# Second example: reading time-stamped data

by Mauri Favaron

## Introduction

### Time stamps, and their importance in data logging

*Time stamp* is a date and time value, attached as a label to a data item and positioning it with respect to time.

Not all data need to be associated with a time stamp; for example, weight measurements collected during a population survey are usually not time stamped, the only data collected with time being in case the date when measurements were taken, maybe for billing purposes, or under the need to discriminate among different treatments.

There *do* exist cases, however, in which the precise knowledge of the date and time associated to a data item is of paramount importance. This is the case, for example, of the time series of individual temperature measurements: time here is very important, as it allows sorting measurements in a "before" and "after". It is this time to allow deciding, eventually, to which hourly or daily average does a specific individual data item contribute.

So time stamping is important, in many cases. Another interesting question is, what can you do after you got a time stamp. Here are a few uses:

- If you want to compute wide-span time statistics (for example hourly averages) starting from fine-grained "raw" data, the latter must be all time stamped so that you can easily understand what averaging period do they actually belong - and then the "arithmetical box" you may fit them in.
- Or, maybe, you have two instruments each providing its own observation set from a same phenomenon, or two among which some relation exists, or is to be investigated. In these cases you should make sure the two observation series refer to a unique time base. Or, maybe, to two distinct time bases, but whose reciprocal shift is known. Here too you need time stamps in both series, and a good sorting algorithm.
- Or, perhaps, you may want to analyse a long measurements series looking for seasonal or daily patterns (think a biorhythm, like the human "monthly" period, the sleep-awake alternation, the stomatal activity cycle of a plant, or the yearly change of the nocturnal average of heart beat rate, or the evolution of temperature along days on a month). If the cycle span is regular enough (which rules the human "monthly" period quite out) you may want to use a "typical period" in which you consider time-based classes basically as if they are categorical data, whose computation once again demands knowledge of time stamps.

These are just few examples, and you may yourself devise many more. The task is not difficult, and this lack of difficulty has very much to do with our shared passion for clocks, mechanical and virtual. We love to control time (or illude to), and so the same measurement "with time stamp" gains a big value compared to the same without.

## Types of time stamps

If the concept of time stamp itself looks trivial on the first instance (it is not, as we'll see soon), the objects it's attributed shows quite a considerable span.

A time stamp may, for example, be attributed to an event so concentrated in time to be safely considered an instant.

Or, it may be attributed to an interval as a whole.

The first case is conceptually simple (and technically tricky): all a data logger should do is to find out a time label from its RTC whenever it is needed, the more accurately possible.

The second, connected typically with relatively long averaging times, is conversely technically simple, but inherently ambiguous. The troubles originate from the fact an interval without gaps inside is delimited by *two* time stamps, let's say them $t_-$ and $t_+$ respectively, with $t_- < t_+$, with belonging times $t$ satisfying the relation $t_- \leq t < t_+$.

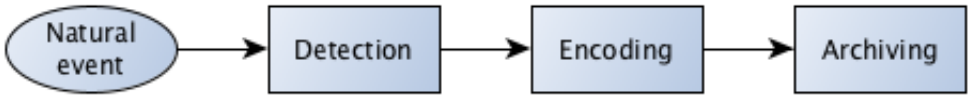
In practice, however, a *single* time stamp $t_0$ is used, and so the problem arises about when it does exactly occur in the interval. Is maybe $t_0 = t_-$? Or perhaps $t_0 = t_+$? $t_0 = \frac{t_- + t_+}{2}$? Schools of thoughts abound, and none of the possibilities (but maybe stamping both the beginning and the end of time interval) provides a definitive answer.

## The inherent difficulty in time-stamping instantaneous events

Assigning a data item an instantaneous time stamp may prove less trivial than one might expect on a first instance.

This is, because of the inherent (and mostly hidden) complexity of the data acquisition process. We'll see it, with reference to the simple case of labeling with a time stamp a threshold exceedance by temperature in a ideal climate chamber.

Here are the phases:




Let's assume the "natural event" is a temperature step change, occurring (say) when an operator pushes a button. As the climate chamber is "ideal", we can assume the step change occurs all of a sudden, immediately as the button is pushed. But as the operator has her/his own times and priority, as far as we're concerned the step change occurs "anywhen in time".

The first step of the data acquisition phase is "detecting" the change. This happens due to a sensor's *primary transducer* like a resistive thermometer whose resistance exceeds a threshold corresponding to the temperature we want to see crossed.

