MSP430

Pierwsze kroki, obsługa portów wejścia/wyjścia

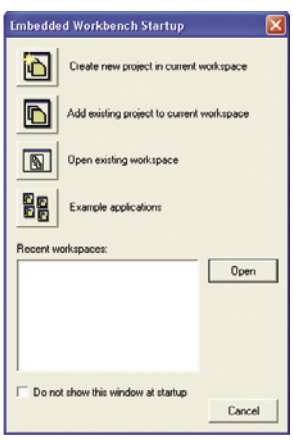
Nauka obsługi portów wejścia/wyjścia zazwyczaj stanowi pierwszy krok podczas pracy z „nową" rodziną mikrokontrolerów. Mówi się, że pierwszy krok bywa najtrudniejszy. Rzeczywiście jest w tym stwierdzeniu ziarno prawdy. My jednak zrobimy go wspólnie, a co za tym idzie nie napotkamy na trudności.

Sprzęt

W artykule będą omawiane praktyczne przykłady bazujące na łatwej w użyciu płytce ewaluacyjnej eMeSPek. Oczywiście Czytelnik może wykorzystać inny zestaw ewaluacyjny z mikrokontrolerem MSP430 serii 1xx, jednak wszelkie odniesienia do sprzętu zawarte w artykule będą dotyczyły tylko i wyłącznie płytki ewaluacyjnej eMeSPek.

Programator

Do pracy ze sprzętem będziemy również potrzebowali programatora. Płytka eMeSPek determinuje jego wybór. Programator musi być kompatybilny ze standardem MSP–FET430PIF firmy TI (jego specyfikacja znajduje się w dokumencie *slau138j.pdf*, do pobrania ze strony Texas Instruments). Nie musimy jednak zaopatrywać się w oryginalny typ programatora. Jego odpowiednik jest do kupienia w sklepie AVT (AVT1409). Ewentualnie Czytelnik może samodzielnie skonstruować programator, w czym pomocna będzie lektura archiwalnych numerów EP3/2005, 6/2007 i 12/2007.



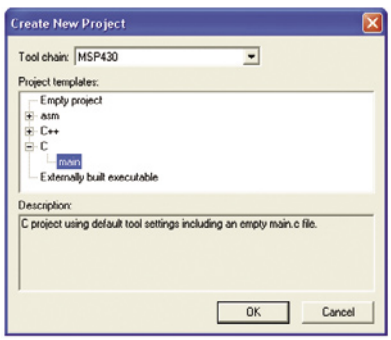
Rys. 1.

Środowisko programistyczne

Zanim rozpoczniemy programowanie, musimy również wybrać środowisko programistyczne. Po wnikliwych rozważaniach zdecydowałem, że będzie to środowisko IAR Embedded Workbanch. Wybór był trudny, gdyż istnieje kilka sprawdzonych środowisk m.in.: CrossWorks, CCE430, MSPGCC. O moim wyborze zdecydował fakt, że środowisko IAR od wielu lat jest wspierane przez producenta MSP430, firmę Texas Instruments. Niestety, nie jest to środowisko darmowe, jednak producent, firma IAR, na swojej stronie internetowej (*www.iar.com*) udostępnia dwie wersje demonstracyjne. Pierwsza z nich (*30-day evaluation edition*) jest wersją w pełni funkcjonalną, jednakże ograniczoną czasowo – 30 dni. Druga wersja (*KickStart Edition*) nie ma limitu czasowego, ograniczona jest natomiast wielkość kodu wynikowego. Dla programów pisanych w C, C++ jest to 4 kB. Programy pisane w asemblerze nie posiadają takiego ograniczenia. Na potrzeby artykułu została wybrana wersja z limitem kodu dla języków wysokiego poziomu, czyli *KickStart Edition*. Program w wersji 4.11 znajduje się na płycie CD EP1/2009B. Instalacja środowiska firmy IAR jest na tyle prosta, że nie wymaga opisu. Podczas instalacji należy wybrać wersję pełną (spośród możliwości wyboru pomiędzy opcjami „full" a „custom" wybieramy „full").

