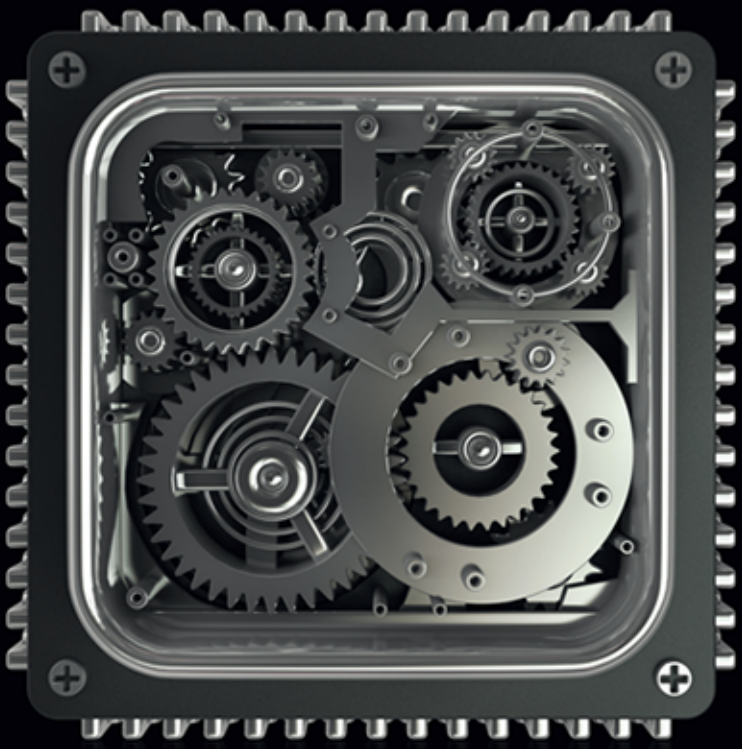
Tomasz Francuz

AVR

Praktyczne projekty



Programowanie mikrokontrolerów to nic trudnego!
Czas się o tym przekonać!

Poznaj mikrokontrolery AVR z rodziny XMEGA
Naucz się praktycznie programować je w języku C
Zdobądź doświadczenie w stosowaniu układów AVR

Helion

Redaktor prowadzący: Michał Mrowiec

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Fotografia na okładce została wykorzystana za zgodą Shutterstock.com

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie?avrppr
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Kody źródłowe wybranych przykładów dostępne są pod adresem:
ftp://ftp.helion.pl/przyklady/avrppr.zip

ISBN: 978-83-246-7877-8

Printed in Poland.

Spis treści

Rozdział 10.

Przerwania i kontroler przerwań

Przerwania

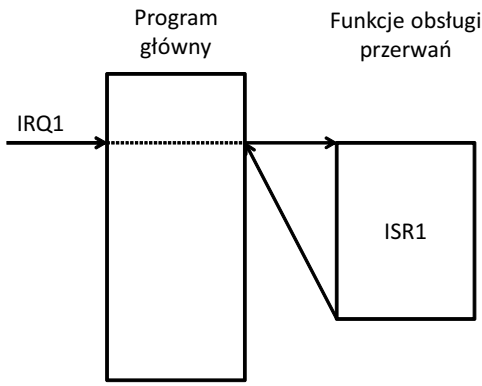
Przerwania są jednym z najważniejszych elementów w świecie mikrokontrolerów, z drugiej strony ich poprawne wykorzystanie i oprogramowanie nastręcza najwięcej trudności. Rozdział ten większość z nich powinien wyjaśnić i zachęcić nas do szerokiego stosowania narzędzia, które właściwie wykorzystane daje niesamowite możliwości. Jest to o tyle istotne, że przykłady znajdujące się w kolejnych rozdziałach w znakomitej większości bazują na przerwaniach.

Czym są przerwania?

