Ball The Game

Dokumentacja

Systemy Wbudowane
2021/2022

Zajęcia:

Systemy Wbudowane, środa 12:00

Skład zespołu:

Nikodem Kirsz	236559	
Bartosz Siekański	236646	
Oskar Trela	236677	(lider)

Urządzenie wykorzystane w projekcie:

LPC2138 + Expansion Board

Procentowy wkład pracy:

Nikodem Kirsz	33.(3)%
Bartosz Siekański	33.(3)%
Oskar Trela	33.(3)%

Spis treści

1. Opis projektu

1.1. Spis funkcjonalności

Tabela 1 - Spis funkcjonalności wykorzystywanych w "Ball the Game"

Funkcjonalność	Stan	Osoba odpowiedzialna
Interfejs GPIO (Joy-stick)	działa	Oskar Trela
Obsługa ADC	działa	Nikodem Kirsz
Akcelerometr (ACC)	działa	Nikodem Kirsz
Interfejs SPI	działa	Oskar Trela
Ekran LCD	działa	Bartosz Siekański
Interfejs I^2C	działa	Bartosz Siekański
Diody przy LCD (pca9532)	działa	Bartosz Siekański
Timer	działa	Oskar Trela

1.2. Opis gry

“Ball the Game” jest grą, która polega na sterowaniu małym kwadratem („piłką”) za pomocą akcelerometru w takim sposób, aby omijać spadające prostokąty („przeszkody”).

Po uruchomieniu programu naszym oczom ukazuje się ekran powitalny, a samą rozgrywkę rozpoczynamy przesuwając Joystick w górę. Pokazuję nam się wtedy ekran gry z białą piłką na środku ekranu, którą teraz możemy kontrolować pochylając urządzenie w wybranym kierunku. Kąt pochylenia urządzenia wpływa na prędkość poruszania się piłki. Naszym celem jest teraz jak najdłuższe pozostanie „żywym” na planszy, kiedy to poziom trudności stale rośnie.

Gra kończy się w momencie uderzenia w przeszkodę, a użytkownikowi wyświetla się wtedy na ekranie uzyskany wynik. Ponowną rozgrywkę może on rozpocząć tym samym przyciskiem, co poprzednio (Joystick-Up).

Kontrolki użytkownika:

Joystick-Up	–	Rozpoczęcie gry
Joystick-Down	–	Zatrzymanie gry
Akcelerometr	–	Ruch piłki

1.3. Opis algorytmu

Program zaczyna się w funkcji *main(),* która to tworzy proces inicjalizacyjny *initProc(void* arg)*, który to z kolei tworzy proces inicjalizujący ekran, konwerter analogowo-cyfrowy oraz wyświetlający ekran powitalny – procedura *drawWelcome().* Następnie wchodzimy w nieskończoną pętlę, w której to możemy wybierać Joystickiem żądaną akcję.
Procedura *startGame()* jest odpowiedzialna za wywołanie procedur inicjalizujących stan gry, wykonującą sekwencję diod, za sprawdzenie obecności i zainicjowanie *pca9532* oraz za stworzenie procesów umożliwiających kontrolę pochylenia urządzenia za pomocą akcelerometru – *accXCtrlProc(void *arg)* i *accYCtrlProc(void *arg)* – procesu kontrolującego ruch przeszkód – *obstaclesCtrlProc(void *arg)* – oraz procesu zliczającego wynik i podnoszącego poziom trudności w zależności od długości rozgrywki – *gameTimeProc(void *arg).*
Procedura *stopGame()* zatrzymuje rozgrywkę poprzez ustawienie wartości 0 na zmiennej *isInProgress,* dzięki której kończą się wszystkie pętle związane z samą rozgrywkę, następnie wywołuje procedurę wykonującą sekwencję diod, po czym wywołuje procedurę wyświetlającą wynik końcowy gracza.

