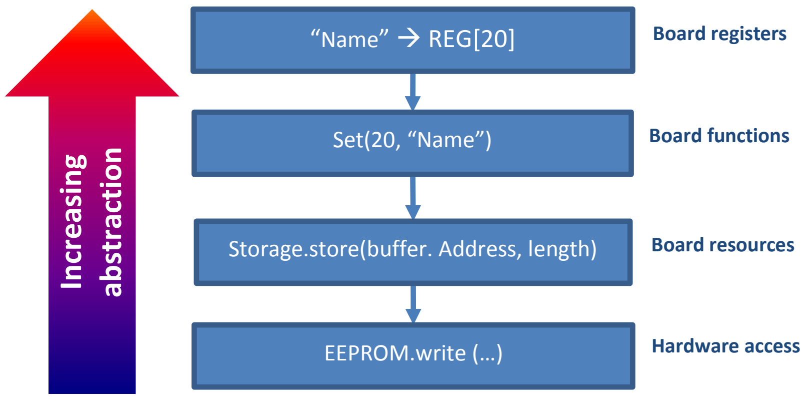


FIGURE 7 – EXAMPLE OF ABSTRACTION

Register access

The board presents the available resources as a set of registers which can be read and/or written. The read/write operations are performed by means of commands which are exchanged with the board.

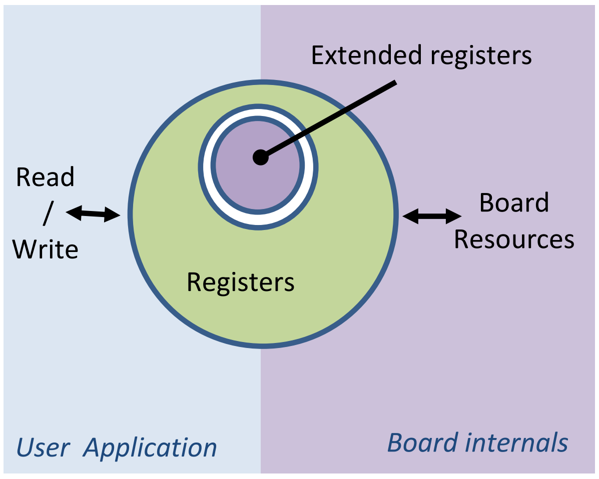


FIGURE 8 - REGISTER SET ACCESS

As the extended board will inherit the set of registers from the base board, only the base set needs to be specified in this phase of the design.

Implementation of the Base Board Layer

This layer implements the generic board with some basic functionalities which can be divided into two sublayers: communication layer and resources layer

Communication sublayer

The board has some bus interfaces devoted to the exchange of requests and replies. Although all the interfaces can be used to transfer data to/from the board, each of them has been chosen for a specific purpose. The primary interface is a serial link over USB and it should be used to easily connect the board to an external computer which acts as a controller. Another interface is a I^{2}C controller and it is intended to be used to interconnect many boards together in a complex system. Other communication interfaces could be added for special purposes. The main communication functionalities implemented in the communication sublayer are

- data transfer to/from every board regardless of the physical connection (USB or I^{2}C or other)
- decoding of a set of messages exchanged on the bus
- request of execution of decoded messages
- data forwarding from one bus to another when are data addressed to other boards

Board messages

In the current implementation, messages are text strings. Although the ASCII representation gives more overhead with respect to binary encoding, the chosen representation gives the immediate advantage of assuring the compatibility between the board and the communication software. Indeed any existing program able to send and receive text over a serial line, is suitable to operate the board without installation of any additional software. This avoids also the concern of debugging the client side of the application during the development phase of the firmware.

Messages are part of the messaging layer of the firmware and are defined as *c defines* in `classmessage.h`

```
#define PROTOCOL_VERSION  "ASCII 1"

// MESSAGES WITHOUT PARAMETERS
#define PROTOCOL_PROTOCOL "p"
#define PROTOCOL_WHO      "?"
#define PROTOCOL_ACK      "a"

// MESSAGES WITH ADDRESS AND/OR PARAMETERS: MSG [reg [val]]
#define PROTOCOL_IDENTIFY "i"
#define PROTOCOL_SYS      "*"
#define PROTOCOL_SET      "w"
#define PROTOCOL_GET      "r"
#define PROTOCOL_REPLY    "-"
#define PROTOCOL_REMARKS  "#"

#define PROTOCOL_ACCEPTED "ok"
#define PROTOCOL_FAIL     "fail"
```

The protocol provides for a reply to every message addressed to the board.

When a message to the board does not return a value, for example a board reset request, a reply is anyway generated depending on the command execution success or failure.

The reply to a successfully executed message is defined by PROTOCOL_ACCEPTED, and a rejected command (for example a wrong command or a message with missing/invalid parameters) gets a PROTOCOL_FAIL reply.

The current Communication Protocol is identified as “ASCII 1” (textual data, protocol version 1).

The board messages are listed in Table 1 and a short description of the message and an example of data exchange are given as well.

Code	Mnemonic	Description	Example
p	PROTOCOL	Request the Board Communication protocol	Request the protocol `Command: p` `Reply: - ASCII 1`
?	WHO	Request the Board ID	Request the ID to board #37 `Command: ?` `Reply: - 37`
*	SYSTEM [op]	Request a special operation	Reboot the board `Command: * restart` `Reply: - rebooting`
w	SET <reg> <val>	Write a register	Set the board name `Command: w 20 This is a board` `Reply: - ok`
r	GET <reg>	Read a register	Get the board name `Command: r 20` `Reply: - This is a board`
-	REPLY <data>	Inform that the message is a reply to a previous command	Request the ID to board #37 `Command: ?` `Reply: - 37`
#	REMARKS <data>	Inform that a part of the message should be ignored	Read register 50 in a SPI debug session `Command: r 50` `Reply:` `# spi(2 bytes) Tx: 00 12 Rx: 00 00` `# spi(5 bytes) Tx: 84 00 00 00 00` `# Rx: 00 00 83 12 6F` `- 1.000000047` *(Actual reply)*
??	LIST	Show the IDs of the boards connected to the I^2C bus	Boards #40, 50 on I^2C bus `Command: ??` `Reply: - 40 50`
i	IDENTIFY <id>	Start and identification process The process ends with ACK	Assign ID 40 `Command: i 40` `Reply: - ok`
a	ACK	An ID has been accepted or the identification process must end	Stop the identification `Command: a` `Reply: - ok`
f	*FORWARD <id>*	*Forward the messages to the specified board* [1]	*Board #37 connected to USB, forward messages to board #68* `Command: r 20` `Reply: - This is a board` `Command: f 68` `Reply: - ok` `Command: r 20` `Reply: - This is another board`

TABLE 1 - BOARD MESSAGES

A detailed description of the commands can be found in

[1] As many debug functions are embedded in the test release of the firmware, the message-forwarding feature has been temporarily disabled in order to reduce firmware size and RAM usage.

Appendix.

Message structure and line terminator convention

Each message sent from the external controller to the board must end with a line terminator. In order to resolve the compatibility among different software implementations of line terminator sequences (Window, OSX, Unix, Linux, etc), the terminator character can be both ASCII character 10 or 13. The first received character terminates the line and if the second character is present, it is ignored. The resulting structure of a message sent to the board is

Identifier		Data	Line terminator
Any character sequence among `“p” “?” “*” “w” “r” “-” “#” “??” “i” “a” “f”`	blank	Sequence of characters (if present)	CHR(10) or CHR(13) or both in any order
1 or 2 bytes	1 byte	Variable length	1 or 2 bytes

The structure of the messages is the same also for the replies from the board but the line terminator is one single character, ASCII code 10 (*newline* or ‘\n’ in *c* convention).

Identifier		Data	Line terminator
`“-” or “#”`	blank	Sequence of characters	CHR(10)
1 byte	1 byte	Variable length	1 byte

Board Messages execution

Due to the different board interconnections configurations discussed in section “Arduino Nano board overview and interconnections”, there are two message passing situations as depicted in Figure 9 and Figure 10.

Messages in Single Board Mode

When the Board is operating in Single Board Mode, there is a point to point connection between the board and the controller. Thus every message sent from one device is intrinsically addressed to the other one.

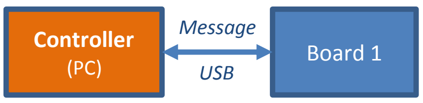


FIGURE 9 - MESSAGES IN SINGLE BOARD MODE

Messages in Bus Connected Mode

When the Boards are operating in Bus Connected Mode there are situations where the receiver of a message is not the intended final destination. With respect to Figure 10, for example when the Controller needs to send a message to Slave Board 2, the Board 1 is the only physical connection between the two points so it needs to forward the message. The same mechanism must be implemented also when the Board Slave 2 sends back a reply to the Controller. In these situations Board 1 must enter into a *Master Board Mode* which allows forwarding the messages to/from the I^2C bus.

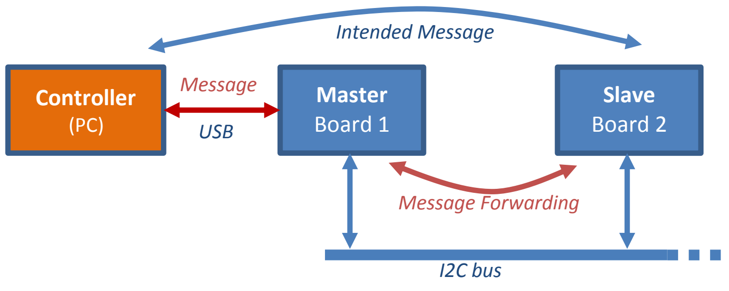


FIGURE 10 – MESSAGES IN BUS CONNECTED MODE

For this reason a dedicated register has been implemented in the board object which is accessed through the dedicated FORWARD command.

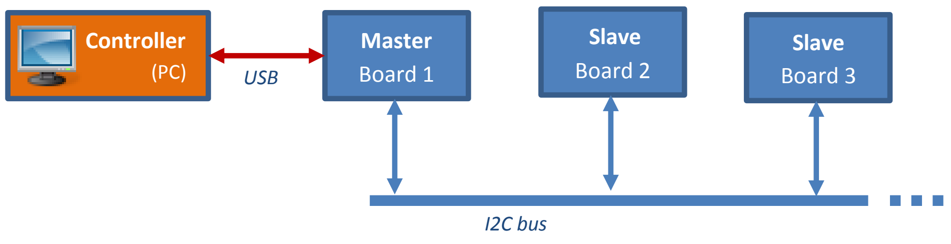
When a board receives a *FORWARD <id>* message, the board changes its role into Master and the received *id* is stored in an internal register.

When the board is in Master mode, the forwarding algorithm acts in two ways according to the direction of the data flow.

- Message from Master to Slave: the Master receives the message, after having checked the message type, the Master forwards to the I^2C bus. No reply is sent back at this point.
- Reply from Slave to Master: the Master receives the reply message and sends it as is via the USB connection.

In Master mode, the massages are not automatically forwarded and a check must be performed to avoid undesired behaviour of the devices on the bus.

The diagram below shows an example of a unexpected condition.