Na to pytanie najprościej odpowiedzieć, pokazując przykład z życia codziennego. Wyobraź sobie, że grasz w swoją ulubioną grę i nagle dzwoni telefon. W tym momencie istnieją dwie możliwości:

1. Zignorowanie telefonu (brak odpowiedzi na przerwanie).
2. Odebranie telefonu (obsługa przerwania).

Jednak trudno jest jednocześnie skupić się na dwóch różnych rzeczach (np. grze we wciągającą grę i rozmowie przez telefon). Trzeba więc je zrobić sekwencyjnie — przerwać grę, zatrzymać ją w pewnym punkcie (np. zapisać stan gry), a następnie odebrać telefon i rozpocząć rozmowę. Po jej zakończeniu można wrócić do przerwanej gry, odtwarzając jej stan np. z zapisanego pliku na dysku. Dokładnie tak samo wygląda obsługa przerwań w świecie mikrokontrolerów. Ponieważ CPU mikrokontrolera XMEGA w danym czasie potrafi realizować tylko jeden wątek, na czas przerwania jego wykonanie musi zostać przerwane, a jego stan zapisany. Dzięki temu procesor będzie mógł rozpocząć realizację innego wątku (obsługi przerwania) — rysunek 10.1.



Rysunek 10.1. *Po otrzymaniu sygnału przerwania procesor przerywa bieżący wątek i rozpoczyna funkcję obsługi przerwania. Po jej zakończeniu wraca do przerwanego wcześniej wątku*

W prawdziwym świecie potrzebujemy na zmianę wykonywanego zadania pewien czas (zapisanie gry i podniesienie słuchawki), także mikrokontroler potrzebuje trochę czasu na obsłużenie przerwania. Czas ten — nazywany czasem latencji (ang. *Latency time*) — jest mu potrzebny na dokończenie aktualnie wykonywanej instrukcji oraz zapisanie stanu bieżącego wątku (automatycznie zapisywany jest wyłącznie rejestr stosu `SP`), dzięki temu po zakończeniu obsługi przerwania będzie mógł go wznowić. Z drugiej strony funkcja obsługi przerwania (ISR, ang. *Interrupt handler/service routine*) też potrzebuje trochę czasu — w tak zwanym prologu funkcji (generowanym automatycznie przez kompilator) zapisywane są wszystkie rejestry procesora, które funkcja zmienia. Przed jej zakończeniem ich stan jest odtwarzany w czasie wykonywania tzw. epilogu funkcji (on również jest generowany automatycznie przez kompilator). Dzięki temu przerwany wątek jest wznawiany w dokładnie takim samym stanie, w jakim został przerwany — a więc z jego punktu widzenia procesor zachowuje się tak, jakby przerwania w ogóle nie było.

Pamiętaj, że kompilator generuje wyłącznie kod zachowujący stan rejestrów procesora. Jeśli funkcja zmieni stan tzw. rejestrów IO lub pamięci, to nie zostanie on odtworzony po wyjściu z funkcji obsługi przerwania.

Przerwania maskowalne

Wróćmy jeszcze do przedstawionej wcześniej analogii. W chwili zgłoszenia przerwania (usłyszenia dzwonka telefonu) mamy możliwość zignorowania tego sygnału (żądania przerwania, ang. *Interrupt request*) lub rozpoczęcia czynności przygotowujących do obsługi tego zdarzenia. Podobny wybór ma także procesor. Przerwania można zablokować całkowicie, co jest realizowane poprzez wyzerowanie specjalnej flagi (`I`, ang. *Interrupt*) rejestru stanu procesora instrukcją `CLI` (w języku C jej odpowiednikiem jest makrodefinicja `cli()`); można je także odblokować instrukcją asemblera `SEI` (w C jej odpowiednikiem jest `sei()`). Przerwania, które możemy blokować, nazywane są **przerwaniami maskowalnymi**.

Po zablokowaniu przerwań nie będą one obsługiwane, lecz ciągle procesor będzie zapamiętywał fakt, że odpowiednie żądania zostały zgłoszone, zostaną one zrealizowane zaraz po odblokowaniu przerwań.