Procedury kontrolujące ruch piłki sczytują wartości przyspieszenia mierzone przez akcelerometr w osiach X oraz Y. Następnie porównują je z przyspieszeniem referencyjnym, dzięki czemu mogą wyliczyć pochylenie urządzenia. Na podstawie tego pochylenia wyliczają kierunek i siłę z jaką przesunie się piłka, oraz odstęp czasu przed następnym przesunięciem piłki.

Piłkę jak i odpowiednią ilość przeszkód (zdefiniowaną dyrektywą preprocesora *#define MAX_OBSTACLES*) tworzymy tylko raz, a następnie do dalszego generowania przeszkód wykorzystaliśmy metodę Object Pooling'u, która polega na ponownym wykorzystywaniu „zużytych" obiektów (czyli u nas – przeszkód, które już nie znajdują się na obszarze ekranu) i resetowaniu ich właściwości, oszczędzając na tym bardziej kosztowne instancjonowanie nowych obiektów.

Przesuwanie piłki jak i przeszkód polega na uprzednim zamazaniu tych obiektów kolorem tła, wyliczeniu ich nowych pozycji na podstawie ich szybkości oraz narysowaniu tych obiektów w nowych pozycjach na ekranie w kolorze pierwszoplanowym.

Kolizje piłki z przeszkodami sprawdzamy porównując najbardziej wysuniętą część piłki w osi X z początkiem przeszkody, a następnie odwrotnie – najbardziej wysuniętą część przeszkody w osi X z początkiem piłki. Jeżeli kolizja na tej osi nie jest już możliwa to zwracamy wartość 0, oszczędzając zbędnych obliczeń dla osi Y. Jeżeli jednak kolizja jest możliwa to powtarzamy ten algorytm dla osi Y. Jeżeli kolizja wystąpiła to funkcja zwraca wartość 1.

2. Opis funkcjonalności

2.1. Interfejs SPI

SPI (*Serial Peripheral Interface*) jest interfejsem wykorzystywanym do komunikacji między systemami mikroprocesorów, a układami peryferyjnymi. W naszym przypadku układem peryferyjnym jest ekran LCD umieszczony na Expansion Board'zie.

Rys.1. Protokół SPI z jednym urządzenie nadrzędnym i jednym podrzędnym. [1]

SCK (na rys.1. jako **SCLK)** – (*Serial Clock*) sygnał zegarowy,

MOSI – (*Master Output Slave Input*) dane wejściowe dla układu peryferyjnego,

MISO – (*Master Input Slave Output*) dane wyjściowe z układu peryferyjnego,

SS – (*Slave Select*, zwany również *Chip Select*) aktywny w przypadku braku napięcia na pinie, wyznacza który *Slave* jest wybrany do uczestnictwa w transferze danych.

Ze względu na charakter transakcji między urządzeniami (LCD nie wysyła danych zwrotnych) nie będziemy potrzebowali sygnału **MISO**.

Aby skorzystać z SPI należy go najpierw zainicjalizować:

1. Ustawiamy kierunek dla każdego z pinów (sygnał wysoki, czyli jako wyjście):

```
IODIR |= (LCD_CS | LCD_CLK | LCD_MOSI);
```

Gdzie:

LCD_CS	==	**SS**,	pin P0.7 – adres 0x00000080
LCD_CLK	==	**SCK**,	pin P0.4 – adres 0x00000010
LCD_MOSI	==	**MOSI**,	pin P0.6 – adres 0x00000040

Wyłączamy kontroler z uczestnictwa w transferze danych ustawiając wartość pinu **SS** na stan wysoki, gdyż to tylko inicjalizacja.

```
IOSET = LCD_CS;
```

Żeby włączyć kontroler do uczestnictwa w transferze ustawiamy wartość pinu **SS** na stan niski.

```
IOCLR = LCD_CS;
```

Następnym krokiem jest podłączenie interfejsu do pinów.

```
PINSEL0 |= 0x00001500;
```

Ostatnią rzeczą jest ustawienie prędkości interfejsu oraz ustawienie w tryb *Master*.

```
SPI_SPCCR = 0x08;
SPI_SPCR = 0x20;
```

`SPI_SPCCR` – wyznacza rejestr (*SPI Clock Count Register*), który determinuje prędkość interfejsu.

