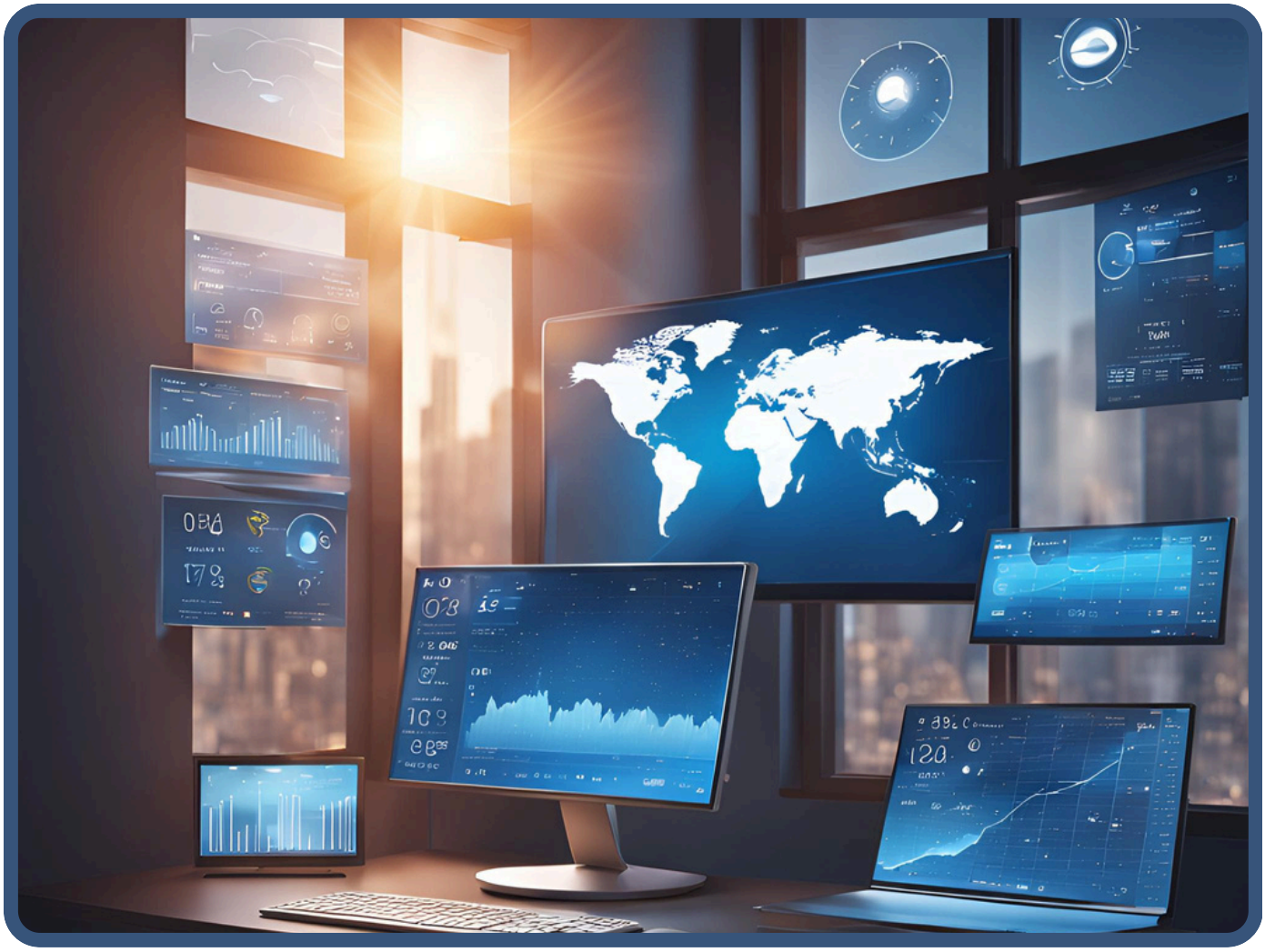
EQUIPE 7

LIVRABLE 2 :
WORLDWIDE WEATHER WATCHER



PITOIS MAËL

GLAIROT VALENTIN

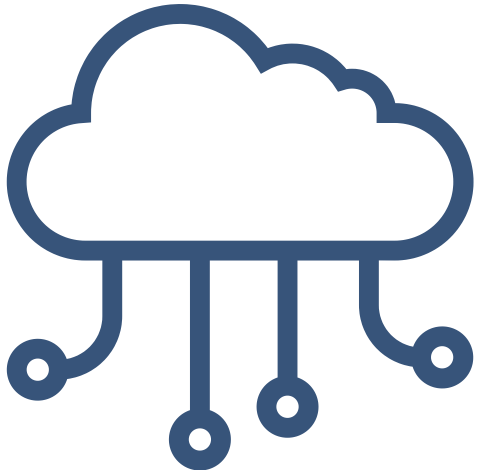
CARMONA JULIAN

SELLE BAPTISTE

SOMMAIRE :

INTRODUCTION :

Nous avons été mandatés par l'Agence Internationale pour la Vigilance Météorologique (AIVM) pour développer un prototype de station météorologique embarquée à bord de navires de surveillance. Ce système vise à mesurer des paramètres cruciaux liés à la formation des cyclones et autres phénomènes naturels. En utilisant un microcontrôleur AVR ATmega328 (Arduino), nous devons concevoir une solution intégrant des capteurs variés et des composants essentiels, tout en garantissant simplicité et facilité d'utilisation pour l'équipage. Ce livrable marque une deuxième étape dans la réalisation de ce projet ambitieux.



ARCHITECTURE DU PROGRAMME

- **États des modes :**
  - Interruptions mode économique et maintenance
- **État des LEDs :**
  - *etat_led (char etat) : permet de modifier la couleur de la LED*
- **Boutons :**
  - *int boutonVert;* : État du bouton vert.
  - *int boutonRouge;* : État du bouton rouge.
- **Temps :**
  - *unsigned long previousMillis;* : Pour la gestion du temps (détection des appuis longs ou délai de 30 minutes).
  - *const long interval30min = 1800000;* : Intervalle de 30 minutes pour passer au mode configuration.
- **Capteurs et autres composants :**
  - *int luminosite;* : Valeur du capteur de luminosité.