Przypomina to trochę sytuację, w której grając, co prawda nie odebraliśmy telefonu, ale zerknęliśmy, aby sprawdzić, kto dzwonił. Po zakończeniu gry (o ile będziemy o tym ciągle pamiętać) możemy do takiej osoby oddzwonić. Dokładnie tak samo zachowują się żądania przerwań w sytuacji, gdy przerwania są zablokowane — zostaną one zrealizowane w chwili, w której obsługa przerwań zostanie ponownie włączona. Jednak procesor ma ograniczone zdolności do zapamiętywania żądań przerwań. Zazwyczaj z każdym źródłem przerwań związana jest flaga określająca, czy dane żądanie wystąpiło, czy nie. Jedna flaga przechowująca informację bitową nie jest w stanie zapamiętać, ile żądań przerwań z danego źródła zostało zgłoszonych — w efekcie jeśli żądań było więcej niż jedno, po odblokowaniu przerwań zostanie zrealizowane wyłącznie jedno — o pozostałych procesor „zapomni". Wracając do naszej analogii, sytuacja wygląda tak, jakby osoba o danym numerze telefonu dzwoniła do nas wielokrotnie. Niezależnie od tego, ile razy zadzwoni, to i tak oddzwonimy wyłącznie raz.

Przerwania niemaskowalne

Jednak przerwania w życiu codziennym mają różną wagę. O ile możemy zignorować dzwoniący telefon, to nie powinniśmy zignorować np. informacji z naszego UPS-a, że kończy się zasilanie i komputer za chwilę zostanie wyłączony. Zignorowanie takiego ostrzeżenia może zakończyć się katastrofą (np. niezapisaniem stanu gry i utratą dotychczasowych osiągnięć). Mikrokontrolery XMEGA również dysponują taką „gorącą linią" — są to tak zwane przerwania niemaskowalne (NMI, ang. *Non-maskable interrupts*). Jak sama nazwa wskazuje, przerwań tego typu nie da się zablokować. Ich wystąpienie jest związane z zajściem jakichś krytycznych zdarzeń, które nie mogą pozostać bez obsługi. Dla XMEGA jedynym takim krytycznym zdarzeniem jest uszkodzenie zewnętrznego rezonatora kwarcowego, co powoduje utratę źródła sygnału zegarowego. Co prawda procesor XMEGA ciągle potrafi w takich okolicznościach prawidłowo działać, lecz program powinien podjąć jakieś działania — np. poinformować użytkownika o uszkodzeniu części układu elektronicznego.

Źródła przerwań

W mikrokontrolerach XMEGA każdy układ peryferyjny może być źródłem jednego lub więcej przerwań. Najczęściej wykorzystywane są przerwania generowane przez:

- porty IO (określony poziom logiczny na pinie IO lub jego zmiana);
- ADC (zakończenie konwersji, określony wynik porównania uzyskanej wartości ze wzorcem);
- timery (przerwania przepełnienia licznika, osiągnięcie przez licznik określonej wartości lub reakcja na zdarzenie zewnętrzne);

- interfejsy komunikacyjne (USART, SPI, I2C) określające wysłanie lub odebranie porcji danych;
- DMA — określające gotowość na transfer kolejnej porcji danych.

Inne układy peryferyjne, takie jak kontroler pamięci EEPROM, FLASH, moduł kryptograficzny i inne, również mogą być źródłem przerwań.

Aby móc z nich skorzystać, musimy przeprowadzić kilka etapów związanych z ich konfiguracją.

Konfiguracja i obsługa przerwań

Podsystem przerwań to układ, który w stosunku do innych rodzin AVR uległ olbrzymim zmianom. Dzięki temu XMEGA dysponują uniwersalnym kontrolerem, zdolnym realizować przerwania o różnych priorytetach, a także rozstrzygać konflikty pomiędzy przerwaniami zgłaszanymi jednocześnie. Dobre zapoznanie się z podsystemem PMIC (ang. *Programmable multi-level interrupt controller*) i umiejętność właściwego wykorzystania i zrozumienia przerwań gwarantuje sukces w świecie mikrokontrolerów.

Aby odblokować przerwania w XMEGA, musimy spełnić kilka warunków:

- należy napisać funkcję obsługi przerwania i powiązać ja z odpowiednim wektorem przerwania;
- należy ustalić poziom i odblokować możliwość zgłaszania przerwania przez wybrany układ peryferyjny;
- należy zezwolić na dany poziom przerwań w kontrolerze PMIC;
- należy odblokować globalną flagę zezwolenia na przerwania.

