Real-time data acquisition with Arduino & buddies

Real-time?

Laypeople equates "real time" with being so blazingly fast you don't even notice it to happen.

But this definition is not what engineers intend, and a bit of clarification may be needed.

The proper definition of "real time" is, "occurring deterministically in time". It by no means signifies "fast", and indeed many slow real-time processes exist, like the hourly averaging of meteorological data.

Thus, a code excerpt like this,

```
while(true) {
    ...
    delay(1000);
}
...```
```

does *not* operate in real-time, as interrupts (and machine load on a multiyasking system) may induce departures from the imagined 1s delay and reality.

In the context of cataloguers, real-time mean (also) "locating measurements precisely with respect to time"., and this in turn requires using some form of clock allowing to measure time flow independently of processor activity. Most often, this clock takes the form of a hardware device, the Real Time Clock ("RTC"), either incorporated in the processor (as on Cortex M microcontrollers), or external; years ago, software real time clocks were not uncommon.

Our real-time data logging loop

Real-time behavior may be programmed in many way, and the one most often used in industry is adopting a real-time operating system and using it.

The way we'll use here is different, in line with the "understand how it works" used in this space, and we'll see how a decent real-time data logging behavior may be obtained using a standard Arduino sketch and the Cortex M0's built-in hardware real-time clock.

As this RTC has 1s resolution, this will also be the smallest time we could theoretically wait for - but bear in mind *accuracy* will be smaller, for smaller times.

The following code excerpt illustrates the concept of a delay-based wait-until-next-time-slot, with 1s resolution.

```
long int waitUntilTime(
  long int slotLength,   // The length of the time step ("slot") to use (s)
  int granularity = 10   // The time granularity of waits (ms; default =
10)
) {

  // Compute next slot Epoch time
  long int iEpochTime = PackTime();
  long int iStep = iEpochTime / slotLength;
  long int iNextEpoch = (iStep + 1L) * slotLength;

  // Wait until next Epoch time just found
  while(iEpochTime < iNextEpoch) {
      delay(granularity);
      iEpochTime = PackTime();
  }

  // Leave
  return(iEpochTime); // Return real time on last delay step

}
```

The core of code above is the `while` loop, designed to locate the check the current time reaches or exceeds the desired final time, whose prescribed value has been computed and stored in `iNextEpoch` variable.

But: what about `PackTime()`? This is not a standard Arduino function, and its purpose is to return the number of seconds between the "Epoch time", the instant at which the World began according to UNIX fans, that is, 01. 01. 1970 00:00:00 (we don't care of time zones), and now. Here is an implementation, ported from the *pbl_met* library (you may find it on GitHub).

```
// Compute Epoch time, as in same name routine within
//
//    https://github.com/serv-terr/pbl_met/blob/master/core/pbl_time.f90
//
long int PackTime(void) {

  const long int BASE_DAY = 2440588L; // 01.01.1970, by convention the
Epoch

  long int iJulianDay = JulianDay() - BASE_DAY;

  // Get time
  int iHour   = rtc.getHours();
  int iMinute = rtc.getMinutes();
  int iSecond = rtc.getSeconds();

  // Scale date to second form, and add time contribution
```

```
  long int iJulianSecond = iJulianDay * 24L * 3600L;
  long int iTime         = iJulianSecond +
                           (long int)iSecond +
                           60L*((long int)iMinute + 60L*(long int)iHour);

  // Leave
  return(iTime);

}
```

