Hes·so FRIBOURG/ FREIBURG
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz

Systèmes embarqués

Projet Intégré

Rapport de conception

Auteur :

Romain Froidevaux, T-2a, Groupe E romain.froidevaux@edu.hefr.ch

Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

Avril 2014

Table des matières

I. Introduction

Dans le cadre du cours de Systèmes Embarqués 2, nous avons pour objectif de réaliser un projet par groupe de 3 personnes. Cela nous permettra de mettre en application toute la matière vue au cours théorique ou en travaux pratiques.

Chacun des membres du projet a des tâches bien définies. Ce rapport a pour but de nous faire réfléchir sur la conception nos différentes méthodes pour nous mener plus facilement à la phase de réalisation.

Mes points sont les suivants :

- Afficher les différentes vues sur l'écran LCD de la cible
- Implémenter les commandes de la Shell

Ainsi à la fin de cette analyse de conception, j'aurai une vue précise de mes points à implémenter.

II. Rappel du projet

Toute la classe a une base commune pour le projet avec des points obligatoires :

- Affichage d'informations sur l'écran LCD
- Lancement des applications via l'écran tactile
- Affichage en temps réel de la température sur des 7 segments
- Réalisation d'un chronomètre avec les boutons poussoirs *(via interruptions)*
- Implémentation de la commande Ping pour vérifier la connectivité à distance
- Création d'un jeu multi-joueurs au travers du réseau
- Implémentation d'une Shell pour lancer/stopper/lister les applications
- Utilisation d'un noyau temps réel coopératif

Dans mon groupe, composé de L. Gremaud, D. Rossier et moi-même, nous avons choisi de développer le jeu de la bataille navale en multi-joueurs. Nous devrons donc gérer les tours de jeu et afficher en alternance la grille du joueur 1 et celle du joueur 2.

Comme fonctionnalités optionnelles qui seront développées selon notre avancement dans le projet, nous avons rajouté :

- Deuxième jeu : Tic Tac Toe
- Application de visualisation des scores
- Utilisation d'un noyau préemptif

Pour connaître la répartition des tâches précise, se référer au cahier des charges.

III. Architecture générale de notre projet

Dans le rapport d'Architecture et de Spécification, nous avons détaillé la structure de notre développement, dont voici le schéma des parties majeures.

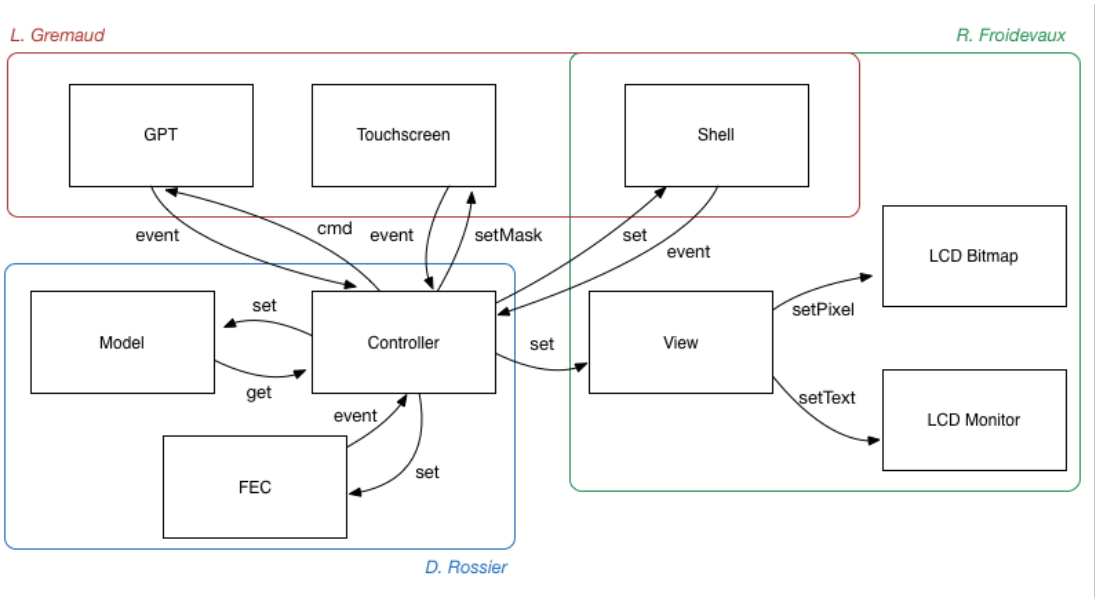

Fig. 01 : Modèle de développement

Grâce à une telle structure de développement, nous pouvons développer en parallèle nos différents éléments et de manière générique, afin de pouvoir réutiliser ceux-ci ultérieurement dans d'autres projets.

