REFLEKS

FTIMS Systemy Wbudowane 2016/2017

LPC 2138

Grupa D08
Łukasz Kuta - 195637
Patryk Błaziński - 195564
Konrad Klimczak - 195614
Rafał Koper - 195624

FUNKCJONALNOŚCI

L.p.	Funkcjonalność	Opis
1.	FIQ	Wywołanie przerwania pinem p0.14 powoduje zmianę zapalonej diody na kolejną
2.	TIMER	Pauzuje wykonywany proces gry
3.	Diody LED	Informacja o ilości żyć, oraz wskazująca przycisk do wciśnięcia
4.	UART	Kontakt płytki z komputerem
5.	EEPROM	Zapis wyników
6.	I2C	Komunikacja z extension board
7.	SPI	Komunikacja z wyświetlaczem LCD i kartą SD
8.	wyświetlacz LCD	Wyświetlanie stanu gry
9.	SD	Odczyt pliku uruchamiający grę
10.	logika gry	synchronizacja wszystkich funkcjonalności w logiczną całość
11.	RTC	Obliczanie czasu gry
12.	Potencjometr	Zmiana ilości żyć gracza

ZAKRES OBOWIĄZKÓW

Nr. funkcjonalności	Autor	Udział pracy
1.	Łukasz Kuta	Zadania były wykonywane wspólnymi siłami. Każdy z członków zespołu równomiernie został obciążony pracą.
2.	Patryk Błaziński	
3.	Łukasz Kuta	
4.	Konrad Klimczak	
5.	Rafał Koper	
6.	Rafał Koper	
7.	Patryk Błaziński	
8.	Patryk Błaziński	
9.	Łukasz Kuta	
10.	Rafał Koper	
11.	Konrad Klimczak	
12.	Konrad Klimczak	

OPIS DZIAŁANIA PROGRAMU

INSTRUKCJA UŻYTKOWNIKA

Na początku, gracz musi umieścić kartę SD w slocie. Po podłączeniu urządzenia do zasilania, jeżeli system poprawnie zidentyfikuje plik z kodem do gry, gra się uruchomi. W przeciwnym wypadku - tj. w momencie braku karty SD z plikiem z kodem gry, lub posiadania błędnego kodu, ekran wyświetli stosowny komunikat, zaś sama gra się nie uruchomi.

Po udanym ‘uwierzytelnieniu’ użytkownik ma możliwość ustawienia poziomu trudności gry. Robi to z wykorzystaniem czerwonego potencjometru (p0.28).
Zmianę aktualnej ilości żyć można zaobserwować na diodach wokół wyświetlacza LED - para diod, leżąca po przeciwnych jego stronach, oznacza jedno życie.

Aby rozpocząć rozgrywkę, gracz musi trafić w odpowiedni przycisk - p0.20 do p0.23.
To, który przycisk należy w danym momencie wcisnąć, jest podyktowane losowo zapalającymi się diodami umieszczonymi tuż nad przyciskami.
Po tym jest liczony czas reakcji gracza, a na wyświetlaczu pojawia się aktualny winik gracza i ilość jego ruchów. Im szybciej odpowiedni przycisk zostanie wciśnięty tuż po zapaleniu się diody, tym gracz otrzyma więcej punktów. Czyli - im więcej punktów, tym lepszy wynik.
Za każdym, pięciokrotnie dobrze wybranym przyciskiem, czas odstępu między kolejnymi zmianami diod zostaje pomnożony przez 0.8 (oznacza to przyspieszenie migania - czyli stopniowe zwiększanie poziomu trudności).
Wciśnięcie nieodpowiedniego przycisku powoduje stratę jednego życia (tj. zgaśnięcie pary diod).

Po wyczerpaniu żyć na ekranie wyświetla się ostateczny wynik użytkownika wraz z ilością wykonanych ruchów. Za pomocą joysticka można zapisać ten wynik na EEPROM - wystarczy poruszyć w odpowiednim momencie ów joystick do góry.
Po chwili gra uruchomi się od nowa.