Minimalna wartość jaka może zostać wpisana to 0x08 oraz wartość ta musi być parzysta (bit 0 musi mieć wartość 0, a wzór opisujący prędkość zegara wygląda następująco:

$$prędkość\ zegara\ = \frac{\boldsymbol{PCLK}}{\boldsymbol{SPCCR}},$$

gdzie:

PCLK – czas taktowania zegara dla danych wyjściowych z Master'a.

Po wzorze widać, że najszybsza prędkość będzie dla najmniejszej wartości w rejestrze **SPCCR**.

`SPI_SPCR` – wyznacza rejestr kontrolujący operacje na interfejsie. Do ustawienia w tryb Master należy przypisać wartość 0x20.

2.2. Konwerter analogowo-cyfrowy

Konwerter analogowo-cyfrowy służy (jak zresztą sama nazwa wskazuje) do konwersji sygnału analogowego na sygnał cyfrowy, którym możemy się następnie posługiwać w naszym programie. W naszym projekcie wykorzystujemy go do konwertowania analogowego odczytu akcelerometru (opisanego w rozdziale **2.3.**) na sygnał cyfrowy w postaci 10-bitowej liczby dodatniej całkowitej.

Wykorzystujemy tutaj rejestry:
`AD1CR` – (ADC1 Control Register) 32-bitowy rejestr służący do wybierania trybu pracy konwersji: kanału, częstotliwości zegara, liczby zegarów, czy samego startu konwersji
`AD1DR` – (ADC1 Data Register) 32-bitowy rejestr, w którym przechowywane są informacje o wykonanej konwersji: wynik konwersji, flagi określające powodzenie konwersji

Rejestry te mają adres:

```
AD1CR      0xE0060000
AD1DR      0xE0060004
```

Przed użyciem konwertera trzeba go zainicjować, co wykonujemy instrukcjami:

```
PINSEL1 &= ~((1<<10)|(1<<11));   (ustawienie bitów nr 10 i 11 na stan niski)
PINSEL1 |=  (1<<11);             (ustawienie bitu nr 11 na stan wysoki)
PINSEL1 &= ~((1<<12)|(1<<13));   (ustawienie bitów nr 12 i 13 na stan niski)
PINSEL1 |=  (1<<12);             (ustawienie bitu nr 12 na stan wysoki)
```

Następnie musimy ustawić potrzebne wartości w rejestrze kontrolnym AD1CR:
- (bity 0-7, SEL) ustawiamy przykładowy kanał 0
- (bity 8-15, CLKDIV) ustawiamy częstotliwość zegara na 4.5 MHz, czyli na jego maksymalną dopuszczalną wartość
- (bit 16, BURST) ustawiamy na 0, jako że konwersja jest kontrolowana przez program
- (bity 17-19, CLKS) ustawiamy bit na 0, czyli ilość zegarów na 11, co daje nam 10-bitowy wynik konwersji
- (bity 24-26, START) ustawiamy bit 24 na 1, aby zacząć teraz testową konwersję, ponieważ kiedy wartość BURST jest ustawiona na 0 to właśnie te bity decydują o starcie konwersji
- (bit 27, EDGE)) bit ten ustawiamy na 0, lecz nie ma on znaczenia, gdy wartość START jest równa 000-001

Następnie czekamy krótką ilość czasu i wykonujemy testowy odczyt.

