UNER Facultad de Ingeniería

Trabajo Práctico VII

Fundamentos de Programación de Sistemas Embebidos
Especialización en Sistemas Embebidos
Cohorte 2018

Objetivos

- Comprender la aplicación de punteros a funciones, estructuras y arreglos en la implementación de aplicaciones para sistemas embebidos.
- Logre implementar aplicaciones que utilicen estructuras de datos para la implementación de interfaces de usuario
- Aplique el uso de modificadores de almacenamiento.
- Aplique diferentes técnicas para la implementación de módulos en aplicaciones de sistemas embebidos.

Herramientas requeridas

Para el desarrollo del presente trabajo práctico deberá contar con:

- Equipo PC utilizando Linux como sistema operativo.
- Paquete de compilación de GNU GCC para ARM incluyendo binutils-arm-none-eabi, gcc-arm-none-eabi y make.
- Open OCD
- Eclipse IDE for GNU ARM (versión recomendada Oxygen 3a)
- Placa de desarrollo EDU-CIAA NXP.

Interfaz de comandos

Las interfaces de comandos basadas en texto constituyen una herramienta de interacción con sistemas embebidos de diferente tipo, desde configuración de equipos de comunicaciones hasta monitoreo remoto.

Desde los primeros sistemas informáticos, se han utilizado diferentes tipos de terminales con diferentes capacidades de interacción, uno de los más populares son las terminales VT100. Estas, se comunicaban a través de una línea de conexión serie con un servidor. A través de una pantalla y un teclado interactuaban con un operador, enviando peticiones al servidor y visualizando su contenido. Una de las características significativas que incorpora este tipo de terminales es la posibilidad de incluir secuencias de caracteres especiales (secuencias de escape) que logran mostrar texto invertido, cambiar el color del texto, borrar la pantalla, etc.

Hoy en día, el uso de este tipo de terminales se han reemplazado por terminales gráficas. Sin embargo, el principio de funcionamiento sigue siendo útil y aplicable a sistemas embebidos. En estos casos, el sistema embebido hace las veces de servidor y, a través de una comunicación serie, se interactúa desde un software que emula el comportamiento de la terminal antes descripta. El resultado es la posibilidad de interacción más avanzada que simplemente mostrar caracteres a través del puerto.

Aún así, existe la necesidad de agregar un procesamiento adicional a los datos que se envían al sistema embebido (servidor), de modo que sea posible la edición de línea de órdenes. Si conectamos un sistema embebido con emulador de terminal, cada carácter que envíe el sistema embebido será representado y cada tecla que pulsemos en la terminal será enviada al servidor. Por lo que no es posible, por ejemplo, ver la representación del carácter enviado, corregir un valor mientras se escribe o guardar un histórico de comandos. Este problema ha sido abordados por diferentes librerías con mayor o menor complejidad, dos casos muy utilizados son las librerías `microrl` (https://github.com/Helius/microrl) y `ntshell` (https://www.cubeatsystems.com/ntshell/index.html).

Utilice la librería `microrl` para implementar una interfaz de comandos, documentando a través de `doxygen` las aplicaciones que desarrolle. Realice cada una de las actividades indicadas a continuación para en el proceso de construcción de la aplicación.

1. A partir de un proyecto nuevo, agregue al mismo los archivos `sysUtils.h` y `sysUtils.c` en las carpetas correspondientes. Agregue en ellos función de inicialización que permita configurar adecuadamente el reloj del sistema, la unidad de punto flotante y los pines asociados a los leds, pulsadores y la `USART2`.