IV. Interfaces graphiques

a. Contexte

Les interfaces graphiques sont relativement semblables les unes des autres. Pour ne pas devoir recoder plusieurs fois les mêmes éléments, il a été décidé *(Cf. documentation d'Architecture et de Spécifications)* de réaliser une première classe servant d'API pour agir directement sur l'écran LCD et dessiner les éléments désirés.

Ensuite, les classes de niveau supérieur appelleront ces différentes méthodes de l'API pour créer les vues désirées.

b. Description des classes et méthodes

1. Driver : lcdc

Le driver de l'écran est repris tel quel du TP05 pour les fonctions de base.
Une méthode sera rajoutée pour définir l'adresse mémoire pour le bitmap afin de réaliser simplement l'opération `swap()` au cours du jeu de la bataille navale permettant de passer de l'affichage de la grille du joueur 1 au joueur 2 et inversement.

```
/**
 * Set the bitmap
 *
 * Set the bitmap to use and display it into the LCD
 *
 * @param bitmap the bitmap to use
*  @return the current used bitmap
 */
extern enum view_bitmap lcdc_set_bitmap (enum view_bitmap bitmap);

//Depending on the selected bitmap, set the bitmap memory address into lcdc_bitmap variable
```

2. API : lcddisplay

Cette classe `lcddisplay` fait office d'API pour toutes les classes supérieures permettant ainsi de dessiner simplement les différents éléments nécessaires aux vues.

```
/**
 * Initialize the LCD display
 *
 * First call of the API, to initialize the LCD display and display a white screen
 */
extern void lcddisplay_init();
```

```
/**
 * Print a grid
 *
 * Print a grid on the LCD screen
 *
 * @param color the color of the grid
 * @param size the size of the grid
 */
extern void lcddisplay_print_grid(enum view_color color, enum view_grid_size size);

//Print a grid by using the print_zone method implemented into the same source file
```

```
/**
 * Print an image
 *
 * Print an image on the LCD screen
 *
 * @param start_pos the top-left position of the image
 * @param icon structure of the XPM image
 */
extern void lcddisplay_print_image(struct utils_position start_pos, struct xpm_image icon);
```

```
/**
 * Print text
 *
 * Print a string test in the LCD display
 *
 * @param string array of chars, string text to display
 * @param start_pos high-left position of the first char
 * @param color color of the text
 * @param whiteBackground color background in white if set; else, don't modify the pixel
 */
extern void lcddisplay_print_text(char* string, struct utils_position start_pos, enum
view_color color, bool whiteBackground);
```

```
/**
 * Print zone
 *
 * Print a colored zone (rectangular)
 *
 * @param start_pos top-left position of the zone
 * @param stop_pos bottom-right position of the zone
 * @param color color of the zone
 */
extern void lcddisplay_print_zone(struct utils_position start_pos, struct utils_position
stop_pos, enum view_color color);
```

Ainsi, avec ces 4 méthodes, il sera possible de dessiner toutes les vues nécessaires à notre projet.

3. Vue : view_home

La vue pour l'écran d'accueil est composée principalement d'éléments fixes, hormis le chronomètre et le thermomètre. Ainsi, deux modificateurs devront être implémentés pour mettre à jour ces valeurs sur l'écran.

```
/**
 * Initialize view
 *
 * Initialize the home screen view with basics elements
 */
extern void view_home_init();
```

```
/**
 * Set time
 *
 * Set the time value of the chronometer
 * @param time current time value of the chronometer
 */
extern void view_home_set_time(struct utils_time time);
```

```
/**
 * Set temprature
 *
 * Set the temperature value of the LM75
 * @param temperature current temperature of the LM75 thermometer
 */
extern void view_home_set_temperature(struct utils_temperature temperature);
```

4. Vue : view_battleship

La vue de positionnement des bateaux requière principalement deux setters. Le premier pour colorer une case voulue *(afin d'afficher où le bateau est positionné)*, et le second afin de définir l'état du bouton `Play`. En effet, celui ne sera actif que lorsque tous les bateaux seront correctement positionnés.

```
/**
 * Initialize Battleship Positioning view
 *
 * Initialize the Battleship Positioning view with basics elements
 */
extern void view_battleship_init();
```

```
/**
 * Set case
 *
 * Set grid case with different elements
 * @param position the position of the case into the grid
 * @param case_type the type of the case
 */
extern void view_battleship_set_case(struct utils_position position, enum
view_battleship_case case_type);

// The enum case_type is used to make it general. But in this view, only the Ship_present
// value of the enum will be used to displays where is the ship.
```

```
/**
 * Set play button state
 *
 * Set the state of the play button
 * @param state the state of the button
 */
extern void view_battleship_set_play_button_state(enum utils_button_state state);
```