Aby dokonać konwersji musimy kolejno:
1. Czyścimy pierwsze 8 bitów odpowiedzialne za wybór kanału, ustawiamy wartość bitu odpowiadającemu wybranemu przez nas kanałowi na 1, a następnie wstawiamy wartość 1 do START, co rozpoczyna konwersję.

```
AD1CR = (AD1CR & 0xFFFFFF00) | (1 << channel) | (1 << 24);
```

2. Następnie czekamy aż konwersja się zakończy (bit nr 31 przyjmie wartość 1).

```
while((AD1DR & 0x80000000) == 0);
```

3. Czytamy wynik z rejestru danych oraz rzutujemy go na 10-bitowy Integer
($0x3FF_{(16)} == 1023_{(10)} == 2^{10} - 1_{(10)}$)

```
return (AD1DR>>6) & 0x3FF;
```

2.3. Akcelerometr

Na Expansion Board'zie znajduje się 3-osiowy akcelerometr (*Freescale MMA7260QT*), który wysyła analogowe napięcie dla każdego z mierzonych kierunków. Sygnały wyjściowe są podłączone do pinów P0.21, P0.22, P0.30. Piny P0.13 i P0.14 odpowiedzialne są za kontrolowanie czułości akcelerometru, który to może pracować w 4 trybach czułości: 1.5g, 2g, 4g lub 6g. W naszej grze wykorzystaliśmy go jako kontroler ruchu piłki, wykorzystując tylko osie X oraz Y, mierząc różnice wartości mierzonych do pewnych wartości referencyjnych, zapisanych na początku gry.

Do odczytywania wartości zmierzonych przez to urządzenie wykorzystujemy funkcję *getAnalogueInput1(tU8 channel)*, która to przyjmuje odpowiedni kanał jako argument.
Tymi kanałami są:

```
#define ACCEL_X AIN6
#define ACCEL_Y AIN7
```

gdzie

```
#define AIN6 6
#define AIN7 7
```

Następnie konwertuje sygnał analogowy na sygnał cyfrowy, w postaci 10-bitowej dodatniej liczby całkowitej (sama konwersja jest opisana w rozdziale **2.2**).

2.4. Interfejs GPIO

GPIO (*General Purpose Input Output*) służy do sterowania wejściem lub wyjściem pinów. Dostępne mamy dwa porty P0 oraz P1 i odpowiednio dla nich 32 piny P0.0 – P0.31 oraz P1.16-P1.31. Dla każdego portu mamy przydzielone po 4 32-bitowe rejestry obsługujące piny:

1. **IOPIN** – przechowuje informacje na temat aktualnego stanu pinów,
   Dla portu P0:

   IO0PIN - 0xE002 8000

   Dla portu P1:

   IO1PIN - 0xE002 8010

2. **IOSET** – ustawia stan wysoki na linii portu,

   Dla portu P0:
   IO0SET - 0xE002 8004
   Dla portu P1:
   IO1SET - 0xE002 8014

3. **IOCLR** – ustawia stan niski na linii portu,

   Dla portu P0:
   IO0CLR - 0xE002 800C
   Dla portu P1:
   IO1CLR - 0xE002 801C

4. **IODIR** – definiuje czy konkretna linia portu będzie pracowała w trybie wyjścia (stan wysoki) czy wejścia (stan niski),

   Dla portu P0:
   IO0DIR - 0xE002 8008
   Dla portu P1:
   IO1DIR - 0xE002 8018

Piny mogą wykonywać różne funkcje. Każdy posiada do 4 możliwości, które mogą zostać wybrane za pośrednictwem jednego z 3 rejestrów **PINSEL**.

1. **PINSEL0 –** dla P0.0 – P0.15 adres - **0xE002 C000**
2. **PINSEL1 –** dla P0.16 – P0.31 adres - **0xE002 C004**
3. **PINSEL2 –** dla P1.16 – P1.31 adres - **0xE002 C014**

Interfejs GPIO wykorzystujemy przy funkcjonalności dotyczącej joy-stick'a.

```
IODIR &= ~(KEYPIN_CENTER | KEYPIN_UP | KEYPIN_DOWN | KEYPIN_LEFT | KEYPIN_RIGHT);
```

Wykonując powyższą instrukcję ustawiamy kierunek **wejścia** dla każdego z pinów joy-stick'a.