Pierwszy projekt

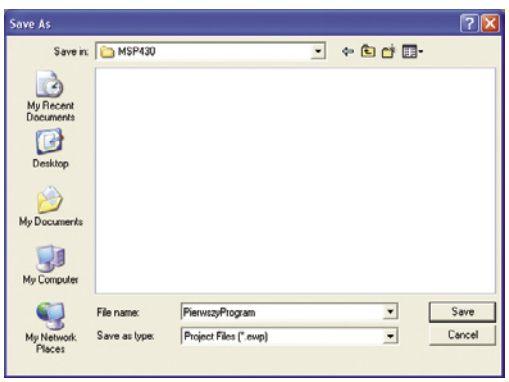
Po pomyślnym zainstalowaniu środowiska IAR można je uruchomić. Na ekranie komputera pojawi się okno jak na na **rys. 1**. Spośród dostępnych kontrolek wybieramy *Create new projekt in current workspace*, czyli utworzenie nowego projektu w bieżącym obszarze roboczym. Pojawi się wówczas okno (**rys. 2**), w którym możemy dokonać wyboru rodzaju uruchamianego projektu. Do wyboru mamy: projekt pusty, asm, C++ oraz C. Zaznaczmy projekt w C oraz zatwierdzamy wybór przy użyciu przycisku OK. Następnie zapisujemy projekt pod wybraną przez nas nazwą – przykładowo *PierwszyProgram* (**rys. 3**) oraz oczekujemy otwarcia okna z nowym projektem (**rys. 4**).



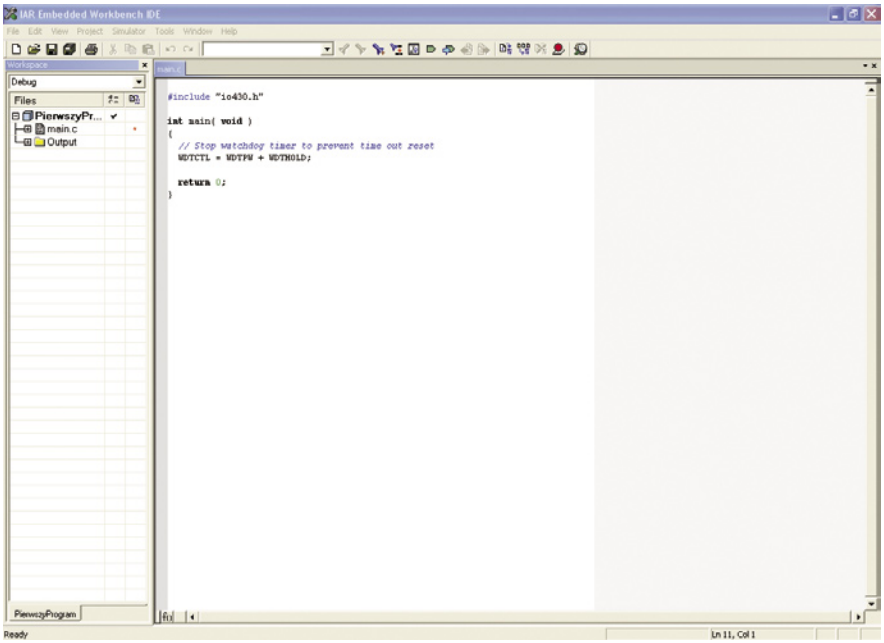
Rys. 2.

Pierwszy program – założenia

Nowo utworzony projekt w pliku *main.c* zawiera zaproponowany przez środowisko IAR kod programu. Ten kod to szablon pustego projektu (załączenie definicji rejestrów, zatrzymanie *watchdog*). My jednak chcemy napisać swój własny, pierwszy program i uruchomić go na płytce ewaluacyjnej – eMeSPek. Nasuwa się pytanie: Jaki to ma być program? W przypadku programistów dla komputera PC odpowiedź byłaby prosta, gdyż zazwyczaj pierwszym programem jest wyświetlenie komunikatu „Hello World". Podążając tym tropem my również napiszmy „Hello World", co w naszym przypadku będzie oznaczało, że zostanie zaświecona dioda LED. Będziemy musieli skonfigurować odpowiednie piny, czyli przejść do meritum naszego kursu – obsługi portów I/O. Zanim jednak to zrobimy, Czytelnik musi zapoznać się ze specyfiką obsługi tych portów w mikrokontrolerach MSP430.



Rys. 3.



Rys. 4.

Ramka 1. Pobieranie środowiska IAR KickStart
Krok 1.
Na stronie producenta *www.ti.com/msp430* spośród dostępnych zakładek wybieramy Tools & Software.
Krok 2.
U dołu strony odszukujemy akapit Software Development Tools oraz z ramki o takiej samej nazwie wybieramy środowisko IAR-KICKSTART.
Krok 3.
W przeglądarce otworzy się nowe okno z informacjami o kompilatorze. Po ewentualnym zapoznaniu się z nimi przejdźmy do kolejnego etapu pobierania środowiska IAR-KISKSTART (wciśnięcie przycisku Download & Register).
Krok 4.
Również i tym razem w przeglądarce otworzyło nam się nowe okno – z tą różnicą, że teraz zostało użyte bezpieczne połączenie internetowe https. Dlaczego został zwiększony poziom bezpieczeństwa? Otóż pobranie środowiska IAR-KISKSTART wymaga podania danych do konta TI. W tym celu przejdźmy do widocznego na stronie formularza i w pola Email Address oraz Password odpowiednio wpiszmy adres e-mail oraz hasło podane podczas rejestracji konta TI, po czym logując się (Log In) rozpocznijmy pobieranie środowiska IAR-KISKSTART. W przypadku, gdy użytkownik nie posiada konta TI; a co za tym idzie danych do uzupełnienia pól formularza; to konieczne będzie jego utworzenie („Sign Up").

