Systèmes embarqués 2

Projet intégré

Loïc Guibert, Raphaël Pittet | Juin 2019

Table des matières

Introduction

Plusieurs concepts et mécanismes ont été enseignés lors des cours théoriques de systèmes embarqués. Dans le but de les comprendre et de les utiliser, un projet intégré doit être conçu par groupe de deux. Ce projet se présente sous la forme d'une application, dont la forme est libre. Chaque groupe doit passer par l'analyse du programme, sa conception, sa réalisation et enfin ses tests finaux. Toutes ces étapes sont nécessaires pour de développement d'un programme dans les règles de l'art.

Ce projet, conçu en langage C, repose sur un système d'exploitation élémentaire coopératif qui a la possibilité de communiquer et de capter son environnement à l'aide de plusieurs périphériques d'entrée/sortie.

Nous avons décidé de concevoir un jeu de bataille navale à deux joueurs, chacun possédant un dispositif BeagleBone Black. Ce jeu utilise l'écran LCD OLED, le bouton rotatif, un module de communication nRF ainsi que les boutons poussoirs.

Analyse

Comme expliqué dans la section « Introduction », nous avons réalisé un jeu de bataille navale (aussi appelé « touché-coulé »). Ce jeu consiste à placer un ou plusieurs navires, de taille différente ou non, sur une grille. Chaque joueur doit, à tour de rôle, deviner l'emplacement des bateaux de son adversaire dans le but d'envoyer des tirs sur les cases où sont placés les bateaux. Lorsque toutes les cases d'un bateau sont touchées, ce dernier est « coulé ». Le gagnant est le joueur ayant réussi à couler tous les bateaux de son adversaire.

Voici un exemple du plateau de ce jeu :

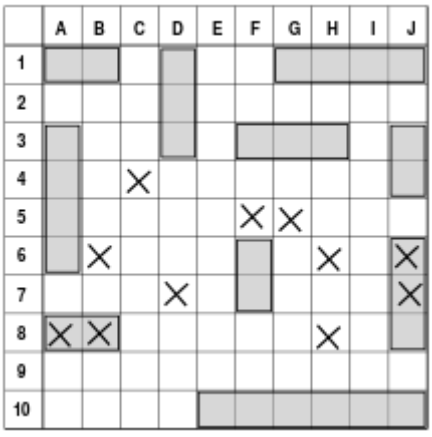


Figure 1 - Exemple d'un plateau de jeu (source : wikimédia)

ELEMENTS DU JEU

La grille est affichée sur l'écran, avec une largeur et une profondeur de 6 cases. Nous nous retrouvons donc avec une matrice à deux dimensions de 36 cases.

Les cases peuvent être vide ou inclure une partie d'un bateau. Cette dernière indication est donnée par une couleur spécifique, alors qu'une case sans contenu est affichée vide. Lorsqu'un coup est porté sur une case, un rond y est affiché.

Les bateaux sont au nombre de trois, avec des tailles différentes : nous avons deux bateaux de deux cases de long et un troisième de trois cases.

DEROULEMENT D'UNE PARTIE

Voici les différentes étapes d'une partie de bataille navale, que cela soit en jeu de société ou en jeu digital :

1. Une grille vide s'affiche.
2. Chaque joueur doit placer ses différents bateaux sur la grille. Un joueur dispose de cinq bateaux.
3. Dès que chaque personne a fini son placement, le jeu démarre et demande à un joueur de commencer.
4. Les tours s'enchaînent jusqu'à ce qu'un joueur ait réussi à éliminer tous les navires adverses. Il sera déclaré vainqueur.
    a. Si un joueur touche toutes les parties d'un bateau, ce dernier est déclaré comme coulé.
5. Si les joueurs le désirent, ils peuvent relancer une partie en cliquant sur le bouton poussoir prévu à cet effet. Les deux joueurs doivent avoir cliqué sur le bouton reset.

CONTROLES

Nous avons plusieurs besoins d'interface homme-machine afin de rendre ce jeu jouable.

Le **déplacement** entre les cases est fait avec le bouton rotatif du BeagleBone : la grille est parcourue itérativement. Lorsque le joueur souhaite **sélectionner** une case, il doit appuyer sur un bouton poussoir. Un autre bouton poussoir sert à pivoter le navire à placer de nonante degrés sur la grille, lorsque les joueurs sont demandés de placer leurs bateaux. Un troisième bouton poussoir sert à relancer une nouvelle partie.

Conception

Voici les différents aspects que nous avons utilisés afin de mener à bien notre projet intégré.

THREADS

L'utilisation de threads est demandée pour ce travail, mais elle est tout de même nécessaire pour le bon fonctionnement de notre jeu. En effet, la communication nRF doit être implémentée avec cette méthode afin d'assurer des transferts de données performants au sein de notre programme.

Ces threads permettent donc de réaliser des tâches en parallèle. C'est un des mécanismes principaux d'un OS.