Widzimy, że procedura ta jest wieloetapowa i pozornie skomplikowana, jednak aby osiągnąć sukces z przerwaniami, należy dobrze przemyśleć każdy z wymienionych kroków. W dalszej części rozdziału zostaną szerzej omówione poszczególne punkty związane z konfiguracją i obsługą przerwań. W dalszej części książki zostaną też pokazane liczne przykłady praktycznego użycia przerwań.

Czas odpowiedzi na żądanie przerwania

Pomiędzy zajściem zdarzenia wywołującego przerwanie a jego obsługą przez mikrokontroler mija pewien czas. Jest on zależny od aktualnego stanu mikrokontrolera (od tego, jaką realizuje instrukcję). W związku z tym czas odpowiedzi nie jest stały, co w pewnych zastosowaniach może mieć znaczenie. Jeśli czas odpowiedzi musi być stały, musimy zastosować specjalne techniki. W XMEGA od chwili zgłoszenia przerwania do chwili wejścia do funkcji jego obsługi mija co najmniej 8 cykli, z czego 5 cykli zajmuje odłożenie na stosie adresu powrotu, a kolejne 3 cykle zajmuje wykonanie instrukcji skoku znajdującej się pod adresem danego wektora przerwania do funkcji

obsługi przerwania. Kiedy procesor w chwili zgłoszenia przerwania wykonuje instrukcję zajmującą więcej niż jeden cykl, to przed wejściem w przerwanie instrukcja ta jest najpierw kończona, co wydłuża czas potrzebny do wejścia do funkcji obsługi przerwania. Każda funkcja obsługi przerwania musi kończyć się specjalną instrukcją asemblera `RETI` — dzięki temu PMIC wie, że realizacja odpowiedniej funkcji uległa zakończeniu, i może przygotować się do obsługi kolejnego przerwania.

> Kompilując kod obsługi przerwania w języku C, kompilator sam zadba o właściwe zakończenie funkcji obsługi przerwania. Jednak wymaga to użycia specjalnego makra `ISR`, informującego kompilator, że ma do czynienia z funkcją obsługi przerwania.

Wyjście z funkcji obsługi przerwania trwa kolejne 5 cykli — w tym czasie procesor pobiera ze stosu adres powrotu i ładuje go do licznika rozkazów oraz odtwarza wartość wskaźnika stosu.

Funkcja obsługi przerwania

Przed odblokowaniem danego przerwania musimy określić funkcję, która będzie odpowiadać za jego obsługę. Funkcje obsługi przerwań, tzw. handlery (ang. *Interrupt service routines*, *interrupt handlers*), muszą być „opakowane" w specjalne atrybuty informujące kompilator, że jest to specjalna funkcja i należy ją skompilować w inny sposób niż zwykłe funkcje języka C. Ponieważ standard języka C nie definiuje sposobu, w jaki mają być przez kompilator traktowane takie funkcje, ani nawet nie określa, w jaki sposób je wyróżnić, każdy kompilator cechuje się nieco innym podejściem. Aby nie wchodzić w zbędne szczegóły, wystarczy powiedzieć, że `avr-gcc` wymaga, aby funkcje obsługi przerwań były wyróżnione atrybutem `interrupt`, wymagającym kilku parametrów, m.in. numeru wektora przerwań, z którym dana funkcja zostanie powiązana. Jednak *AVR-libc* wprowadza wygodne makro o nazwie `ISR`. Dzięki temu w najprostszym przypadku funkcję obsługi przerwania definiuje się poprzez podanie nazwy wektora przerwania, którego obsługą ma zająć się pisana funkcja:

```
ISR(TCC0_OVF_vect)
{
   if(ACA_STATUS & AC_AC0STATE_bm) PORTA.OUTSET=0b01000000;
    else PORTA.OUTCLR=0b01000000;
}
```

Powyższy przykładowy kod będzie wywoływany, zawsze kiedy wystąpi przerwanie nadmiaru timera `TCC0`[1].

> Pamiętaj, że funkcje obsługi przerwań nie wymagają wcześniejszego utworzenia prototypu funkcji. Nigdy więc ich prototypów nie umieszcza się w plikach nagłówkowych.