KEYPIN_CENTER	**0x00000100**
KEYPIN_UP	**0x00000400**
KEYPIN_LEFT	**0x00000200**
KEYPIN_RIGHT	**0x00000800**
KEYPIN_DOWN	**0x00001000**

Podczas działania gry w tle jest wykonywane sprawdzenie czy stan któregoś z pinów nie uległ zmianie.

Przykładowy warunek sprawdzenia czy joystick został wciśnięty.

```
((IOPIN & KEYPIN_CENTER) == 0)
```

Analogiczne sprawdzenie zastosowaliśmy dla reszty kierunków.

2.5. Ekran LCD

Ekran LCD (umieszczony w Expansion Board'zie) jest wykorzystywany do wyświetlania obrazu powitania oraz obrazu samej gry. Wyświetlacz jest kolorowy, jego wymiary to 128x128 pikseli. Posiada podświetlenie LED, które jest kontrolowane przez stałoprądowe urządzenie LT1932. Aby skorzystać z ekranu, niezbędne będzie wcześniejsze zainicjalizowanie interfejsu SPI – zostało opisane to szczegółowo w rozdziale 1.

Przesyłanie danych do wyświetlacza LCD odbywa się w następujący sposób:

Wyłączamy SPI:

```
IOCLR = LCD_CLK;
PINSEL0 &= 0xffffc0ff;
```

Następnym krokiem jest skonfigurowanie MOSI. Jeśli przesyłane dane mają zostać wyświetlone na ekranie, wykonana zostanie następująca instrukcja:

```
IOSET = LCD_MOSI;
```

Jeśli natomiast zamierzamy przesłać dane, które dotyczą ustawień ekranu (nastąpi wtedy zresetowanie MOSI), należy wykonać następującą instrukcję:

```
IOCLR = LCD_MOSI;
```

Następnie, ustawiamy wartość pinu **SCK** (sygnał zegarowy) na stan wysoki.

```
IOSET = LCD_CLK;
```

Po czym, ustawiamy wartość pinu **SCK** (sygnał zegarowy) na stan niski.

```
IOCLR = LCD_CLK;
```

W kolejnym kroku włączamy ponownie SPI. Inicjalizujemy ten interfejs poprzez kolejno ustawienie prędkości interfejsu oraz ustawienie go w tryb Master.

```
SPI_SPCCR = 0x08;
SPI_SPCR = 0x20;
```

Podłączamy ponownie interfejs SPI do pinów

```
PINSEL0 |= 0x00001500;
```

Ostatnim działaniem jest wysłanie danych

```
SPI_SPDR = data;
while((SPI_SPSR & 0x80) == 0);
```

`SPI_SPDR` – wyznacza rejestr danych (*SPI Data Register*). To dwukierunkowy rejestr umożliwiający transmisję oraz odbiór danych dla interfejsu SPI. Ustawienie trybu Master sprawia, że nadpisanie tego rejestru rozpocznie transfer danych.

`SPI_SPSR` – wyznacza rejestr statusu (*SPI Status Register).* To rejestr tylko do odczytu, informujący o statusie interfejsu SPI.

2.6. Diody przy LCD (pca9532)

Diody przy ekranie LCD (osiem sztuk po lewej i osiem sztuk po prawej stronie wyświetlacza) zostały wykorzystane jako wizualne urozmaicenie rozgrywki. Diody zapalają się z obu stron wprost proporcjonalnie do pozycji na osi Y obiektu sterowanego przez gracza. Ich funkcjonowanie możliwe jest dzięki ekspanderowi pca9532, który podłączony jest do magistrali I^2C. Aby skorzystać z diod, niezbędne będzie wcześniejsze zainicjalizowanie magistrali I^2C – zostało opisane to w rozdziale **2.7**.