The external controller sends two forwarding requests:

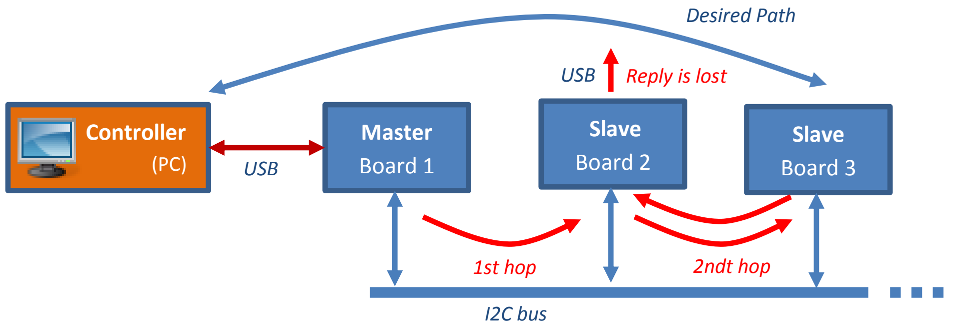
```
f 2
f 3
```

Upon receiving the first request, Board 1 becomes the master and Board 2 remains a Slave and this is correct.

If no message check is performed, after receiving the second request, the message is executed on the currently addressed board (Board 2) which becomes itself Master.

In the end, the status of the system is as follows

- The Controller knows that the Master is Board 1 because it is connected through USB
- Board 1 acts as a Master and it is not informed about the Master role of Board 2
- Board 2 acts as a Master and it is not informed about the presence of another Master (Board1)
- Board 3 knows that the master is Board 2



The result of the multi-hop message forwarding is the loss of the Board 3 reply.

Although the forwarding algorithm could be corrected in order to gather the incoming messages in Board 2 and sending them to the I^2C bus instead of USB, there is no immediate advantage in having a multi-hop routing of packets and it would lead to ambiguity of the system and complexity of the firmware.

For these reason, some messages need not to be forwarded by the Master even when forwarding is enabled and they must be processed and executed locally on the USB-attached board.

Message	Action on the Master when forwarding is enabled
WHO LIST ACK IDENTIFY FORWARD	Execute locally. No forwarding.
SYSTEM GET SET REPLY REMARKS	Forward to the other bus.

Resources sublayer

The base layer grants access to the board resources through a set of messages which are decoded by the communication layer. Upon an execution request the decoded command is then executed in the *resources sublayer* through dedicated callback functions.

The functionalities implemented in the *resources sublayer* can be

- board hardware access (SPI interface, GPIO lines, AD channels, embedded timers, EEPROM, etc)
- access to virtual registers used to abstract the board capabilities
- system-related functions (e.g. board identification on the bus, debug and firmware information)

At this point, the base board layer implements all the functions required by a generic board to operate in both a standalone or bus-connected configuration.

This approach offers some advantages in terms of future development time and reliability

- it allows to develop and debug the code even in absence of the application-specific hardware
- once the base code has been tested, its development can be frozen and the stable version can be reused for new applications

Base Board Registers

The following registers have been defined in the header file of the board class

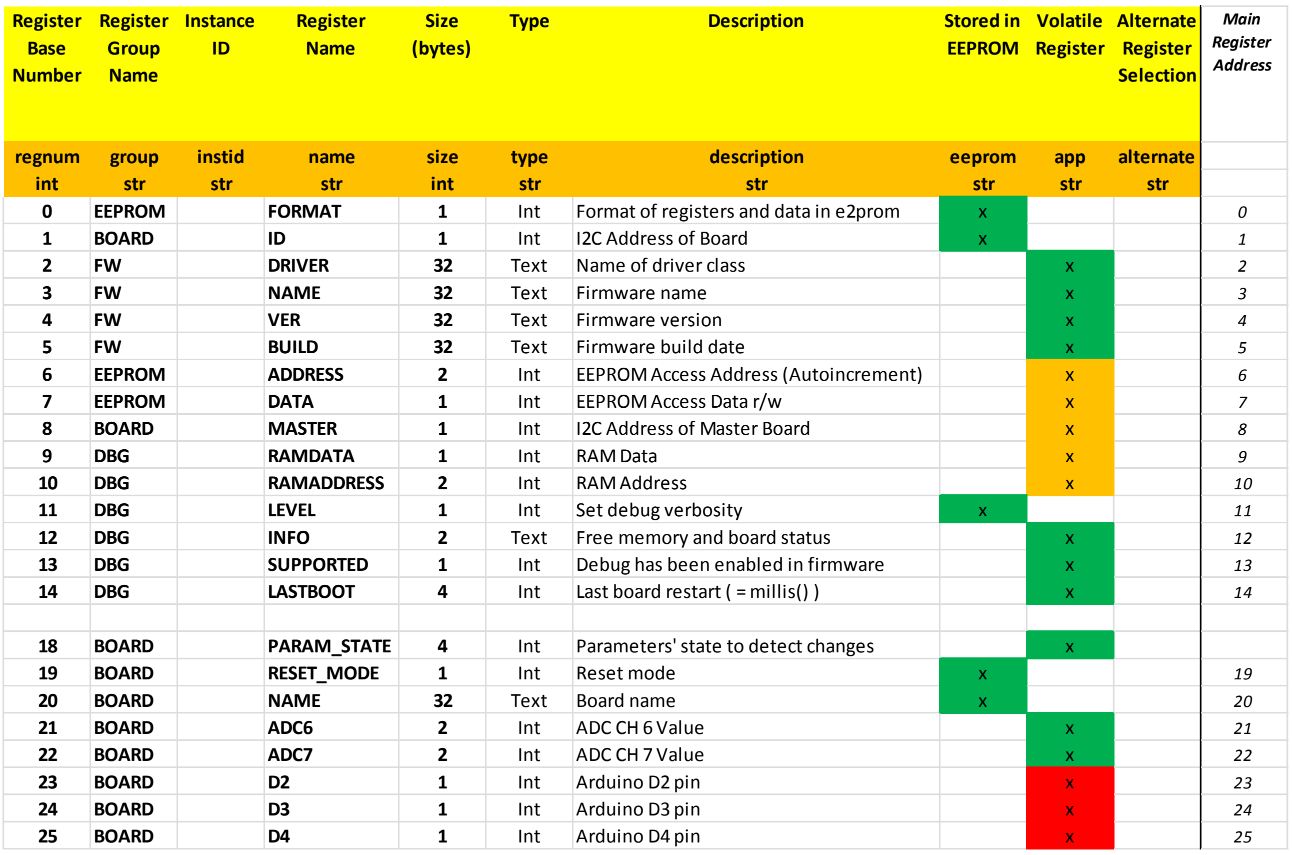
```
#define STO_EEPROM_FORMAT       0  // Format of registers and data in e2prom
#define STO_BOARD_ID            1  // I2C Address of Board
#define REG_FW_DRIVER           2  // Name of driver class
#define REG_FW_NAME             3  // Firmware name
#define REG_FW_VER              4  // Firmware version
#define REG_FW_BUILD            5  // Firmware build date
#define REG_EEPROM_ADDRESS      6  // EEPROM Access Address (Autoincrement)
#define REG_EEPROM_DATA         7  // EEPROM Access Data r/w
#define REG_BOARD_MASTER        8  // I2C Address of Master Board
#define REG_DBG_RAMDATA         9  // RAM Data
#define REG_DBG_RAMADDRESS      10 // RAM Address
#define STO_DBG_LEVEL           11 // Set debug verbosity
#define REG_DBG_INFO            12 // Free memory and board status
#define REG_DBG_SUPPORTED       13 // Debug has been enabled in firmware
#define REG_DBG_LASTBOOT        14 // Last board restart ( = millis() )
/*
    ADDRESS RANGE FOR FUTURE DEFINITION OF OTHER REGISTERS
*/
#define REG_BOARD_PARAM_STATE   18 // Parameters' state, used to detect changes
#define STO_BOARD_RESET_MODE    19 // Reset mode
#define STO_BOARD_NAME          20 // Board name
#define REG_BOARD_ADC6          21 // ADC CH 6 Value
#define REG_BOARD_ADC7          22 // ADC CH 7 Value
#define REG_BOARD_D2            23 // Arduino D2 pin
#define REG_BOARD_D3            24 // Arduino D3 pin
#define REG_BOARD_D4            25 // Arduino D4 pin
```

A detailed description of the registers can be found in

Appendix.

The above C code snippet has been generated by a code-generator written in Python, fed by the description of the data though an Excel file shown in Figure 11.

A detailed description of the code generator is given in (5).



Register Base Number	Register Group Name	Instance ID	Register Name	Size (bytes)	Type	Description	Stored in EEPROM	Volatile Register	Alternate Register Selection	*Main Register Address*
regnum int	group str	instid str	name str	size int	type str	description str	eeprom str	app str	alternate str	
0	EEPROM		FORMAT	1	Int	Format of registers and data in e2prom	x			*0*
1	BOARD		ID	1	Int	I2C Address of Board	x			*1*
2	FW		DRIVER	32	Text	Name of driver class		x		*2*
3	FW		NAME	32	Text	Firmware name		x		*3*
4	FW		VER	32	Text	Firmware version		x		*4*
5	FW		BUILD	32	Text	Firmware build date		x		*5*
6	EEPROM		ADDRESS	2	Int	EEPROM Access Address (Autoincrement)		x		*6*
7	EEPROM		DATA	1	Int	EEPROM Access Data r/w		x		*7*
8	BOARD		MASTER	1	Int	I2C Address of Master Board		x		*8*
9	DBG		RAMDATA	1	Int	RAM Data		x		*9*
10	DBG		RAMADDRESS	2	Int	RAM Address		x		*10*
11	DBG		LEVEL	1	Int	Set debug verbosity	x			*11*
12	DBG		INFO	2	Text	Free memory and board status		x		*12*
13	DBG		SUPPORTED	1	Int	Debug has been enabled in firmware		x		*13*
14	DBG		LASTBOOT	4	Int	Last board restart (= millis())		x		*14*
18	BOARD		PARAM_STATE	4	Int	Parameters' state to detect changes		x		
19	BOARD		RESET_MODE	1	Int	Reset mode	x			*19*
20	BOARD		NAME	32	Text	Board name	x			*20*
21	BOARD		ADC6	2	Int	ADC CH 6 Value		x		*21*
22	BOARD		ADC7	2	Int	ADC CH 7 Value		x		*22*
23	BOARD		D2	1	Int	Arduino D2 pin		x		*23*
24	BOARD		D3	1	Int	Arduino D3 pin		x		*24*
25	BOARD		D4	1	Int	Arduino D4 pin		x		*25*

FIGURE 11 - SAMPLE OF CODE GENERATOR INPUT

The Specialized board layer

This layer acts as an extension to the *Base board layer* and it is intended to give access to the application-specific hardware functionalities of the board through a set of virtual registers specific.

A detailed description of a DDS-Board specialization can be found in report (4).