Należy też zawsze sprawdzić poprawność pisowni makra `ISR` — zamiana literek spowoduje, że kompilator skompiluje kod, tworząc funkcję domyślną, jednak nie będzie to funkcja obsługi przerwania i nie zostanie ona powiązana z odpowiednim wektorem.

[1] O ile odblokujemy taką możliwość.

W efekcie program nie będzie działał zgodnie z założeniami. Jeśli na przykład zmienimy pokazaną wcześniej funkcję na:

```
IRS(TCC0_OVF_vect)
{
   if(ACA_STATUS & AC_AC0STATE_bm) PORTA.OUTSET=0b01000000;
    else PORTA.OUTCLR=0b01000000;
}
```

to otrzymamy tylko ostrzeżenia:

```
Warning   1   return type defaults to 'int' [enabled by default]
Warning   2   type of '__vector_14' defaults to 'int' [enabled by default]
Warning   3   control reaches end of non-void function [-Wreturn-type]
```

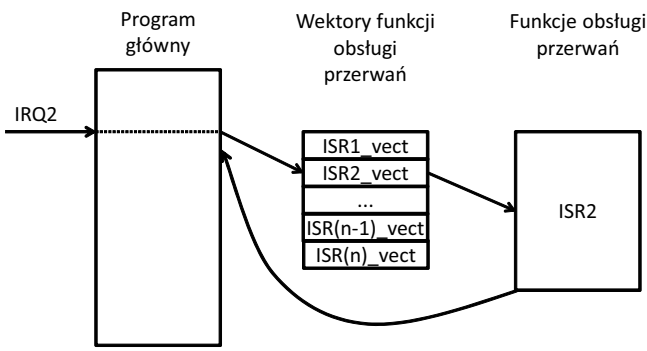
Dzieje się tak, ponieważ kompilator tworzy domyślną funkcję, a funkcje takie w języku C zwracają jeden wynik o typie `int`. Takie ostrzeżenia łatwo przegapić, tym bardziej że jeśli program składa się z wielu plików źródłowych, zostaną one wygenerowane wyłącznie w sytuacji, gdy kompilowany jest plik zawierający powyższą definicję.

> Jest to jeszcze jeden powód, aby pisać programy tak, aby po kompilacji było zawsze 0 błędów i 0 ostrzeżeń. Jeśli nie będziemy się trzymać tej zasady, to w gąszczu innych ostrzeżeń pojawienie się takiego jak powyższe łatwo przeoczyć. Wykrycie miejsca błędnego działania programu będzie ekstremalnie trudne.

Jak widzimy, każda funkcja obsługi przerwania powiązana jest z tzw. wektorem przerwania. Każdy układ mogący generować przerwanie posiada przeznaczony wyłącznie dla niego wektor przerwania. Jeśli układ peryferyjny może generować kilka typów przerwań, to zazwyczaj posiada oddzielne wektory dla każdego typu.

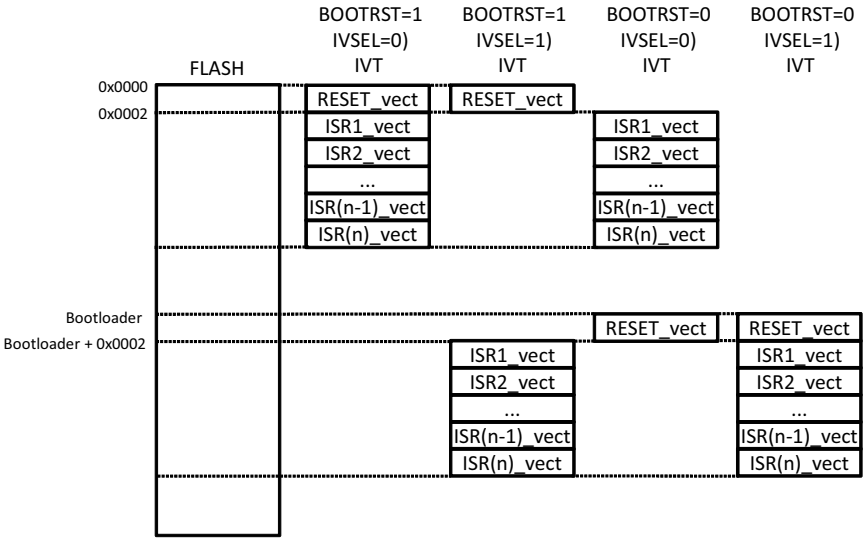
Wektory przerwań

Układy peryferyjne posiadają przypisane im tzw. wektory obsługi przerwań (ang. *ISR vectors*). Każdy podsystem posiada jeden lub więcej takich wektorów. Zgłoszenie przerwania powoduje wywołanie funkcji obsługi przerwania (ISR) powiązanej z jego wektorem (rysunek 10.2).