This function relies on another one, `JulianTime`, also not in Arduino standard libraries. Here is an implementation, also ported from the *pbl_met*.

```
// Compute Julian day, as in same name routine within
//
//    https://github.com/serv-terr/pbl_met/blob/master/core/pbl_time.f90
//
long int JulianDay(void) {

  // Constants, used within this function
  const long int DATE_REFORM_DAY =  588829L;
  const long int BASE_DAYS       = 1720995L;
  const float    YEAR_DURATION   =     365.25f;
  const float    MONTH_DURATION  =      30.6001f;

  // Get current date
  int iYear  = rtc.getYear();
  int iMonth = rtc.getMonth();
  int iDay   = rtc.getDay();

  // Preliminary estimate the Julian day, based on
  // the average duration of year and month in days.
  int iAuxYear  = (iMonth > 2) ? iYear : iYear - 1;
  int iAuxMonth = (iMonth > 2) ? iMonth + 1 : iMonth + 13;
  long int iTryJulianDay = (long int)(YEAR_DURATION * iAuxYear) +
                           (long int)(MONTH_DURATION * iAuxMonth) + (long
int)iDay +
                           BASE_DAYS;

  // Correct estimate if later than calendar reform day
  long int iNumDays = (long int)iDay + 31*(long int)iMonth + 372*(long
int)iDay;
  long int iJulianDay = iTryJulianDay;
  if(iNumDays >= DATE_REFORM_DAY) {
    long int iCentury = (long int)(0.01 * iAuxYear);
    iJulianDay += (- iCentury + iCentury/4L + 2L);
  }
```

```
  // Leave
  return(iJulianDay);

}
```