The sensor, however, behaves as a *first-order instrument*, that is, its response may be described by a first-order linear differential equation, which implies a finite *response time*. This in practice means the primary transducer's resistance will cross the threshold value somewhat *after* the real event. Because of this, the time stamp assigned to the threshold crossing event will be something like

$$T_s = T_0 + \varepsilon_1$$

where $T_0$ is the time instant at which the operator has pressed the pushbutton, and $\varepsilon_1$ is the response time of the primary transducer.

The event timing, at this time, has not yet become a real time stamp, but quite rather the possibility of it. For the resistance threshold crossing event to become a time stamp, it should be converted to a voltage and encoded as a number using an Analog-to-Digital Converter (ADC), and subsequently compared to the threshold and acknowledged larger by some computer program on board of the datalogger, and finally archived.

All these times contribute to the time stamp, whose expression may be then written as

$$T_s = T_0 + \varepsilon_1 + \varepsilon_2$$

So we see the attributed time stamp is an overestimate of the real instant in which the event occurred. The question is, whether the overestimation amount is to be treated as systematic, or as a random variable instead.

From a theoretical standpoint the answer looks simple: all phases involved are deterministic, and so their combination is too: provided enough details are available, the sum $\varepsilon_1 + \varepsilon_2$ could be computed exactly.

The real world is a bit nastier than so, however, and "collecting enough details" may prove extremely difficult even in so a simple case. For example, the sampling of electrical signals by the data logger analog to digital circuit occurs at fixed instants which bear no relationship with the original threshold-exceeding event.

Because of this, it makes all sense to treat the sum $\varepsilon_1 + \varepsilon_2$ as a random number, whose distribution and parameters may be in principle estimated if not measured directly.

Estimating the "latency time" of a time stamp is quite an art, and some advanced data acquisition system even have dedicated input channels to accomplish this requirement. Our very simple data reader is not that advanced, however, and so we must stay aware of the problem, and keen to anticipate its consequences.

*En passant*, I mention the fact that it is not in general possible to use an uniform time stamp for data collected using different loggers. This is not an important fact for our application, but in general is an issue, and countermeasures (e.g. by adopting a unique time base via an NTP server, or a GPS) tend to be non-trivial (read: expensive).

This problem may be mitigated during (offline) calculations, for example by "matching" time series stamps by using the "maximum cross-correlation lag" as a guide. This method is used quite frequently in data processing, and you may find extensive literature references.

But maybe the best mitigation method remains on the hardware level using *one* logger for all data, so to automatically ensure the time base is unique. This phrase may look sensible on a first instance, but has an inherent trouble built-in: the presence of a (non-real-time) multitasking operating system, whose intricate time dynamics may add a truly random component to time stamps.

# Dealing with time stamps, in actual terms

## Use constraints, and possible choices

We are striving to build a data *logger*, that is, a device which can collect a wealth of time stamped data (of temperatures in this example) unattended.

A data logger, as able as it is to operate unattended in non-obtrusive manner, should anyway have its time stamp initialized in some way, and then maintained through the whole data acquisition process.

For this to happen, a Real Time Clock ("RTC") unit should be available (in the SAMD21 processor of the AdaLogger I'm using this facility is already present), and some way must exist to set an appropriate initial state.

Now, let's imagine.

Our user unboxes the logger, power it, connect some programming device to set the RTC initial state - say, by assigning the date and time on the acquisition start - then disconnects it leaving the power connected, and leaves hoping the logger will do its assigned job, until the next visit or the next battery run-off.

The SAMD21 exposes two pins for supplying the RTC power even if the rest of the system (and of the SAMD21) is shut off. In a true consumer data logger it would be very important to connect those pins to a backup battery, typically a CR-2032 type. This way, it is sufficient to assign the RTC date and time once: if a power down occurs, the RTC remains powered and continues tracking time. Further user intervents will then be necessary only to adjust the RTC drift, if any visible.

In *practice*, this means we cannot exclude the user will power down the system, maybe transfer it to another location, and switch the power back on after some time. How cumbersome would it be, if on powering back on the system it would be necessary to re-assign the "initial" date and time, just because the logger RTC has forgotten to update the time, being it not connected to an independent power source.