Teoria

W mikrokontrolerach MSP430 porty I/O są rejestrami 8–bitowymi. Wstępnie można je podzielić na porty o funkcjonalności podstawowej oraz na porty mogące pracować jako wejścia przerwania zewnętrznego. W przypadku mikrokontrolera znajdującego się na płytce ewaluacyjnej eMeSPek – MSP430f1232 mamy do dyspozycji trzy porty I/O: P1, P2, P3. Porty P1 oraz P3 są 8–bitowe, natomiast port P2 jest niekompletny, ma wyprowadzone jedynie 6 bitów. W sumie, mamy do dyspozycji 22 wyprowadzenia. Każdy z dostępnych pinów możemy skonfigurować jako wejście, bądź jako wyjście. Aby tego dokonać należy ustawić w rejestrze PxDIR (gdzie x oznacza numer portu) bit odpowiadający za konfigurację odpowiedniego pinu (**Ramka 1**). Zapis jedynki powoduje konfigurację powiązanego pinu jako wyjście, natomiast zapis zera, konfigurację jako wejście. Przykładowo, gdy chcemy, aby pin 3 portu pierwszego był skonfigurowany jako wyjście, powinniśmy użyć zapisu: `P1DIR_3=1`, natomiast gdy chcemy, aby ten pin był skonfigurowany jako wejście, wówczas poprawny jest zapis: `P1DIR_3=0`.

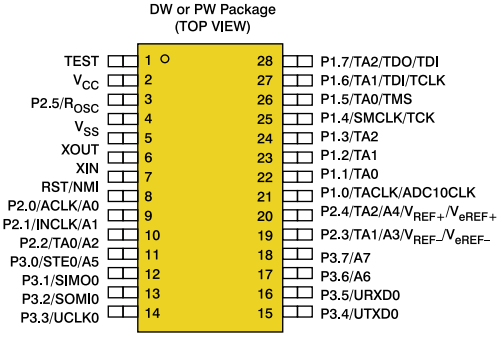
W przypadku, gdy pin został skonfigurowany jako wyjście istnieje możliwość ustawiania jego wartości logicznej. Aby tego dokonać należy ustawić w rejestrze PxOUT (gdzie x oznacza numer portu) bit odpowiadający za wartość logiczną pinu. W momencie, gdy ustawioną wartością będzie zero, pin zostanie ustawiony w stan niski (np. `P1OUT_3=0`. W przeciwnym wypadku, gdy zostanie ustawiona jedynka (`P1OUT_3=1`), pin zostanie ustawiony w stan wysoki. W sytuacji, gdy poprzez ustawienia w rejestrze PxDIR, pin został skonfigurowany jako wejście, wówczas istnieje możliwość odczytywania jego aktualnego stanu. Informacje o aktualnym stanie pinu wejściowego są przechowywane w rejestrze PxIN (gdzie x oznacza numer portu). Odczyt odbywa się poprzez odczytanie wartości bitu odpowiadającego za dany pin (`P1IN_3`). Gdy odczytany bit będzie miał wartość zero, będzie to równoznaczne ze stanem niskim na wejściu, natomiast, gdy bit będzie miał wartość jeden, będzie to oznaczać, że w danym momencie na wejściu jest stan wysoki.

W tej chwili osoby, które uważnie przyjrzały się wyprowadzeniom procesora MSP430f1232 przedstawionym na **rys. 5** zapewne zastanawiają się, dlaczego przy pinach oprócz oznaczeń portów istnieją również inne oznaczenia? Otóż w procesorach MSP430 piny nie pełnią wyłącznie funkcji linii I/O, ale mogą również pełnić funkcję doprowadzeń układów peryferyjnych. I tak na przykład nóżka 15 mikrokontrolera może pełnić funkcję pinu I/O (P3.4), jak również może być wyprowadzeniem TX układu peryferyjnego UART0 (UTXD0). Z kolei nóżka 18 oprócz

Ramka 2. Konfiguracja portów I/O.
Tak jak wcześniej wspominałem porty wejścia/wyjścia są ośmiobitowymi mapowanymi w pamięci FLASH mikrokontrolera rejestrami. W przypadku MSP430f1232 adresy tych rejestrów są następujące.