   Para la configuración de los pines de entrada salida digitales, utilice las siguientes definiciones:

```
/*     sysUtils.h          */
typedef struct
{
  uint8_t hwPort;
  uint8_t hwPin;
  uint8_t gpioPort;
  uint8_t gpioPin;
  uint16_t modo;
}digitalIO;

/*     sysUtils.c          */

const digitalIO leds[] =
{
  {0x02, 0x02, 0x05, 0x02, SCU_MODE_INACT | SCU_MODE_FUNC4},
  {0x02, 0x0A, 0x00, 0x0E, SCU_MODE_INACT | SCU_MODE_FUNC0},
  {0x02, 0x0B, 0x01, 0x0B, SCU_MODE_INACT | SCU_MODE_FUNC0},
  {0x02, 0x0C, 0x01, 0x0C, SCU_MODE_INACT | SCU_MODE_FUNC0},
  {0x02, 0x00, 0x05, 0x00, SCU_MODE_INACT | SCU_MODE_FUNC4},
  {0x02, 0x01, 0x05, 0x01, SCU_MODE_INACT | SCU_MODE_FUNC4}

};

const digitalIO keys[] =
{
  {0x01, 0x00, 0x00, 0x04,
    SCU_MODE_INACT | SCU_MODE_FUNC0 | SCU_MODE_INBUFF_EN},
  {0x01, 0x01, 0x00, 0x08,
    SCU_MODE_INACT | SCU_MODE_FUNC0 | SCU_MODE_INBUFF_EN},
  {0x01, 0x02, 0x00, 0x09,
    SCU_MODE_INACT | SCU_MODE_FUNC0 | SCU_MODE_INBUFF_EN},
  {0x01, 0x06, 0x01, 0x09,
    SCU_MODE_INACT | SCU_MODE_FUNC0 | SCU_MODE_INBUFF_EN}
};
```

   Implemente una función que recorra las estructuras `leds` y `keys` realizando las inicializaciones necesarias a través de invocaciones a las funciones `Chip_SCU_PinMuxSet`, `Chip_GPIO_SetPinDIROutput`, `Chip_GPIO_SetPinOutLow` y `Chip_GPIO_SetPinDIRInput`.

   Configure la USART2 de modo que utilice interrupciones cada vez que se recibe un carácter. Verifique el correcto funcionamiento de esta aplicación utilizando gtkTerm de modo que al recibir un carácter se altere el estado de un led.

2. Incorpore los archivos de la librería `microrl` (`microrl.h`, `microrl.c` y `config.h`) en los directorios correspondientes.

   Modifique el archivo config.h como se indica a continuación:

   Línea 30: Configuración del *prompt*. El texto aquí definido aparecerá como inmutable indicando la disponibilidad del sistema de procesar comandos.

```
#define _PROMPT_DEFAULT ‘‘edu-ciaa# ’’
```

Línea 36: Longitud del *prompt*.

```
#define _PROMPT_LEN 10
```

Línea 42: Completado automático, la librería incorpora una funcionalidad que permite, luego de escribir el comienzo de un comando, presionar la tecla `tab` y agrega el resto del comando. Esto requiere la implementación de una función adicional, por lo que no la utilizaremos en esta primera etapa.

```
// #define _USE_COMPLETE
```

Línea 68: No utilizar librería estándar de lenguaje C, `microrl` utiliza funciones de la librería estándar de c o nó. En este último caso utiliza implementaciones propias que no requieren asignación de memoria dinámica.

```
// #define _USE_LIBC_STDIO
```

Línea 79: Mostrar el *prompt* al encender el sistema. Por defecto está deshabilitado, al habilitar esta función se mostrar la línea de comandos cuando la placa esté lista para operar.

```
#define _ENABLE_INIT_PROMPT
```

Linea 83: Configuración de fin de línea. Dependiendo de las características de los sistemas utilizados, la marca de fin de línea puede representarse de diferentes maneras. En nuestro caso configuraremos el sistema para que utilice como marcador *carrige return*.

```
#define _ENDL_CR
```

Una vez finalizada la configuración de la librería es necesario implementar 3 funciones a través de las cuales se dará la interacción la librería. A partir de las definiciones siguientes implemente estas funciones en los archivos `sysUtils.h` y `sysUtils.c`.

`void print(const char *txt)`

Envía la cadena de caracteres dada como argumento a través de la USART2.