Top level file of the Firmware

The main tasks of the Firmware are

- the Initialization of the board at boot time – Arduino environment `setup()` function
- the repeated handling of messages from all busses – Arduino environment `loop()` function
- handling of time-critical tasks inside an Interrupt Service Routine – `ISR(TIMER1_COMPA_vect)`

```
// ArduinoNano - DDS Board controller
// Written by Claudio Francese - 2017

#include <Arduino.h>
#include "utils.h"

// FIRMWARE INFORMATION
char* __MAIN_FILE_NAME__        = __FILE__;     // NAME OF THIS FILE
char* __FW_DRIVER__             = "dds";        // DRIVER NAME
char* __FW_VERSION__            = "1.0";        // FIRMWARE VERSION
char* __FW_NAME__               = "DDS BASE";   // FIRMWARE NAME / SHORT DESCRIPTION

#include "classextendedboard.h"
ClassExtendedBoard *Board = new ClassExtendedBoard(); // POINTER TO ARDUINO BOARD EXTENDED OBJECT

ISR(TIMER1_COMPA_vect) { // FUNCTION FOR INTERRUPT SERVICE ROUTINE - RUNS @ 4 kHz
        // PWM FOR ONBOARD ARDUINO LED
        static uint8_t counter=0;
        counter ++;
        digitalWrite(LED_ATTENTION, Board->LedLevel>counter?HIGH:LOW);
}

void setup() { // SETUP THE OBJECTS, ARDUINO PERIPHERALS, DDS
        Board->begin();
}

// MAIN APPLICATION LOOP
void loop() {
        Board->ProcessIO();             // READ, PROCESS AND UPDATE THE ARDUINO I/O SIGNALS
                                        // (FRONT AND REAR PANEL)
        Board->ProcessMessage(USB);     // PROCESS MESSAGES IN FIFO 1
        Board->ProcessMessage(I2C);     // PROCESS MESSAGES IN FIFO 2

}
```

FIGURE 12 - FIRMWARE MAIN FILE

The complexity of the application is hidden in the class `ClassExtendedBoard` which extends the base class `ClassBoard.`

Figure 13 graphically shows the program flow of the main loop of the firmware. With reference to the diagram, the abstract operations on the left (ProcessIO and Process the Message Queue) are turned into access to resources on the right through callback functions, each accessing the physical resources when needed.

The program flow also shows that two tasks run in parallel, as well. The first pushes messages in the appropriate Message Queue when there are new incoming data on a bus. The other task serves the time critical portion of the firmware trough the interrupt handler.

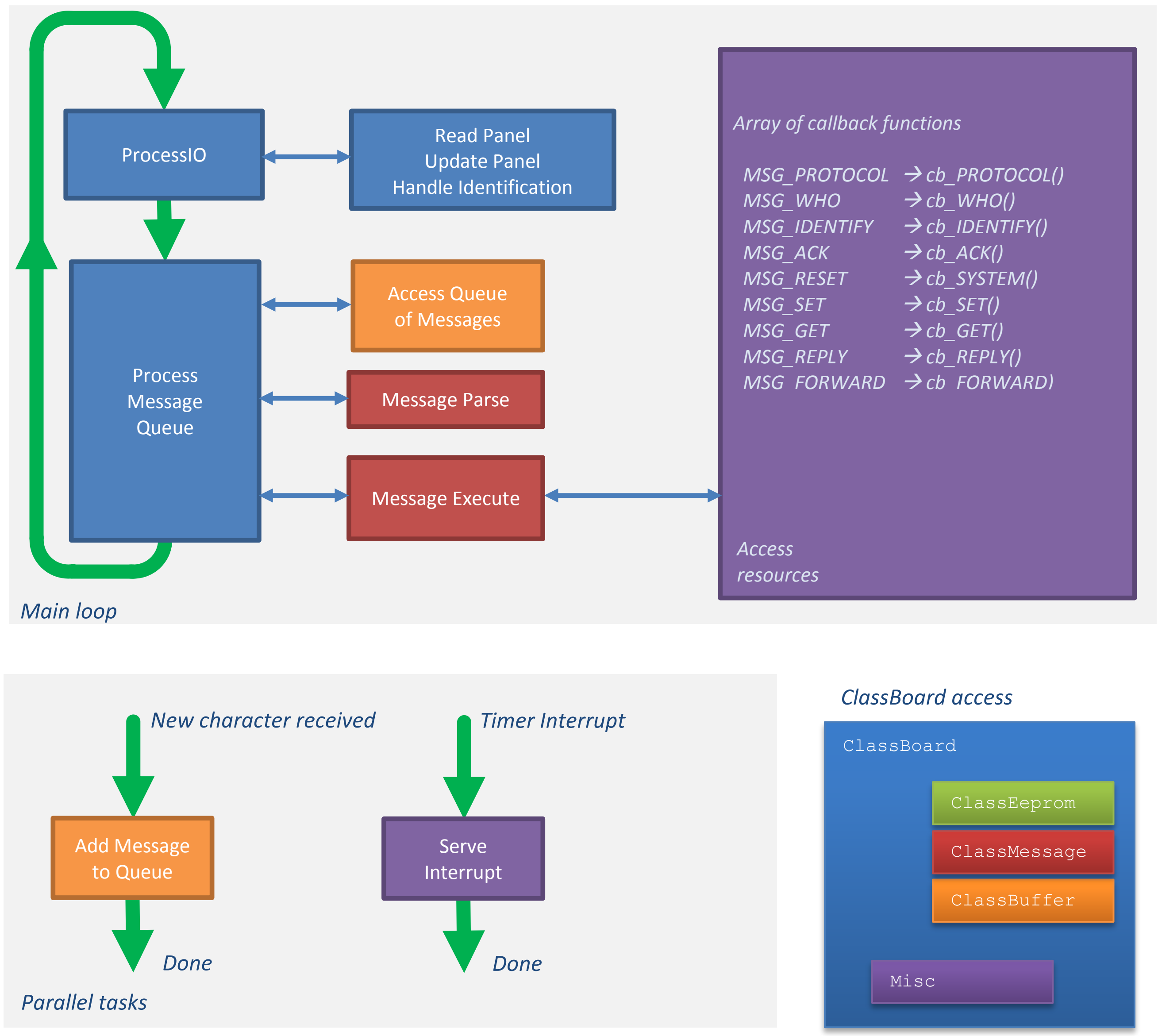


FIGURE 13 – FIRMWARE MAIN FILE PROGRAM FLOW

A description of the involved classes follows.

Class ClassBoard

This class is the core of the applications because it serves many purposes and provides

- a base board able to operate standalone or in bus-connection mode with a fully functional messaging mechanism accessed with only one function call per bus
- support to up to 8 bus connections for message exchange

- a set of standard registers accessible via read/write messages and/or dedicated messages
- handle of the internal state of the board (e.g. identification, normal operation)
- support to the parameter change detection, useful with concurrent accesses to the board
- mechanism to assign the ID to the board

The declaration of ClassBoard is shown in Figure 14

```
class ClassBoard {
  public:
    ClassBoard();// Class constructor, initializes obj properties

    // Virtual methods overridden by derived class to add application-dependant functionalities
    virtual void begin(void);           // Initialize Arduino and board peripherals
    virtual void System(void);       // Reset handler for RESET message
    virtual void Recall(void);       // Recall parameters from EEPROM
    virtual void Store(void);           // Store parameters to EEPROM

    // Default GET / SET handler for unknown register
    virtual bool DefaultSet(addtype reg, addtype eeaddr, uint8_t size, char* string);
    virtual bool DefaultGet(char* retval, addtype reg, addtype eeaddr, uint8_t size, char* value);

    // Convert a numeric (32 bit integer) value to / from physical units
    virtual void ConvertFromPhysicalUnits(addtype reg, char* string);// Convert from physical units
    virtual void ConvertToPhysicalUnits(addtype reg, char* string) ; // Convert to physical units

    // Base board methods
    void ProcessMessage(bustype);
    bool Set(addtype Reg, char* value); // Write Value string to Board Register,
                                            // returns TRUE on success, FALSE on failure
    bool Get(char*, addtype reg, char*);    // Read Board Register as String,
                                            // returns TRUE on success, FALSE on failure

    void ReplyToSerial(char* string);       // Send a message to serial
    void SendReply(C_STRING Reply) ;        // Send a message
    void ProcessIO();                           // Acquire the front panel
    valtype  getID();                       // Read the board ID
    void setID(valtype) ;                       // Write the board ID

    // STORE / RETRIEVE len BYTES TO / FROM EEPROM
    void store(uint8_t *buf, addtype addr, uint8_t len);
    void retr (uint8_t *buf, addtype addr, uint8_t len);

    void spi(uint8_t cspin, uint8_t *data, unsigned char len) ;   // SPI Transmit / Receive
    void StopIdentification();                  // Terminate identification process
    void AcceptIdentifier();                            // Accept the transmitted ID, send ACK
    void ParameterGroupChange(uint8_t group);   // INCREMENT PARAMETER-CHANGES COUNTERS

    uint8_t LedLevel;           // Brightness of Arduino LED (PWM in ISR)
    uint8_t DebugLevel; // Debug level (set debug verbosity)
    bool WarmBoot : 1;          // True if Arduino was reset upon serial reconnection.
                             // False after a cold boot.
    bool Identification : 1;    // True if the board in identification mode
    uint8_t IdentificationID;   // Proposed ID for board identification
    void StartIdentification(unsigned char);    // Start identification, enable led blinking
    ClassMessage *Msg;                          // MESSAGE HANDLING CLASS
    ClassBuffer IncomingData[2];        // ALLOCATE 2 FIFOs FOR INCOMING DATA CHANNELS

    ClassEeprom Storage;// EEPROM STORAGE FOR PERMANENT BOARD PARAMETERS
    uint8_t ResetMode;          // BOARD RESET MODE

  private:
    uint16_t RamAddress;// USED FOR RAM READ/WRITE
    uint16_t E2PromAddress;     // USED FOR EEPROM READ/WRITE
    uint16_t LedFlashCounter;   // USED TO MAKE A LED BLINK IN IDENTIFICATION MODE
    uint8_t  cached_boardid;    // ID OF THE BOARD (REDUCES THE NUMBER OF EEPROM R/W OPERATIONS)
    bool     Key_ID : 1;         // USED FOR IDENTIFICATION
    bool     Key_IDLast : 1;             // USED FOR IDENTIFICATION
    bool     ConditionAcceptedID : 1; // TRUE WHEN THE PROPOSED ID HAS BEEN ACCEPTED

  protected:
    uint32_t parametergroupcounter;     // STATE OF SOME PARAMETERS, USED TO DETECT CONCURRENT CHANGES
  } ;
```

FIGURE 14 - CLASSBOARD.H

Among the methods declared in the class, three are worth to be cited: `begin()`, `ProcessIO()` and `ProcessMessage()` because they are used in the main file of the firmware as shown in Figure 12.