OPIS DZIAŁANIA PROGRAMU

OPIS ALGORYTMU

1. Start programu.
2. Sprawdzenie karty SD w poszukiwaniu pliku
3. Uruchomienie procesu obsługi diod oraz przycisków.
4. Oczekiwanie na start rozgrywki (dowolny aktywny przycisk diody) i wybór ilości szans potencjometrem.
5. Inicjalizacja rozgrywki, uruchomienie procesu odczytującego ruchy graczy, wejście do głównej pętli rozgrywki, naliczanie wyniku.
6. Po utraconych szansach wyświetlanie wyniku, ruchów i czasu gry na ekranie z możliwością jego zapisu do pamięci EEPROM za pomocą górnego joysticka.

FIQ

Mikrokontrolery z rodziny LPC 2xxx obsługują trzy typy przerwań:

- IRQ
- VIRQ
- FIQ

Przerwanie FIQ, które wykorzystaliśmy, ma najwyższy priorytet i najkrótszy czas reakcji. Najszybsze możliwe opóźnienie FIQ jest osiągalne wtedy, gdy tylko jedno wywołanie jest zakwalifikowane jako FIQ, ponieważ wtedy ‘FIQ service routine’ może w łatwy sposób obsłużyć to urządzenie.

Przypisaniem do metody atrybutu interrupt("FIQ") oznacza tyle, że ta metoda może obsługiwać *szybkie przerwania*

```
void _fiqHandler(void) __attribute__((interrupt("FIQ")));
```

Zmieniamy domyślną obsługę FIQ na rzecz naszej własnej implementacji

```
pISR_FIQ = (unsigned int)_fiqHandler;
```

Ustawienie EINT1 wrażliwego na krawędzie

```
EXTMODE  = 0x00000002;
```

Ustawienie EINT1 wrażliwego na zbocze opadające

```
EXTPOLAR = 0x00000000;
```

Przypisanie pinu P0.14 do zewnętrznego przerwania EINT1

```
PINSEL0 &= ~0x30000000;
```

```
PINSEL0 |=     0x20000000;
```

Wyczyszczenie flagi przerwania dla EINT1

```
EXTINT   = 0x00000002;
```

Przypisanie EINT jako przerwanie FIQ

```
VICIntSelect |= 0x00008000;
```

Włączenie przerwania EINT1

```
VICIntEnable  = 0x00008000;
```

Natomiast w samym ciele obsługi przerwania, na końcu metody, podalismy:

```
EXTINT = 0x00000002;      //reset flagi przerwania
VICVectAddr = 0x00;       //zasygnalizowanie końca przerwania
```

TIMER

W naszym projekcie został wykorzystany TIMER1 jest to jeden z dwóch timerów występujących w mikrokontrolerze lpc2138. Timer posłużył nam do zatrzymania wykonywania obecnych procesów w celu wykonania opóźnienia (np w przypadku opóźnienia między zapalaniem się kolejnych diod), aby Timer działał prawidłowo należy zainicjować jego działanie.

```
TIMER1_TCR = 0x02;          - zatrzymanie i resetowanie timera
TIMER1_PR = 0x00;           - ustawiamy wartość preskalera na 0
TIMER1_MR0 = delayInMs * ((CRYSTAL_FREQUENCY * PLL_FACTOR) / (1000
                    * VPBDIV_FACTOR)); - ustawianie do jakiej wartości timer przelicza
TIMER1_IR = 0xff;           - resetowanie wszystkich flag przerwań
TIMER1_MCR = 0x04;          - timer zatrzymuje się gdy osiągnie wymaganą wartość
TIMER1_TCR = 0x01;          - uruchomienia timera
```

```
while (TIMER1_TCR & 0x01); - oczekiwanie na osiągnięcie wartości ustawionej w MR0
```

DIODY LED

Diody led są jednym z elementów GPIO - a jest to interfejs służący do komunikacji pomiędzy elementami systemu komputerowego, takimi jak mikroprocesor czy różne urządzenie peryferyjne.

Często nasze diody są powiązane również z innymi elementami systemu (m.in z Timerem opisanem poniżej).

Aby spowodować świecenie diody, należy skonfigurować następujące rejestry:
IODIR pozwala określić czy pin pracuje jako wejście czy wyjście
IOCLR, **IOSET** ustawiają na liniach stan wysoki lub niski.

