stDAQ Reference Manual - r.1.0 (Antille)

S.Furlan

July 4th, 2021

Contents

1 stDAQ - Data Acquisition System with ST-NUCLEO

"Let's be quantitative" - Ray.E.G. (DAMTP Prof., Univ. of Cambridge, UK)

The only way to recognize progress is by measuring it. Whether it is a research project, a new design for experiment for an industrial production system or your latest DIY hobby, there is always the need of a handy tool that can collect, process and save the data logs on your PC. This tool is a data acquisition system or alternatively referred as DAQ. DAQ suppliers like National Instruments recognized this need and built LabView, a specilized interface to virtualize and program their proprietary mixed-signals acquisition board. Though, both the interface and the board come with a certain cost for the license that may not justify a commitment for your initial startup budget. Luckly, the recent introduction of mixed-signal computing platforms powered by ARM, like the Raspberry PI and many high-performance microcontrollers, made the need of a DAQ more affordable and customizable. Yet still remains the assumption that you are a skilled programmer mastering all the details involved in the setting of the many registers in a ARM processor.

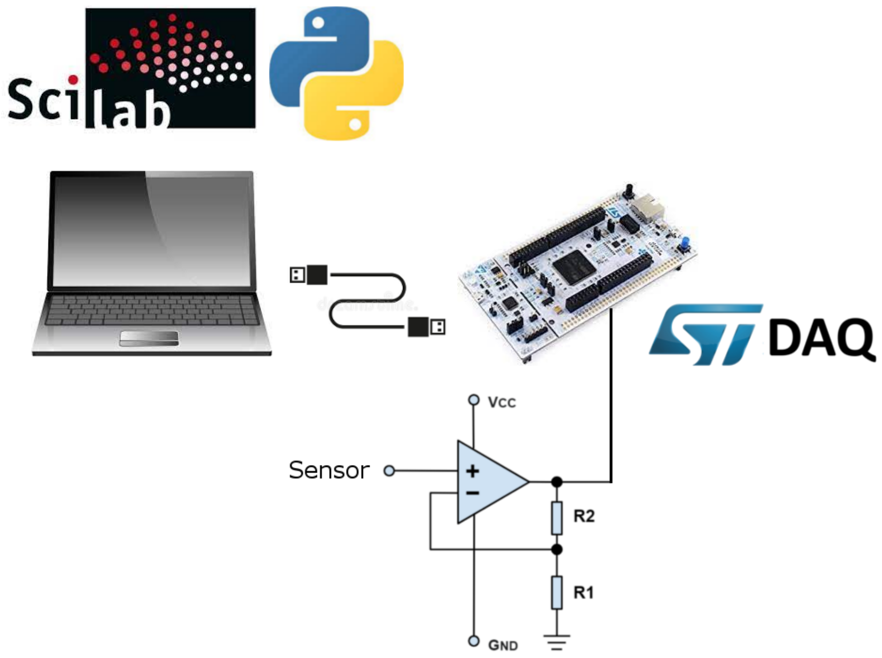

Figure 1: stDAQ data acquisition system setup.

In this scenario the stDAQ project sparked from the personal quest of the author to provide to the general public a data acquisition system that is: i. affordable and scalable (less than 20$ for each ST-NUCLEO board and freeware), ii. easy to setup and program (using well-known data processing environment like Scilab and Python), iii. performing and reliable (you need to trust the author and the many tests performed and reported in this manual).

The stDAQ data acquisition system runs on a ST-NUCLEO board, connected to laptop PC via USB, see figure 1. The ST-NUCLEO board runs the stDAQ firmware which setups up the board as a mixed-signals front-end, while the laptop PC is used as a programming interface to the stDAQ, either running through a Scilab or a Python environment. Setup and installation details are explained later. On the other side of the ST-NUCLEO, you can plug in any output of your dedicated signal conditioning circuit or sensor front-end.

1.1 Specifications

stDAQ is designed to support a selected number of peripherals from the ones available on the microcontroller mounted on the board. Currently, only the NUCLEO-F413ZH can support stDAQ. We have provision to extended the project soon to all the ST-NUCLEO F4 family through compatibility. Additional features and peripharals will be also addressed in new releases.