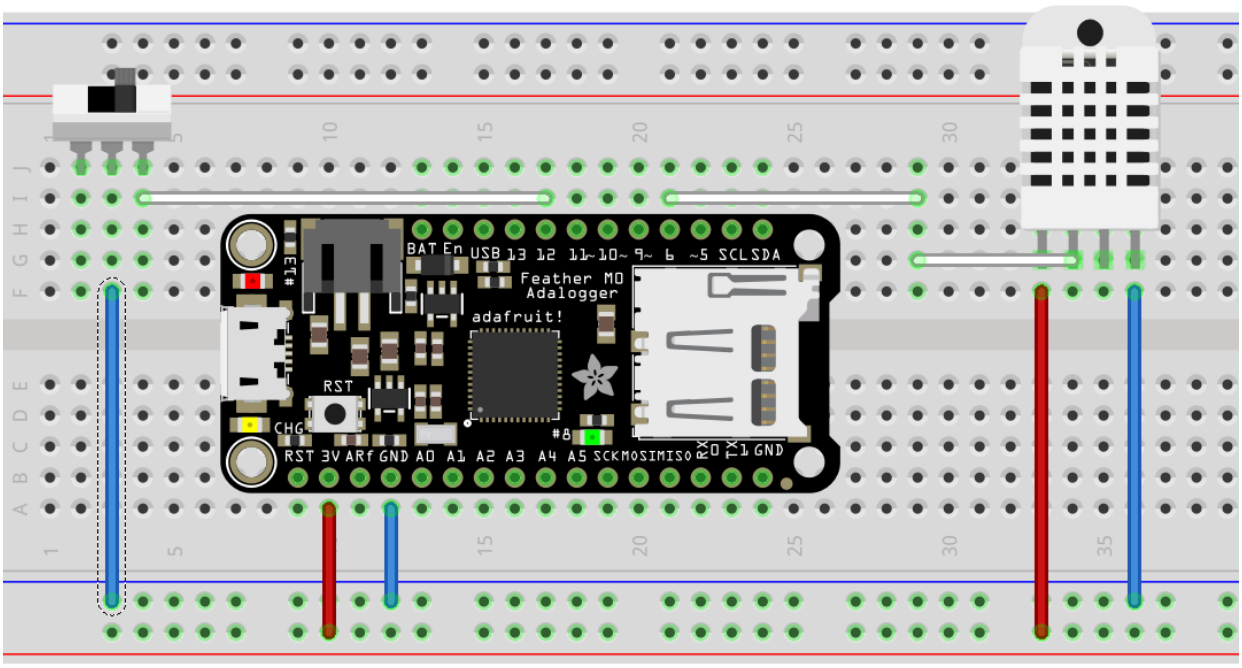
Sure, an entirely automatic time alignment method might make user life simpler. And ways to do this exist, for example by using a GPS receiver: the GPS, GLONASS and Galileo constellations maintain precision reference time bases, which are transmitted to the GPS receivers all over the World. This reference time is itself transmitted to the system periodically: using off-the-shelf, low-cost GPS receivers for example transmit position (when available) and time data to their

host by a TTL serial port, periodically (usually every second).

The GPS solution has one problem, however: GPS chips are quite power-thirsty devices, and their continuous use would tax a battery-based (perhaps!) solution. Intermittent use, although in theory possible, would also expose to problems, due to the comparatively long cold (or warm) start time of GPS, during which time is i fact unknown.

The pragmatic solution used here is then a switch, whose ON state is used to enter a date/time programming mode. The system, battery-powered, will then maintain this date/time until energy is available. We'll return to power and power use in a further article.

This is the revised system design:




## Reading data reliably from a serial channel using Arduino devices

The SAMD21-based Arduino Zero and Feather32 have two serial ports: a 3.3V TTL port mapped to Serial1, and the usual virtual USB port mapped to Serial.

Writing to them is simple enough: in most cases all you have to do is invoking one or more calls to Serial.Print/() or Serial1.Println() or whatever, and the library code does all the dirty job of splicing your messages to bytes and sending them to the UART for you, using an interrupt based mechanism to make this happen while allowing the functional code to execute without delays.

Reading data is another story: there is nothing like an operating system serial input buffer where to peek characters orderly from, so checking data are available and timely collect them into well-formed strings is up to you.

The Serial and Serial1 objects support reading characters from the UART, by allowing used programs to inspect the number of input characters in the UART queue (think to it as a very small "operating system" buffer), and getting the oldest character in queue (while removing it). For these operations to allow reading a *string*, and not just a single character, you should provide two things:

- First and foremost, a convention saying without ambiguity when the input string has ended (maybe, using a line-termination character like on UNIX/Linux, or a special character sequence as in Windows; or, by transmitting the character count before the actual string; or any other way your fancy may suggest).
- Second, a mean (maybe a function) by which characters are quickly identified as soon as arrived, and sent to a destination string until its end is reached.