Ustawiamy cztery diody (te nad przyciskami kontrolującymi gre) jako wejście

```
IODIR1 |= 0x000F0000;
```

Wyłączamy wszystkie diody

```
IOSET1 = 0x000F0000;
```

Następnie, aby zapalić i zgasić określoną diodą, wpisujemy takie wartości :

```
IOCLR1 = address;      //zapalenie diody
IOSET1 = address;      //zgaszenie diody
```

Gdzie adresy do tych czterech diod to:

```
static tU32 LED_ADRESS_1 = 0x00010000;
static tU32 LED_ADRESS_2 = 0x00020000;
static tU32 LED_ADRESS_3 = 0x00040000;
static tU32 LED_ADRESS_4 = 0x00080000;
```

Aby odnieść się do diod z extension board, tj. tych wokół wyświetlacza, musimy posłużyć się I2C oraz Pca9532 - gdzie ten drugi przeznaczony jest do obsługi diod.

Po zainicjowaniu ich działania, aby zapalić bądź zgasić odpowiednią diodę używamy funkcji setPca9532Pin(numerDiody, wartosc), gdzie:

- numerDioty - numer z przedziału <0, 15>
- wartosc - 1 = zapalone, 0 = zgaszone.

Na początku wysyłamy na magistrale I2C adres urządzenia, a następnie dane.
Wykorzystujemy do tego funkcję tS8 i2cPutChar(tU8 data).
Wysyłamy dane tylko wtedy, gdy magistrala nie jest zajęta

```
if((I2C_CONSET & 0x08) != 0)  /* if SI = 1 */
{
        /* send data */
        I2C_DATA   = data;
        I2C_CONCLR = 0x08; /* clear SI flag */
        retCode    = I2C_CODE_OK;
}
```

Jeśli magistrala nie jest zajęta i zwraca informację, że magistrala jest wolna:
retCode = I2C_CODE_OK;
W przeciwnym wypadku zwracana jest informacja o zajętości magistrali:
retCode = I2C_CODE_BUSY;

Czekamy, aż magistrala zostanie zwolniona

```
while(retCode == I2C_CODE_BUSY)
{
        retCode = i2cPutChar(data);
}
if(retCode == I2C_CODE_OK)
        retCode = i2cWaitTransmit();
```

Następnie czekamy na otrzymanie od I2C ACK.

UART

UART pozwala nam na komunikowanie się z urządzeniami podłączonymi przez port szeregowy. By móc korzystać z jego dobrodziejstw musimy wywołać metodę eaInit(), która ma w sobie metodę consolInit(), która w tej funkcjonalności bardzo nas interesuje:

```
#if (CONSOL_UART == 0)
  //włączenie uarta dla pinów #0 w GPIO (P0.0 = TxD0, P0.1 = RxD0)
  PINSEL0 = (PINSEL0 & 0xfffffff0) | 0x00000005;
#else
  //włączenie uarta dla pinów #1 w GPIO (P0.8 = TxD1, P9.1 = RxD1)
  PINSEL0 = (PINSEL0 & 0xfff0ffff) | 0x00050000;
#endif

  //inicjalizacja bitrate
  UART_LCR = 0x80; //DLAB-bit = 1
  UART_DLL = (unsigned char)(UART_DLL_VALUE & 0x00ff);
  UART_DLM = (unsigned char)(UART_DLL_VALUE>>8);
  UART_LCR = 0x00;

  //inicjalizacja LCR: 8N1
  UART_LCR = 0x03;

  //reset kolejki FIFO
  UART_FCR = 0x00;

  //wyczyszczenie bitów przerwań
  UART_IER = 0x00;
```

Najpierw ustawiamy piny odpowiedzialne za odbiór (TxD0 lub TxD1) i nadawanie danych (TxR0 lub TxR1). Następnie inicjalizujemy bitrate przesyłu danych, ale trzeba pamiętać by przed tym ustawić rejestr LCR w DLAB-bit na 1, aby rejestry DLL i DLM były dostępne. Do rejestrow DLL i DLM zapisuje odpowiednio mniej i bardziej ważny bit wartości bitrate. Po ustaleniu tej wartości bit DLAB ustawia na 0, aby zablokować dostęp do rejestrów DLL i DLM. Inicjalizując LCR funkcja ustawia długość przyjmowanego słowa danych na 8-bitowe.

Dzięki temu można przesyłać znaki. Ustawia także 1 bit stopu oraz brak sprawdzania parzystości.