Board setup : ClassBoard::begin()

Following the Arduino naming convention, this functions sets up the internals of the Board. So, after instantiating the Board object and having initialized the objects' properties by means of the constructor `ClassBoard()`, the firmware calls the `begin()` method which is used to initialize the Arduino onboard peripherals and the application all at once. As visible in Figure 14, the `begin()` method is marked with the **virtual** specifier, thus allowing the derived class to override it to extend the method's functionalities.

```
void ClassBoard::begin(void) {  // Initialize Arduino board
        // SET ARDUINO ONBOARD PERIPHERALS
        Serial.begin(115200);   // INITIALIZE SERIAL PORT

        Storage.retr((uint8_t*) &DebugLevel, EE_DBG_LEVEL, 1); // RECALL LAST DEBUG MODE
        Storage.retr((uint8_t*) &ResetMode, EE_BOARD_RESET_MODE, 1); // RECALL RESET MODE

        pinMode(SWITCH_ID, INPUT);              // Arduino reads  the Switch
        digitalWrite(SWITCH_ID, HIGH);          // Enable internal pullup


        // SETUP TIMER 1
        TCCR1A = 0;                       // normal operation
        TCCR1B = bit(WGM12) | bit(CS10);  // CTC, no pre-scaling
        OCR1A = 1999;                     // compare A register value
        TIMSK1 = bit(OCIE1A);             // interrupt on Compare A Match

        pinMode(LED_ATTENTION, OUTPUT);   // Enable led

        setID(getID());                   // Calls i2c setup

        // DETECT WARM / COLD BOOTSTRAP
        if (strcmp(BootstrapSignature, COLD_BOOT_MAGIC_STRING) == 0)
                WarmBoot = true;
        else {
                strprintf(BootstrapSignature, "%s", COLD_BOOT_MAGIC_STRING);
                WarmBoot = false;
        }
        if (DebugLevel == 255) WarmBoot = false;
}
```

Board acquisition and update of I/O signals: ClassBoard::ProcessIO()

The `ProcessIO()` method reads all the front panel inputs and updates the outputs. This method provides also support for the identification mechanism by updating the `ConditionAcceptedID` flag when it detects a keypress during identification. The identification mode is shown by making the on-board led blink fast.

```
void ClassBoard::ProcessIO() {
        // KEYS USED FOR IDENTIFICATION ACK
        Key_IDLast = Key_ID;
        Key_ID = digitalRead(SWITCH_ID);

        ConditionAcceptedID = Key_IDLast ^ Key_ID;

        // accept identification if requested by the user
        if (Identification && ConditionAcceptedID) AcceptIdentifier();

        /*
         * If in identification mode, check the panel button(s), blink the led(s)
         */
        static bool flag = false;

        if ( LedFlashCounter > (Identification?50:400)) {
                if (flag? (LedLevel<250):(LedLevel>1)) LedLevel = LedLevel + (flag?+4:-1);
                else flag = !flag;

                LedFlashCounter = 0;
        }
        LedFlashCounter++;
}
```

Board messages handler: ClassBoard::ProcessMessage()

This method has one parameter, a bus identifier. When called, the method checks if a pending message exists for the given bus. If available, the method decodes the message and tries to execute it returning the result of the operation to the bus. The message queue associated to the bus is cleared upon message execution.

```
void ClassBoard::ProcessMessage(bustype bus) {
        if (IncomingData[bus].Ready) {
                C_STRING NewMessage;        // CHARACTER BUFFER FOR NEW INCOMING MESSAGE
                IncomingData[bus].pop(NewMessage);
                Msg->Parse(bus, NewMessage);
                Msg->Exec();
        }
}
```

Such a class provides the functions necessary to operate a generic board which is connected to a USB and to an optional I^2C bus as shown in the main file of the firmware in Figure 12.

Extending the Register Set

As stated before, the base board capabilities need to be extended by extending the set of registers. At design time of the base class, the set of new registers which will be added in future is not known so the register handler must be implemented as a **switch** statement divided into two parts: the first part handles the known registers, while the **default** case calls a default handler for unknown registers. Figure 15 shows the implementation of the read access through the `Get()` method of the base class.

```
bool ClassBoard::Get(char* retval, addtype reg, char* extravalue) { // Read Board Register as a string

// INITIALIZATION CODE OMITTED

switch (reg) {
        // ACCESS REGISTERS KNOWN AT DESIGN TIME
        case REG_EEPROM_ADDRESS:
                value = E2PromAddress;
                break;
        case REG_EEPROM_DATA:
                Storage.retr((uint8_t*) &value, E2PromAddress, 1);
                E2PromAddress++;
                break;
        case STO_DBG_LEVEL:
                value = DebugLevel;
                break;

        // ... OTHER BASE BOARD REGISTERS

        case REG_BOARD_PARAM_STATE:
                value = parametergroupcounter;
                break;

        case REG_DBG_LASTBOOT:
                value = millis();
                break;

        // HANDLE REGISTERS NOT KNOWN TO THE BASE CLASS
        default:
                if (DefaultGet(retval, reg, eeaddr, size, extravalue)) { // operations … }
        }
}
bool ClassBoard::DefaultGet(char* retval, addtype reg, addtype eeaddr, uint8_t size, char* value) {
        return false;
} // Virtual method overridden in derived class
```

FIGURE 15 - REGISTER ACCESS IMPLEMENTATION FOR EXTENSION

The default handler has an empty implementation in the base class `ClassBoard` because the actual object instantiated in the firmware will be of class `ClassExtendedBoard` and the called method which will have to be called is the `ClassExtendedBoard::DefaultGet()` not `ClassBoard::DefaultGet()`. The same applies to write access. This is the reason why the methods `DefaultGet()` and `DefaultSet()` are marked as **virtual** in the base board class.

As an example, Figure 16 shows the implementation of the `ClassExtendedBoard ::DefaultGet`() method.

```
bool ClassExtendedBoard::DefaultGet(char* retval, addtype reg, addtype eeaddr, uint8_t size, char* extravalue) {
        // Handles read  access to registers not handled by the base class

        // INITIALIZATION OMITTED

        bool retstatus = true;
        valtype value = 0;

        switch (reg) {

        case REG_CH0_FREQ:
        case REG_CH1_FREQ:
        case REG_CH2_FREQ:
        case REG_CH3_FREQ:
                value =  GetFrequency(reg-REG_CH0_FREQ);
                break;

        case REG_CH0_AMPL:
        case REG_CH1_AMPL:
        case REG_CH2_AMPL:
        case REG_CH3_AMPL:
                value = GetAmplitude(reg-REG_CH0_AMPL);
                break;

        case REG_CH0_PHAS:
        case REG_CH1_PHAS:
        case REG_CH2_PHAS:
        case REG_CH3_PHAS:
                value = GetPhase(reg-REG_CH0_PHAS);
                break;

        default:
                retstatus = false;
        }

        if (retstatus) strprintf(retval, "%lu", value);

        return retstatus;
}
```

FIGURE 16 - EXAMPLE OF DEFAULTGET FOR A DDS

The graphical flow of the code is shown in Figure 17.

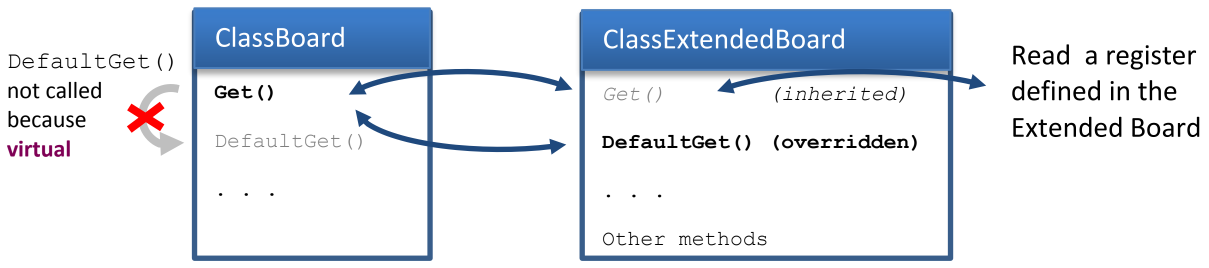
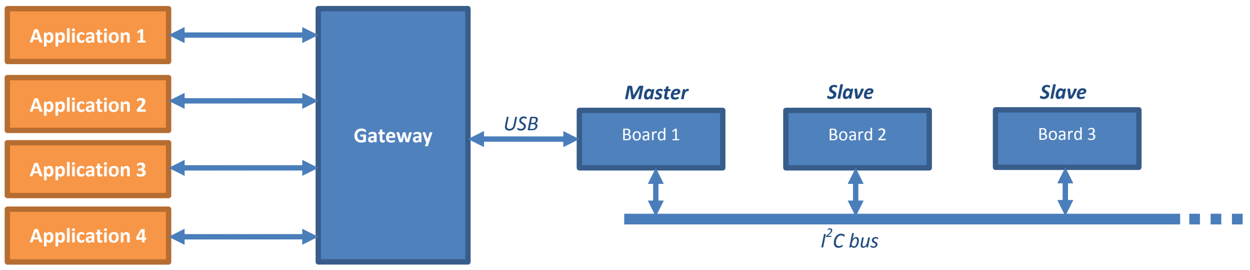


FIGURE 17 - CODE FLOW FOR A REGISTER READ ACCESS

Parameter change detection

When the board is operated in a more complex system, a dedicated machine could run a gateway software among the USB-connected Master Board and the controlling user software. The connection type between the gateway and the user application is not part of this report and will be discussed in (6).



In this concurrent configuration, a user application could not be aware whether some board's parameters have been changed by another user application. Fort this reason a write-access counter to the registers has been implemented. The registers are grouped in 4 sets according to priority and 4 counters are available, one for each priority group.

```
#define PARAM_GROUP_PRIORITY_HIGHEST    0
#define PARAM_GROUP_PRIORITY_HIGH       1
#define PARAM_GROUP_PRIORITY_LOW        2
#define PARAM_GROUP_PRIORITY_LOWEST     3
```

When a register in a group is written, the corresponding counter is increased by one. A user application can poll the register counter and verify which groups of registers should be reloaded.

Table 2 shows the association among registers and counters of the base board.

Register	Priority group
STO_BOARD_ID STO_DBG_LEVEL	PARAM_GROUP_PRIORITY_LOW
STO_BOARD_NAME	PARAM_GROUP_PRIORITY_LOWEST

TABLE 2 - REGISTERS ACCESS PRIORITY GROUPS

The four counters are actually stored into a single 32-bit register `REG_BOARD_PARAM_STATE` which can be read with the command GET.

The value of a counter is updated by calling the `ClassBoard::ParameterGroupChange()` method with the given priority group as a parameter

```
void ClassBoard::ParameterGroupChange(uint8_t group) {   // INCREMENT THE COUNTER OF PARAMETER-GROUP CHANGES
        // 8 bit FOR EVERY COUNTER
        // 0: BOARD LOW LEVEL PARAMETERS
        // 1: BOARD HIGH LEVEL PARAMETERS
        // 2: APPLICATION LOW LEVEL PARAMETERS
        // 3: APPLICATION HIGH LEVEL PARAMETERS

        uint32_t mask   = ((uint32_t) 0xFF) << (8*group);
        uint8_t count   = (parametergroupcounter & mask) >> (8*group);
        parametergroupcounter = (parametergroupcounter & ~mask) | ((uint32_t) (count + 1) << (8*group))  ;
        }
```

The method `ClassBoard::ParameterGroupChange()` is called in the firmware where the concerned registers are accessed. For example, the following code inside the `ClassBoard::Set()` method performs the job at each write access to any of the three specified registers.

```
        switch (reg) {
        case STO_BOARD_ID:
        case STO_DBG_LEVEL:
                ParameterGroupChange(PARAM_GROUP_PRIORITY_LOW);
                break;
        case STO_BOARD_NAME:
                ParameterGroupChange(PARAM_GROUP_PRIORITY_LOWEST);
                break;
        }
```

Other registers can be added to other groups by inserting a similar code in the relevant part of the firmware, for example in the derived Board object.