The task in itself is easy, and just demands being executed in order. The following snippet contains my implementation, in which in addition to reading I allow using one port or the other at will, with 0 identifying the USB serial, and 1 the Serial1.

```
// Read a string from serial USB line, until its terminating character.
// This function replaces the standard Serial.readBytesUntil(), which I
// wasn't able to convince yielding characters - surely I did something,
// but wasn't able understanding what from the documentation. This routine
// works instead, provided it is preceded by a wait-for-first character
// command.
void receiveUntilLineEnd(const int port=0, const char terminator = '\n') {

  char        ch;                     // Character received
  size_t      n   = sizeof(buffer);  // Number of characters available
  boolean     lineEnd = false;

  // Get pending characters
  while (Serial_available(port) > 0 && lineEnd == false) {

    // Get one pending character
    ch = Serial_read(port);

    // Add character to string, if not the terminator; otherwise,
    // declare the string complete.
    if (ch != terminator) {
      buffer[idx] = ch;
      idx++;
      if (idx >= n) {
        idx = n - 1;
      }
    }
    else {
      // terminate the string
      buffer[idx] = '\0';
      idx = 0;
      lineEnd = true;
```

```
    }

  }

}

// Get the number of pending character over serial port
int Serial_available(const int port=0) {
  int numCharsWaiting = 0;
  switch(port) {
    case(0):
      numCharsWaiting = Serial.available();
      break;
    case(1):
      numCharsWaiting = Serial1.available();
      break;
  }
  return(numCharsWaiting);
}

char Serial_read(const int port=0) {
  char ch;
  switch(port) {
    case(0):
      ch = Serial.read();
      break;
    case(1):
      ch = Serial1.read();
      break;
  }
  return(ch);
}
```

The characters read are placed in a global variable you should declare somewhere in your code:

```
// Buffer, for reading initial configuration
#define MAX_CHAR 128
char buffer[MAX_CHAR];
int  idx;
```

The variable 'idx', also a global, is used to locate the position of the next character in 'buffer'. And, the actual size of 'buffer', 'MAX_CHAR', should be large enough to surely get the longest string you may expect.

Now, which string we're waiting for? An ISO date-time, like this:

2018-03-08 08:00:00

This date-and-time, parsed to integer values, will be used to program the real-time clock (RTC) chip on acquisition start, if the programming switch is set on. Of course, you may imagine a more sophisticate interactive control if you wish. We'll not pursue this line of action in this code, concentrating rather on the more interesting (in Hypatia Project context) subject on performing "better" measurements.