W naszym projekcie program wysyła dane na konsolę podłączonego komputera. W tym celu korzystamy z metody printf() która jest opakowaniem dla metody consoleSendChar(char * str).

```
void
consolSendChar(char charToSend)
{
  //Wait until THR is empty
  while(!(UART_LSR & 0x20))
    ;
  UART_THR = charToSend;
}
```

W tej funkcji na początku oczekujemy aż pierwszy bit rejestru UART_LSR będzie równy 0, co oznacza że żadne bity nie czekają na odczyt.
UART_THR jest to rejestr służący jako bufor dla wychodzących bajtów.

EEPROM

EEPROM (ang. Electrically-Erasable Programmable Read-Only Memory) – rodzaj nieulotnej pamięci komputerowej. Może być kasowana tylko przy użyciu prądu elektrycznego. Liczba zapisów i kasowań jest ograniczona, w zależności od typu i producenta pamięci wynosi od 10 000 do 1 000 000 cykli. Po przekroczeniu tej wartości pamięć ulega uszkodzeniu, natomiast liczba odczytów pamięci EEPROM jest nieograniczona. W naszym programie wykorzystujemy tą pamięć do zapisu wyników gry.

Metoda zapisująca nasz wynik jako 2 bajty w pamięci:

```
tU8 tmp[2];                          tworzenie tablicy znaków 2-elementowej
tmp[0] = score & 0xFF;               8 bitów naszego wyniku
tmp[0] = (score >> 8 )& 0xFF;        kolejne 8 bitów naszego wyniku
```

//wynik rozbijany na znaki ponieważ używamy funkcji i2cPutChar() która zapisuje pojedynczy znak

```
eepromPoll();                        pętla czekająca na zwrócenie przez I2C flagi ACK
eepromWrite(0x0000, tmp, 2)          zapis w adresie 0x0000  tablicy znaków tmp o dlugosci
2
eepromPoll();
```

Analogicznie do zapisu tworzymy odczyt:

```
tU8 tmp[2];
eepromPoll();                          pętla czekająca na zwrócenie przez I2C flagi ACK
eepromPageRead(0x0000, tmp, 2)   odczyt w adresie 0x0000 do tablicy tmp o dlugosci 2
eepromPoll();
```

Sam EEPROM zaś w mikrokontrolerze znajduje się pod adresem 0xA0 i może pomieścić 64kbit (8KB) danych.

I2C

Szeregowa, dwukierunkowa magistrala służąca do przesyłania danych pomiędzy urządzeniami elektronicznymi. W naszym przypadku odpowiada za komunikację mikrokontrolera z expanderem pod adresem 0xC0.

Inicjacja I2C:

```
PINSEL0  |= 0x50;                          Uruchomienie magistrali I2C (piny P.02 i P0.3)
I2C_CONCLR = 0x6c;                        Wyczyszczenie rejestru kontrolnego

I2C_SCLL = ( I2C_SCLL & ~0x0000FFFF) | 0x00000100;          Reset rejestrów I2C
I2C_SCLH = ( I2C_SCLH & ~0x0000FFFF) | 0x00000100;
I2C_ADDR = ( I2C_ADDR & ~0x000000FF) | 0x00000000;
I2C_CONSET = ( I2C_CONSET & ~0x0000007C) | 0x00000040;
```

Aby móc współpracować z diodami używając tego interfejsu używamy PCA9532 , inicjalizowana jest sekwencją: 0x12, 0x97, 0x80, 0x00, 0x40, 0x00, 0x00, 0x00, 0x00 która ustawia właściwości expandera. Po instrukcji pca9532Init() możemy już współpracować z diodami.

SPI

SPI (Serial Peripheral Interface) jest szeregowym interfejsem. Wykorzystywany przez mikrokontrolery do szybkiej, synchronicznej komunikacji z jednym lub wieloma urządzeniami peryferyjnymi lub między dwoma mikrokontrolerami. W komunikacji z użyciem SPI zawsze jedno urządzenie jest typu master. W naszym przypadku SPI użyte jest zarówno w

przypadku wyświetlacza LCD jak i odczytu z karty sd. Szczegóły implementacji w konkretnych przypadkach w nagłówkach dotyczących LCD oraz SD.