Class ClassBuffer

This class implements a FIFO which is used to queue the incoming data from the bus and to yield the data to the upper layer of the software (ClassMessage) which decodes the message. In addition to the constructor, three methods have been implemented to add a new character, retrieve the buffer, clear the data buffer. An access to the `millis()` function of Arduino, grants the receive timeout handling capability. In case of timeout during data reception, the buffer is automatically cleared.

```
ClassBuffer::ClassBuffer() {
        flush();
        lastrx = millis();
}

// Add a character to the buffer
// When Newline is received, set Ready = true
void ClassBuffer::push(char inChar) {
        if ((unsigned long) (millis() - lastrx) >= 1000) flush(); // RX TIMEOUT = 1.0 s
        int buflen = strlen(Buf);
        if (((inChar == '\n') | (inChar == '\r'))) {     // WIN/UNIX COMPATIBILITY
                if (buflen > 0) Ready = true;
                }
        else // add it to the inputString:
        {
                Buf[buflen] = inChar;
                Buf[buflen + 1] = '\0';
        }
        lastrx = millis();      // UPDATE THE TIMEOUT COUNTER
}

// return the buffer and prepare for new incoming data (clear buffer, set Ready=false)
void ClassBuffer::pop(char* retbuf) {
        strcpy(retbuf, Buf);
        flush();
}

void ClassBuffer::flush() {
        Ready = false;
        for (int i = 0; i < STRING_MAXLEN; i++)
                Buf[i] = '\0';
}
```

Class ClassEeprom

This class implements the Read/Write access to the Arduino internal EEPROM memory. Currently two methods have been implemented to read / write an arbitrary number of bytes from/to the EEPROM.

```
void ClassEeprom::store(uint8_t *buf, addtype addr, uint8_t len) {  // STORE len BYTES TO EEPROM
  uint8_t tmp;

  for (int i=0; i<len; i++) {
    tmp = *(buf+i);
    EEPROM.write(addr+i,   tmp);
    }
  }

void ClassEeprom::retr(uint8_t *buf, addtype addr, uint8_t len) {   // RETRIEVE len BYTES FROM EEPROM TO BUFFER
buf
  for (int i=0; i<len; i++)  {
    *(buf+i) = EEPROM.read(addr+i);
    }
  }
```

Class ClassMessage

This class serves many purposes related to messages

- defines the format of messages
- provides a method to decode a message verifying the correct number of parameters
- provides an entry point to execute the decode message by means of callback functions
- associates a callback function to every expected type of message

```
enum messages { // KNOWN MESSAGES
    MSG_WHO,
    MSG_LIST,
    MSG_IDENTIFY,
    MSG_ACK,
    MSG_RESET,
    MSG_SET,
    MSG_GET,
    MSG_MASTER,
    MSG_REPLY,
    MSG_UNKNOWN // MUST BE THE LAST ELEMENT OF messages ENUM
    };

class ClassMessage {  // MESSAGE MANAGING (PARSING AND EXECUTION)
  public:
        ClassMessage(ClassBoard *b);
    void Parse(bustype source, C_STRING inputString); // SPLIT THE MESSAGE INTO COMPONENTS
    void Exec();                                        // EXECUTE THE COMMAND
    void RegisterCallback(messages m, MessageCallback *c);  // ASSOCIATE A CALLBACK
                                                        // FUNCTION TO A COMMAND

    C_STRING string;        // COPY OF INPUT MESSAGE
    bool CanExecute : 1;    // TRUE IF THE COMMAND IS CORRECT
    bustype Bus : 3;        // SOURCE BUS OF MESSAGE - 3 bits : 8 Busses
    messages Message : 4;   // MESSAGE TOKEN : 16 Messages
    uint16_t Parameter;     // REGISTER OR BoardID (IF PRESENT IN MESSAGE)
    C_STRING Value;         // DATA (IF PRESENT IN MESSAGE)

    MessageCallback *callbacks[MSG_UNKNOWN + 1];    // ARRAY OF COMMAND CALLBACK FUNCTIONS
    bool tokenizer(char* input, char * tok, char sep, uint8_t maxlen) ; // SPLIT A STRING
                                                                    // INTO TOKENS
  private:
    ClassBoard *Board;
        };
```

FIGURE 18 - CLASSMESSAGE DECLARATION

As the board supports multiple bus connections for messages, the class also associates the identifier of the bus which the massage was received from (field bustype Bus).

The implementation of the Parse() function, which decodes the received message, is quite simple yet annoying. Thus only a snippet of code is reported here as an example

```
void ClassMessage::Parse(bustype source, C_STRING inputString) { // Parse the input message,
splitting it in fields
        Bus = source;
        CanExecute = false;
        Message = MSG_UNKNOWN;
        Parameter = 0;
        Value[0] = (char) 0;
        strcpy(string, inputString);
        char tok[16];

        // PARSE INPUT STRING AND FIND THE NUMERIC ID OF THE COMMAND/MESSAGE
        if (tokenizer(inputString, tok, ' ', STRING_MAXLEN)) {
                if (!strcmp(tok, PROTOCOL_WHO))
                        Message = MSG_WHO;
                else if (!strcmp(tok, PROTOCOL_IDENTIFY))
                        Message = MSG_IDENTIFY;
                else if (!strcmp(tok, PROTOCOL_ACK))
                        Message = MSG_ACK;
                else if (!strcmp(tok, PROTOCOL_SYS))
                        Message = MSG_RESET;
                else if (!strcmp(tok, PROTOCOL_SET))
                        Message = MSG_SET;
                else if (!strcmp(tok, PROTOCOL_GET))
                        Message = MSG_GET;
                else if (!strcmp(tok, PROTOCOL_REPLY))
                        Message = MSG_REPLY;
        }
        else return;    // EMPTY COMMAND

        switch (Message) {      // ACK and RESET are always enabled
                case MSG_ACK:
                case MSG_RESET:
                        CanExecute = true;
                        break;

                default:        // ENABLE/DISABLE OTHER MESSAGES DURING IDENTIFICATION
                        CanExecute = !Board->Identification;
                        break;
                }

        if (CanExecute) {
                switch (Message) {      // GET 1st PARAMETER (Register or BoardID)
                case MSG_SET:
                case MSG_GET:
                case MSG_IDENTIFY:
                        CanExecute = strcmp(inputString, ""); // MISSING PARAMETER
                        if (!CanExecute) {
                                Message = MSG_UNKNOWN;
                                break;
                                }
// … continues …
```

As shown in Figure 19, the execution of a command specified in a message can take place only if the decoder marked if as executable (field `bool CanExecute`) or if the received message was unknown (`MSG_UNKNOWN == Message`). In the latter case, the callback function associated to the message must handle the condition, typically sending an error code back to the source of the message.

```
void ClassMessage::Exec() { // Execute the last parsed command
        if (CanExecute | (MSG_UNKNOWN == Message)) { // MESSAGE IS ADDRESSED TO THIS BOARD
                                                     // OR HAS NO ADDRESS OR IT IS UNKNOWN -> EXECUTE
                if (this->callbacks[Message])
                        this->callbacks[Message](); // USE REGISTERED CALLBACK FUNCTION
        }
}
```

FIGURE 19 - MESSAGE EXECUTION

The method which associates a message to its callback function quite simply stores the function pointer in the callback-functions array.

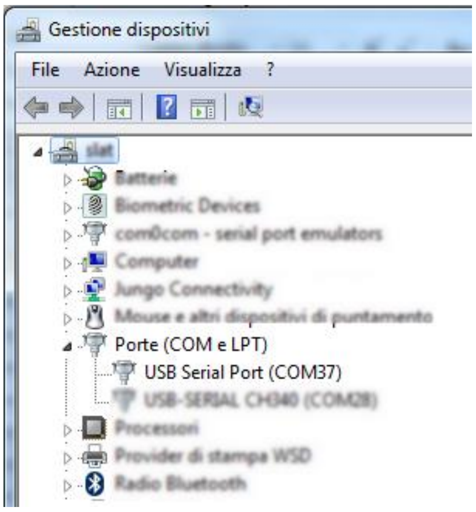
```
typedef void (MessageCallback)(void) ; // CALLBACK PROTOTYPE FOR MESSAGES HANDLING

void ClassMessage::RegisterCallback(messages m, MessageCallback *c) {
        callbacks[m] = *c;
} // Associate a callback function to a given message_id
```

Operating the board

Once connected the Arduino board to an external controller via the USB cable, the board is seen as a serial communication device.

Serial port

In a Windows Operating System, the device manager can be used to show the name of the port to connect to.



The serial port settings are:

Baudrate	115200 kbaud
Data size	8 bit
Parity	None
Stop	1
Flow Control	None

Communication Protocol

Before starting to use the Board, the communication protocol version should be checked. Currently only version “**ASCII 1**” has been implemented but future releases could define new data transfer encodings and protocols.

Identification procedure

Every board must have a unique ID on the I^{2}C bus. Some method could be used to set the ID (hardcoding in the firmware, using a dedicated configuration tool to transfer the information into the non volatile memory of each board through the serial port, etc) but none is comfortable for the end user, especially when there are many boards installed inside a rack and/or the only accessible USB connector is on the master board. For these reasons it has been decided to implement the ID assignment function in the board firmware. This allows to reconfigure the boards’ IDs without connecting them to any programming tool.

In the identification process, the bus master sends to all the listening boards a *Start of Identification* message followed by the numeric ID which has been proposed for assignment.

For example:

```
        i 40
```

Each board enters in the *Identification Mode* and waits for an action on any of its front panel switches. During this phase, all the boards keep on listening at the bus without sending any message.

The user decides which board to assign the ID to by acting any panel switch on the chosen board. Immediately after this operation, the board sends a broadcast message onto the bus to inform all the other boards and the controller that the ID has been accepted.

After sending the message, the board stores the ID into its EEPROM and reverts to the *Normal Mode*.

The identification process can also be interrupted by injecting an acknowledgment message onto the bus. Typically this can be done by the external controller upon software or user request. Receiving the message puts all the other boards in the *Normal Mode* again and ends the assignment process of that ID.