The complete code for our real-time data logger

And here is ths full code, so you can see how the code described integrates into the whole data logger.

```
#include "SigFox.h"
//#include "ArduinoLowPower.h"

const bool DEBUG = true;

#define WATER0_ID           2
#define WATER1_ID           3
#define MNQ_ID              4
#define GPS_ID              5

#define MAX_CHAR          256

///////////////////////////////
// ST-standard IoT interface //
///////////////////////////////

typedef struct __attribute__ ((packed)) iot_message {
  uint8_t  messageType;
  uint8_t  iState;
  int16_t  lightValue0;
  int16_t  lightValue1;
  int16_t  lightValue2;
  int32_t  heavyValue3;
} IoT_Message;

int idx;

// ST-standard IoT section. Actual contents depends on the chosen IoT
stack.
// Currently, SigFox on Arduino MKRFOX 1200 is used.
void IoT_Begin(
  const bool dbg     // true if debug messages are to be printed, false for
normal operation (no debug messages)
) {

    // Start SigFOX chip
    if (!SigFox.begin()) {
      // Something is really wrong, try rebooting
```

```
      if(dbg) Serial.println("Sigfox module not properly configured, or no
SigFox module at all");
        if(dbg) {
          Serial.println("Unexpected error, attempting a reboot");
        }
        NVIC_SystemReset();
        while (1);
    }

    if(dbg) Serial.println("SigFox chip configured.");

    // Print SigFox identification data
    if(dbg) {
      String version = SigFox.SigVersion();
      String ID      = SigFox.ID();
      String PAC     = SigFox.PAC();
      Serial.print("SigFox version: ");
      Serial.println(version);
      Serial.print("SigFox ID:      ");
      Serial.println(ID);
      Serial.print("SigFox PAC:     ");
      Serial.println(PAC);
    }

    // Go standby mode
    SigFox.end();
    if(dbg) Serial.println("SigFox chip set to standby mode.");

    // Enable Sigfox debug if requested (warning: large power consumption)
    if(dbg) SigFox.debug();

};


void IoT_Send(
  IoT_Message msg,
  const bool dbg
) {

    SigFox.beginPacket();
    SigFox.write((uint8_t*)&msg, 12);
    int retCode = SigFox.endPacket();
    if(retCode != 0) {
      if(dbg) {
        Serial.print("SigFox message: error transmitting - Return code: ");
        Serial.println(retCode);
      }
      return;
    }
```

```
    if(dbg) Serial.println("Message sent to SigFox, no error reported.");

};


///////////////////////
// Utility functions //
///////////////////////

float parseVal(const char* sString, const float invalid=-9999.9f) {
  if(strcmp(sString, "NAN") == 0) {
    return(invalid);
  }
  else {
    return(atof(sString));
  }
};


// Clean IoT chip state, remove pending interrupts if any, and generally
// speaking predispose the IoT to accept and execute data send commands
// (that is, calls, to IoT_Send(...) routine).
void IoT_PrepareToTransmit(const bool dbg) {

    // Start SigFOX chip
    SigFox.end();
    delay(100);
    SigFox.begin();
    delay(100);

    // Clean all pending SigFOX interrupts
    SigFox.status();
    delay(1);
    if(dbg) Serial.println("SigFox interrupt list cleaned.");

};


////////////////////////////////////////
// This application's functional code //
////////////////////////////////////////

// Serial-related buffers
int  iCharPos = 0;       // Next character position in receive buffer
char buffer[MAX_CHAR];   // Receive buffer
char msgStr[MAX_CHAR];   // Secondary buffer, holding complete strings

// State
byte          messageType;
```

```
unsigned long timeSpent;
unsigned int  ledState = 0; // Corresponding to LOW

// Auxiliary variables used to get data from serial port
int iNumBytes = 0;
int iByte     = 0;

// Function calls
void strProcessAndSend();

int readFromPort1() {

  // Wait data from serial port
  while(true) {

    int iNumBytes = Serial1.available();
    if(iNumBytes > 0) {
      for(int i = 0; i < iNumBytes; i++) {
        char iByte = Serial1.read();
        if(iByte == '\r') {

          // Ensure the terminating zero is there...
          if(iCharPos < MAX_CHAR) {
            buffer[iCharPos] = '\0';
            iCharPos++;
          }
          else {
            buffer[MAX_CHAR-1] = '\0';
          }

          // Save completed string, allowing further characters
          // to be added to the temporary string
          strcpy(msgStr, buffer);
          Serial.print("Message to parse: ");
          Serial.println(msgStr);

          // Clean character receive buffer
          for(int j = 0; j < MAX_CHAR; j++) buffer[j] = '\0';
          iCharPos = 0;

          // Leave routine - mission accomplished
          return(strlen(msgStr));

        }
        else {

          if(iCharPos <= 127) {
            buffer[iCharPos] = iByte;
            iCharPos++;
```

```
          }
          else {
            buffer[127] = '\0';
          }

        }

      }
    }

    delay(5);

  }

}




//////////
// INIT //
//////////

void setup() {

  if(DEBUG) {
    pinMode(LED_BUILTIN, OUTPUT);
    digitalWrite(LED_BUILTIN, LOW);
  }

  // Set up serial and wait until active and available
  Serial.begin(9600);
  //Serial1.begin(9600);
  //while(!Serial);
  Serial.println("Started");

  // Clean receive buffer
  for(int i = 0; i < MAX_CHAR; i++) buffer[i] = '\0';

  if(DEBUG) Serial.println("Memory cleaned, starting transmission
sequence.");

  // Execution begins on wakeup, as soon as the interrupt associated to
WAKEUP_PIN
  // is triggered.
  if(DEBUG) {
    pinMode(LED_BUILTIN, OUTPUT);
    digitalWrite(LED_BUILTIN, HIGH);
  }
```

```
  Serial.println("Boot completed.");

  // Initialize buffer
  strcpy(buffer, "1,125,20.00,30.00,10000.0,NAN,0.,25.00,12.38,A98765\n");
  // strcpy(buffer, "2,125,CPGN01,PNT1,4500000,10000000\n");
  strcpy(msgStr, buffer);
  int iNumChars = strlen(buffer);
  // 11184810 = 0x00AAAAAA

  // Get data from serial port
  //int iNumChars = readFromPort1();