W celu włączenia lub wyłączenia diody, musimy na początku zainicjalizować piny wejścia/wyjścia pca9532 metodą *pca9532Init()*. Dokonujemy tego poprzez komunikację z pca9532, wysyłając następujące komendy inicjalizujące:

```
tU8 initCommand[] = {0x12, 0x97, 0x80, 0x00, 0x40, 0x00, 0x00, 0x00,
                     0x00};
```

Jeśli inicjalizacja zakończyła się sukcesem, możemy przejść do operacji na diodach. W tabeli 1 przedstawiono rejestry oraz bity odpowiadające konkretnym diodom.

Tabela 2. Rejestry oraz bity odpowiadające konkretnym diodom.

Rejestr	Bit	Numer diody
LS0 (0x06)	7:6	3
	5:4	2
	3:2	1
	1:0	0
LS1 (0x07)	7:6	7
	5:4	6
	3:2	5
	1:0	4
LS2 (0x08)	7:6	11
	5:4	10
	3:2	9
	1:0	8
LS33 (0x09)	7:6	15
	5:4	14
	3:2	13
	1:0	12

Ustawienie wartości 0x01 na konkretnych bitach, spowoduje ustawienie stanu niskiego (operacja włączenia) dla konkretnej diody.
Ustawienie wartości 0x00 na konkretnych bitach, spowoduje ustawienie stanu wysokiego (operacja wyłączenia) dla konkretnej diody.

Ustawienie wartości 0x10 na konkretnych bitach, spowoduje mruganie z częstotliwością PWM 0 dla konkretnej diody.

Ustawienie wartości 0x11 na konkretnych bitach, spowoduje mruganie z częstotliwością PWM 1 dla konkretnej diody.

gdzie:

`PWM0` (Pulse With Modulation 0), `PWM1` (Pulse with Modulation 1) – rejestry określające cykle pracy mrugania diod.

2.7. Interfejs I^2C

I^2C, którego rozwinięcie to *Inter-Integrated Circuit* (z ang. "pośrednik pomiędzy układami scalonymi") to szeregowa, dwukierunkowa magistrala używana do transferu danych w urządzeniach elektrycznych. W naszym projekcie została wykorzystana do komunikacji z diodami LED, których funkcjonowanie zostało opisane w rozdziale **2.6**. Magistrala I^2C korzysta z pinów P0.2 (I^2C-SCL) i P0.3 (I^2C-SDA).

Interfejs I^2C posiada następujące rejestry:

`I2C_CONSET` – wyznacza rejestr kontroli (*I^2C Control Set Register*).

`I2C_STAT` – wyznacza rejestr statusu (*I^2C Status Register*). Podczas wykonywania operacji dostarcza kody statusu, które pozwalają określać następne działanie.

`I2C_DATA` – wyznacza rejestr danych (*I^2C Data Register*). Podczas transmisji w trybie slave lub master, transmitowane dane są przypisywane do tego rejestru, a podczas odbierania danych są z niego odczytywane.

`I2C_ADDR` – wyznacza rejestr adresu (*I^2C Slave Address Register*). Zawiera 7 bitowy adres slave dla operacji na interfejsie I^2C pracującym w trybie slave, nie jest używany w trybie master.

`I2C_SCLH` – wyznacza rejestr cyklu pracy HIGH (*I^2C Duty Cycle Register High Half Word*). To rejestr 16 bitowy, który określa czas HIGH zegara I^2C.

`I2C_SCLL` – wyznacza rejestr cyklu pracy LOW (*I^2C Duty Cycle Register Low Half Word*). To rejestr 16 bitowy, który określa czas LOW zegara I^2C.

`I2C_CONCLR` – wyznacza rejestr czyszczenia (*I^2C Control Clear Register*).

Rejestry SCLH i SCLL razem ustalają częstotliwość taktowania zegara I^2C wygenerowaną za pomocą następującego wzoru:

$$I^2C_{bit\ frequency} = \frac{PCLK}{I^2CSCLH + I^2CSCLL}$$

gdzie:

PCLK – częstotliwość taktowania zegara peryferyjnego.