Notre thread principal est celui du jeu en lui-même, **game_process**. C'est ce thread qui contient la machine d'états qui est décrite au point suivant. Toutes les données à traiter ainsi que les directives d'affichage sont gérées à cet endroit.

Nous avons deux threads qui assurent le fonctionnement de la communication : **thread_nrf_rx** et **thread_nrf_tx**, qui s'occupent respectivement de la réception et du transfert des données.

Ces trois threads sont coopératifs, c'est-à-dire qu'ils se passent la main lorsqu'ils ont fini leur traitement. La commutation de contexte cause de légères pertes de temps, mais la robustesse ainsi que la fiabilité générale du jeu se retrouve largement améliorée.

Il aurait été possible d'appeler les deux méthodes dont s'occupent les deux threads de communication à la fin de la machine d'états puis de stocker les valeurs obtenues en tant que variables statiques, mais ceci n'aurait pas été optimal. Nous avons donc opté pour l'utilisation de threads.

Grâce aux threads, ces trois fonctionnalités se donnent régulièrement la main afin qu'ils puissent s'exécuter séquentiellement de façon optimale.

MACHINE D'ETATS

Etant donné que le jeu que nous avons choisi passe par plusieurs étapes spécifiques, nous avons décidé de construire la logique principale de notre programme sous la forme d'une machine d'états.

Cette manière de faire comporte plusieurs avantages. Premièrement, elle permet d'éviter des appels de fonctions indésirés, en passant par des modes précis et à nombre fini. Le passage d'un mode à l'autre est initié par plusieurs événements, qu'ils proviennent du système ou de l'utilisateur. De plus, la logique du jeu ne peut être qu'uniquement dans un état à la fois. Cette méthode permet donc de spécifier ce comportement de façon sécurisée.

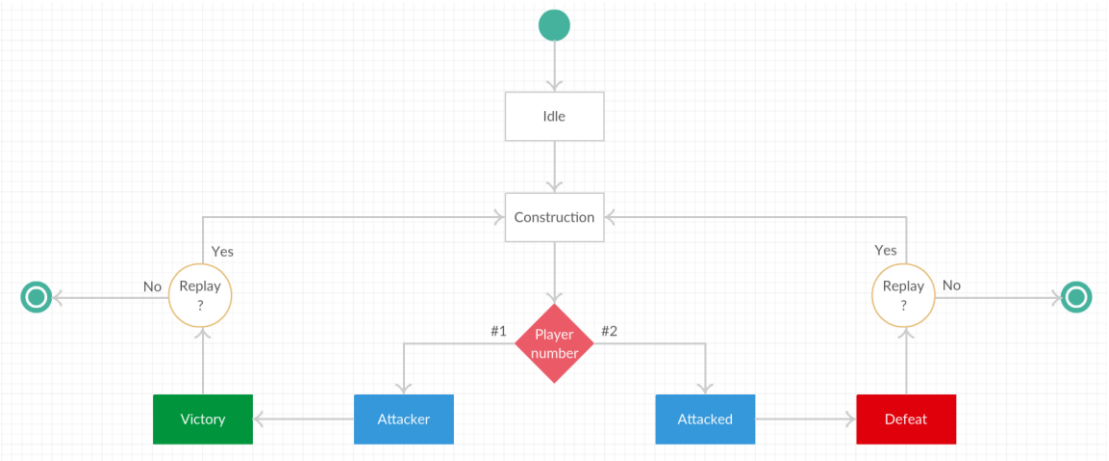


Figure 2 - Machine d'états

IDLE	Etat lors du lancement du jeu. Le choix du numéro du joueur (1 ou 2) est fait ici
CONSTRUCTION	Les joueurs choisissent les emplacements de leurs bateaux.
ATTACKER	L'attaquant doit choisir l'emplacement de la grille qu'il souhaite attaquer. Il reçoit le résultat de cette attaque pour (re)devenir attaqué.
ATTACKED	L'attaqué ne peut rien faire et commute en tant qu'attaquant lorsque son appareil envoie le résultat de l'attaque subie.
VICTORY	Si tous les bateaux adverses sont coulés, la victoire est annoncée. Il est possible de relancer une partie.
DEFEAT	Si tous les bateaux du joueur sont coulés, un texte s'affiche sur son écran afin d'annoncer sa défaite. Il est possible de relancer une partie.

Nous avons réalisé cette étape par le biais d'un switch case.

STRUCTURES DE DONNEES

Ici sont exposés les aspects des données principales que nous utilisons, Ce n'est pas une liste exhaustive : nous présentons ici les éléments les plus intéressants.

Données de jeu

Nous devons stocker plusieurs informations afin d'assurer le bon déroulement du jeu.

Pour chaque case du plateau de jeu, nous devons savoir si un bateau y est placé, si l'opposant a tiré dessus et quel est le statut de cette case.

```
struct cell {
        bool is_touched;
        struct boat* boat;
        int cell_status;
};
```

Figure 3 - Structure d'une cellule

Le statut de ces cellules peut prendre différentes valeurs, que nous avons rendues accessibles par le biais d'une énumération possédant ces valeurs : touched, water, sunk, nothing, selection ou boat.