  if(iNumChars <= 0) {
    Serial.println("No inbound data read from serial port");
    if(DEBUG) {
      Serial.println("Error starting system, attempting a reboot");
    }
    NVIC_SystemReset();
    while (1);
  }
  else {

    if(DEBUG) {
      Serial.print("Inbound message received: ");
      Serial.println(msgStr);
    }

    // Initialize IoT using the ST-standard function
    IoT_Begin(DEBUG);

    // Process data, and send them if all rights
    // (checking is done inside)
    strProcessAndSend();
    if(DEBUG) Serial.println("Processing-and-Sending completed.");

    if(DEBUG) digitalWrite(LED_BUILTIN, LOW);

    // Inform users all went good
    if(DEBUG) Serial.println("Successful completion.");

  }

}


//////////
// LOOP //
//////////
```

```
void loop() {
}


////////////////
// Processing //
////////////////

void strProcessAndSend() {

  // Locals
  byte          cr300msgType = 0;
  byte          packetId;
  int           level, temp, ph, redox, oxygen, battV;
  long int      conductivity, validCount;
  int           acr;
  long int      lon, lat;
  char          cpgn[7];
  char          pnt[5];
  int           retCode;

  // Message which will be sent
  IoT_Message msg1, msg2, msg3, msg4;

  cr300msgType = atoi(strtok(msgStr, ","));
  if(DEBUG) Serial.print("Message type: ");
  if(DEBUG) Serial.println(cr300msgType);
  if(msgStr[0] - '1' == 0) {

    if(DEBUG) Serial.println("Type 1 (multiparametric probes only)
message.");

    // Parse as data string
    packetId     = atoi(strtok(NULL, ","));
    level        = parseVal(strtok(NULL, ","), -99.99f) * 100.0f;
    temp         = parseVal(strtok(NULL, ","), -99.99f) * 100.0f;
    conductivity = parseVal(strtok(NULL, ","), -9999.0f) * 10.0f;
    ph           = parseVal(strtok(NULL, ","), -99.99f) * 100.0f;
    redox        = parseVal(strtok(NULL, ","), -3000.0f) * 10.0f;
    oxygen       = parseVal(strtok(NULL, ","), -99.99f) * 100.0f;
    battV        = parseVal(strtok(NULL, ",")) * 100.0f;
    validCount   = strtol(strtok(NULL, ","), NULL, 16);
    if(DEBUG) {
      Serial.println("Message parsed:");
      Serial.print("Packet ID: ");
      Serial.println(packetId);
      Serial.print("Level:     ");
      Serial.println(temp);
      Serial.print("Conductiv: ");
```

```
    Serial.println(conductivity);
    Serial.print("pH:        ");
    Serial.println(ph);
    Serial.print("Redox:     ");
    Serial.println(redox);
    Serial.print("Oxygen:    ");
    Serial.println(oxygen);
    Serial.print("Battery:   ");
    Serial.println(battV);
    Serial.print("% valid:   ");
    Serial.println(validCount);
  }

  // Encode for SigFox transfer

  msg1.messageType = WATER0_ID; // Multiparametric water sensors, block 1
  msg1.iState      = packetId;
  msg1.lightValue0 = level;
  msg1.lightValue1 = temp;
  msg1.lightValue2 = ph;
  msg1.heavyValue3 = conductivity;

  msg2.messageType = WATER1_ID; // Multiparametric water sensors, block 2
  msg2.iState      = packetId;
  msg2.lightValue0 = redox;
  msg2.lightValue1 = oxygen;
  msg2.lightValue2 = battV;
  msg2.heavyValue3 = validCount;

  // Prepare to send first message by IoT
  IoT_PrepareToTransmit(DEBUG);

  // Send water packet 1 over SigFox
  if(DEBUG) Serial.println("SigFox about to transmit message 1.");
  IoT_Send(msg1, DEBUG);
  delay(1000);

  // Prepare to send second message by IoT
  IoT_PrepareToTransmit(DEBUG);

  // Send water packet 2 over SigFox
  if(DEBUG) Serial.println("SigFox about to transmit message 2.");
  IoT_Send(msg2, DEBUG);
  delay(1000);

}
else if(msgStr[0] - '2' == 0) {

  if(DEBUG) {
```

```
    Serial.println("Type 2 (MONIQA position) message.");
  }

  // Parse as GPS fix string
  packetId = atoi(strtok(NULL, ","));
  strcpy(cpgn, strtok(NULL, ","));
  strcpy(pnt,  strtok(NULL, ","));
  lon   = atoi(strtok(NULL, ","));
  lat   = atoi(strtok(NULL, ","));
  if(DEBUG) {
    Serial.println("Message parsed.");
    Serial.print("PacketId: ");
    Serial.println(packetId);
    Serial.print("Campaign: ");
    Serial.println(cpgn);
    Serial.print("Point:    ");
    Serial.println(pnt);
    Serial.print("Lat:      ");
    Serial.println(lat);
    Serial.print("Lon:      ");
    Serial.println(lon);
  }

  // Encode for SigFox transfer

  msg3.messageType = GPS_ID; // GPS fix
  msg3.iState      = packetId;
  msg3.lightValue0 = (lat & 0xffff0000) >> 16;
  msg3.lightValue1 = lat & 0xffff;
  msg3.lightValue2 = 0;
  msg3.heavyValue3 = lon;

  unsigned long int b0 = cpgn[0];
  unsigned long int b1 = cpgn[1];
  unsigned long int b2 = cpgn[2];
  unsigned long int b3 = cpgn[3];
  unsigned long int b4 = cpgn[4];
  unsigned long int b5 = cpgn[5];
  unsigned long int b6 = cpgn[6];
  unsigned long int c0 = pnt[0];
  unsigned long int c1 = pnt[1];
  unsigned long int c2 = pnt[2];
  unsigned long int c3 = pnt[3];
  msg4.messageType = MNQ_ID;
  msg4.iState      = packetId;
  unsigned int total = (b0 | (b1 << 8)) & 0x0000FFFF;
  msg4.lightValue0 = total;
  total = (b2 | (b3 << 8)) & 0x0000FFFF;
  msg4.lightValue1 = total;
```

```
    total = (b4 | (b5 << 8)) & 0x0000FFFF;
    msg4.lightValue2 = total;
    unsigned long int total32;
    total32 = c0;
    Serial.println(total32);
    total32 += (c1 << 8);
    Serial.println(total32);
    total32 += (c2 << 16);
    Serial.println(total32);
    total32 += (c3 << 24);
    Serial.println(total32);
    msg4.heavyValue3 = total32;
    Serial.println(total32);

    // Prepare to send first message by IoT
    IoT_PrepareToTransmit(DEBUG);

    // Send GPS packet over SigFox
    if(DEBUG) Serial.println("SigFox about to transmit message 3.");
    IoT_Send(msg3, DEBUG);
    delay(1000);

    // Prepare to send second message by IoT
    IoT_PrepareToTransmit(DEBUG);

    // Send second message
    if(DEBUG) Serial.println("SigFox about to transmit message 4.");
    IoT_Send(msg4, DEBUG);
    delay(1000);

  }

}
```