Table 1: stDAQ support and compatibility.

ST-NUCLEO	NUCLEO-F413ZH
Operating Systems	Windows 8.1 and newer
Environments	Scilab 6.0.1/6.0.2, Python 3.4

In the next table are resumed the main features for each peripheral.

Peripherals	Feature	Min.	Typ.	Max.
ADC	resolution		12 bits	
	MUX channels			14
	channel sampling frequency			1 MHz
	inter-sequence sampling frequency			1 KHz
DAC	resolution		12 bits	
	MUX channels			1
	DAC update latency		1 msec.	
GPIO	input pins			4
	output pins			8
	IO update latency		2 msec.	
	IO voltage (TTL/CMOS)	0V		+3.3V
TIMER				
PWM				
I2C				

1.2 Pinout

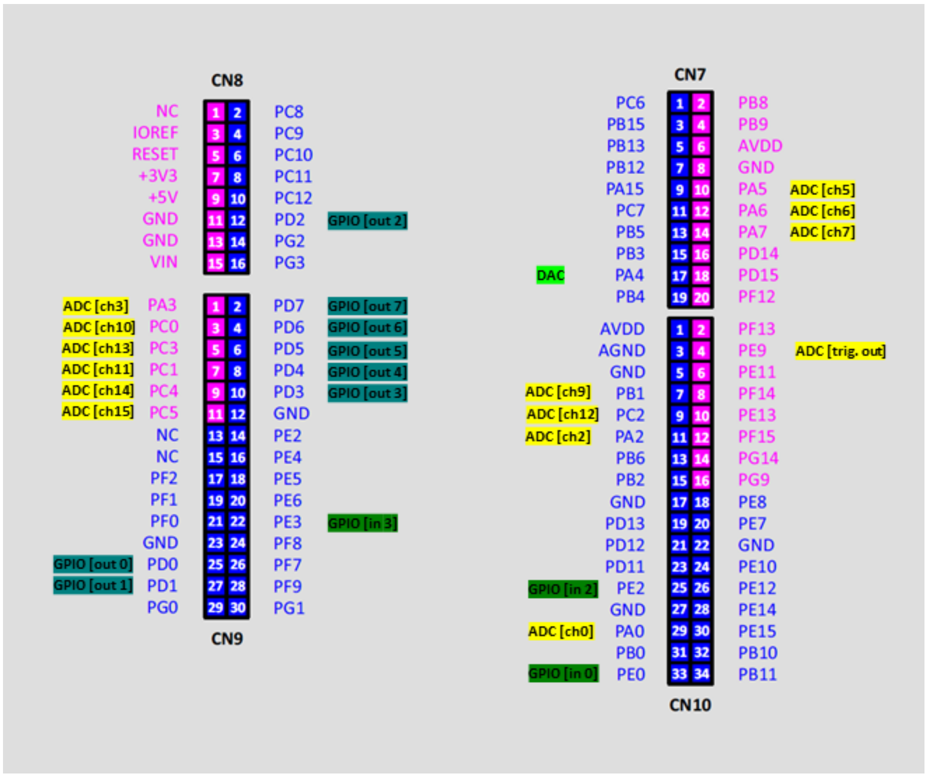

Figure 2: Pinout for the Arduino and Zio extension of the NUCLEO-F413ZH.

2 Peripheral interfaces

stDAQ provides the access to a selection of peripherals of the STM32F413ZH micro-controller through a programming interface. The scope of the stDAQ is to keep the programming interface as simple as possible, without sacrificing on performance, so to provide to the user a flexible programming environment to quickly setup its next data acquisition experiment. The supported programming environments are either the Scilab console or Python. In both cases, the peripheral programming interfaces are linked to the environments with a dedicated library, as explained in the Setup & Installation section.
The access to the stDAQ system is obtained by opening the USB communication on a virtual COM port, execute a sequence of setup commands, run the acquisition and close the communication.
In the following sections, the programming functions related to each peripherals are highlighted in red, and usage examples are described along with their timing performance.