La grille de jeu agit différemment si le joueur a le rôle d'attaquant ou celui d'attaqué. Dans le but de simplifier le traitement des données, nous avons créé deux matrices locales de ces cellules, une pour chaque mode.

```
static struct cell grid_attacker[6][6];
static struct cell grid_attacked[6][6];
```

Figure 4 - Les deux plateaux de jeu

Pour agir et lire ces données, nous avons implémenté un getter et un setter.

Concernant le mode de jeu dans lequel se situe le joueur, nous avons placé une variable stockant une valeur de l'énumération contenant les différents modes au sein du module game.c. Il est possible de consulter cette valeur en utilisant un getter.

Enfin, nous avons une structure qui contient les informations concernant chaque bateau :

```
struct boat {
        int length;
        int x_pos;
        int y_pos;
        enum orientation orientation;
        uint8_t cells;
        bool is_sunk;
};
```

Figure 5 - Données pour les bateaux

L'emplacement du bateau est indiqué par son point le plus au fond à gauche de sa position, qui est donné par les integers x_pos et y_pos. Son orientation est donnée par l'énumération orientation, qui peut prendre les valeurs « horizontal » ou « vertical ». Un bitset nommé « cells » permet d'indiquer quelles parties du bateau ont déjà été touchées, ce qui permet de tester si le navire est coulé. Le cas échéant, nous portons le boolean is_sunk à true.

Affichage

Pour l'affichage, nous avons mis en place une manière de sélectionner les différentes couleurs des cellules. Ces dernières possèdent des codes couleurs très spécifiques qui indiquent leur état.

Les couleurs sont données par une énumération, qui identifie leur valeur directement avec une donnée RGB565 qui est utilisé pour l'écran LDC.

Afin de pouvoir faire le lien entre les statuts et les couleurs, nous avons mis en place un tableau de correspondance qu'il est possible de consulter via un getter. Ce tableau reste local au module.

```
static uint16_t status2color[NB_OF_STATUS] = {
                [TOUCHED]               = YELLOW,
                [WATER]                 = BLUE,
                [SUNK]                  = RED,
                [NOTHING]               = BLACK,
                [SELECTION]             = WHITE,
                [BOAT]                  = GREEN
};
```

Figure 6 - Tableau de conversion entre les statuts et les couleurs

Communication

Pour assurer le transfert d'informations entre les deux joueurs, nous avons décidé d'utiliser une structure contenant les quatre types de données que nous utilisons pour communiquer.

```
struct message {
        uint8_t x_pos;
        uint8_t y_pos;
        uint8_t status;
        uint8_t mode;
};
```

Figure 7 - Structure contenant les données à communiquer

Cette structure est ensuite englobée dans une union, ce qui nous permet par la suite d'encapsuler nos données sous forme de pointeur général dans un message queue.

```
union payload {
        struct message msg;
        void* raw;
};
```

Figure 8 - Union pour l'encapsulation

Nous avons choisi d'utiliser une logique de message queue pour transmettre nos messages car ils permettent de communiquer des données entre plusieurs threads sans rencontrer de problèmes avec les commutations de contexte. Nous avons dû utiliser une union pour notre structure afin de disposer d'un pointeur générique void*, car ce type est bien mieux géré par les messages queues que nous utilisons pour ce projet.

PSEUDO-CODE

Voici le pseudo-code de la logique principale du jeu :

```
void game_process() {
  game_init();

  switch (mode) {
    case IDLE:
      // Until player is ready
      while(!is_ready) {};
      // Init communication module based on player number
      comm_init(channel);
      mode = CONSTRUCTION;
      break;

    case CONSTRUCTION:
      game_boat_placing();
      if (is_ready) {
        if (player_1) mode = ATTACKER;
        if (player_2) mode = ATTACKED;
      }
      break;

    case ATTACKER:
      if (choice_done) {
        comm_send_data(x, y);
      }
      if (response_received) {
        game_process_data(x, y, status);
        if (victory) {
          mode = VICTORY;
        } else {
          mode = ATTACKED;
        }
      }
      break;

    case ATTACKED:
      if (attack_received) {
        game_process_data(x, y);
        if (game_lost) {
          mode = DEFEAT;
        } else {
          mode = ATTACKER;
        }
        comm_send_data(x, y, status);
        break;
      }

    case VICTORY:
      display_victory();
      if (restart) {
        game_init();
        mode = CONSTRUCTION
      }
      break;

    case DEFEAT:
      display_defeat();
      if (restart) {
        game_init();
        mode = CONSTRUCTION
      }
      break;
  }
}
```

ALGORITHMES

Ici sont expliqués quelques algorithmes que nous avons mis en place. Tous n'y sont pas expliqués, seuls les plus intéressants.