  - *DateTime dateRTC;* : Données récupérées du module RTC.
  - *float dataGPS;* : Données GPS.
  - *File dataSD;* : Fichier pour la carte SD.

EXPLICATION DES FONCTIONS

• Initialisation

```
void setup {

   Initialisation LED {}
   Initialisation bouton vert {}
   Initialisation bouton rouge {}

   Bouton vert = digital_read(pinBoutonVert)
   Bouton rouge = digital_read(pinBoutonRouge)

   Initialisation interface {}
   Initialisation horloge {}
   Initialisation carte SD {}
   Initialisation GPS {}

      Initialisation interrupt 1 {Bouton vert appuyé, Mode économique} // Mode économique
    Initialisation interrupt 2 {Bouton rouge appuyé, Mode maintenance} // Mode Maintenance

   global eco = 0;
   global maintenance = 0;
   global configuration = 0;
   global LOG_INTERVALL = 10;


   si bouton rouge pressé :
      Mode Configuration()
}
```

On initialise d'abord les composants essentiels du système, tels que **les LED**, les boutons, l'interface utilisateur, l'horloge, la carte SD, et le module GPS. Ensuite, on lit l'état des boutons (vert et rouge) afin de détecter les actions de l'utilisateur. Deux interruptions **sont mises en place** : l'une pour passer en mode économique lorsque le bouton vert est pressé, et l'autre pour passer en mode maintenance lorsque le bouton rouge est activé. Plusieurs **variables globales** sont initialisées pour gérer les modes de fonctionnement, et si le bouton rouge est pressé, on passe en mode configuration.