Rysunek 10.2. *Zgłoszenie przerwania (IRQ2) powoduje przerwanie aktualnego wątku i odczytanie wektora przerwania z nim powiązanego (ISR2_vect). Wektor ten zawiera adres funkcji obsługi przerwania (ISR2). Po jej zakończeniu wznawiane jest wykonywanie przerwanego wątku*

Wszystkie wektory są zgrupowane razem, tworząc tzw. tablicę wektorów przerwań (ang. *Interrupt vector table*, *IVT*). Tablica ta rozpoczyna się na początku pamięci FLASH (od adresu 0x0002), ale jej początek może być zmieniony na początek sekcji tzw. *bootloadera* — wydzielonego obszaru pamięci, w którym rezyduje program umożliwiający zmianę pozostałej części pamięci FLASH — w celu np. uaktualnienia znajdującej się w niej aplikacji. Tabelę wektorów przerwań możemy przenieść do obszaru bootloadera, ustawiając bit IVSEL rejestru kontrolnego PMIC (PMIC_CTRL) — rysunek 10.3.



Rysunek 10.3. *Położenie wektora RESET i tablicy wektorów przerwań (IVT) w zależności od ustawienia fusebitu BOOTRST i bitu konfiguracyjnego IVSEL*

Tu jednak mogą kryć się pewne problemy. Otóż bit konfiguracyjny IVSEL jest chroniony przed zmianą za pomocą znanego już nam rejestru ochrony konfiguracji CCP. Aby go zmienić, musimy wcześniej do rejestru CCP wpisać odpowiedni kod odblokowujący — wartość 0xD8 lub symbolicznie CPP_IOREG_gc. Jednak zmieniając bit IVSEL, nie chcemy zmieniać stanu innych bitów tego rejestru. W tym celu naturalna wydaje się być sekwencja:

```
CCP=CCP_IOREG_gc;              //Odblokuj możliwość zmiany IVSEL
PMIC_CTRL|=PMIC_IVSEL_bm;      //Przenieś IVT w obszar normalnej pamięci FLASH
```

Niestety powyższy kod, o czym łatwo możemy się przekonać w symulatorze, nie spowoduje zmiany stanu bitu IVSEL. Przyczyna tego staje się jaśniejsza po przeanalizowaniu wygenerowanego dla powyższych instrukcji kodu asemblerowego:

```
  CCP=CCP_IOREG_gc;              //Odblokuj możliwość zmiany IVSEL
226:   94 bf          out   0x34, r25   ; 52
  PMIC_CTRL|=PMIC_IVSEL_bm;      //Przenieś IVT w obszar normalnej pamięci FLASH
228:   80 81          ld    r24, Z
22a:   80 64          ori   r24, 0x40   ; 64
22c:   80 83          st    Z, r24
```

Rejestr CCP po odblokowaniu zezwala na zmianę konfiguracji rejestru IO w czasie maksymalnie 4 kolejnych cykli zegara. Niestety kod zmieniający stan IVSEL trwa znacznie

dłużej, w efekcie zmiana jest nieskuteczna. Musimy więc nieco zmodyfikować sposób zmiany stanu IVSEL:

```
uint8_t tmp=PMIC_CTRL | PMIC_IVSEL_bm;      //Przenieś IVT w obszar bootloadera
CCP=CCP_IOREG_gc;                           //Odblokuj możliwość zmiany IVSEL
PMIC_CTRL=tmp;
```