2.1 DEVICE

2.1.1 stdaq_open()

The stdaq_open() function opens the communication on the selected COM port.
The prototype for the function call is:

```
res = stdaq_open(port)
```

The entries to the functions are:

[port (IN)] is a string that specifies the input COM port, i.e.: ”COM1”

[res (OUT)] returns the status of the opening: 0 if successfull, -1 if not successfull.

A function call example in Scilab:

```
--> res = stdaq_open("COM0");
```

2.1.2 stdaq_close()

The stdaq_close() function closes the communication on the selected COM port.
The prototype for the function call is:

```
stdaq_close()
```

Currently this function does not require any entry and does not have returns. It just closes the active COM port.

2.1.3 stdaq_version()

The stdaq_version() function gets the current version of the stDAQ firmware programmed on the ST-NUCLEO.
The prototype for the function call is:

```
[package,release,subrelease] = stdaq_version()
```

The entries to the functions are:

[none (IN)]

[res (OUT)] returns a list with both the package name and the release number of the stDAQ firmware.

A function call example in Scilab:

```
--> if (stdaq_open("COM0")>=0) then
    ... [pkg,rel,sub] = stdaq_version();
    ... mprintf("\n stDAQ version: (%s) r.%d.%d",pkg,rel,sub);
    ... stdaq_close();
    end
```

2.2 DAC

The stDAQ supports a single 12 bits DAC, with output on pin PA4.
The programming of the DAC is obtained with the following functions:

2.2.1 stdaq_set_dac()

The stdaq_set_dac() function sets the value in the DAC register used to set the analog voltage output on the pin PA4. To physically set the analog output voltage on the pin PA4 the DAC must be enabled (see stdaq_dac_enable()).
The prototype for the function call is:

```
stdaq_set_dac(value)
```

The entries to the functions are:

[value (IN)] is an integer value in the range of [0-4095]. The output voltage on the pin corresponds to the scaling of the value by $Vout = value/4096 * 3.3V$.

[none (OUT)]

A function call example in Scilab:

```
--> stdaq_set_dac(2048);
```

this will put on the output about 1.65V, once the DAC is enabled.

2.2.2 stdaq_enable_dac()

The stdaq_enable_dac() function enables the analog output voltage on the pin PA4. If DAC is enabled before setting its value in the register, the default output is 0V.
The prototype for the function call is:

```
stdaq_enable_dac()
```

Currently this function does not require any entry and does not have returns. It just enables the output analog voltage of the DAC.

2.2.3 stdaq_disable_dac()

The stdaq_disable_dac() function disables the analog output voltage on the pin PA4.
The prototype for the function call is:

```
stdaq_disable_dac()
```

Currently this function does not require any entry and does not have returns.

2.2.4 Examples

As an example we will generate a periodic function in Scilab, command the DAC through increments and analyze its performance.

First we generate a triangular wave spanning from 0 to 3.3V with an incremental step of 12, equivalent to a step of about 10 mV.

```
--> stdaq_set_dac(4095);
--> stdaq_enable_dac();
--> step = 12; taps = 341; periods = 10;
--> for i=1:(2*taps*periods)
    ... value = abs(step*(pmodulo(i,2*taps)-taps));
    ... stdaq_set_dac(value);
    end
```

This is recommended since the USB communication has a delay of about 1 msec. Longer sleeps can be performed, but shorter ones perform an unpredicatable update.

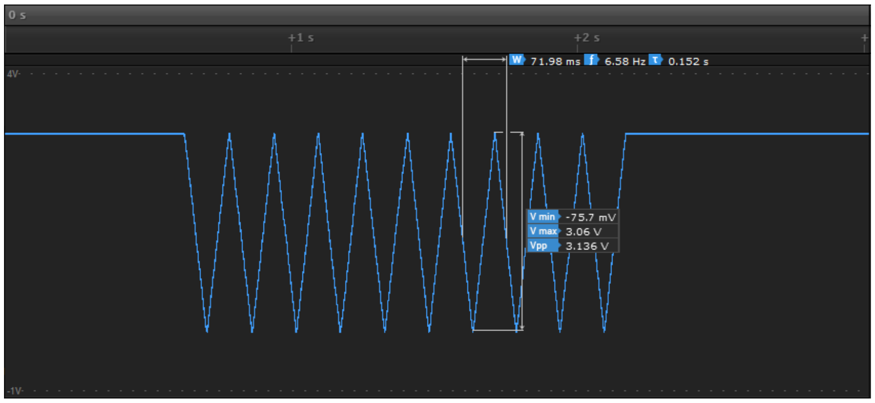

Figure 3: Triangular wave measured on PA4.