`void cmdExecute(int argc, const char * const * argv)`

Cada vez que la librería `microrl` identifica que se ha enviado un comando, ejecutará esta función. Los argumentos tienen el mismo sentido que `argc` y `argv` de la función `main`. Impleméntela de modo que muestre el comando identificado:

```
void cmdExecute(int argc, const char * const * argv)
{
  print("Rcv:");
  print(argv[0]);
  print("\r");
}
```

`void sigint (void)`

Cuando se envía la combinación `Ctrl+C` (o `Ctrl + D`) es posible establecer una respuesta particular. En nuestro caso, forzaremos el reinicio del sistema.

```
void sigint (void)
{
   Chip_RGU_TriggerReset(RGU_CORE_RST);
}
```

Modifique la implementación de la función `main` como se muestra a continuación:

```
1  #include <stdint.h>
2  #include "sysUtils.h"
3  #include "chip.h"
4  #include "microrl.h"
5
6  microrl_t rl;
7
8  void UART2_IRQHandler(void)
9  {
10   uint8_t data = Chip_UART_ReadByte(LPC_USART2);
11   microrl_insert_char(&rl, data);
12 }
13
14 int main(void)
15 {
16
17   uint8_t ch;
18   sysInit();
19
20   microrl_init (&rl, print);
21   microrl_set_execute_callback (&rl, cmdExecute);
22   microrl_set_sigint_callback (&rl, sigint);
23   while(1);
24   return 0;
25 }
```

La librería `microrl` utiliza una estructura para almacenar la información asociada, esta es de tipo `microrl_t` y se encuentra definida en el *header* `microrl.h` (línea 6).

En la línea 8 se ha implementa la función de manejo de interrupciones de la USART2, la que se ha configurado de dispararse cada vez que se tiene un valor disponible en el *buffer* de entrada. El valor recibido es enviado a la motor de la librería a través de la función `microrl_insert_char`.

Dentro de la función `main` encontramos la función `sysInit`, responsable de inicializar el sistema y las invocaciones a diferentes funciones para configurar la librería. En la línea 20 se inicializa la estructura con la configuración de trabajo, luego se indica cuál es la función a utilizar para imprimir un texto por la terminal serie, para ejecutar un comando y al recibir la secuencia `Ctrl+C`. Note el uso de punteros a funciones como argumentos, esto permite el uso de funciones con diferentes nombres e incluso trabajar con diferentes instancias de la librería con diferentes configuraciones.

En esta actividad, en favor de simplificar la implementación, se utilizaron los nombres de funciones indicados.

Compruebe el correcto funcionamiento de la aplicación, para esto deberá utilizar `gtkTerm` y configurarlo de modo que maneje automáticamente los indicadores de *Carrige Return* y *Line Feed*. Esto se configura desde el menú de la aplicación `Configuration/CR LF auto`. Al escribir "hola mundo.[en] el terminal debería observar Rcv: hola mundo".

3. Utilizando el depurador y agregando un punto de interrupción (*break*) dentro de la primera línea de código ejecutable de la función `cmdExecute`, analice el valor de los argumentos `argc` y `argv` con las siguientes entradas:

   ```
   test 1 2
   test 1
   test hola mundo
   ```

4. Desarrolle una función que permita reproducir secuencias de leds. Para esto, agregue dos archivos al proyecto (`secuencia.h` y `secuencia.c`) para implementar la función 'secuencia':

```
void secuencia(uint8_t secNro, uint8_t cntRep, uint16_t delayMs);
```

Donde:

- `secNro`: número de secuencia. En principio podrá tener valores 0 o 1 correspondiéndose cada uno con un patrón de encendido y apagado de leds.
- `cntRep`: cantidad de repeticiones que se dará a la secuencia.
- `delayMs`: tiempo de espera entre el encendido de dos patrones de una secuencia particular.

Existe diferentes alternativas para la generación de secuencias como las planteadas, en este trabajo se propone utilizar la interrupción del `SysTick` como base de tiempo, una variable para indicar la secuencia a reproducir y la cantidad de repeticiones como un contador descendente que al llegar a cero detiene las interrupciones del reloj utilizado.