• Boucle Principale

```
loop {
   Mode standard()
}
```

Dans cette boucle principale, on exécute en continu la fonction “**mode standard**”. Cela signifie que tant qu'aucun autre événement, comme l'activation d'un mode différent via les interruptions, n'intervient, le système **fonctionne en mode standard**.

EXPLICATION DES MODES

```
void Mode configuration() {
    si 30 min timeout {
        return
    } sinon {
        led_etat("jaune")
        Etat Systeme()
        Configuration des paramètres
        acq_capt_des()
    }
}
```

Dans la fonction **Mode configuration**, on vérifie d'abord si un délai de 30 minutes est atteint. Si c'est le cas, on sort de la fonction. Sinon, on allume une LED en jaune pour indiquer que le système est en mode configuration. Ensuite, on affiche **l'état actuel du système** et on permet à l'utilisateur de modifier certains paramètres. Pendant cette phase, l'acquisition des données des capteurs **est désactivée** avec la fonction acq_capt_des().

```
void economie_energie() {
    LOG_INTERVALL = LOG_INTERVALL * 2
}
```

Dans la fonction **economie_energie**, on double l'intervalle d'acquisition des données en multipliant **la variable LOG_INTERVALL** par deux. Cela permet de réduire la fréquence des enregistrements, **optimisant** ainsi la consommation d'énergie du système.

```
void changer SD() {
    // Permet de changer la carte SD
}
```

La fonction **changer SD** est utilisée pour permettre le remplacement de la carte SD. Cette fonction pourrait inclure des instructions pour **éjecter en toute sécurité** l'ancienne carte SD et préparer le système à en accepter une nouvelle, garantissant ainsi que les données soient **correctement sauvegardées** et que la nouvelle carte soit prête à l'emploi.

```
void Configuration des paramètres() {
Input Paramètre à accéder
   si LOG_INTERVALL :
      Input Nouveau LOG_INTERVALL
   si TIMEOUT :
      Input Nouveau TIMEOUT
   si FILE_MAX_SIZE :
      Input Nouveau FILE_MAX_SIZE
   si VERSION :
      Output Version
   si CLOCK :
      input Nouveau CLOCK
   si DATE :
      input Nouveau DATE
   si DAY :
      input Nouveau DAY
   si RESET :
      LOG_INTERVALL= 10
      TIMEOUT = 30
      FILE_MAX_SIZE = 4096
}
```

La fonction **Configuration des paramètres** permet d'accéder et de modifier différents paramètres du système. On commence par demander à l'utilisateur de choisir le paramètre à configurer. Si c'est **LOG_INTERVALL**, on demande un nouveau délai d'intervalle. Si c'est TIMEOUT, on entre un nouveau délai de temporisation. Pour **FILE_MAX_SIZE**, on configure une nouvelle taille maximale de fichier. Si VERSION est sélectionné, on affiche la version actuelle. De plus, l'utilisateur peut ajuster l'heure (CLOCK), la date (DATE), ou le jour (DAY). Enfin, si RESET est activé, **tous les paramètres** reviennent aux valeurs par défaut.

```
void acq_capt_des() {
   // Permet l'acquisition des capteurs
   désactivée
}
```

La fonction **acq_capt_des** désactive l'acquisition des données provenant des capteurs. Cela permet d'arrêter temporairement la collecte d'informations des capteurs comme poue mode maintenance ou configuration.

```
void accederDonnees() {
   // Permet d'accéder aux données
}
```

La fonction **accederDonnees** permet d'accéder aux données stockées dans le système. Cela inclut la lecture de fichiers enregistrés sur la carte SD, l'affichage des données collectées par les capteurs, ou l'accès à d'autres informations stockées dans le système pour consultation ou analyse.

```
void Récupération Donnees() {
 Lire capteurs
 Lire horloge
 Lire GPS
 Open Carte SD
 Stocker les données
 Close Carte SD
}
```