2.3 ADC

The stDAQ supports a single 12 bits ADC, muxed over 14 input channels and a temperature sensor. In the following table there is the correspondence between channel and pin on the ST NUCLEO-F413ZH. Channel 16 corresponds to the temperature sensor. Note that channel 4 and channel 8 are missing because the pins are dedicated to other functions.

The programming of the ADC is obtained with the following functions:

Table 2: ST NUCLEO-F413ZH ADC channels pinout.

PIN	**CHANNEL**	**PIN**	**CHANNEL**
PA0	channel 0	PB1	channel 9
PA1	channel 1	PC0	channel 10
PA2	channel 2	PC1	channel 11
PA3	channel 3	PC2	channel 12
PA5	channel 5	PC3	channel 13
PA6	channel 6	PC4	channel 14
PA7	channel 7	PC5	channel 15

2.3.1 stdaq_set_adc()

stdaq_set_adc() sets the ADC channels required for the acquisition.
The prototype for the function call is:

```
stdaq_set_adc(channelsequence, clockdivision)
```

The entries to the functions are:

[channelsequence (IN)] is an array of max. 16 entries with values ranging from [0-16] with the exclusion of 4 and 8, which are channels not available. Repetition of the same value is possible inside the channelsequence.

[clockdivision (IN)] is a value in range [0-9] corresponding to a specific clock frequency, which trigges the acquisition of the next channel inside the channelsequence. Refers to the following table for the corresponding frequencies.

[none (OUT)]

Table 3: ST NUCLEO-F413ZH clockdivision frequencies used to trigger the acquisition.

clockdivision	frequency	clockdivision	frequency
0	1 MHz	5	31.25 KHz
1	500 KHz	6	15.625 KHz
2	250 KHz	7	7.8125 KHz
3	125 KHz	8	3.9062 KHz
4	62.5 KHz	9	1.9531 KHz

A function call example in Scilab:

```
--> chseq = [0,5,6,6,16]; // [ch0, ch5, ch6, ch6, temp_sensor]
--> clkdiv = 5; // 31.25 KHz
--> stdaq_set_adc(chseq,clkdiv);
```

2.3.2 stdaq_get_adc()

The prototype for the function call is:

```
samples = stdaq_get_adc(channelsequence, numsamplesperchannel)
```

The entries to the functions are:

[channelsequence (IN)] is an array of max. 16 entries with values ranging from [0-16] with the exclusion of 4 and 8, which are channels not available. Repetition of the same value is possible inside the channelsequence.

[numsamplesperchannel (IN)] is the number of samples per channel returned by the acquisition;

[samples (OUT)] returns a matrix of [numchannels x numsamplesperchannel] with the 12 bits ADC values scaled to 0-3.3 V. For the temperature sensor, the value is already returned scaled in Celsius.