Port 1		Port 2		Port 3	
Tag	Adres	Tag	Adres	Tag	Adres
P1SEL	0x26	P2SEL	0x2E	P3SEL	0x1B
P1IE	0x25	P2IE	0x2D	P3DIR	0x1A
P1IES	0x24	P2IES	0x2C	P3OUT	0x19
P1IFG	0x23	P2IFG	0x2B	P3IN	0x18
P1DIR	0x22	P2DIR	0x2A		
P1OUT	0x21	P2OUT	0x29		
P1IN	0x20	P2IN	0x28		

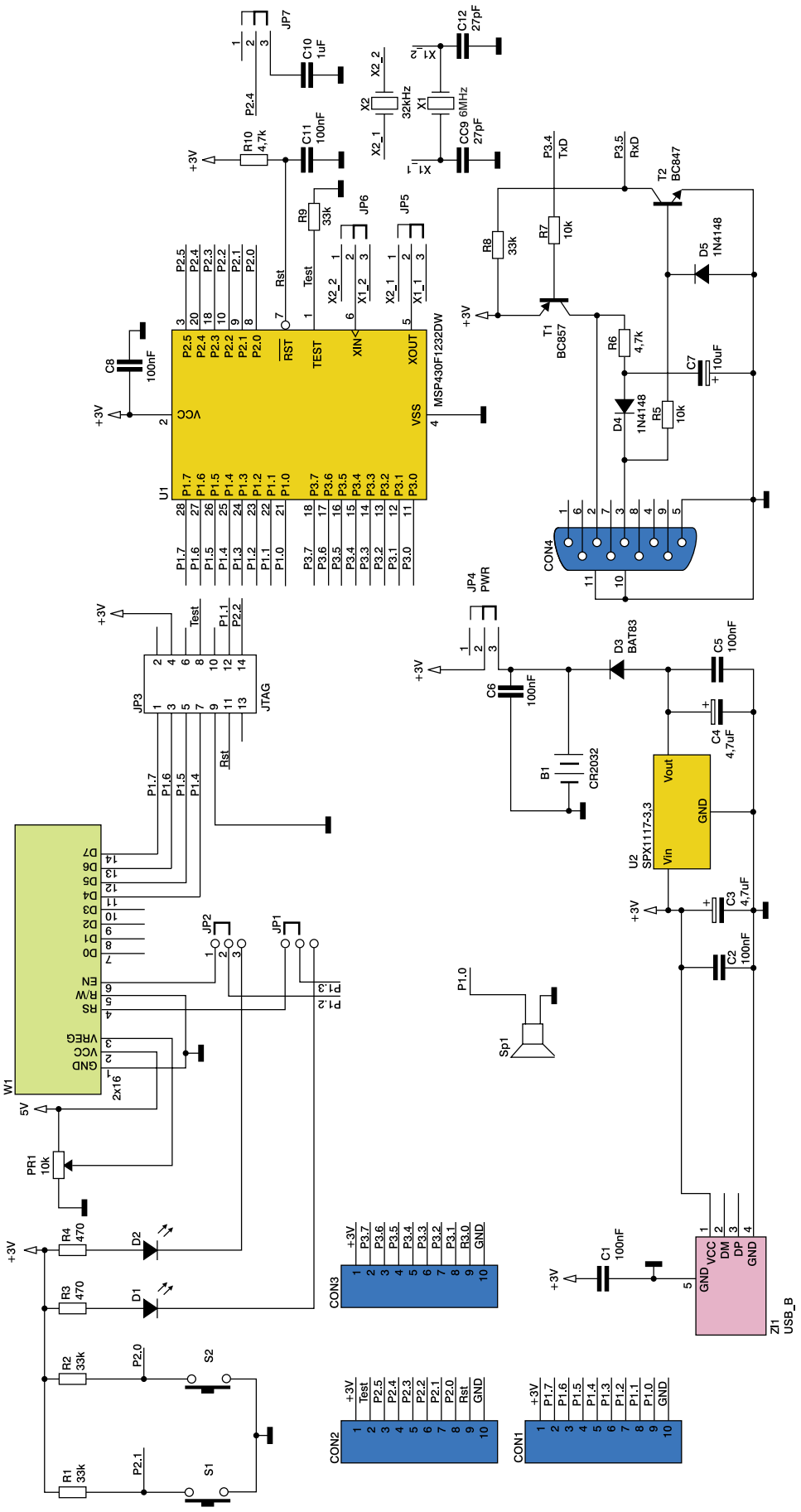
Programista podczas konfigurowania portów wejścia/wyjścia wykonuje operacje na adresach rejestrów. Przykładowo ustawienie wyjść portu pierwszego w stan wysoki wyglądało by tak: `*(unsigned char *)0x22 = 0xff;`. Jak widać, taki zapis choć poprawny nie jest zbyt wygodny. Programista musi bowiem znać adresy rejestrów. Dlatego też, w celu ułatwienia życia programiście, adresy rejestrów zostały przypisane do tagów. Aby programista w swoim projekcie mógł korzystać z tagów, musi oczywiście dołączyć plik z ich definicjami. W przypadku IAR'a programista nie musi załączać pliku pod konkretny typ mikrokontrolera (w naszym przypadku tym plikiem byłby io430x12x2.h) wystarczy, że załączy plik io430.h, a całą resztę zajmie się za niego kompilator.
W pliku z definicjami zostały również zdefiniowane struktury opisujące rejestry. Korzystanie z tych struktur umożliwia dostęp do poszczególnych bitów w rejestrze. Taka możliwość jest bardzo przydatna w przypadku konfigurowania portów wejścia/wyjścia, gdyż łatwy dostęp do bitów w rejestrze przekłada się na łatwy dostęp do pinów w porcie. Przykłady jak korzystać ze struktur znajdują się w dalszej części artykułu. Dlatego też proponuję powrócić do lektury.