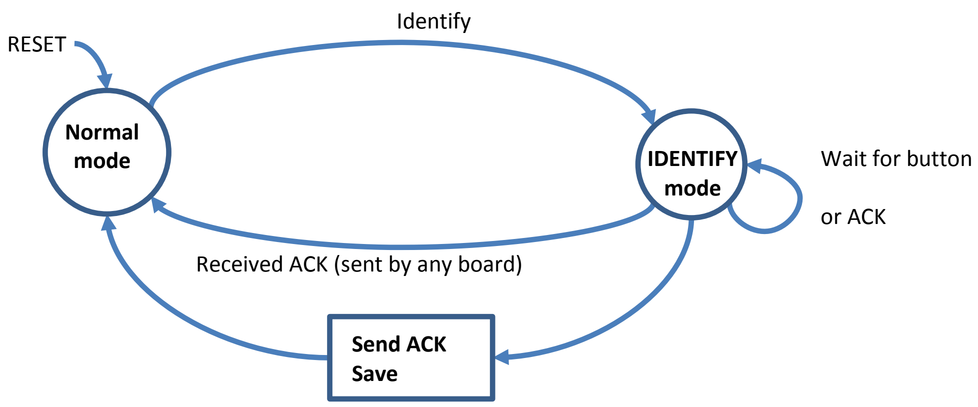


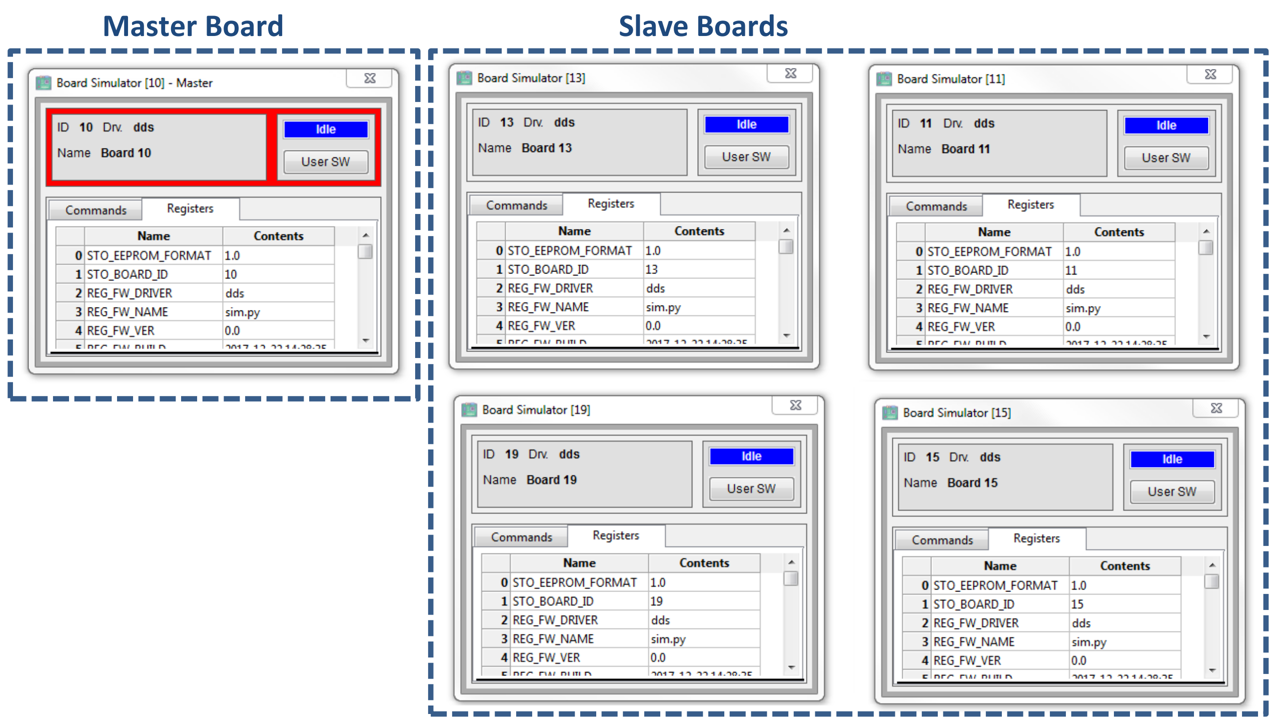
FIGURE 20 - STATE DIAGRAM OF THE BOARD ID ASSIGNMENT

Appendix

Board Simulator

During the development of the firmware, the board simulation environment shown in Figure 21 was developed as well. The environment is written in Python / wxPython and it was used in Windows 7 OS.

The purpose of the simulator was the test of the protocol and board abstraction during their definition.



Board Simulator [10] - Master

ID **10** Drv. **dds**
Name **Board 10**

	Name	Contents
0	STO_EEPROM_FORMAT	1.0
1	STO_BOARD_ID	10
2	REG_FW_DRIVER	dds
3	REG_FW_NAME	sim.py
4	REG_FW_VER	0.0

Board Simulator [13]

ID **13** Drv. **dds**
Name **Board 13**

	Name	Contents
0	STO_EEPROM_FORMAT	1.0
1	STO_BOARD_ID	13
2	REG_FW_DRIVER	dds
3	REG_FW_NAME	sim.py
4	REG_FW_VER	0.0

Board Simulator [11]

ID **11** Drv. **dds**
Name **Board 11**

	Name	Contents
0	STO_EEPROM_FORMAT	1.0
1	STO_BOARD_ID	11
2	REG_FW_DRIVER	dds
3	REG_FW_NAME	sim.py
4	REG_FW_VER	0.0

Board Simulator [19]

ID **19** Drv. **dds**
Name **Board 19**

	Name	Contents
0	STO_EEPROM_FORMAT	1.0
1	STO_BOARD_ID	19
2	REG_FW_DRIVER	dds
3	REG_FW_NAME	sim.py
4	REG_FW_VER	0.0

Board Simulator [15]

ID **15** Drv. **dds**
Name **Board 15**

	Name	Contents
0	STO_EEPROM_FORMAT	1.0
1	STO_BOARD_ID	15
2	REG_FW_DRIVER	dds
3	REG_FW_NAME	sim.py
4	REG_FW_VER	0.0

Figure 21 - Board Simulator

The simulator is able to mimic the behaviour of boards acting both as a Master or Slaves. It implements the base registers as stored values in a table and the effect of those registers which do not require a dedicated hardware (i.e. signal generation, data conversion, etc)

The simulator of the Master Board is launched with the command in Windows 7

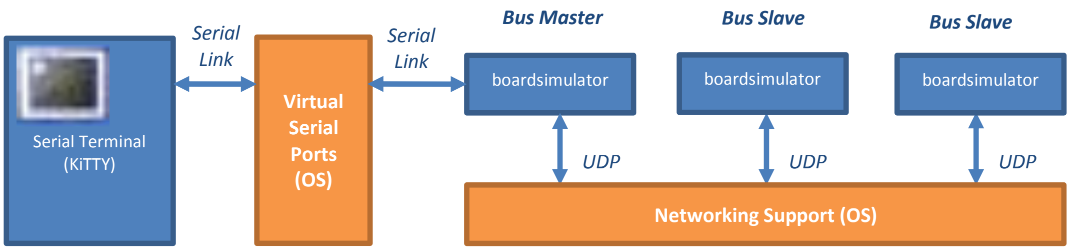
```
start boardsimulator -i 10 --master --driver=dds --serial=cncb0
```

where the parameters mean: Board ID 10, Master Mode, mimic a board which responds "dds" when queried for its driver type, connect the board to the serial port cncb0. This serial port, actually, is a virtual serial port emulated by the Operating System (7). So, the emulated virtual port allows to connect an application on the other hand of the "virtual cable" without having two physical ports. In this case, the other end is cnca0 which is accessed through a terminal emulator (KiTTY).

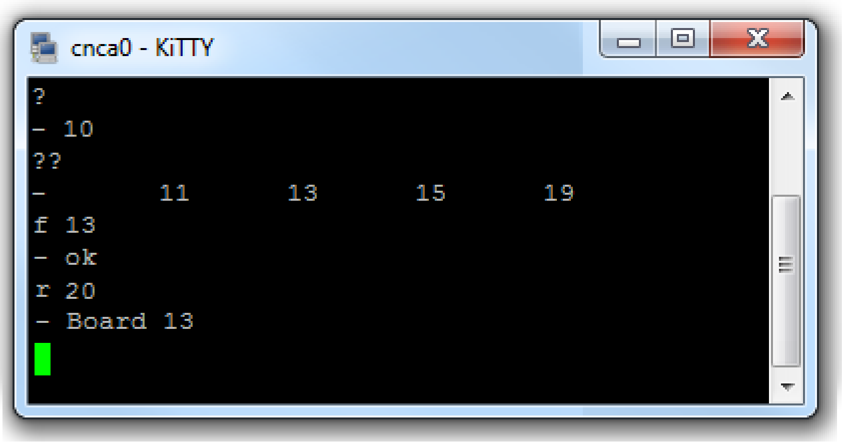
The emulated board system is competed by launching the other board simulator instances as Slaves.

```
start boardsimulator -i 11 --driver=dds
start boardsimulator -i 13 --driver=dds
start boardsimulator -i 15 --driver=dds
start boardsimulator -i 19 --driver=dds
```

The board to board interconnection in the physical system is the I^{2}C bus but it is not available in the emulated environment, so also the I^{2}C bus communication has been emulated by letting every `boardsimulator` instance send and receive broadcast UDP datagrams.



As Figure 22 shows, the User Application (in this case KiTTY) through the serial access to the Master, can control all the boards connected to the system.



```
?
- 10
??
-        11       13       15       19
f 13
- ok
r 20
- Board 13
```

FIGURE 22 - BOARD SIMULATOR RESULTS

This allows also to start the development of the User Applications or Mid-Tier applications (6) even when the firmware has not been completed yet.

Board commands

Command PROTOCOL – Request Communication Protocol

SYNOPSIS

p

Request the implemented communication protocol to the Board.

In a complex system, the controlling software should send this command before starting to send other messages in order to verify the compatibility or to adapt to the given protocol.

Example. Request the communication protocol to a Board with ASCII version 1 implementation

```
p
- ASCII 1
```

Command WHO – request the board ID

SYNOPSIS

?

The WHO command is used to query the connected board for its numeric ID used for I^{2}C addressing. **The valid addresses are in the range starting from 8 to 119.**

The board ID is stored in EEPROM and is R/W accessible through Register #1 as well.

Example. Request ID to USB-connected Board with ID = 37

```
?
- 37
```

The I^{2}C addressing scheme is 7 bit. Although 128 addresses are possible, 16 of them are reserved according to the following table leaving 112 available addresses for normal the bus operation.

Slave Address	R/W Bit	Address Range	Description
000 0000	0	0	General call address
000 0000	1	0	START byte
000 0001	X	1	CBUS address
000 0010	X	2	Reserved for different bus format
000 0011	X	3	Reserved for future purposes
000 01XX	X	4 … 7	Hs-mode master code
111 10XX	X	120…123	10-bit slave addressing
111 11XX	X	124 … 127	Reserved for future purposes

For further details about the I^{2}C bus, please see (8).

Command LIST – list the I^{2}C attached boards

SYNOPSIS

??

The LIST command, asks the Master to scan the I^{2}C for attached *devices*.

The ID of the Master board is not included in the results.

The scan process is performed at a low-level by addressing a device and waiting for an I^{2}C-acknowledge, thus a *device* can be both an Arduino board running the described firmware or any I^{2}C compatible device (including bare integrated circuits).

Example. Boards 37, 40, 41 connected through I^{2}C; Board 37 connected via USB as well. Request the list of boards connected to Board with ID = 37

```
??
- 40 41
```

Command FORWARD - Enable Message Forwarding

SYNOPSIS

f ID

f

Upon receiving the FORWARD Message, the Message Forwarding mode is enabled on the Master Board.