Latency

The time step ("slot") used for data acquisition in former code has been set to 5s (in an industrial-strength data logger this duration would have been presumably been user-settable). So, we would expect on intuition that the time stamps of data gathered is an exact multiple of 5s.

But if we start the datalogger, we discover it is not:

```
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) >>
2018-08-05 18:10:06 - Ta: 27.20 °C    RH: 58.00 %
2018-08-05 18:10:11 - Ta: 27.70 °C    RH: 56.80 %
2018-08-05 18:10:16 - Ta: 27.70 °C    RH: 56.70 %
2018-08-05 18:10:21 - Ta: 27.70 °C    RH: 56.50 %
```

As you can see, the time at which acquisition steps take place occur *one second* past the 5s multiple. Why?

The answer is intuitively simple: because the wait time does only account for the wait itself, not the data gathering from the sensor *and* reporting measurements to serial port, *and* writing data plus time stamp to the SD card, *and* flushing contents to SD Card to ensure no data would be lost in case of power failure.

All these operations must by their very nature (and by the single-task paradigm underpinning Arduino code) occur sequentially, and this takes quite a long time: approximately one second, overall.

Then, we can expect a time latency between the instant the real time clock says "OK, it's time to proceed", and the data acquisition chores did actually complete.

This fact is normally harmless - it changes very little the real-timeness of our system - but we have to know of it.

In industrial-strength data loggers, the use of powerful CPUs and fast communication buses often makes the data acquisition latency time close to zero, in fact small enough it can be safely ignored in most cases. There exist instances I've seen, however, where latency should be taken into account - one I'm thinking of was a thermoelectric power plant turbine vibration monitoring system, in which data were collected at a rate of 20000 samples per second, and an anomalous vibration pattern had to be "immediately" acknowledged and acted upon, doing what possible to destroy the turbine and/or the generator in the meanwhile.

Fortunately, environmental monitoring applications, especially when based on relatively slow sensors, are not so critical, and all we shall do with latency time is to just know it - plus, ensuring it is possibly shorter than the time slot selected.