Rys. 5. Procesor HSP430f1232

funkcji I/O (P3.7) może pełnić funkcję pinu peryferyjnego układu ADC10 (A7 – 7 kanał układu przetwornika A/D).

Aby skonfigurować pin jako peryferyjny, należy w rejestrze PxSEL (gdzie x oznacza numer portu) ustawić bit odpowiadający za jego funkcję. Gdy taki bit zostanie ustawiony (np. `P1SEL_3=1`), wówczas zostanie włączona peryferyjna funkcja pinu (pin przestaje funkcjonować jako pin I/O, gdy był wejściem przerwania traci swoją funkcjonalność). Wyzerowanie bitu (np. `P1SEL_3=0`) powoduje przywrócenie uprzedniej funkcji (wejścia/wyjścia, bądź wejścia przerwania). W przypadku niektórych peryferiów (przykładowo TimerA) i związanych z nimi doprowadzeń (P1.1...P1.3), skonfigurowanie pinu jako peryferyjnego nie jest wystarczające do uzyskania funkcjonalności peryferyjnej. Dodatkowo należy skonfigurować PxDIR pinu (w przypadku TimerA jako wyjście). Dzieje się tak, gdyż komenda PxSEL nie konfiguruje automatycznie rejestru PxDIR dla określonego pinu. Do tych odstępstw od reguły również powrócimy przy innej okazji.



Rys. 6. Schemat elektryczny płytki ewaluacyjnej eMeSPek

Każdy pin portu pierwszego P1 i portu drugiego P2, może zostać skonfigurowany jako wejście przerwania. Konfigurację taką możemy osiągnąć w wyniku ustawiania rejestrów PxIFG, PxIE, PxIES (gdzie x oznacza numer portu – 1 lub 2). Dodatkowo, z każdym z portów powiązana jest procedura obsługi przerwania. Aby skonfigurować pin jako wejście przerwania, należy aktywować przerwania od tegoż pinu. Do tego celu wykorzystywany jest rejestr PxIE. Ustawienie w tym rejestrze bitu właściwego dla danego pinu aktywuje przerwania (np. `P1IE_3=1` oznacza aktywowanie przerwań od trzeciego pinu portu pierwszego), natomiast wyzerowania tegoż bitu dezaktywuje przerwania (`P1IE_3=0`). W momencie, gdy przerwanie od danego pinu zostało aktywowane, należy jeszcze zadecydować, które zbocze sygnału będzie je wyzwalało. Taką decyzję możemy podjąć dokonując wpisu do rejestr PxIES. Gdy chcemy, aby przerwanie było aktywowane w momencie przejścia ze stanu wysokiego na niski, wówczas w rejestrze ustawiamy określony bit (np. `P1IES_3=1`), natomiast w przypadku, gdy chcemy, aby przerwanie było aktywowane w momencie zmiany stanu z niskiego na wysoki, w rejestrze odpowiedni bit zerujemy (np. `P1IES_3=0`). W chwili, gdy zaistnieją skonfigurowane przez nas warunki wystąpienia przerwania, procesor w rejestrze PxIFG automatycznie ustawi bit informujący o zajściu przerwania na danym pinie oraz rozpocznie wykonywanie procedury obsługi przerwania.

Należy zwrócić uwagę na to, że bit wystąpienia przerwania (w rejestrze PxIFG) nie jest automatycznie zerowany w chwili zakończenia procedury obsługi przerwania. Jego wyzerowanie (np. `P1IFG_3=0`) jest obowiązkiem programisty!