# The full code

And here is the whole source code for our second example.

```
// Settable (but still "very wrong") temperature reader.
//
// This Arduino sketch is part of the Hypatia project.
//
// Parts:
//
// - An Arduino (an "Uno" is just sufficient; I'm using a Feather M0
AdaLogger, by AdaFruit)
// - A DHT-22 temperature and relative humidity sensor (good for indoor,
mainstream accuracy)
// - A switch with two stable open-close states (or a pin header)
// - A breadboard
//
// Libraries (if you have not yet installed):
//
// - DHT sensor library (from AdaFruit)
// - Adafruit Unified Sensor library (it may be necessary to install
//   this dependency of DHT lib by hand).
// - RTCZero (for RTC functionality)
//
// What you may expect from this sketch:
//
// 1) A simple data acquisition with time stamping example
//
// 2) An example on how to read safely a string from the USB-emulated
serial port
//
// 3) Indirectly, a test that your DHT-22 and the RTC
//
// 4) A way to change the system behavior based on a switch
//
// 5) How-to-read reliably from a serial port
//
// This is open-source software, covered by the MIT license.
//
//
// Written by: Mauri Favaron
```

```
// Library imports:
#include <DHT.h>      // DHT-11, DHT-22 temp/relh sensors
#include <RTCZero.h>  // Accessing the logger's Cortex-M0+ RTC functions

// Global variable, referring to the DHT-22 used in this project
#define DHT_PIN 6
#define DHT_TYPE DHT22
DHT dht(DHT_PIN, DHT_TYPE);

// Global variable, referring to the Cortex-M0+ RTC
RTCZero rtc;

// Global variables, containing the values of temperature and
// relative humidity just as coming from the DHT-22 sensor
float rTemperature;
float rRelHumidity;

// Mode selector, and mode-related data (date and time)
#define PROGRAMMING_PIN 12

// Buffer, for reading initial configuration
#define MAX_CHAR 128
char buffer[MAX_CHAR];
int  idx;


// Get the number of pending character over serial port
int Serial_available(const int port=0) {
  int numCharsWaiting = 0;
  switch(port) {
    case(0):
      numCharsWaiting = Serial.available();
      break;
    case(1):
      numCharsWaiting = Serial1.available();
      break;
  }
  return(numCharsWaiting);
}

char Serial_read(const int port=0) {
  char ch;
  switch(port) {
    case(0):
      ch = Serial.read();
      break;
    case(1):
      ch = Serial1.read();
      break;
```

```
  }
  return(ch);
}

// Read a string from serial USB line, until its terminating character.
// This function replaces the standard Serial.readBytesUntil(), which I
// wasn't able to convince yielding characters - surely I did something,
// but wasn't able understanding what from the documentation. This routine
// works instead, provided it is preceded by a wait-for-first character
// command.
void receiveUntilLineEnd(const int port=0, const char terminator = '\n') {

  char        ch;                     // Character received
  size_t      n   = sizeof(buffer);  // Number of characters available
  boolean     lineEnd = false;

  // Get pending characters
  while (Serial_available(port) > 0 && lineEnd == false) {

    // Get one pending character
    ch = Serial_read(port);

    // Add character to string, if not the terminator; otherwise,
    // declare the string complete.
    if (ch != terminator) {
      buffer[idx] = ch;
      idx++;
      if (idx >= n) {
        idx = n - 1;
      }
    }
    else {
      // terminate the string
      buffer[idx] = '\0';
      idx = 0;
      lineEnd = true;
    }

  }

}


// Print a number with leading zero, if 2 digits; if more than 2 digits,
print as it is
void print2digits(int number) {
  if (number < 10) {
    Serial.print("0"); // print a 0 before if the number is < than 10
  }
```

```
  Serial.print(number);
}


void blink(const int n) {
  for(int i = 0; i < n; i++) {
    digitalWrite(LED_BUILTIN, HIGH);
    delay(100);
    digitalWrite(LED_BUILTIN, LOW);
    delay(100);
  }
}


// Initialization
void setup() {

  // Setup sensors
  dht.begin();        // Clean-up and start DHT-22

  // Initialize serial port preliminarily (will be refined, if in
programming mode)
  Serial.begin(9600); // Will be used to monitor the system activity

  // Setup internal LED so that it may be used to inform about the
  // programming mode on start
  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED_BUILTIN, LOW);

  // Setup and read programming mode switch
  pinMode(PROGRAMMING_PIN, INPUT_PULLUP);
  int iProgState = digitalRead(PROGRAMMING_PIN);
  if(iProgState == 1) {
    // Wait for Arduino IDE terminal window to open
    // and reset serial port circuitry, allowing users to
    // see the informative messages
    while(!Serial) {}
  }
  Serial.print("Programming switch reading:");
  Serial.println(iProgState);

  // Setup serial, taking programming mode into account
  if(iProgState == HIGH) {

    // Inform we're in programming mode
    blink(5);

    // Get programming
    Serial.setTimeout(65535L);
```

```
    while(true) {

      // Clean serial buffer before to proceed
      for(int i = 0; i < MAX_CHAR; i++) buffer[i] = '\0';
      idx = 0;

      // Ask interactively the user to provide a date and time for
      // initializing the RTC.
      Serial.println("Please enter current (local, no time saving) date and
time");
      Serial.println("in ISO form (e.g. 2018-03-08 10:11:12) >> ");
      while(Serial.available() <= 0);   // Wait until a character becomes
available
      receiveUntilLineEnd(0, '\n');

      // Check the line just read has the correct number of characters (19
for an ISO string)
      int len = 0;
      for(int i = 0; i < MAX_CHAR+1; i++) {
        if(buffer[i] != '\0') {
          len++;
        }
        else {
          break;
        }
      }
      if(len == 19) {

        // Extract all parts of date-time to strings on their own
        char sYear[5];
        char sMonth[3];
        char sDay[3];
        char sHour[3];
        char sMinute[3];
        char sSecond[3];
        int i, j;
        j=0;
        for(i = 0; i < 4; i++) {
          sYear[i] = buffer[j];
          j++;
        }
        sYear[4] = '\0';
        j=5;
        for(i = 0; i < 2; i++) {
          sMonth[i] = buffer[j];
          j++;
        }
        sMonth[3] = '\0';
        j=8;
```

```
        for(i = 0; i < 2; i++) {
          sDay[i] = buffer[j];
          j++;
        }
        sDay[3] = '\0';
        j=11;
        for(i = 0; i < 2; i++) {
          sHour[i] = buffer[j];
          j++;
        }
        sHour[3] = '\0';
        j=14;
        for(i = 0; i < 2; i++) {
          sMinute[i] = buffer[j];
          j++;
        }
        sMinute[3] = '\0';
        j=17;
        for(i = 0; i < 2; i++) {
          sSecond[i] = buffer[j];
          j++;
        }
        sSecond[3] = '\0';

        // Try converting parts of date-time to numbers, and check they
more or less make sense
        byte iYear   = atoi(sYear) % 100;
        byte iMonth  = atoi(sMonth);
        byte iDay    = atoi(sDay);
        byte iHour   = atoi(sHour);
        byte iMinute = atoi(sMinute);
        byte iSecond = atoi(sSecond);
        iYear   = constrain(iYear, 1, 9999);
        iMonth  = constrain(iMonth, 1, 12);
        iDay    = constrain(iDay, 1, 31);
        iHour   = constrain(iHour, 0, 23);
        iMinute = constrain(iMinute, 0, 59);
        iSecond = constrain(iSecond, 0, 59);

        // Set the Cortex M0+ RTC to reflect the desired date and time
        rtc.begin();
        rtc.setYear(iYear);
        rtc.setMonth(iMonth);
        rtc.setDay(iDay);
        rtc.setHours(iHour);
        rtc.setMinutes(iMinute);
        rtc.setSeconds(iSecond);

        // Resume initialization code, exiting from current while() loop
```

```
        break;

      }
      else {
        Serial.println("The string just read is not an ISO date-time:
please repeat");
      }
    }
  }
  else{

    // Inform the user we're about to start immediately
    blink(4);

  }

}

void loop() {

  // Wait some time (the amount should be larger than the interval
  // the slowest sensor takes to get a new reading; 1s is sufficient
  // for the DHT-22.
  delay(1000);

  // Get readings
  rTemperature = dht.readTemperature();
  rRelHumidity = dht.readHumidity();

  // Show us what the sensor did read
  print2digits(rtc.getYear()+2000);
  Serial.print("-");
  print2digits(rtc.getMonth());
  Serial.print("-");
  print2digits(rtc.getDay());
  Serial.print(" ");
  print2digits(rtc.getHours());
  Serial.print(":");
  print2digits(rtc.getMinutes());
  Serial.print(":");
  print2digits(rtc.getSeconds());
  Serial.print(" - ");
  Serial.print("Ta: ");
  Serial.print(rTemperature);
  Serial.print(" °C    RH: ");
  Serial.print(rRelHumidity);
  Serial.println(" %");

  // Successful loop instance completion: inform users with a single blink
```

```
  blink(1);

}
```