Wyświetlacz LCD

Wyświetlacz jest jednym z urządzeń peryferyjnych dostarczonych w extension board. Wyświetlacz ma 128x128 pikseli format koloru 8-bitowy. Komunikacja z wyświetlaczem odbywa się za pomocą SPI. Zgodnie z dokumentacją wyświetlacz wykorzystuje piny P0.4, P0.6, P0.7,P0.15 oraz reset:

P0.4 – SPI-SCK (Serial Clock)
P0.6 – SPI-MOSI (Master Output, Slave Input)
P0.7 – SPI-SSEL (Slave Select)
P0.15 – Backlight (podświetlenie wyświetlacza)
Reset – reset

Komunikacja jednokierunkowa ekran nie wysyła informacji do mikrokontrolera. Wszystkie piksele mieszczące się na ekranie traktowane są jako wektor. Na SPI wysyłane jest 9 bitów. Pierwszy z nich „mówi" o tym, czy dane przesłane po nim są zwykłymi danymi czy też poleceniem. W przypadku 0 polecenie 1 dane.

Na początku musimy ustawić interfejs SPI dla LCD. By tego dokonać należy wykonać poniższe czynności.

IODIR |= (LCD_CS | LCD_CLK | LCD_MOSI); - ustawianie wyjścia dla interfejsiu SPI
PINSEL0 |= 0x00001500; - podłączenie SPI do pinów
SPI_SPCCR = 0x08; - ustawienie mnożnika zegara
SPI_SPCR = 0x20; - dane dotyczące transmisji

Kolejność działań w przypadku wysyłania danych do lcd:

IOCLR = LCD_CLK; - wyłączenie SPI
PINSEL0 &= 0xffffc0ff; - wyłączenie SPI
IOCLR = LCD_MOSI; - jeśli dane o konfiguracji LCD
IOSET = LCD_MOSI; - jeśli przesyłanie danych
IOSET = LCD_CLK; - wysoki sygnał na linii sygnału zegarowego
IOCLR = LCD_CLK; - niski sygnał na lini sygnału zegarowego

Ponowna inicjalizacja SPI
SPI_SPCCR = 0x08;
SPI_SPCR = 0x20;
PINSEL0 |= 0x00001500;

SPI_SPDR = data; - ustawienie danych do wysłania
while((SPI_SPSR & 0x80) == 0) - wysyłanie danych

SD

Czytnik kart został podłączony za pomocą SPI z wykorzystaniem rejestrów:
- SCK - zegar taktujący
- MOSI - dane do układu
- MISO - dane z układu
- SSEL - wybór slave'a

Dane przesyłane są za pomocą 8 bitów. Aby odczytać dane z karty należy wysłać (przez SPI, w rejestrze S0SPDR umieszczamy dane, gdy zostaną przesłane, zmienia się stan rejestru S0SPSR) najpierw nr polecenia odczytu (17) oraz numer sektora, który ma być odczytany. Przyjęcie polecenia oraz gotowość do nadawania danych jest potwierdzane 2 bajtami o określonej wartości. Po czym następuje dopiero przesłanie bajt po bajcie zawartości całego sektora (czyli 512B) oraz 2 bajtów sumy kontrolnej

Logika gry

Po pomyślnym odczytaniu karty i "uwierzytelnieniu" użytkownik potencjometrem wybiera ilość żyć które reprezentowane są przez rzędy diod przy wyświetlaczu. Jednym z przycisków przy diodach P0.20, P0.21, P0.22, P0.23 uruchamia grę. Losowana jest wartość z przedziału [0 , 3] która zapala jedną z diod przy podanych przyciskach na określony przez nas czas (delay), który ulega zmianie co 5 ruchów użytkownika. Gracz wciskając odpowiednią diodę uzyskuje wynik równy:

```
*result = (1000 - *result) + TIMER1_TC / 100000;
```

który sumuje się z każdym kolejnym naciśnięciem odpowiedniego przycisku diody. Naciśnięcie złej diody lub diody po czasie jest równoznaczny z zabraniem jednego życia(które gracz ustawiał na początku). Im większy wynik tym lepsza reakcja gracza. Po straceniu wszystkich możliwych szans wynik, ilość ruchów, czas gry oraz zapisany wynik jest wyświetlany na ekranie końcowym gdzie można zapisać uzyskany wynik za pomocą górnego wciśnięcia joysticka lub wyczyścić wynik wciśnięciem dolnym.