W **ramce 2** zawarto swojego rodzaju ściągę z obsługi portów I/O w procesorach MSP430 z rodziny 1xx.

Pozostała jeszcze nierozwiązana kwestia konfiguracji niewykorzystywanych pinów. Należy je konfigurować jako wejścia (niefunkcyjne) (rejestr PxDIR) i wymusić na nich stan niski. Ma to ścisły związek z oszczędnością energii.

Pierwszy program – implementacja

Skoro „uporaliśmy" się już z częścią teoretyczną, przejdźmy do części praktycznej. Na początek, zgodnie z naszymi wcześniejszymi ustaleniami, napiszmy program „migająca dioda". Przyjmijmy, że częstotliwość migania diody będzie wynosiła 1 Hz, a wypełnienie sygnału będzie równa 50%. Musimy zatem doprowadzić do sytuacji, w której dioda będzie zgaszona przez pół sekundy, po czym będzie się świecić przez pół sekundy, a cały proces będzie się powtarzał cyklicznie. W ten sposób uzyskamy jedno mignięcie diody na sekundę, z czasem świecenia/zgaszenie równym pół sekundy. Do odmierzania czasu wykorzystajmy wbudowaną procedurę kompilatora IAR `__delay_cycles(unsigned long __cycles)`. Jej deklaracja znajduje się w pliku nagłówkowym *intrinsics.h*, dlatego konieczne będzie jego załączenie (`#include „intrinsics.h";`). Gdy mamy już sprecyzowany algorytm pracy, nie pozostaje nam nic innego, jak dobrać wartość parametru `__cycles` wykorzystywanego w procedurze `__delay_cycles`, tak aby móc odczekać wymagany czas. Po starcie mikrokontroler pracuje z częstotliwością około 720 kHz, czyli na pół sekundy przekłada się 360000 cykli zegara. Skoro wiemy już ile cykli zegarowych ma trwać opóźnienie, możemy wrócić do projektu „PierwszyProgram" utworzonego na początku artykułu, i postępując zgodnie z algorytmem działania, zaimplementować kod programu.

Jak widać na schemacie z **rys. 6**, eMeSPek posiada dwie diody podłączone do pinów portu pierwszego mikrokontrolera. Włączanie diod odbywa się niskim stanem na wyjściu mikrokontrolera, wyłączanie natomiast wysokim. Należy również pamiętać o aktywowaniu diod, a mianowicie o ustawieniu zworek JP1 oraz JP2 w pozycji 2,3. Ustalmy jeszcze, że dioda, która będzie migać, to dioda D1 (P1.3). Znając wszystkie te szczegóły przyjedźmy do implementacji algorytmu. Zadanie to pozostawiamy Czytelnikowi do samodzielnego wykonania.

Pierwszy program – uruchomienie

Zakładam, że Czytelnik uporał się z implementacją algorytmu oraz, że projekt, po uprzednim zapisaniu na dysku (*PierwszyProgram.eww*) kompiluje się (*Project –> Compile*) bezbłędnie i jest gotowy do zaprogramowania urządzenia. Przejdźmy, zatem do opcji projektu (*Project –> Options*). Z dostępnych kategorii wybierzmy *Debugger*. W zakładce *Setup* zmieniamy *Driver* z *Simulator* na *FET Debugger*, po czym zatwierdzamy dokonane zmiany przy użyciu przycisku „OK". W tym momencie środowisko IAR przestało pracować w trybie symulatora, a rozpoczęło w trybie *debuggera*, co oznacza, że możliwe będzie programowanie oraz debuggowanie urządzenia. Zanim do tego przejdziemy, ustawmy dodatkowo w opcjach projektu weryfikację poprawności wgranego do urządzenia programu oraz wybierzmy rodzaj stosowanego przez nas programatora i typ używanego mikrokontrolera. Aby wymusić weryfikację procesu programowania ponownie przejdźmy do opcji projektu (*Project –> Options*) oraz w kategorii *FET Debugger* w zakładce *Download* zaznaczmy *Verify download*. Teraz wybierzmy rodzaj stosowanego przez nas programatora. W tym celu przełączmy się pomiędzy zakładkami – z *Download* na *Setup* i spośród dostępnych programatorów wybierzmy ten, który zastosujemy, a mianowicie *Texas Instruments LPT–IF*. Na koniec, przejdźmy do kategorii *General Options* oraz w rodzinie MSP430x1xx odszukajmy mikrokontroler MSP430f1232. Zatwierdźmy nasz wybór i po uprzednim podłączeniu układu startowego do programatora oraz podaniu zasilania (z jednego z dostępnych źródeł) rozpocznijmy programowanie (*Projekt –> Debug*). Podczas programowania ujrzymy informacje o kasowaniu pamięci mikrokontrolera oraz o wgrywaniu i sprawdzaniu poprawności programu, po czym mikrokontroler zostanie zaprogramowany. Wówczas należy rozpocząć wykonywanie programu (*Debug –> Go*). Zakładam, że nie było żadnych problemów i w tym momencie dioda miga w taki sposób, jak sobie zakładaliśmy. W takim razie zakończmy działanie debugerra, wracając do edytora projektu (*Debug –> Stop Debugging*) i wzbogaćmy nasz program o obsługę wejść.