Inicjalizacja magistrali I^2C dokonywana jest za pomocą następujących instrukcji:

Ustawienie linii zegara i linii danych poprzez ustawienie wartości 01 na pinach P0.2 i P0.3.

```
PINSEL0 |= 0x00001500;
```

Wyczyszczenie flag poprzez wpisanie do rejestru I2C_CONCLR wartości 0x06. Poprzez te operacje bity 2,3,5 oraz 6 tego rejestru zostają ustawione na wartość 0.

```
I2C_CONCLR = 0x6c;
```

Ostatnim krokiem jest zresetowanie następujących rejestrów.

```
I2C_SCLL = (I2C_SCLL & ~I2C_REG_SCLL_MASK) | I2C_REG_SCLL);
I2C_SCLH = (I2C_SCLH & ~I2C_REG_SCLH_MASK) | I2C_REG_SCLH);
I2C_ADDR = (I2C_ADDR & ~I2C_REG_ADDR_MASK) | I2C_REG_ADDR);
I2C_CONSET = (I2C_CONSET & ~I2C_REG_CONSET_MASK) |
I2C_REG_CONSET);
```

gdzie:

```
I2C_SCLL = 0xE001C014;
I2C_SCLH = 0xE001C010;
I2C_ADDR = 0xE001C00C;
I2C_CONSET = 0xE001C000;


I2C_REG_SCLL_MASK = 0x0000FFFF;
I2C_REG_SCLH_MASK = 0x0000FFFF;
I2C_REG_ADDR_MASK = 0x000000FF;
I2C_REG_CONSET_MASK = 0x0000007C;
I2C_REG_SCLL = 0x00000100;
I2C_REG_SCLH = 0x00000100;
I2C_REG_ADDR = 0x00000000;
I2C_REG_CONSET = 0x00000040;
```

2.8. Timer

Timery wykorzystywane są w mikrokontrolerach do kontrolowania upływającego czasu. Nasze urządzenie (LPC2138) ma dwa 32-bitowe timery, które korzystają z **PCLK*** Aby skorzystać z timera mamy do dyspozycji kilka rejestrów na każdy timer:

* prefix **T0** oznacza rejestr odpowiadający timerowi pierwszemu, a **T1** analogicznie drugiemu.

IR – (*Interrupt Register*) rejestr 8-bitowy służący do obsługi przerwań. Można dzięki niemu zidentyfikować które z 8 przerwań jest w oczekiwaniu.

T0IR - 0xE000 4000
T1IR - 0xE000 8000

TCR – (*Time Control Register*) rejestr służący do kontroli funkcji **TC** (*Timer Counter*). **TC** może zostać przez niego zresetowany lub wyłączony.

T0TCR - 0xE000 4004
T1TCR - 0xE000 8004

TC – (*Timer Counter*) rejestr 32-bitowy zliczający czas, który jest inkrementowany, gdy **PC** (*Prescale Counter)* osiągnie wartość zapisaną w **PR** (*Prescale Register)*. Liczy do osiągnięcia wartości **0xFFFFFFFF**, a następnie wstecz do wartości **0x00000000**.

T0TC - 0xE000 4008
T1TC - 0xE000 8008

PR – (*Prescale Register*) rejestr, którego wartość wyznacza ilość przerwań po których resetuje się **PC**, a inkrementuje **TC**.

T0PR - 0xE000 400C
T1PR - 0xE000 800C

PC – (*Prescale Counter*) rejestr 32-bitowy zliczający ilość przerwań **PCLK** od ostatniego resetu.

T0PC - 0xE000 4010
T1PC - 0xE000 8010

MCR – (*Match Control Register*) rejestr używany do kontroli jakie operacje powinny zostać wykonane w przypadku gdy **MR(0-3)** pokrywają się z **TC**.

T0MCR - 0xE0004014
T1MCR - 0xE000 8014

MR0 – (*Match Register*) rejestr porównywany do **TC** w celu wykonania jakiejś akcji w przypadku równości wartości tych rejestrów.