Communication

La logique de communication a été conçue afin de communiquer des données dans l'état de jeu concerné. En effet, il serait problématique de recevoir des informations sur une attaque alors que nous sommes censés envoyer une nouvelle attaque.

Afin de résoudre ce problème, nous avons placé quelques conditions lors de la réception de données placée au sein du module de jeu, qui est appelée par thread_nrf_rx :

```
void game_receive_data(struct message msg) {
          if (game_mode == ATTACKED) {
                    if(msg.mode == ATTACKER){
                    attacked_data_rx = game_cell_attacked(msg.x_pos, msg.y_pos);
                    grid_attacked[msg.x_pos][msg.y_pos].cell_status = attacked_data_rx;

                    }
                    //if we lose, it's handled in the state machine
          }
          if (game_mode == ATTACKER) {
                    if(msg.mode == ATTACKED){
                              attacker_data_rx = msg.status;

          grid_attacker[current_position.x_pos][current_position.y_pos].cell_status =
msg.status;
                    }
                    if(msg.mode == DEFEAT){
                              attacker_data_rx = DEFEAT;
                    }
          }
}
```

Au niveau de l'envoi des données depuis le jeu, nous appelons une méthode du module de communication qui place simplement les données désirées sous la forme de pointeur générique dans un message queue.

Pour la réception des données, nous cherchons toute information partagée sur des PIPES. Le cas échéant, nous plaçons les données reçues dans une union payload, que nous renvoyons dans la méthode game_receive_data sous sa forme de structure.

Le thread permettant d'envoyer des données est périodiquement appelé lors de yields. Nous testons si des données sont à envoyer avant de procéder à l'envoi : ceci nous permet d'éviter des erreurs liées aux messages queue lorsqu'ils sont vides :

```
static void thread_nrf_tx(void* p) {
    (void)p;
    //int msg_nr = 0;
    while (1) {
        if (msgq_level(msgq_tx) != 0) {

            union payload payload = {
                .raw = msgq_fetch (msgq_tx),
            };

            char tx_data[NRF_TX_SIZE] = {0, };
            memcpy (tx_data, &payload, sizeof(payload));
            int status = nrf24_write(tx_data, NRF_TX_SIZE);
            if (status == -1) {
                printf("error while sending data...\r\n");
            }
        }
        thread_yield();
    }
}
```

Wheel

Au niveau de son fonctionnement, la roulette interagit avec le programme sous forme d'interruption, grâce à la fonction « wheel_handler ». Cependant, suivant le mode de jeu dans lequel se trouve le joueur, l'interruption n'interagira pas avec le programme afin d'éviter l'incompréhension du joueur.

La fonction « wheel_handler » contient l'algorithme principal de la gestion des déplacements. Pour cela, elle prend en compte la position actuelle du curseur et, en cas de nécessité, l'éventuel bateau à placer.

Voici le fonctionnement de cet algorithme :

Lorsque le joueur est en mode ATTACKER, un curseur blanc s'affiche en position (0, 0). Si le joueur tourne la roulette, une interruption est levée et le curseur se déplace de +1 ou -1 en X, en fonction de la direction. Enfin, on remet à jour la cellule sur laquelle il était en prenant compte son état.

Si le joueur est en mode CONSTRUCTION, le curseur devient le bateau à placer. Il est possible de modifier son orientation grâce au bouton 2. Lorsqu'un bateau doit être placé, l'algorithme prend en compte sa taille et son orientation afin d'interdire au joueur de placer un bateau avec une partie de celui-ci hors de la grille.

Nous avons quelques améliorations que nous pourrions porter sur cet algorithme :

L'algorithme en lui-même n'est pas des plus performants, car il est codé avec une marche à suivre différente en fonction de chaque cas. Nous aurions pu réaliser un algorithme plus générique, ne prendrait plus en compte les position (x ; y) mais fonctionnant uniquement avec un compteur. Nous avons aussi pensé à ajouter un thread avec un listener afin de réaliser l'algorithme dans le thread lorsque la position du curseur est modifiée plutôt que directement dans l'interruption.

Malheureusement, ayant pris pas mal de retard avec d'autresW modules de ce projet, nous n'avons pas eu le temps d'améliorer celui-ci. Etant donné que ce module était fonctionnel, nous avons préféré terminer le reste afin d'avoir un délivrable fonctionnel plutôt que quelques modules réalisés à la perfection mais inutilisables au final.

Bateaux

Le placement des bateaux doit stocker un pointeur sur ledit bateau dans les cases concernées, et sa position doit passer par une validation. En effet, il est impossible de placer un bateau sur la partie d'un autre bateau.