Tab. 1. Wykaz rejestrów

Nazwa rejestru	Opis	Prawa dostępu	Bit jeden	Bit zero
PxDIR	wybór „kierunku" rejestru	odczyt/zapis	wyjście	wejście
PxIN	odczyt stanu wejścia	tylko odczyt	stan wysoki	stan niski
PxOUT	stan wyjściowy	odczyt/zapis	stan wysoki	stan niski
PxSEL	wybór funkcjonalności pinu	odczyt/zapis	pin funkcyjny	pin wejścia/wyjścia
PxIE	włączenie/wyłączenie przerwań	odczyt/zapis	przerwanie włączone	przerwanie wyłączone
PxIES	wybór zbocza sygnału wyzwalającego przerwanie	odczyt/zapis	zbocze opadające (zmiana ze stanu wysokiego na niski)	zbocze narastające (zmiana ze stanu niskiego na wysoki)
PxIFG	flaga przerwania	odczyt/zapis	„wejście" do procedury obsługi przerwania	program nie „wchodzi" do procedury obsługi przerwania

Pierwszy program – obsługa wejść

Tak jak to robiliśmy w przypadku diod, również i tym razem zajrzyjmy do schematu układu startowego (rys. 6) i przyjrzyjmy się podłączeniu przycisków. W momencie, gdy przycisk jest zwolniony, na wejściu pinu mikrokontrolera utrzymuje się stan wysoki, natomiast wciśnięcie przycisku powoduje zwarcie do masy oraz zmianę stanu wejścia mikrokontrolera na stan niski. W naszym programie użyjemy obu przycisków S1 oraz S2. Stan pierwszego przycisku S1 będziemy sprawdzać cyklicznie, co jedną sekundę i odpowiednio sterować diodą D1 (przycisk wciśnięty – brak migania, przycisk zwolniony – dioda D1 miga). Miganie ma się odbywać zgodnie z wytycznymi z pierwszej wersji programu). Drugi przycisk S2 będzie sterował diodą D2. W chwili, gdy przycisk S2 będzie wciśnięty, włączymy diodę D2. Gdy nie będzie wciśnięty, dioda D2 będzie wyłączona. Tak się składa, że przycisk S2 został podłączony do portu drugiego (S2 – P2.0), czyli wejścia przerwania i mamy możliwość reagowania na zmianę jego stanu dzięki obsłudze przerwania. Wystarczy, że odpowiednio go skonfigurujemy. W odniesieniu do naszego przykładu „odpowiednio" oznacza włączenie przerwań od S2, skonfigurowanie S2 jako wejście, wybranie zbocza sygnału (S2 – wciśnięty, S2 – puszczony) po wystąpieniu, którego program ma rozpocząć wykonywanie procedury obsługi przerwania (szablon procedury obsługi przerwania zarówno dla portu pierwszego, jak i drugiego pokazano na **list. 1)**. W procedurze obsługi przerwania od portu drugiego musimy zawrzeć logikę działania algorytmu pracy z przyciskiem S2. Mianowicie w chwili, gdy wejdziemy do procedury obsługi przerwania w wyniku zmiany stanu sygnału na wejściu z wysokiego na niski (wciśnięcie przycisku), powinniśmy włączyć diodę D2. W przeciwnym wypadku, gdy wejście do procedury obsługi przerwania było wywołane zmianą sygnału na wejściu ze stanu niskiego na wysoki (puszczenie przycisku), powinniśmy wyłączyć diodę D2. Każdorazowo przed wyjściem z procedury obsługi przerwania powinniśmy zmienić kierunek zbocza wyzwalającego na przeciwny oraz wyzerować flagę przerwania (`P2IFG_0=0`). Dodatkowo, aby móc obsługiwać przerwania, musimy włączyć ich obsługę. Można to zrobić korzystając z komendy `__enable_interrupt()`. Komenda ta znajduje się w pliku nagłówkowym *intrinsics.h*, który to już wcześniej załączyliśmy do naszego projektu.