T0MR0 - 0xE000 4018
T1MR0 - 0xE000 8018

W naszym przypadku wykorzystujemy drugi timer przy inicjalizacji **ADC**. Inicjalizacja timera przebiega następująco:

1. Należy zatrzymać oraz zresetować timer oraz ustawić wartość Prescaler'a na 0:

```
T1TCR = 0X02; -> resetuje TC oraz PC (0x02 włącza bit 1)
T1PR = 0X00; -> wartość prescaler'a na 0
```

2. Konfigurujemy timer wpisując w rejestr **MR0** sparametryzowaną wartość, która będzie wyznaczała czas opóźnienia.

```
T1MR0 = delayInMs * (CORE_FREQ / PBSD / 1000);
CORE_FREQ = FOSC * PLL_MUL;
FOSC = 14745000;  <- częstotliwość pracy zegara
PLL_MUL = 4;      <- mnożnik częstotliwości zegara
PBSD = 1;         <- (Peripheral bus speed divider)
delayInMs         <- parametr przekazywany przez użytkownika (odpowiada
                     opóźnieniu w ms)
```

3. Resetujemy wszystkie flagi przerwań ustawiając wszystkie bity rejestru na stan wysoki.

```
T1IR = 0XFF;
```

4. Ustawiamy funkcje, która powinna zostać wykonana przy pokryciu się **TC** z **MR0**.
```
T1MCR = 0X04; <- ustawiamy bit 2 pod symbolem MR0S dzięki czemu timer
                 zostanie zatrzymany w przypadku pokrycia
```
5. Startujemy timer i zatrzymujemy się w procedurze.
```
T1TCR = 0X01; <- ustawiamy bit 0 na stan wysoki i rozpoczynamy tym
                 samym pracę timer'a


while(T1TCR & 0x01); <- zatrzymujemy procedure dopóki działa timer
```

***PCLK** – (*Peripherial Clock*) cykle zegara peryferyjnego.

3. Analiza skutków awarii

Ryzyko	Prawdop.	Skutki	(Samo) wykrywalność	Reakcja i jej koszt	Iloczyn
Mikrokontroler	0.01	Krytyczne (10)	Niewykrywalne (10)	10.0 Wymiana płytki?	10
GPIO	0.05	Krytyczne (10)	Niewykrywalne (10)	8.0 Sprawdzenie poprawności działania pinów.	40
ADC	0.05	Krytyczne (10)	Brak reakcji na ruch płytką. (10)	6.0 Ponowna inicjalizacja.	30
ACC	0.02	Krytyczne (10)	Brak reakcji na ruch płytką. (10)	10.0 Sprawdzenie napięcia na analogowym wyjściu.	20
SPI	0.05	Krytyczne (10)	Brak komunikacji z ekranem. (10)	6.0 Zresetowanie rejestrów i ponowna inicjalizacja	30
Ekran LCD	0.05	Krytyczne (10)	Wyświetla się tylko czarny ekran. (10)	10.0 Brak reakcji.	50
I^2C	0.05	Znikome (2)	Sprawdzenie rejestru (I2C_STAT – 0xE001C004) wykrycie na podstawie kodu błędu. (5)	4.0 Brak reakcji.	5
pca9532	0.1	Znikome (2)	Brak inicjalizacji. Nie działają ledy przy ekranie. (10)	2.0 Ponowienie próby inicjalizacji przy nowej grze.	4
Timer	0.05	Znikome (2)	Niewykrywalne (10)	1.0 Brak reakcji.	2

4. Bibliografia

[1] Serial Peripheral Interface – Wikipedia, wolna encyklopedia
[2] Analog-to-Digital Converter – ElectroWings
[3] Experiment_Expansion_Board_Users_Guide-Rev_A.pdf
[4] QuickStart_Program_Development_Users_Guide-Version_1.0_Rev_A.pdf
[5] UM10120.pdf