5. Vue : view_battleship_play

La vue de jeu pour le joueur courant à la bataille navale comporte deux compteurs de victoires. Il est donc nécessaire d'implémenter un setter pour aller mettre à jour ces valeurs.
Au même titre que pour la vue `view_battleship`, le bouton `Validate` ne sera actif que lorsque la case ciblée sera correctement positionnée dans la grille. Il est donc nécessaire d'y lier également un setter pour définir l'état du bouton.

```
/**
 * Initialize Battleship Play view
 *
 * Initialize the Battleship Play view with basics elements
 */
extern void view_battleship_play_init();
```

```
/**
 * Set score
 *
 * Set the score (win and lose points)
 * @param hit counter of win for the current play
 * @param missed counter of lose for the current play
 */
extern void view_battleship_play_set_score(uint8_t hit, uint8_t missed);
```

```
/**
 * Set validate button state
 *
 * Set the state of the validate button
 * @param state the state of the button
 */
extern void view_battleship_play_set_validate_button_state(enum utils_button_state state);
```

6. Vue : view_battleship_wait

La vue d'attente pour le jeu de la bataille navale est sensiblement identique à celle où le joueur doit sélectionner une case. Elle bénéficie donc des mêmes méthodes, hormis celle faisant référence au bouton `Validate`, qui n'est pas présent.

```
/**
 * Initialize Battleship Wait view
 *
 * Initialize the Battleship Wait view with basics elements
 */
extern void view_battleship_wait_init();
```

```
/**
 * Set score
 *
 * Set the score (win and lose points)
 * @param hit enemy counter of hits for the current play
 * @param missed enemy counter of missed for the current play
 */
extern void view_battleship_wait_set_score(uint8_t hit, uint8_t missed);
```

7. Vue : view_scores

La vue étant statique, elle ne possède publiquement qu'une méthode d'initialisation prenant en paramètre les scores actuels.

```
/**
 * Initialize view
 *
 * Initialize the score screen view with basics elements
 * @param player_score score of the player
 * @param enemy_score score of the enemy
 * @param button_type the type of the button
 */
extern void view_score_init(uint8_t player_score, uint8_t enemy_score, enum
view_score_button_type button_type);
```

Au sein du fichier source par contre, il sera judicieux de ne pas tout réaliser au sein de la même méthode, mais de procéder au découpage suivant :

```
void view_score_init(uint8_t player_score, uint8_t enemy_score, enum view_score_button_type
button_type) {

	//Reinitialize the screen
	//Print elements of the view

	view_score_button_print(button_type);
	view_score_text_print();
	view_score_values_print(player_score, enemy_score);

	if (player_score > enemy_score)
		view_score_image_print(win);
	else if (player_score < enemy_score)
		view_score_image_print(lose);
	else
		view_score_image_print(equality);
}
```

```
void view_score_button_print(enum view_score_button_type button_type) {
	//Print the replay or reset button depends the button_type value
}
```

```
void view_score_values_print(uint8_t player_score, uint8_t enemy_score) {
	// Print the player's score value
	// Print the enemy's score value
}
```

```
void view_score_text_print() {
	//Print the global scores label
	//Print the Your score label
	//Print the Your enemy's score label
}
```

```
void view_score_image_print(enum view_score_image score_image) {
	//Print the image depends if the player win, lose or be in equality
}
```

8. Vue : view_shell

La vue du Shell est celle qui nécessite du plus de fonctions. Nous avons en effet voulu prendre en compte le fait que l'utilisateur peut supprimer un caractère ou une ligne en cas d'erreur.

```
/**
 * Initialize Shell view
 *
 * Initialize the Shell view with basics elements
 */
extern void view_shell_init();
```

```
/**
 * Print chars
 *
 * Print chars into the Shell space
 * @param s string of chars to print
*/
extern void view_shell_print_chars(char* s);

// The string s can contains a '\n' character to make a carriage return. If the string is too long to
// be displayed in one line, this method will continue to write the rest of the string on the next
// line
```

```
/**
 * Delete char
 *
 * Delete the last char into the Shell space
 */
extern void view_shell_delete_char();
```

```
/**
 * Delete line
 *
 * Delete the current Shell line
 * @param shell_intro print a shell intro for new entry if true
 */
extern void view_shell_delete_line(bool shell_intro);
```

```
/**
 * Print new line
 *
 * Create a new line (Carriage return)
 * @param shell_intro print a shell intro for new entry if true (sh-groupe_e #)
 */
extern void view_shell_print_new_line(bool shell_intro);
```

```
/**
 * Clear
 *
 * Clear all printed lines into the Shell
 */
extern void view_shell_clear();
```

Au niveau de la réalisation, l'utilisation d'un type `utils_position` global paraît judicieuse. Il sera en effet beaucoup plus facile de se référer au pointeur pour chaque opération sur un caractère.