Pierwszy program – opcje debuggera

Gdy program będzie już gotowy, można zaprogramować mikrokontroler (*Project –> Debug*). Po zaprogramowaniu, zanim uruchomimy program, otwórzmy okno podglądu rejestrów (*View –> Register*) oraz podglądu zmiennych (*View –> Watch*), dodatkowo zamknijmy niewykorzystywane okno *disassembly*. Nasz projekt powinien wyglądać tak, jak to pokazano na **rys. 7**.

Gdy tak jest, można uruchomić program (*Debug –> Go*) oraz sprawdzić prawidłowość jego działania. Zakładam, że program działa zgodnie z naszymi ustaleniami. Podbudowani sukcesem nie kończmy jeszcze pracy z debuggerem i zatrzymajmy pracę programu ustawiając pułapkę w dowolnym miejscu pętli `while` (*Edit –> Toggle Breakpoint*. Pułapka sprzętowa zostanie ustawiona w linii, w której znajdował się kursor. Zrobiliśmy to, by poznać sposób zmiany wartości zmiennych oraz wartości rejestrów w chwili debuggownia programu w środowisku IAR.

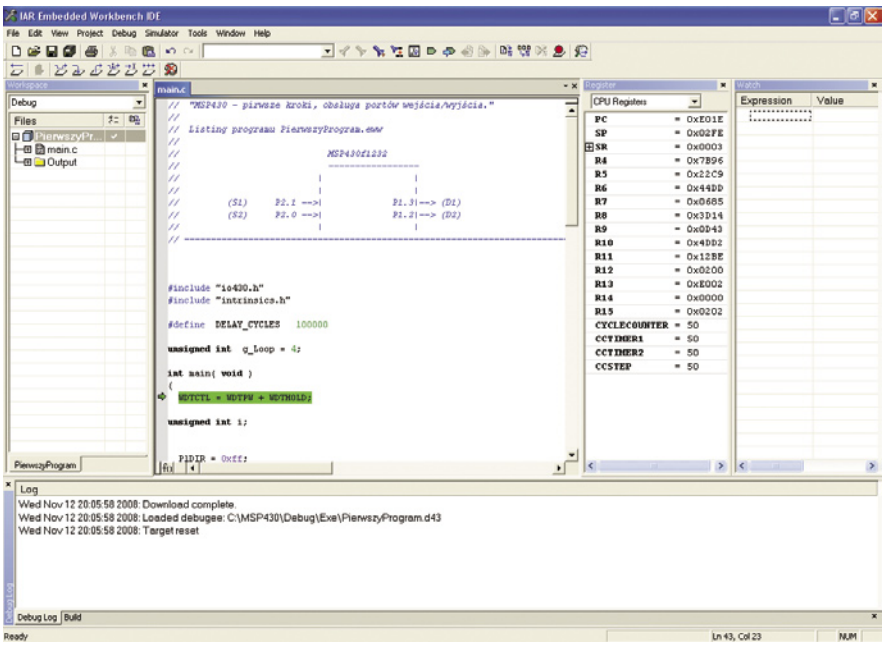
Przejdźmy, zatem do okna z rejestrami i z dostępnych na liście rozwijalnej parametrów wybierzmy *Port 1/2*. Następnie w rejestrze P1OUT, zmieńmy wartość drugiego lub trzeciego bitu. Spowoduje to zmianę stanu diody, co jest dowodem na fizyczną zmianę stanu na odpowiednim wyjściu mikrokontrolera. Korzystając z faktu, że mamy otwarte okno podglądu zmiennych (*Watch*) dodajmy do tego okna zmienną `g_Loop`. Gdy już to zrobimy, to momentalnie przy zmiennej pojawi się jej wartość, czyli 4. Zmieńmy ją na 2, następnie zdejmijmy pułapkę (*Edit –> Toggle Breakpoint*) i uruchommy pogram (*Debug –> Go*). Jak się można było domyślić, dioda D1 miga dwukrotnie szybciej, co jest powiązane z dwukrotnym zmniejszeniem wartości zmiennej `g_Loop`. W podobny sposób możemy podglądać i ustawiać pozostałe zmienne oraz rejestry. Zachęcam Czytelnika do samodzielnych eksperymentów. Będzie to znakomity sposób na utrwalenie wiedzy nabytej podczas lektury tego odcinka kursu, jak również skłoni do własnych przemyśleń.

Łukasz Krysiewicz

List. 1. Sposób implementacji procedur obsługi przerwań

```
/*
  Procedura obsługi przerwania, dla portu pierwszego mikrokontrolera.
*/
#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
{
        ......
}

/*
  Procedura obsługi przerwania, dla portu drugiego mikrokontrolera.
*/
#pragma vector=PORT2_VECTOR
__interrupt void Port_2(void)
{
        ......
}
```



Rys. 7.