Pour ce faire, nous avons mis en place ce code :

```
bool is_pos_valid = true;

for (int i = 0; i < boat->length; i++) {
          if (boat->orientation == HORIZONTAL) {
                    if
(grid_attacked[current_position.x_pos+i][current_position.y_pos].boat) {
                              buttons_set_state(BUTTONS_1, false);
                              game_choose_boat_placement(boat);
                              is_pos_valid = false;
                    }
          }

          if (boat->orientation == VERTICAL) {
                    if
(grid_attacked[current_position.x_pos][current_position.y_pos+i].boat) {
                              buttons_set_state(BUTTONS_1, false);
                              game_choose_boat_placement(boat);
                              is_pos_valid = false;
                    }
          }
}
```

Nous testons donc toutes les cases concernées par le bateau placé: si un bateau y est déjà positionné, nous portons un boolean à false, ce qui ne permet pas de valider le navire sur la case et de passer à l'étape suivante. Le joueur reste donc dans la validation de la position.

Lorsque l'on touche un bateau, nous devons calculer quel emplacement de celui-ci est concerné car chaque case sur lesquelles il est placé contiennent un pointeur sur lui.

Dans un premier temps, nous devons trouver quelle case du navire est touchée en se basant sur la position de l'attaque ainsi que sur l'origine du bateau (qui est toujours au point le plus en bas à gauche du bateau) :

```
int boat_part = 0;

// method to know wich part of the boat is touched
if (x > boat->x_pos && boat->orientation == HORIZONTAL){
                    boat_part = x-boat->x_pos;
}
if (y > boat->y_pos && boat->orientation == VERTICAL){
                    boat_part = y-boat->y_pos;
}
```

Lorsque nous avons obtenu la partie du bateau, nous devons actualiser l'état du bateau. Pour ce faire, nous fonctionnons avec un bitset présente au sein de la structure « boat ». Selon la part du bateau, nous appliquons un masque adapté :

```
uint8_t mask = 0;
switch (boat_part){
case 0 :
          mask = 0x001;
          break;

case 1:
          mask = 0x02;
          break;

case 2:
          mask = 0x04;
          break;
}

boat->cells |= mask;
```

Ensuite, nous vérifions si le bateau est coulé. Etant donné que son bitset indique uniquement des cases touchées, il suffit de vérifier la valeur dudit bitset et de la comparer à la taille du bateau :

```
if (boat->cells == 3 && boat->length == 2){
          boat->is_sunk = true;
          return SUNK;
}
if (boat->cells == 7 && boat->length == 3){
          boat->is_sunk = true;
          return SUNK;
}
```

Attaqué et attaquant

L'algorithme en pseudo-code est déjà affiché dans la partie « Pseudo-code » de ce document.

Affichage

L'écran de la cible doit être actualisé de façon régulière, selon l'état du jeu. Les modes attaquant et attaqué s'affichent de façon différentes (ronds ou carré), de même que pour le placement des bateaux en mode construction. De plus, il faut tenir compte de l'état de chaque case du jeu, car la couleur de ces dernières est différente pour chaque situation.

```
case ATTACKER:
          for (int i = 0; i < NB_OF_ROWS; i++) {
                    for (int j = 0; j < NB_OF_COLS; j++) {
                              cell_status = game_get_cell(i, j, ATTACKER);
                              display_modifie_cell(i, j, cell_status, ATTACKER);
                    }
          }
          break;
case ATTACKED:
          for (int i = 0; i < NB_OF_ROWS; i++) {
                    for (int j = 0; j < NB_OF_COLS; j++) {
                              cell_status = game_get_cell(i, j, ATTACKED);
                              display_modifie_cell(i, j, cell_status, ATTACKED);
                    }
          }
          break;
case CONSTRUCTION:
          for (int i = 0; i < NB_OF_ROWS; i++) {
                    for (int j = 0; j < NB_OF_COLS; j++) {
                              cell_status = game_get_cell(i, j, ATTACKED);
                              display_modifie_cell(i, j, cell_status, CONSTRUCTION);
                    }
          }
}
```

La méthode display_modifie_cell() prend aussi en compte l'état du jeu pour actualiser l'affichage de chaque case.