# An example output

Here is the output, as prited to the USB virtual serial:

```
Programming switch reading:1
Please enter current (local, no time saving) date and time
in ISO form (e.g. 2018-03-08 10:11:12) >>
2018-06-15 07:05:01 - Ta: 25.30 °C    RH: 52.70 %
2018-06-15 07:05:02 - Ta: 25.30 °C    RH: 52.70 %
2018-06-15 07:05:04 - Ta: 25.40 °C    RH: 51.50 %
2018-06-15 07:05:05 - Ta: 25.40 °C    RH: 51.50 %
2018-06-15 07:05:06 - Ta: 25.30 °C    RH: 51.40 %
2018-06-15 07:05:08 - Ta: 25.30 °C    RH: 51.40 %
2018-06-15 07:05:09 - Ta: 25.30 °C    RH: 51.40 %
2018-06-15 07:05:10 - Ta: 25.30 °C    RH: 51.40 %
2018-06-15 07:05:12 - Ta: 25.30 °C    RH: 51.40 %
2018-06-15 07:05:13 - Ta: 25.30 °C    RH: 51.40 %
2018-06-15 07:05:15 - Ta: 25.30 °C    RH: 51.40 %
2018-06-15 07:05:16 - Ta: 25.30 °C    RH: 51.40 %
```

You see? We've *time stamps* now!

This is an important step towards a data logger.

Notice these time stamps are *instant-wise*, each referring to an individual data item: we're dealing with *raw data*.

Also: the first three lines refer to the dialogue between user and system. You do not see the date and time as inserted on system start, because it appears in another sub-window.