Ainsi : > Ecrire un caractère déplace le pointeur de 8 pixels vers la droite
> Supprimer un caractère déplace le pointeur de 8 pixels vers la gauche
> Supprimer une ligne remet le pointeur au début de la ligne courante

Tout en prenant compte des cas limites bien sur *(fin de ligne, début de ligne, fin de la Shell, etc ...)*.

Lorsque toute la Shell sera remplie sur l'écran, il a été décidé de la remettre à zéro *(réimpression d'une zone noire par dessus)* et de replacer le curseur tout en haut à droite.

9. Vue : view_tictactoe

La vue pour le jeu du Tic Tac Toe est semblable (au niveau des méthodes) à celles pour le jeu de la bataille navale. Ainsi, il doit être possible d'initialiser la vue, de définir le contenu de chacune des cases de la grille, de setter la valeur du champ de texte dédié au tour du jeu et définir l'état du bouton valider. Celui-ci ne doit être actif que lorsque le joueur a placé sont élément dans la grille.

```
/**
 * Initialize TicTacToe view
 *
 * Initialize the TicTacToe view with basics elements
 */
extern void view_tictactoe_init();
```

```
/**
 * Set case
 *
 * Set a grid case with different elements
 * @param position the position of the case into the grid
 * @param case_type the type of the case
 */
extern void view_tictactoe_set_case(struct utils_position position, enum view_tictactoe_case
case_type);
```

```
/**
 * Set text
 *
 * Set the text of the tournaments label
 * @param turn turn of player
 */
extern void view_tictactoe_set_turn_text(enum view_tictactoe_turn turn);
```

```
/**
 * Set validate button state
 *
 * Set the state of the validate button
 * @param state the state of the button
 */
extern void view_tictactoe_set_validate_button_state(enum utils_button_state state);
```

V. Implémentation des commandes Shell

a. Contexte

Depuis un ordinateur connecté sur le port série de la cible, il doit être possible de lister, lancer et arrêter les applications. L. Gremaud est en charge d'interpréter les commandes entrées par l'utilisateur dans le terminal, et moi-même de développer les méthodes permettant ensuite de traiter la commande et d'exécuter les actions nécessaires.

b. Description des méthodes

```
/**
 * Initialize Shell back-end
 *
 * Initialize the Shell back-end with basic operations
 */
extern void shell_program_init();
```

```
/**
 * Ping
 *
 * Ping an IP address
 * @param ip_address the destination IP address of the ping
 * @return 0 on success, a negative number on failure
 */
extern int shell_program_ping(struct ipaddr ip_address);

// The God of the FEC David Rossier will provide the low-level ping function. The goal's function
// is to interpret the result and return it to the user and the Shell program (via LCD)
```

```
/**
 * Set IP
 *
 * Set the IP address of the embedded system
 * @param ip_address the new IP address of the embedded system
 * @return 0 on success, a negative number on failure
 */
extern int shell_program_set_ip(struct ipaddr ip_address);

// The IP address of the embedded system will be stored into the Model. Accessible via the
// function ip_set_ip_address(struct of the IP address)
```

```
/**
 * Start an application
 *
 * Start a process (application) on the embedded system
 * @param app the application to start
 * @return 0 on success, a negative number on failure
 */
extern int shell_program_app_start(enum model_application app);

// This function will call the function into the Kernel to give the order to launch the application
// given by parameter
```

```
/**
 * Stop an application
 *
 * Stop a process (application) on the embedded system
 * @param app the application to stop
 * @return 0 on success, a negative number on failure
 */
extern int shell_program_app_stop(enum mode_application app);

// As the shell_program_app_start function, this one will call the function into the Kernel
// to give the order to stop the application given by parameter
```

```
/**
 * List all applications
 *
 * List all applications of the embedded system
 * @return array of chars with all available applications
 */
extern char * shell_program_list_app_all();

// The list of all applications available on this embedded system is stored on the Model. This
// method will get the list and pass it to the Shell view.
```

```
/**
 * List running applications
 *
 * List running applications on the embedded system
 * @return array of chars with all running applications
 */
extern char * shell_program_list_app_run();

// The kernel knows the current running app. This function will get it and call the Shell
// view to display it.
```

```
/**
 * List halted applications
 *
 * List halted applications on the embedded system
 * @return array of chars with all halted applications
 */
extern char * shell_program_list_app_halt();

// This function will call the kernel to get all app (process) in WAITING mode and return the list
// to the Shell view
```

```
/**
 * Help
 *
 * Displays all available commands into the shell
 * @return array of chars with all available commands
 */
extern char * shell_program_help();

// This function will list all available shell commands in out embedded system (the list is stored
// into the Model) and return it to the Shell View.
```

```
/**
 * Chrono start
 *
 * Start the chronometer
 * @return 0 on success, a negative number on failure
 */
extern int shell_program_chrono_start();

// This function will send the order to the Kernel to set the chrono process state as READY to
// start it asap using the GPT.
```

```
/**
 * Chrono stop
 *
 * Stop and reset to zero the chronometer
 * @return 0 on success, a negative number on failure
 */
extern int shell_program_chrono_stop();

// This function will just call the stop function of the chrono.
```

```
/**
 * Chrono pause
 *
 * Pause the chronometer
 * @return 0 on success, a negative number on failure
 */
extern int shell_program_chrono_pause();

// This function will just call the pause function of the chrono.
```

```
/**
 * Chrono show
 *
 * Displays into the shell the current value of the chronometer
 * @return utils_time the current chronometer value
 */
extern struct utils_time shell_program_chrono_show();

// This function will get the current value of the chrono to send it to the Shell View.
```