Obsługe joysticka inicjalizujemy za pomocą initKeyProc gdzie tworzony jest oddzielny proces specjalnie do obsługi kontrolera zaś same kierunki opisane są następująco:

```
#define KEYPIN_CENTER 0x00000100
#define KEYPIN_UP 0x00000400
#define KEYPIN_DOWN 0x00001000
#define KEYPIN_LEFT 0x00000200
#define KEYPIN_RIGHT 0x00000800

//ustawiajace te kierunki jako wejscie
IODIR &= ~(KEYPIN_CENTER | KEYPIN_UP | KEYPIN_DOWN | KEYPIN_LEFT |
KEYPIN_RIGHT);
```

Sam wciśnięty kierunek pozyskujemy zaś z funkcji checkKey() w która pozyskuje wartość, ustawianą wcześniej w funkcji sampleKey() gdzie sprawdzamy wciśnięte kierunki co 50ms.

RTC

W naszym projekcie został również użyty Real Time Clock - według specyfikacji urządzenia jest to zestaw liczników do pomiaru czasu, kiedy do urządzenia jest dostarczany prąd, choć nie jest to wymagane.
Zegar napędza dedykowany oscylator o częstotliwości działania 32kHz.

W naszym programie wykorzystaliśmy jako stopera, który mierzy czas od początku rozgrywki (pierwsze poprawne kliknięcie przycisków pod diodami) do zakończenia poprzez utracenie wszystkich żyć.

Obsługę RTC inicjujemy metodą initRtc() zawartą main.c:

```
RTC_CCR  = 0x00000010;
RTC_ILR  = 0x00000000;
RTC_CIIR = 0x00000000;
RTC_AMR  = 0x00000000;
```

Pierwszy rejestr jest to CCR, który odpowiada za kontrolę nad zegarem. Ustawiamy tutaj wartość binarną na 10000, gdzie czwarty bit przestawia nam źródło zegara na dedykowany oscylator.
Następne trzy rejestry odpowiedzialne. za przerwania od RTC, za przerwania od zmian rejestrów czasu oraz alarmów zerujemy, gdyż nie będą nam potrzebne.

Następnie, w trakcie rozpoczęcia rozgrywki jest uruchamiana metoda startClock():

```
RTC_SEC = 0;
RTC_MIN = 0;
RTC_HOUR = 0;
RTC_CCR = 0x00000011;
```

która resetuje nam rejestry odpowiedzialne za sekundy, minuty oraz godziny.
Na koniec zegar jest uruchamiany, gdyż wartość na bicie zerowym zostaje zmieniona z 0 na 1 co daje nam 10001.

Po utraceniu wszystkich żyć czas zostaje odczytany z rejestrów RTC_SEC oraz RTC_MIN oraz wyświetlony na ekranie.

POTENCJOMETR

Potencjometr jest urządzeniem analogowym, w celu jego użycia niezbędny jest przetwornik ADC, który zmieni sygnał analogowy na sygnał cyfrowy przetwarzany przez mikrokontroler. Na płytce LPC2138 zostały umieszczone 3 potencjometry w naszym przypadku używamy potencjometru AIN1 - pin P0.28. W celu rozpoczęcia należy najpierw zainicjować ADC dla odpowiedniego potencjometru. Czynność ta polega na określeniu dla jakich urządzeń ADC ma zostać zainicjowane. Dla AIN1:

```
PINSEL1 &= ~0x03000000;
PINSEL1 |= 0x01000000;
```

Oraz właściwa inicjalizacja samego ADC

```
ADCR = (1 << 0)|((CRYSTAL_FREQUENCY *PLL_FACTOR /VPBDIV_FACTOR) / 4500000 - 1) << 8 |  (0 << 16)|(0 << 17)|(1 << 21)|(1 << 24)|(0 << 27);
```

Następnie należy dokonać konwersji:

```
ADCR = (ADCR & 0xFFFFFF00) | (1 << channel) | (1 << 24);
while((ADDR & 0x80000000) == 0); - pętla, aż do skończenia konwersji
(ADDR>>6) & 0x3FF; - konwertowanie do wartości całkowitej
```