Réalisation

ARCHITECTURE

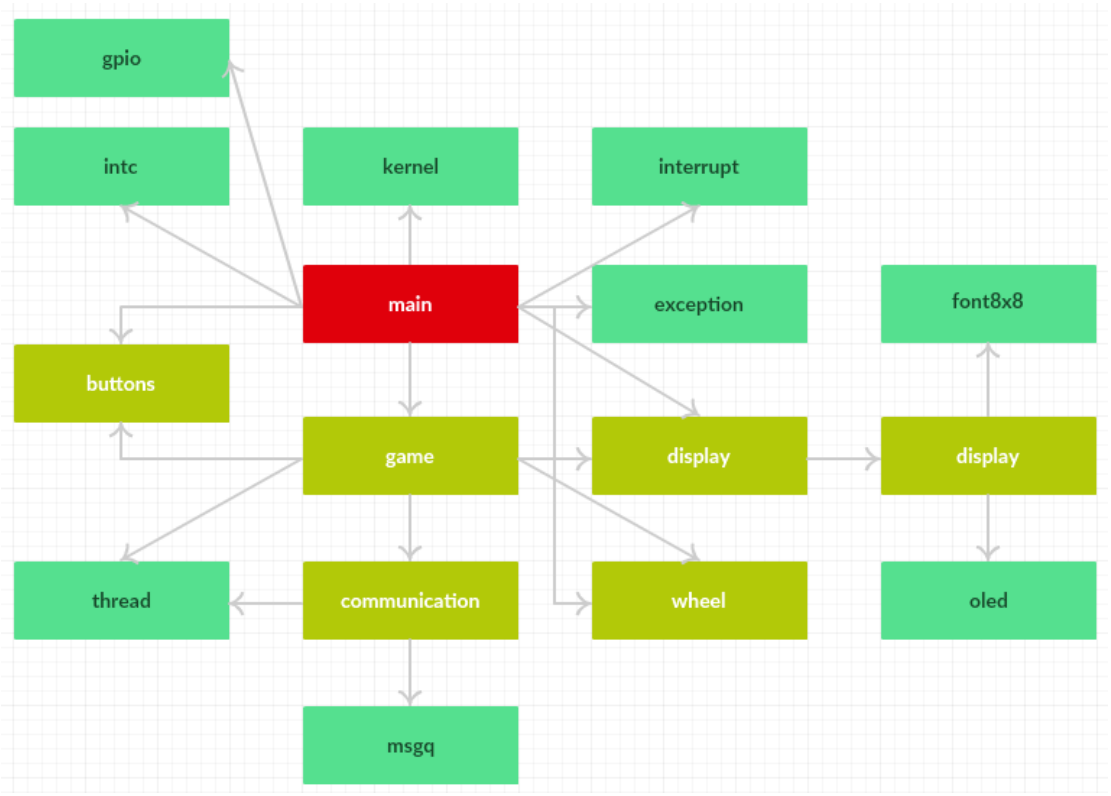
L'architecture a été décrite précédemment. Etant donné que le langage C n'est pas orienté objet, nous avons fonctionné par modules. Cette séparation a été faite selon chaque utilisation spécifique du code : chaque module contient uniquement des fonctions leur étant propres. Les différentes données sont aussi stockées selon leur portée.

Nous n'avons pas utilisé de variables globales, car leur utilité est très restreinte. Cette façon de faire peut entrainer quelques difficultés au niveau des compilations, ce que nous voulons éviter un maximum.

Vu que nous travaillons avec des données locales, nous avons mis en place plusieurs getters / setters afin de modifier des données de façon efficace et robuste.

STRUCTURE DU CODE

Nous avons réutilisé des modules précédemment conçus lors d'antérieures séances de travail pratique, comme par exemple gpio, intc, interrupt, exception ou encore oled. Nous avons aussi utilisé les modules propres au système d'exploitation basique coopératif, fournis par le professeur.



Tests et validation

Nous avons lancé plusieurs tests afin de nous assurer que notre programme soit robuste et ne connaisse pas de problème.

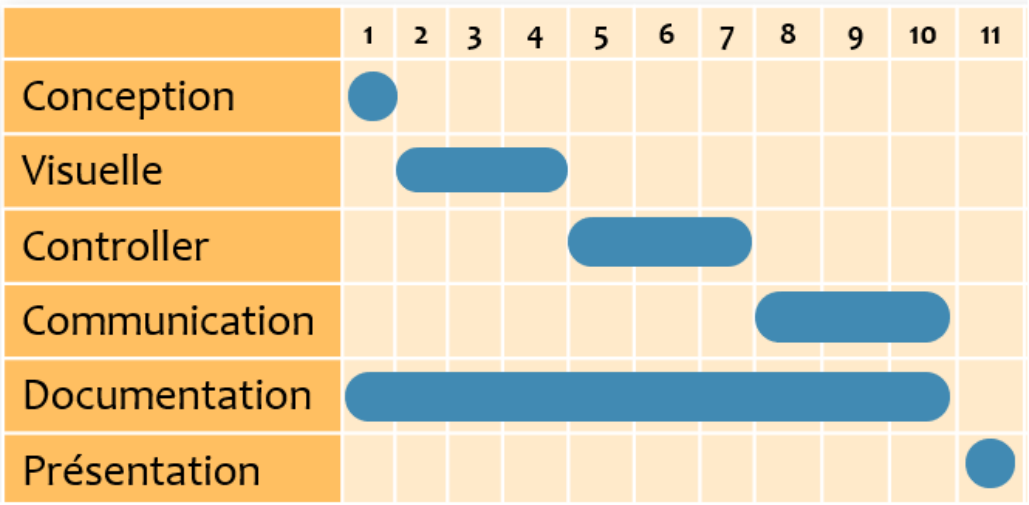
Tous ces tests ont été réalisés en binôme.