VI. Structures et énumérations communes

Nous avons très vite remarqué lors de cette analyse de conception, que bon nombre de structures ou énumérations doivent être utilisées par plusieurs d'entre nous. Pour faciliter leur utilisation, nous avons décidé de créer un ficher `utils.h` dans lequel nous allons tous les regrouper.

Voici celles qui sont utilisées par la vue ou le programme du Shell :

```
/**
 * The position
 * A position according to the axes X and Y
 */
struct utils_position {
        uint16_t x;         //!< X axis value
        uint16_t y;         //!< Y axis value
};
```

```
/**
 * A size according to the Height and Width values
 */
struct utils_size {
        uint16_t h;         //!< Height value
        uint16_t w;         //!< Width value
};
```

```
/**
 * A time set (hours, minutes and seconds)
 */
struct utils_time {
        uint8_t h;          //!< Hour value
        uint8_t m;          //!< Minutes value
        uint8_t s;          //!< Seconds value
};
```

```
/**
 * A temperature (XX.YY °C)
 *
 * Number range from -128.99 to 127.99
 */
struct utils_temperature {
        int8_t bdv;         //!< Number before dot
        uint8_t adv;        //!< Two digit after dot
};
```

```
/**
 * States in which a button may be
 */
enum utils_button_state {
        active,             //!< Active and visible
        inactive,           //!< Visible but inactive
        invisible           //!< Invisible and inactive
};
```

VII. Définition de la stratégie de vérification et validation

Afin de pouvoir tester et valider les différentes méthodes en phase de réalisation, il est nécessaire d'utiliser une méthodologie stricte.

- Une fois une méthode développée, écrire sur papier **toutes les combinaisons** différentes de paramètres possibles et décrire pour chacune le comportement attendu.
- Ensuite, tester sur la cible chacun des cas :
  - Si le résultat est conforme à celui attendu, le test est considéré comme réussi.
  - S'il n'est pas conforme, corriger le problème.
    Attention corriger un problème peut en apporter un autre. Il est donc nécessaire de reprendre tous les cas décrits depuis le début !

Appliquer cette logique de bas en haut. Pour les vues par exemple, il faut commencer par tester le driver, suivi de l'API, puis l'initialisation des vues, pour ainsi valider chacun des étages du programme.

VIII. Conclusion

Au travers de cette réflexion liée à la conception de notre projet, j'ai maintenant une vue plus précise quant à l'implémentation de mes différentes méthodes.

Afin de bien définir lesquelles sont nécessaires, j'ai tout au long de la conception codé des petits squelettes de code, principalement au niveau des interfaces graphiques.

A la fin de ce rapport, j'en ai profité pour compléter les différents squelettes et arriver à afficher les vues de manière correctes sur la cible, telles que nous les avions dessinées. Ainsi, cela nous facilitera le travail pour les tests visuels des différents éléments qu'il nous reste encore à implémenter.

Signature

Froidevaux Romain