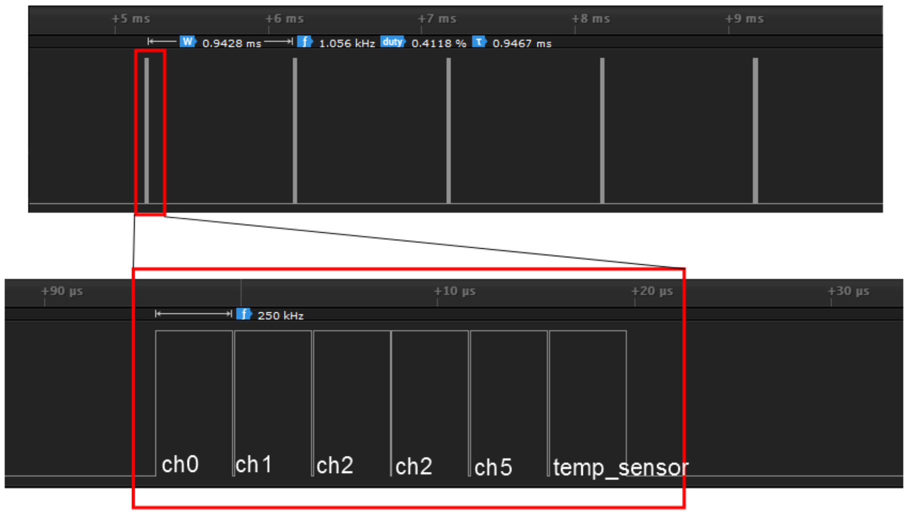

Figure 4: Acquisition timing measured on PE9.

A function call example in Scilab:

```
--> chseq = [0,1,2,2,5,16]; // [ch0, ch1, ch2, ch2, ch5, temp_sensor]
--> clkdiv = 2; // 250 KHz
--> stdaq_set_adc(chseq,clkdiv);
--> samples = stdaq_get_adc(chseq,30);
```

As seen in figure 4 top, the inter-sequence sampling frequency can reach values up to 1 KHz, where the limit is mainly due by the USB communication to request for new data. In the bottom of the same figure, the channel sampling frequency is measured on PE9 as the 250 KHz set in the clockdivision entry. Note that pin PE9 is the trigger to the acquisition of a new channel in the selected sequence list, where the acquisition trigger is set as rising edge.

2.3.3 Examples

As an example we will use the DAC to generate a sinusoid function, command the DAC output, record with the ADC channel 0 and display the result in Scilab. It is required to connect with a jumper wire the output of the DAC (pin PA4) with the input of the channel 0 of the ADC (pin PA0).

```
--> chseq = [0]; // [ch0]
--> clkdiv = 0; // 1 MHz
--> stdaq_set_adc(chseq,clkdiv);
--> stdaq_set_dac(0);
--> stdaq_enable_dac();
--> tapsperperiod = 20; periods = 10; out = [];
--> for i=1:(tapsperperiod*periods)
    ... value = 2048*(1 + sin(2*%pi*i/tapsperperiod));
    ... stdaq_set_dac(value);
    ... samples = stdaq_get_adc(chseq,1);
    ... out = [out, samples];
    end
--> figure; plot(1:length(out),out);
```

2.4 GPIO

The GPIOs of the ST-NUCLEO are configured both as digital output (8 pins) and as digital input (4 pins). A list of pins are referenced in the following tables. The digital outputs are set in push-pull mode.

Table 4: ST NUCLEO-F413ZH GPIO output.

PIN	OUTPUT	PIN	OUTPUT
PD0	0	PD4	4
PD1	1	PD5	5
PD2	2	PD6	6
PD3	3	PD7	7

2.4.1 stdaq_get_gpio()

This function reads the specified input port pin and returns its value to be either set (3.3V) or reset (0V). The prototype for the function call is:

Table 5: ST NUCLEO-F413ZH GPIO input.

PIN	INPUT
PE0	0
PE1	1
PE2	2
PE3	3

```
value = stdaq_get_gpio(pin)
```

The entries to the functions are:

[pin (IN)] output number corresponding to the pin in the GPIO output table;

[value (OUT)] returns binary value defined as set = 1 or reset = 0;

A function call example in Scilab:

```
--> pin = 1; // PE1
--> value = stdaq_get_gpio(pin);
```

2.4.2 stdaq_set_gpio()

This function sets or clears the selected port bit. The prototype for the function call is:

```
stdaq_set_gpio(pin, value)
```

The entries to the functions are:

[pin (IN)] output number corresponding to the pin in the GPIO output table;

[value (IN)] binary value to define if the output is set = 1 or reset = 0;

[none (OUT)]

A function call example in Scilab:

```
--> pin = 0; // PD0
--> value = 1; // set to 3.3V
--> stdaq_set_gpio(pin,value);
```

2.4.3 stdaq_toggle_gpio()

The prototype for the function call is:

```
stdaq_toggle_gpio(pin)
```

The entries to the functions are:

[pin (IN)] output number corresponding to the pin in the GPIO output table;

[none (OUT)]