Sujet	Etat	Remarque
Tests du matériel	Validé	Intégralité des périphériques
Tests des librairies	Validé	Si nous rencontrions des problèmes d'utilisation
Envoi de messages	Validé	Utilisation nRF24
Robustesse de la machine d'états	Validé	Essai de manipulation non-prévues
Envoi de données de jeu	Validé	Voir « Problèmes » dans conclusion
« Garde-fou » pour le traitement des communications	Validé	Eviter qu'un message attaqué arrive en donnée dans un faux mode
Gestion du déplacement de la wheel	Validé	Ne pas sortir de la grille
Vérification du mode initial	Validé	Eviter deux états pareils des deux côtés (blocage)
Affichage des bons états des cellules	Validé	Si une case est par exemple touchée, il y a bien une partie d'un bateau
Envoi de mise à jour	Validé	Nous envoyons plusieurs fois les messages pour nous assurer de leur réception
Superposition de bateaux	Validé	Nous n'avions pas pensé à la superposition des navires
Mise à jour de l'affichage	Validé	Gestion de cet aspect
Exceptions traitées correctement	Validé	Nous avions obtenu plusieurs ARM Data Abort durant le développement
Le système est bien coopératif	Validé	Sans ceci, pas de communication
Bon fonctionnement du jeu	Validé	Même après les parties finies

L'envoi de message a causé quelques problèmes lorsque nous travaillions à côté d'autres groupes. La raison de ceci était que nous avions oublié de changer notre bande de fréquence.

Planification

Voici la planification que nous avions initialement conçue :



Les parties de conception, de visuel et de documentation ont été commencées et terminées sans délai (sauf concernant quelques adaptations). En revanche, nous avons commencé à prendre du retard à partir de la 5e semaine. Il y a eu plusieurs causes de ce retard, les principales étant que nous avions du travail à côté des cours et que nous avons manqué une séance de travail pratique pour des raisons médicales.

Nous avons néanmoins réussi à terminer ce programme pour la fin de la 10e semaine. Nous avons mis les bouchées doubles afin de clôturer ce jeu dans les temps. Nous avons travaillé chaque soir de la 10e semaine, ainsi que lors du congé de l'ascension.

Nous avons rencontré un grand nombre de problèmes, surtout au niveau de la communication. Cet aspect nous a énormément retardés dans la conception de ce projet.

Finalement, nous avons appris de nos erreurs et nous ferons attention à toute prise de retard conséquente.

Conclusion

DIFFICULTES

Nous avons rencontré un grand nombre de difficultés. Premièrement, nous avions commencé notre projet en stockant les données utiles de façon globales. Le compilateur nous levait un nombre de warnings très élevé, car nous avions des include d'header-files à deux sens. Cela nous a causé quelques difficultés pour les corriger.

Ensuite, nous obtenions souvent des exceptions « ARM Data Error », parfois même sans changer notre code entre deux tests. Après quelques essais et débogages, nous avions trouvé les sources de ces problèmes.

De plus, le module de communication nous a causé beaucoup de difficultés. Nous avons eu beaucoup de peine pour appréhender la librairie fournie ainsi que pour définir et construire l'envoi et la réception de messages. De plus, l'utilisation de messages queue n'a pas aidé : ces derniers nous ont causé quelques difficultés, notamment lors de tests (une fois, nous ne passions pas par l'initialisation du msgq utilisé au sein du jeu).

Toujours dans ce module, nous transportions des données trop larges pour le type utilisé, ce qui nous a causé une redéfinition de plusieurs fonctionnalités déjà existantes. En effet, notre énumération de statut de cellule prenait directement des codes couleurs de RGB565. Or, le module nRF ne supporte seulement 32 bits de données et nous avions besoin de transporter 4 données : le code couleur était trop lourd à transporter et était donc faussé.

Enfin, nous avons eu quelques problèmes pour l'état « START », actuellement supprimé du jeu. Ce mode était utilisé juste après l'état CONSTRUCTION et servait essentiellement à définir de façon aléatoire le joueur 1 et le joueur 2. Nous rencontrions beaucoup de difficultés à implémenter une logique robuste : nous rencontrions énormément de malfonctions. Finalement, nous avons supprimé cet état et avons défini les joueurs 1 et 2 selon leur choix initial dans le mode IDLE. Ce comportement étant beaucoup plus simple et naturel (pour les jeux de société, on s'entend pour définir qui commence), nous l'avons implémenté.

Une partie des solutions que nous avons implémentées ont été trouvées grâce à la consultation de notre professeur de Systèmes Embarqués.

AMELIORATIONS

Quelques améliorations de code pourraient être apportées.

Au niveau des données du jeu, nous aurions pu regrouper les deux grilles attaquant et attaqué. Ceci permettrait d'améliorer l'utilisation de la cache et de la mémoire.

Ensuite, nous aurions pu pointer les cases du jeu par le biais d'une seule et unique structure. Actuellement, certaines positions sont indiquées par deux integer.

De plus, la wheel aurait pu être mieux traitée afin d'améliorer son temps de réaction, par exemple en plaçant le traitement d'interruption dans un thread. Nous n'avons pas eu le temps de changer cet aspect, mais elle fonctionne tout de même correctement.