Para esto realice las siguientes definiciones en el archivo `secuencia.c`:

```
static uint8_t iSecNro = 0;
static uint8_t iCntRep = 0;
static uint16_t iDelayMs = 1000;
static uint8_t iActPos = 0;

const uint8_t ledSec1[] = {8, 1, 2, 4, 8, 8, 4, 2, 1};
const uint8_t ledSec2[] = {4, 9, 6, 6, 9};

const uint8_t * const secuencias [] ={ledSec1, ledSec2};
```

Al invocar a la función `secuencia`, esta deberá inicializar las variables `iSecNro`, `iCntRep` , `iDelayMs` e inicializar el `SysTick` de modo que genere interrupciones cada milisegundo. Luego, en cada interrupción, se deberá incrementar una variable estática (`cntTick`), definida en la ISR del `SysTick`.

Los vectores `ledSec1` y `ledSec2` constituyen la secuencia de encendido de los leds. El primer elemento de cada vector es la cantidad de etapas que constituirá la secuencia y los demás valores representan la máscara de los leds a encender. Así, el valor 1 se corresponde con el primer led y el valor 3 con los dos primeros leds.

El número de secuencia es utilizado como índice de filas de la matriz `secuencias`. Entonces, cada vez que se ejecuta la ISR del `SysTick` se incrementa el contador de milisegundos, cuando se alcanza el retardo indicado, se lo inicializa en cero y se actualiza la secuencia. Para esto se apagan todos los leds y se encienden aquellos que están el la columna `iActPos` para luego incrementar esta variable. Cuando alcanza el fin de una secuencia se decrementa el contador `iCntRep`, si este alcanza el valor cero, debe finalizarse la cuenta del `SysTick` y se ha finalizado la secuencia.

Para encender y apagar los leds, defina el arreglo `leds` como una definición externa y utilícela junto a las funciones `Chip_GPIO_SetPinOutLow` y `Chip_GPIO_SetPinState`.

Compruebe la correcta generación de las secuencias desde la aplicación principal.

5. Agregue al proyecto los archivos `comandos.h` y `comandos.c`, estos contendrán un vector de estructuras que representarán la información asociada a cada comando. En `comandos.h` agregue las siguientes definiciones.

```
typedef void (*doAction)(int, const char * const *);

typedef struct
{
```

```
  char * cmdName;
  doAction cmdAct;
} cmdItem;

void help(int, const char * const *);
void sec1(int, const char * const *);
void sec2(int, const char * const *);

void led1On(int, const char * const *);
void led2On(int, const char * const *);
void led3On(int, const char * const *);
void led4On(int, const char * const *);
void led1Off(int, const char * const *);
void led2Off(int, const char * const *);
void led3Off(int, const char * const *);
void led4Off(int, const char * const *);

void cls(int, const char * const *);
menuItem *getMenuItems(void);
unsigned int getMenuItemsCount(void);
```

La estructura `cmdItem`, representa el nombre de un comando y la función a ejecutar. Las funciones ledXXX deberán encender o apagar un led particular, por su parte `sec1` y `sec2` deberán invocar a la función `secuencia` con diferentes argumento.

La función `help` deberá mostrar (enviar por la USART2) la lista de los comandos válidos.

En `comandos.c` realice las siguientes definiciones:

```
menuItem menu[] =
{
    {"help", help},
    {"led1On", leds1On},
    {"led2On", leds2On},
    {"led3On", leds3On},
    {"led4On", leds4On},

    {"led1Off", leds1Off},
    {"led2Off", leds2Off},
    {"led3Off", leds3Off},
    {"led4Off", leds4Off},

    {"secuencia_1", sec1},
    {"secuencia_2", sec2},
    {"cls", cls}
};
```

Implemente en este archivo la función `getMenuItems` de modo que retorne la dirección del arreglo `menu` y `getMenuItemsCount` la cantidad de elementos en el arreglo.

La función `cls` deberá enviar la secuencia de escape necesaria para que la terminal borre su contenido.

```
void cls(int argc, const char * const * argv)
{
   print("\033[2J\033[;f");

}
```

6. Modifique ahora la función `cmdExecute` de modo que, al ser invocada busque nombre del argumento recibido (`argv[0]`) en el vector de comandos. Al encontrar coincidencia, deberá invocar a la función asociada. En caso de no existir ningún comando con el nombre indicado deberá mostrar un mensaje de error.