La fonction **Récupération Donnees** permet de collecter et stocker les données du système. Elle commence par lire les données des capteurs, puis l'horloge pour obtenir l'heure actuelle, et le module GPS pour la localisation. Ensuite, on **ouvre la carte SD** pour y enregistrer ces informations. Une fois les données stockées, la carte SD est fermée pour garantir une bonne gestion des fichiers et éviter toute corruption de données.

```
void Etat Systeme() {
   si Erreur RTC {
      led_etat("rouge_bleu")
      return
   }
   si Erreur GPS {
      led_etat("rouge_jaune")
      return
   }
   si Erreur GPS données incohérentes {
      led_etat("rouge_vert1")
      return
   }
   si Données capteur incohérentes {
      led_etat("rouge_vert2")
      return
   }
   si Carte SD pleine {
      led_etat("rouge_blanc1")
      return
   }
   si Erreur écriture carte SD {
      led_etat("rouge_blanc2")
      return
   }
}
```

La fonction **Etat Systeme** vérifie différents états d'erreurs critiques dans le système. Si une erreur est détectée avec le module RTC, la LED change en rouge et bleu, puis la fonction se termine. **En cas d'erreur** avec le GPS ou des données incohérentes, la LED passe respectivement en rouge et jaune, ou rouge et vert. Si des incohérences sont détectées dans les données des capteurs, la LED devient rouge et vert (variation 2). Si la carte SD est pleine, la LED affiche rouge et blanc (variation 1), et **en cas d'erreur** d'écriture sur la carte SD, elle affiche rouge et blanc (variation 2), avant de quitter la fonction.

```
void led_etat(char etat) {
   si etats = "bleu"{
      led -> bleu
   }
   sinon si etat == "vert"{
      led -> vert
   }
   sinon si etat == "jaune"{
      led -> jaune
   }
   sinon si etat == "orange"{
      led -> orange
   }
   sinon si etat == "rouge_bleu"{
      led -> intermittence rouge bleu
   }
   sinon si etat == "rouge_jaune"{
      led -> intermittence rouge jaune
   }
   sinon si etat == "rouge_vert1"{
      led -> intermittence rouge vert 1s
   }
   sinon si etat == "rouge_vert2"{
      led -> intermittence rouge vert 2s
   }
   sinon si etat == "rouge_blanche1"{
      led -> intermittence rouge blanc 1s
   }
   sinon si etat == "rouge_blanche2"{
      led -> intermittence rouge blanc 2s
   }

}
```

La fonction **led_etat** permet de changer l'état de la LED en fonction du paramètre etat reçu. Si l'état est "bleu", la LED devient bleue. Pour "vert", elle passe au vert, et ainsi de suite pour "jaune" et "orange". Des états complexes comme "rouge_bleu" **déclenchent une intermittence** entre le rouge et le bleu, tandis que "rouge_jaune" fait alterner le rouge et le jaune. Des variations spécifiques comme "rouge_vert1" et "rouge_vert2" contrôlent des intermittences rouge-vert avec des **durées de 1 seconde ou 2 secondes**, respectivement. De même, "rouge_blanche1" et "rouge_blanche2" gèrent des intermittences rouge-blanc avec **des délais** de 1 ou 2 secondes.

CONCLUSION

Cette étape du projet marque la deuxième phase de notre travail, qui consiste à présenter la structure des fonctions et des composants du système. Nous avons détaillé les principales fonctionnalités comme l'initialisation des modules, la gestion des modes et des erreurs, ainsi que l'acquisition des données. Ces éléments forment la base de notre prototype. Des précisions sur l'implémentation complète des fonctions seront fournies dans les prochaines étapes du projet, où nous approfondirons la logique interne et l'optimisation du code pour garantir un fonctionnement optimal du système.