Le rafraichissement de l'écran aurait pu être fait par un timer du BeagleBone Black. Nous avons fait différemment pour simplifier l'ensemble du code.

COMPTE-RENDU

Ce projet intégré a été très bénéfique pour nous. Nous avons appris à gérer un projet de développement logiciel de A à Z en groupe.

La collaboration sur ce projet a été essentielle et très efficace. Nous avons réussi à agir de façon optimale pour travailler sur différents aspects. Nous avons travaillé de façon égale, personne n'a plus travaillé que l'autre.

L'utilisation d'un système de versioning comme Git a aussi été très avenue, même si nous aurions souhaité mieux maîtriser les mécanismes des merges et de banches. Certains messages de commits ne sont pas très explicites, car nous utilisions cet outil pour se partager le code lors du développement. Cette méthode nous permettait de n'apporter des modifications que sur un seul ordinateur. Mais chaque modification majeure est bien accompagnée d'un message de commit clair et explicite.

Malgré les nombreux problèmes rencontrés et le retard que nous avons pris, nous avons tout de même pris plaisir à développer ce jeu. C'était une expérience très agréable et utile dans le cadre du cours de Systèmes Embarqués.

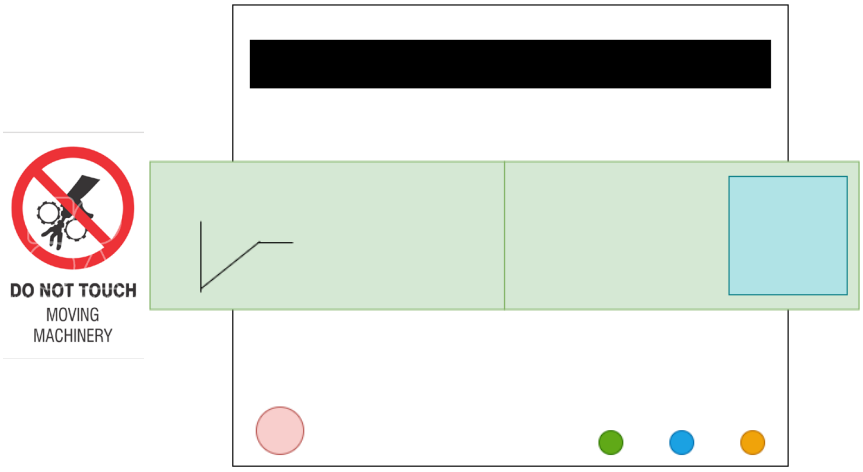
Table des illustrations

Fribourg, le 10 juin 2019

Loïc Guibert Raphaël Pittet

Annexe : Mode d'emploi

Bataille navale
Mode d'emploi



L'écran : le jeu se déroule sur cette partie de l'appareil. Les grilles de jeu ainsi que les textes seront affichés sur cet écran.

La roulette: La roulette est l'appareil qui permet au joueur de déplacer le curseur sur la grille.

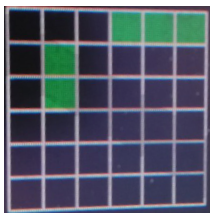
Bouton 2: Le bouton 2 sera utilisé afin d'initialiser la partie, mais aussi pour la rotation des bateaux lors du placement.

Bouton 1: le bouton 1 sera utilisé pour la validation en général. Que se soit pour le placement des bateaux ou pour le choix de l'attaque. Aussi utilisé pour relancer la partie suivante.

Bataille navale
Mode d'emploi

Phase d'initialisation

Lorsque le jeu commence, le choix du device est important. Cliquez sur le bouton 1 pour être attaquant ou le bouton 2 pour être l'attaqué.

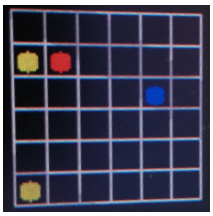


Avant de commencer à jouer il faut placer ses beateaux.
Bouton 1 pour valider.
Bouton 2 pour faire une rotation.
La roulette pour déplacer les bateaux.

Phase de jeu

Mode ATTAQUANT

Lorsque le joueur est dans ce mode, il peux choisir la position de son tire.



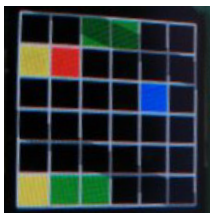
les cases qui ont déjà été tirées dessus durant les tours précédents sont remplies par des points.

La couleur des points signifie:
BLEU : à l'eau.
ORANGE : touché.
ROUGE : coulé.

Le curseur de sélection est aussi un point de couleur BLANCHE.

Mode ATTAQUÉ

Lorsque le joueur est dans ce mode, il ne peut rien faire. il doit attendre que l'attaquant fasse son choix.



Le code couleur des cellules est le même qu'en mode attaquant c'est-à-dire:

BLEU : à l'eau.
ORANGE : touché.
ROUGE : coulé.