In this mode, the messages SYSTEM, GET, SET, REPLY, REMARKS forwarded and executed on the addressed board. The remaining messages WHO, LIST, ACK, IDENTIFY, FORWARD are executed on the Master Board.

Receiving the FORWARD message without the Board ID or a SYSTEM RESET terminate the Forwarding Mode and successive commands are execute locally.

Example. Address successive messages to Board 60

```
f 60
- ok
```

Command SYSTEM – issue a special command to the board

SYNOPSIS	
	*
	* parameters

The SYSTEM command requests a special board function.

The currently system messages for the base board are

* reset	Board RESET	Reboots the board.
* recall	Parameters RECALL	Retrieves the base board parameters from the EEPROM

Example. Restart the Arduino Board

```
*
- rebooting ...
```

Command IDENTIFY - Start Identification

SYNOPSIS
i ID

Upon receiving a IDENTIFY message, the board enters in Identification Mode. In the operating mode, the execution of commands is disabled. The only permitted Messages are ACK and SYSTEM.

Example. Start the identification process trying to assign ID 50

```
i 50
- ok
```

Command ACK - Identification Acknowledge

SYNOPSIS
a

Upon receiving an ACK message, the board exits the Identification Mode (is enabled). If the board was already in Normal Mode, the message has no effect.

Example. Stop the identification process by sending a message to the Master

```
a
- ok
```

Example. Message reported by the Master via USB when an ID is accepted by any of the boards during an Identification process

```
a
```

Command GET - Read a Register

SYNOPSIS

r REG

r REG FORMAT

Read the contents of the specified register *REG*.

If the register is numeric and FORMAT is specified, a conversion is applied according to the following table

FORMAT	RESULT
None	Physical units
d	Decimal output
x X h $	Hexadecimal output

Example. Read the frequency register (# 50) of a 1 MHz generator in native units (MHz)

```
r 50
- 1.000000047
```

Example. Read the frequency register (# 50) of a 1 MHz generator in tuning word units (decimal)

```
r 50 d
- 8589935
```

Example. Read the frequency register (# 50) of a 1 MHz generator in tuning word units (hexadecimal)

```
r 50 x
- 0083126F
```

Command REPLY - Following Data are a Reply

SYNOPSIS	
	- data

Informs that the following data are a Reply to a previous Message.

Example. Read the board name

```
r 20
- Board Name
```

Command REMARKS - Following Data can be discarded

SYNOPSIS	
	# data

Informs that the following data are remarks (mainly used for debugging purposes).

Example. Debug the SPI data transfer with a DDS when reading the channel 0 frequency

```
r 50
# spi(2 bytes) Tx: 00 12 Rx:  00 00
# spi(5 bytes) Tx: 84 00 00 00 00 Rx:  00 00 83 12 6F
- 1.000000047
```

Command SET - Write a Register

SYNOPSIS	
	w REG VALUE
	w REG FORMAT VALUE

Write the value VALUE into register REG.

If a FORMAT prefix is prepended (without spaces) to the VALUE, a conversion is applied according to the following table

FORMAT	RESULT
None	Physical units
d	Decimal input
x X h $ 0x	Hexadecimal input

Example. Set the frequency register (# 50) of a generator to 1 MHz

```
w 50 1
- ok
```

Example. Set the frequency register (# 50) of a generator to decimal 8589935 (tuning word)

```
w 50 d8589935
- ok
```

Example. Set the frequency register (# 50) of a generator to hexadecimal 0083126F (tuning word)

```
w 50 0x0083126F
- ok
```

Board internals

Detecting the Board Reset type

The default bootloader installed in the Arduino boards simplifies the programming process through the USB connection to the external development workstation. Figure 23 and Figure 24 show part of the schematic of an Arduino Nano and Arduino Uno Boards.

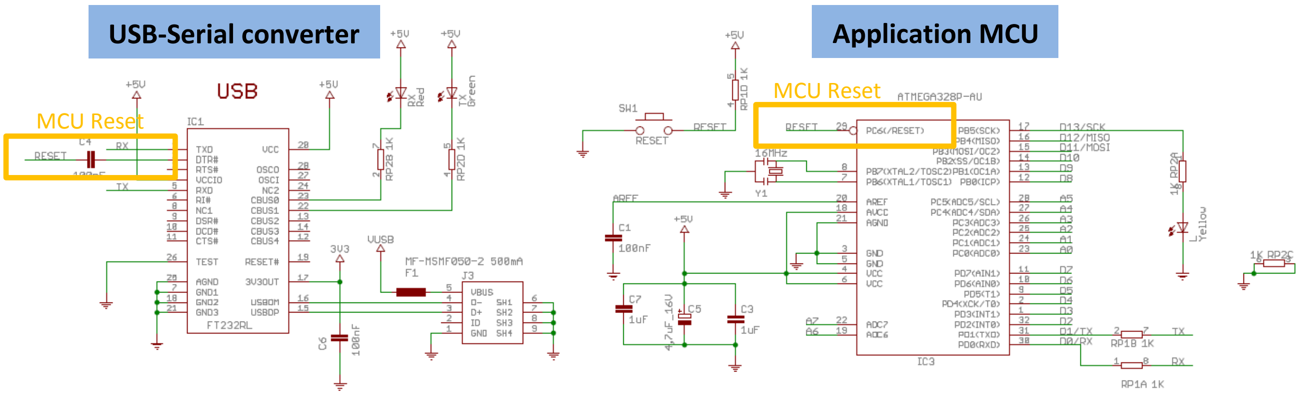


FIGURE 23 - ARDUINO NANO SCHEMATICS

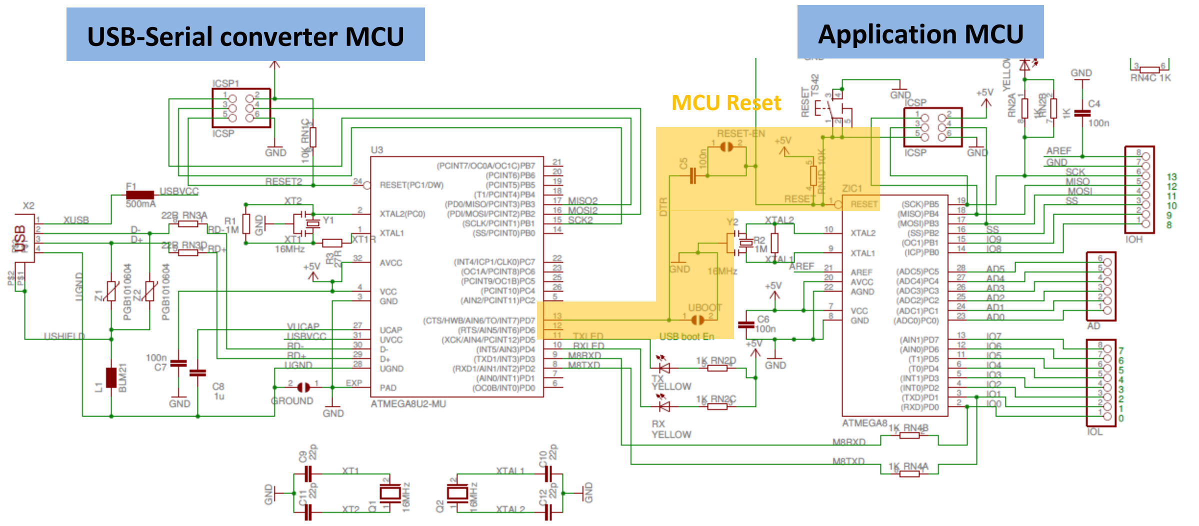


FIGURE 24 - ARDUINO UNO SCHEMATICS

In both cases, the MCU running the user software (in this case the Base Board firmware) can be reset by the USB-Serial converter when data are received from the connected workstation. While this is good in order to ease the code-program-debug lifecycle, on the other hand can be annoying because the MCU gets reset every time the USB connector is plugged or, even worse when the external software is executed on the USB connected pc.

In a situation where an external hardware must be initialized only at power-up (for example a programmable power supply or a continuous signal generator), this problem cannot be tolerated[2]. A hardware modification of the Arduino board can solve the problem for the Arduino Nano or a software modification of the code in the auxiliary MCU of the Arduino Uno which acts as a USB-Serial converter but these approaches go against the design philosophy of using only standard Arduino boards. Thus a software solution has been found to detect the type of reset. The Board firmware is restarted anyway but it can perform different types of initialization depending on the reset type: Cold Boot (upon first Power-Up) or Warm Boot (after a USB connection or a new data exchange session).

The solution is making the firmware check for the presence of a signature in a specific region of RAM memory at reset. Finding the signature means a Warm Boot. Otherwise if the match is not found, the reset is due to a Cold Boot and the signature is written in the memory region.

```
// DETECT WARM / COLD BOOTSTRAP
        if (strcmp(BootstrapSignature, COLD_BOOT_MAGIC_STRING) == 0)
                WarmBoot = true;
        else {
                strprintf(BootstrapSignature, "%s", COLD_BOOT_MAGIC_STRING);
                WarmBoot = false;
        }
```

As all the statically declared variables are automatically initialized at reset by default, the memory region containing the signature must belong to a special segment named `.noinit` . This is done with this snippet of code in the `classboard.cpp` source.

```
C_STRING BootstrapSignature __attribute__ ((section (".noinit"))); // ALLOCATE IN NON-INITIALIZED DATA SEGMENT
```

So, in the remaining part of the firmware, the `WarmBoot` property of the Board class, can be used to perform different operations at reset

```
if (WarmBoot) { // Normal operations }
else {
        // 1st time initialization
        Recall(); // Recall the configuration of external hardware
        }
```

[2] For example, when In a Windows7 64 bit is connected, after powering-up the PC the Board is reset three times: at powerup, at the Windows login screen, after the user login. Attaching and detaching a USB peripheral to the PC, reset the Arduino board as well.

Description of the Base Set Registers

The Board registers are grouped in three sets

General Registers

Register 0: STO_EEPROM_FORMAT - Format of registers and data in e2prom

This register contains the code of the data format which is stored in EEPROM by the firmware. It should be used to track data storing at different addresses or format across different firmware releases.

Register 1: STO_BOARD_ID - I²C Address of Board

This register contains the address of the Board used for I^{2}C communications.

Register 8: REG_BOARD_MASTER – Master Board I²C Address

This is the I^{2}C address of the Master Board and it is stored in the Slave which is addressed through the FORWARD message. The Slave sends to the Master Board the replies to messages.

Register 18: REG_BOARD_PARAM_STATE - Parameters' state, used to detect changes

This is a 32 bit-long register, organized in 4 counters. Each counter is associated to a group of registers of the Board according to a firmware-defined priority policy. When a register belonging to one of the 4 groups is changed, the corresponding counter is incremented as well.

In an environment where the Master Board is controlled by many concurrent applications, polling the REG_BOARD_PARAM_STATE allows an application to detect registers changes, in this case the registers set can be reloaded.

