Analyse du code Cosmonode

Initialisation du GPS, envoi des données NMEA, désactivation du GPS

```
void gps_init() {
    gps_enable();
//    delay(500);
    gps.begin(9600);
//    gps.sendCommand(PMTK_SET_BAUD_115200);
//    SerialGPS.end();
//    gps.begin(115200);
    /**
     * PMTK_SET_NMEA_OUTPUT_RMCGGA fix possible without date.
     */
    gps.sendCommand(PMTK_SET_NMEA_OUTPUT_RMCGGA);
    /**
     * PMTK_SET_NMEA_OUTPUT_RMCONLY no altitude.
     */
    gps.sendCommand(PMTK_SET_NMEA_UPDATE_1HZ);
    gps_synchronize();
    gps_disable();
    SerialGPS.flush();
    SerialGPS.end();
}
```

Lorsque la carte est alimentée la LED D3 clignote toutes les secondes. Quand le GPS est fixé elle s'arrête de clignoter, la procédure pour l'acquisition des données NMEA peut commencer. Comme listé ci-dessus et représenté sur le logigramme, le GPS s'initialise (gps_enable()) et envoie les données NMEA au microcontrôleur via gps.sendCommand. Ensuite avec la fonction gps_synchronize() le microcontrôleur réinitialise toutes ses données (gps_reset()) puis attend que le GPS obtienne une date et une heure valide (gps_date() et gps_time()) et enfin lit, stocke et analyse les données avec gps_read() et gps_parse(). Enfin le GPS se désactive avec gps_disable().

Ceci est la version "#if 1" qui est la version toujours vraie. Il existe une version toujours fausse "#if 0" qui diffère de la première en supprimant la fonction qui désactive le GPS à la fin du code. L'ajout de ces deux versions permet de réaliser des tests en changeant le 1 en 0 et vice-versa.

Lecture et analyse des données GPS

```
void gps_read() {
    logger.printf("[GPS] read date, time and position\n");
    // Date
    worker.date.year = gps.year;
    worker.date.month = gps.month;
    worker.date.day = gps.day;
    rtc.setDate(worker.date.day, worker.date.month,
worker.date.year);
    // Time
    worker.time.hour = gps.hour;
    worker.time.minute = gps.minute;
    worker.time.seconds = gps.seconds;
    worker.time.milliseconds = gps.milliseconds;
    rtc.setTime(worker.time.hour, worker.time.minute,
worker.time.seconds, worker.time.milliseconds);
    // Location
    worker.location.latitude = gps.latitudeDegrees;
    worker.location.longitude = gps.longitudeDegrees;
    worker.location.altitude = gps.altitude;
    // Output
    logger.printf("[GPS] latitude %.3f\n",
worker.location.latitude);
    logger.printf("[GPS] longitude %.3f\n",
worker.location.longitude);
    logger.printf("[GPS] altitude %.3f\n",
worker.location.altitude);
    logger.printf("[RTC] set date and time to %02d/%02d/%02d
%02d:%02d:%02d\n", rtc.getDay(),
            rtc.getMonth(), rtc.getYear(), rtc.getHours(),
rtc.getMinutes(),
            rtc.getSeconds());
}
```

```
void gps_parse() {
    while (gps.newNMEAreceived() == false) {
        gps.read();
    }
    gps_nmea();
    gps.parse(gps.lastNMEA());
```

- Le microcontrôleur lit les données avec gps_read() (position, date, heure, etc...).
- Le microcontrôleur analyse les données reçues avec la fonction gps_parse(). Il lit la dernière phrase NMEA reçue avec gps.parse(gps.lastNMEA()) pour extraire les informations.

Mise à jour et initialisation du RTC (Real Time Clock)

Dans gps_read() ci-dessus on trouve les lignes :

- rtc.setDate(worker.date.day, worker.date.month, worker.date.year)
- rtc.setTime(worker.time.hour, worker.time.minute, worker.time.seconds, worker.time.milliseconds)

Ces lignes permettent de mettre à jour les données du RTC à partir des données transmises par le GPS.

L'initialisation du RTC avec rtc_init() (cf photo ci-dessous) quant à elle ne s'effectue qu'une seule fois. Si FAKEGPS est à 1, le RTC s'allume avec une valeur par défaut et dès que rtc.begin est appelée le RTC commence à compter le temps. Si FAKEGPS est à 0, alors rtc_init() renvoie gps_init() qui appelle gps_read(). Dans ce cas là les données RTC sont mises à jour.

```
void rtc_init() {
    rtc.setClockSource(STM32RTC::LSE_CLOCK);
    rtc.begin();
#if FAKEGPS
#else
    gps_init();
#endif
}
```

Ces deux cas peuvent permettre de faire des tests (en changeant FAKEGPS à 0 ou 1). L'un sans utilisation de GPS grâce à rtc_init(), l'autre avec une mise à jour des données en temps réel grâce à gps_read().

La mise à jour régulière du RTC permet de planifier la mise en veille du microcontrôleur et donc d'économiser de l'énergie.