2.4.4 Examples

As an example we will toggle the pin PD0 at each iteration of a loop cycle, so to evaluate the frequency response and the average latency of the digital output.

```
--> tags = 100;
--> for i=1:tags
    ... stdaq_toggle_gpio(0);
    ... sleep(1);
    end
```

The $sleep(1)$ routine is added in the loop to delay the PC loop of 1 msec. The toggling of the GPIO is in fact limited by the USB refresh rate, which occurs at about 1 msec.. The max. total toggling rate is measured to be 250 Hz, see figure 5. It is observed that some slips of ± 1 msec happens with a rate of 2% of the toggling count. For values of sleeps lower than 1 msec., the toggling is not more reliable and toggling loss may occurs.

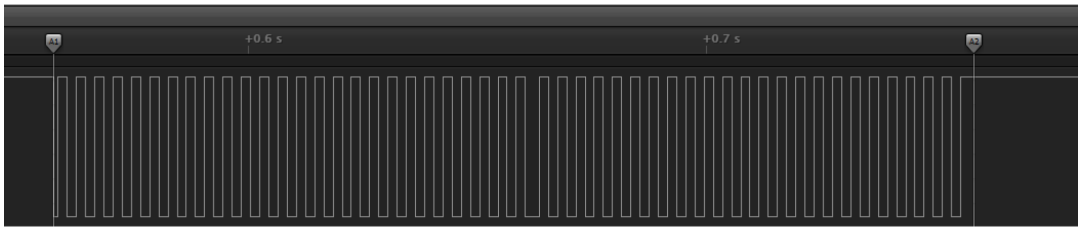

Figure 5: Digital pin toggling measured on PD0.

2.5 LED

The ST-NUCLEO board mounts three color LEDs, typically connected to the pins PB0 (green), PB7 (blue), PB14 (red).

2.5.1 stdaq_toggle_led()

This function toggles one of the three LEDs mounted on the ST-NUCLEO board. The prototype for the function call is:

```
stdaq_toggle_led(color)
```

The entries to the functions are:

[color (IN)] character corresponding to the color of the LED to be toggled, which can be either 'r' (red), 'g' (green), or 'b' (blue).

[none (OUT)]

A function call example in Scilab:

```
--> for i=1:10
    ... stdaq_toggle_led('r');
    ... sleep(500);
    end
```

This example toggles the red LED each 0.5 seconds for 10 times.

2.6 TIMER

2.7 PWM

2.8 I2C

3 Setup & Installation

To operate the ST-NUCLEO as a stDAQ and communicate with your Scilab environment running on your laptop, we recommend the following installation steps:

1. Download and install the latest Scilab freeware from the website.
2. Download and install the ST-Link Utility from the ST website.
3. Plug in the USB PWR of the ST-NUCLEO board to your laptop and open the ST-Link Utility. Go in Target menu and Connect. If sucessfull, the device name is displayed and the memory of the microcontroller opened up in a table.
4. In the ST-Link Utility, open Program... in the Target menu, upload the stdaq_antille_r1s0.bin binary file available in the /nucleo folder of the project, and program the microcontroller.
5. Connect the User USB of the ST-NUCLEO board to your laptop and locate the corresponding COM port.
6. Open Scilab. In the file browser, navigate till the /scilab folder in the stDAQ project and execute runme.sci with the command

   ```
   --> exec('runme.sci',-1)
   ```

   Wait till Link done appears on the console. Now your system is ready to operate.
7. On Scilab console write the command

   ```
   --> stdaq_open('COM0')
   ```

   to start the communication between the stDAQ on the ST-NUCLEO and your Scilab environment.

Note that everytime you restart a new Scilab session, you will always need to execute the runme.sci unless you set it up in the environment variables of Scilab to auto-upload and link the stDAQ dynamic library.
After a power-down or reboot, there is no need to reflash the stDAQ firmware on the ST-NUCLEO. A new reprogramming with the ST-Link Utility is required only if a new firmware release need to be installed on the ST-NUCLEO.

Similarly to Scilab, the Python stDAQ library in the /python folder needs to be used to communicate with the ST-NUCLEO from your .py programming environment.

4 stDAQ license information