The four groups are defined as

```
#define PARAM_GROUP_PRIORITY_HIGHEST     0 // WEIGHT 0x00000001 (1 decimal)
#define PARAM_GROUP_PRIORITY_HIGH        1 // WEIGHT 0x00000100 (256 decimal)
#define PARAM_GROUP_PRIORITY_LOW         2 // WEIGHT 0x00010000 (decimal 65536)
#define PARAM_GROUP_PRIORITY_LOWEST      3 // WEIGHT 0x01000000 (decimal 16777216)
```

each having a different weight in the REG_BOARD_PARAM_STATE register.

Example. Register 50 belongs to the priority group PARAM_GROUP_PRIORITY_HIGH. The register REG_BOARD_PARAM_STATE is incremented by 256 every time register 50 is written.

```
r 18
- 0
w 50 1
- ok
r 18
- 256
```

Register 19: STO_BOARD_RESET_MODE - Reset mode

The register can be used to specify special operations which need to be executed upon reset (for example different types of initialization of an external hardware).

The base board currently does not use this register.

Register 20: STO_BOARD_NAME - Board name
This register contains the ASCII representation of the board name/description. The maximum length of the name is 32 characters.

Register 21: REG_BOARD_ADC6 - ADC CH 6 Value
Reading this register, yields the 10-bit ADC value read by the Arduino Board on channel 6.

Register 22: REG_BOARD_ADC7 - ADC CH 7 Value
Reading this register, yields the 10-bit ADC value read by the Arduino Board on channel 7.

Register 23: REG_BOARD_D2 - Arduino D2 pin
This register is reserved for future expansion of the Board functions, for example reading/writing a value to the Arduino Board digital pin D2.

Access to this register currently has no effect.

Register 24: REG_BOARD_D3 - Arduino D3 pin
This register is reserved for future expansion of the Board functions, for example reading/writing a value to the Arduino Board digital pin D3.

Access to this register currently has no effect.

Register 25: REG_BOARD_D4 - Arduino D4 pin
This register is reserved for future expansion of the Board functions, for example reading/writing a value to the Arduino Board digital pin D4.

Access to this register currently has no effect.

Firmware information
The Firmware Information Group registers, are read only registers and can be used to obtain information on the installed firmware version.

Register 2: REG_FW_DRIVER – Firmware, name of driver class
This register informs the external user application about the Type of a connected Board. The idea is to structure the user application into two layers, the first layer depends on the connected hardware and provides board drivers, the second layer accesses the board though the drivers. A sample of Python code to support DDS boards in a system of I^{2}C interconnected boards a can be found in (6).

Register 3: REG_FW_NAME - Firmware name
This register contains the name of the main source file of the firmware. For example `fw_dds_1_0_small.cpp`

Register 4: REG_FW_VER - Firmware version
This register contains the release information of the firmware. Example `1.0 DEBUG 1`

Register 5: REG_FW_BUILD - Firmware build date
This register contains the date and time of the firmware build. Example `Dec 18 2017 11:47:22`

Debugging and Special Registers

Register 6: REG_EEPROM_ADDRESS - EEPROM Access Address (Autoincrement)

This registers sets the pointer to the EEPROM memory for direct access of stored parameters.

Each access to the EEPROM data automatically increments the address by one.

Register 7: REG_EEPROM_DATA - EEPROM Access Data R/W

This register allows a user application to access the EEPROM data, for example for backup/restore operations of the board parameters or for data inspection during a debug session.

The memory contents pointed by the REG_EEPROM_ADDRESS register can be read or written by getting or setting the REG_EEPROM_DATA register. Data are 8-bit organized.

Each access to the EEPROM data automatically increments the address by one.

Register 9: REG_DBG_RAMDATA - RAM Data

This register allows a user application to access the RAM memory data during a debug session.

The memory contents pointed by the REG_DBG_RAMADDRESS register can be read or written by getting or setting the REG_DBG_RAMDATA register. Data are 8-bit organized.

Each access to the RAM data automatically increments the address by one.

Register 10: REG_DBG_RAMADDRESS - RAM Address

This registers sets the memory pointer for RAM direct access.

Each access to the RAM data automatically increments the address by one.

Register 11: STO_DBG_LEVEL - Set debug verbosity

This register sets the verbosity level of debug information. The register values contents are checked by the debug macro defined in `utils.h` .

Example. Debug of a Board GET operation.

The macro _D_ is inserted in the "GET" callback function with the required debuglevel parameter set to 10.

```
void cb_GET() { // get internal variable         // Get
        C_STRING tmp;
        bool retval = Board->Get(tmp, Board->Msg->Parameter, Board->Msg->Value);
        _D_(10, "This is a test")
        Board->SendReply(retval ? tmp : PROTOCOL_FAIL);
}
```

Performing GET operations with different debug verbosities has these effects

```
w 11 9
- ok
r 11
- 9
w 11 10
- ok
- r 11
This is a test
- 10
r 20
This is a test
- The board name
```

Register 12: REG_DBG_INFO - Free memory and board status

This register shows the contents of the Stack Pointer, Heap Pointer and free RAM memory.

Register 13: REG_DBG_SUPPORTED - Debug has been enabled in firmware

This register shows whether the debug information / macros have been enabled in the firmware.

Register 14: REG_DBG_LASTBOOT - Last board restart

Reading this registers shows the time in milliseconds elapsed since the last Arduino reset (both Warm and Cold boot).

Debug tool Example: Memory leakage in ClassMessage::Parse()

During the development of the firmware instability was observed and the software used to stop responding after some command executions. The number of messages triggering the undesired behavior was random.

As the Arduino and Eclipse lack a debugging environment, some trace tools have been defined and inserted into the firmware to track memory during code execution. Adding some debug macros in `messages.cpp` helped to find the programming error which lead to memory leakage.

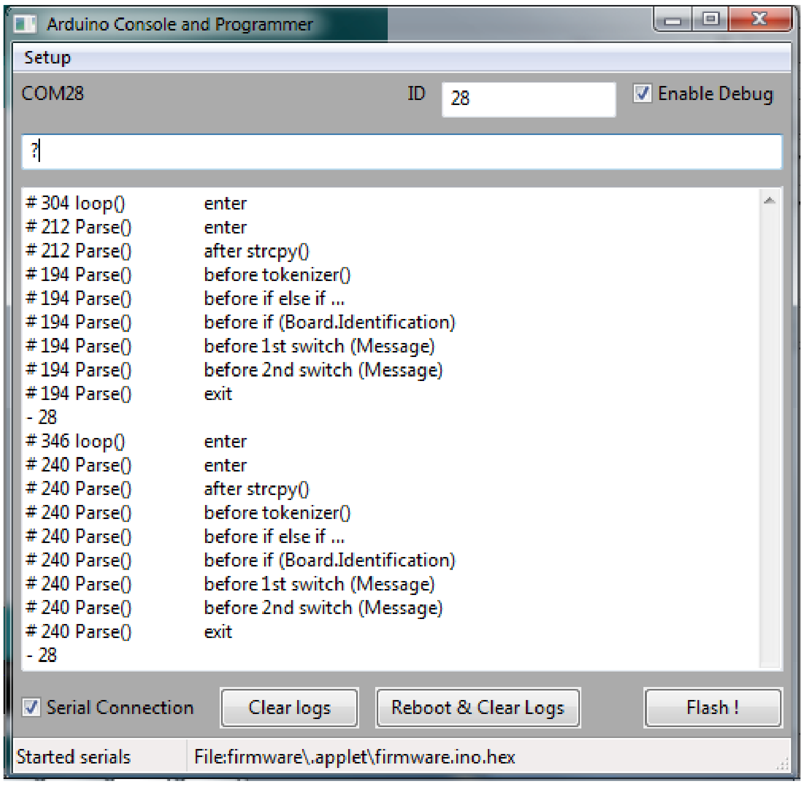
```
  _TRACE_(1, "enter")

  Bus            = source;
  CanExecute     = false;
  Message = MSG_UNKNOWN;
  BoardID = 0;
  Register = 0;
  Value[0] = (char) 0;
  strcpy(string, inputString);
  _ TRACE_(1, "after strcpy()")

  char *tok  = new char[16];

  _ TRACE_(1, "before tokenizer()")
  inputString = tokenizer(inputString, tok, ' ');

  _ TRACE_(1, "before if else if ... ")
...
```

Then a simple yet flexible tool was written in Python / wxPython (9) to ease the debug operation and to allow faster reprogramming with respect to the two mentioned IDEs.

The command WHO ("?") was sent many times to the board and it showed that the memory usage was increasing at each command execution.

By tracking the information, a bug was discovered in the source file and the memory leakage was due to the variable `tok` which was allocated in the heap and not released at the end of the function call.

Using a local variable
`char tok[16];`
solved the problem.


Arduino Console and Programmer

Setup

COM28 ID 28 ☑ Enable Debug

?

```
# 304 loop()        enter
# 212 Parse()       enter
# 212 Parse()       after strcpy()
# 194 Parse()       before tokenizer()
# 194 Parse()       before if else if ...
# 194 Parse()       before if (Board.Identification)
# 194 Parse()       before 1st switch (Message)
# 194 Parse()       before 2nd switch (Message)
# 194 Parse()       exit
- 28
# 346 loop()        enter
# 240 Parse()       enter
# 240 Parse()       after strcpy()
# 240 Parse()       before tokenizer()
# 240 Parse()       before if else if ...
# 240 Parse()       before if (Board.Identification)
# 240 Parse()       before 1st switch (Message)
# 240 Parse()       before 2nd switch (Message)
# 240 Parse()       exit
- 28
```

☑ Serial Connection | Clear logs | Reboot & Clear Logs | Flash !

Started serials | File:firmware\.applet\firmware.ino.hex

Conclusions

Arduino is a low-cost, low-consumption, simple yet flexible platform for small-medium size applications where an electronic device need to be remotely controlled. For other applications, the low pin count, low memory and MCU speed of the board can limit the complexity of the application and of the devices which can be connected.

Nevertheless, if needed the firmware described in this report can be ported to other platforms providing better performances. Research on this topic is in progress in order to develop other scalable applications.

Bibliography

1. **Arduino.** Arduino Nano. [Online] Arduino, 2017. https://store.arduino.cc/arduino-nano.

2. **Atmel Corporation / Microchip.** ATmega328P. *Microchip / Atmel Corporation.* [Online] 2017. http://www.microchip.com/wwwproducts/en/atmega328p.

3. **Python Software Foundation.** Python Homepage. *Python Software Foundation Website.* [Online] 2017. http://www.python.org.

4. **Francese, Claudio.** *Flexible Arduino-Based board - Firmware Extension for DDS.* INRIM. 2017. Technical Report. TR 25/2017.

5. —. *Templating and Automatic Code Generation.* INRiM. 2017. Tecnical Report. TR 27/2017.

6. —. *Flexible Arduino Board - Mid-Tier and Application Software.* INRiM. 2017. Technical Report. TR 26/2017.

7. **com0com.** Null Modem Emulator. [Online] http://com0com.sourceforge.net/.

8. **I2C Bus.** I2C Bus Organization. *I2C Bus Organization.* [Online] 2017. https://www.i2c-bus.org.

9. **The wxPython Team.** WxPython Homepage. [Online] 2017. http://www.wxpython.org.