Configuration de l'alarme

```
void rtc_schedule() {
#if FAKECALENDAR
    worker.calendar.alarm = rtc.getEpoch() + 30;
    rtc.setAlarmEpoch(worker.calendar.alarm,
STM32RTC::MATCH_YYMMDDHHMMSS, ALARM_A);
    rtc.enableAlarm(STM32RTC::MATCH_YYMMDDHHMMSS);
```

- Lorsque FAKECALENDAR est à 1 , le microcontrôleur configure une alarme toutes 30 secondes avec rtc.getEpoch() + 30 afin de tester le mécanisme du réveil.

```
#else
    File file = SD.open(worker.calendar.file);
    if (file) {
        file.seek(worker.calendar.position);
        while (file.available() && (rtc.getEpoch() > worker.calendar.alarm)) {
            worker.calendar.alarm = file.parseInt();
        }
```

- Sinon, lorsque FAKECALENDAR est à 0, le microcontrôleur ouvre le calendrier indiquant les horaires de passage du satellite Kinéis stocké sur la carte SD avec SD.open.
- Tant qu'il y a des données à lire dans le fichier (boucle while) il compare la date actuelle avec la date de l'alarme suivante.
- Si le timestamp du RTC est supérieur au timestamp stocké dans le calendrier alors la boucle continue sinon cela signifie que le timestamp stocké dans le calendrier est dans le futur (prochain réveil) et est donc stocké dans le microcontrôleur (worker.calendar.alarm) pour pouvoir programmer l'alarme RTC.

```
        worker.calendar.position = file.position();
        bool eof = file.position() == file.size();
        file.close();
        if (eof) {
            logger.printf("[ERROR] no more data in calendar file\n");
        } else {
            rtc.setAlarmEpoch(worker.calendar.alarm,
STM32RTC::MATCH_YYMMDDHHMMSS, ALARM_A);
            rtc.enableAlarm(STM32RTC::MATCH_YYMMDDHHMMSS);
            tm* alarm = gmtime(&worker.calendar.alarm);
            // Output
            logger.printf("[RTC] set alarm to %02d/%02d/%02d %02d:
%02d:%02d %lu\n",
                    alarm->tm_mday, alarm->tm_mon + 1, alarm->tm_year
- 100,
                    alarm->tm_hour, alarm->tm_min, alarm->tm_sec,
worker.calendar.alarm);
        }
    } else {
        serial_error("unable to open calendar file");
    }
#endif
}
```

- Ensuite, le microcontrôleur met à jour l'emplacement dans le fichier avec file.position pour qu'il recherche les données du calendrier au bon endroit les prochaines fois. Il ferme alors le fichier.
- Si la fin du fichier est atteinte alors on aura l'erreur suivante : [ERROR] no more data in calendar file\n.
- Sinon l'alarme RTC est configuré avec rtc.enableAlarm et activée avec gmtime().
- Les dernières lignes permettent d'afficher l'alarme (logger.printf) et d'afficher une erreur si le microcontrôleur n'arrive pas à rentrer dans la carte SD.

Lecture et stockage des mesures

```
if (worker.wakeup) {
    worker.calendar.wakeup = rtc.getEpoch();
    logger.printf("[LOOP] wake up at %lu (delta %lu)\n",
            worker.calendar.wakeup,
            worker.calendar.wakeup - worker.calendar.alarm);
    worker.wakeup = false;
    worker.counter++;
    sensor_read();
```

Lorsque worker.wakeup est true c'est à dire lorsque l'alarme RTC se déclenche, le microcontrôleur va lire les données des capteurs avec sensor_read() décrit dans l'image ci-dessous.

```
void sensor_init() {
    Wire.begin();
    sensor_read();
    sensor_output();
}

/**
 * @brief Reads sensor.
 *
 */
void sensor_read() {
    // Read TMP102
    Wire.requestFrom(TMP102_ADDRESS, 2);
    // 12-bit signed raw temperature
    worker.tmp102.raw = Wire.read();
    worker.tmp102.raw <<= 8;
    worker.tmp102.raw |= Wire.read();
    worker.tmp102.raw >>= 4;
    // Convert to °C
    worker.tmp102.celsius = worker.tmp102.raw * 0.0625;
    // Read battery voltage
    worker.battery.voltage = analogRead(VBAT) * 3.3 * 5 / 3.16 / 4095.0;
}

/**
 * @brief Outputs sensor.
 *
 */
void sensor_output() {
    logger.printf("[TMP102] temperature %.2f °C 0x%04X\n", worker.tmp102.celsius, worker.tmp102.raw);
    logger.printf("[BATTERY] voltage %.2f V\n", worker.battery.voltage);
}
```

Les mesures sont lues avec sensor_read(). Par exemple avec le capteur TMP102 :

```
worker.tmp102.raw = Wire.read();
worker.tmp102.raw <<= 8;
```

A la première ligne le microcontrôleur lit le premier octet de la réponse du capteur TMP102 et le stocke puis à la deuxième ligne il lit le deuxième octet et le combine avec le premier pour former une valeur 16 bits (car le TMP102 envoie la température sur 2 octets).

Mise en veille du microcontrôleur

```
Serial.flush();
if (worker.calendar.alarm) {
    LowPower.deepSleep();
}
```

Une fois que les mesures ont été collectées par le microcontrôleur, il se met en veille profonde uniquement si une alarme est déjà programmée (condition if) avec la fonction LowPower.deepSleep().

Réveil du microcontrôleur

```
void rtc_alarm(void*) {
    worker.wakeup = true;
}
```

Le microcontrôleur se réveille ensuite uniquement lorsque worker.wakeup est true. Ainsi il peut commencer à éxecuter les tâches du code.