Tym razem nowa wartość wpisywana do rejestru PMIC_CTRL została przygotowana przed wpisaniem zezwolenia na zmianę konfiguracji do rejestru CCP. W efekcie wygenerowany kod działa prawidłowo i mieści się w limicie czterech cykli:

```
  uint8_t tmp=PMIC_CTRL | PMIC_IVSEL_bm;      //Przenieś IVT w obszar bootloadera
218:   e2 ea          ldi   r30, 0xA2   ; 162
21a:   f0 e0          ldi   r31, 0x00   ; 0
21c:   80 81          ld    r24, Z
21e:   80 64          ori   r24, 0x40   ; 64
  CCP=CCP_IOREG_gc;                           //Odblokuj możliwość zmiany IVSEL
220:   98 ed          ldi   r25, 0xD8   ; 216
222:   94 bf          out   0x34, r25   ; 52
  PMIC_CTRL=tmp;
224:   80 83          st    Z, r24
```

W podobny sposób możemy także skasować bit IVSEL:

```
tmp=PMIC_CTRL&~PMIC_IVSEL_bm; //Przenieś IVT w obszar normalnej pamięci FLASH
CCP=CCP_IOREG_gc;             //Odblokuj możliwość zmiany IVSEL
PMIC_CTRL=tmp;
```

Powiązanie pomiędzy wektorem przerwania a funkcją obsługi przerwania dokonywane jest na etapie konsolidacji programu. Aby to było możliwe, jako argument makra ISR musimy podać poprawną nazwę wektora przerwania, dzięki czemu linker wygeneruje kod odpowiedzialny za wywołanie naszej funkcji.

Puste wektory przerwań

> Każde odblokowane przerwanie musi posiadać funkcję je obsługującą.

Stąd też nigdy nie powinniśmy odblokowywać przerwania, jeśli z jego wektorem nie jest powiązana żadna funkcja obsługi. Domyślnie *AVR-libc* dba, aby z każdym niewykorzystanym wektorem powiązana była prosta funkcja obsługi. Funkcja ta o etykiecie __bad_interrupt ma prostą budowę:

```
0000023e <__bad_interrupt>:
 23e:   0c 94 00 00     jmp   0    ; 0x0 <__vectors>
```

Jak widzimy, jej zadaniem jest skok pod adres wektora resetu procesora. W efekcie jeśli odblokujemy przerwanie, dla którego nie ma funkcji go obsługującej, kontrolę przejmie domyślna funkcja. Stąd też jeśli takie przerwanie zostanie obsłużone, to spowoduje ono reset procesora. Jest to jakieś rozwiązanie, lecz nie jest ono zbyt optymalne — takie zachowanie tak naprawdę nie prowadzi do resetu, lecz do rozpoczęcia wykonywania programu od nowa. Nie jest to w stanie rozwiązać problemu, a nawet potencjalnie

może problem nasilić — w wyniku takiej inicjalizacji nie jest przywracany stan początkowy urządzeń peryferyjnych, w tym urządzeń mogących generować przerwania. W złożonych programach może prowadzić to do kompletnej katastrofy. Oczywiście w poprawnie napisanym programie nigdy nie powinno dojść do sytuacji, w której odblokowane jest przerwanie bez funkcji jego obsługi. Lecz wystarczy pomylić się w nazwie wektora lub zastosować niewłaściwy wektor, aby popaść w problemy. Jak je rozwiązać? Tu znowu z pomocą przychodzi nam *AVR-libc*. Umożliwia ona implementację własnej domyślnej funkcji obsługi przerwań, w tym celu należy ją tylko zdefiniować:

```
ISR(BADISR_vect)
{
  //Tu należy umieścić własną domyślną obsługę
}
```

Tak zdefiniowana funkcja będzie wywoływana w sytuacji, w której brakuje właściwej funkcji obsługi przerwania. Możemy w niej zdefiniować sposób reakcji na taką błędną sytuację, np. zasygnalizowanie w jakiś sposób problemu lub wysłanie informacji do programisty o zaistniałej sytuacji. Funkcja taka ma szczególne znaczenie na etapie debugowania kodu, kiedy tego typu błędy mogą się zdarzyć. Jej właściwa implementacja może zaoszczędzić nam sporo czasu przy debugowaniu kodu stwarzającego problem na przykład na skutek błędnie zapisanego wektora przerwania.

Puste przerwania

W pewnych sytuacjach odblokowujemy przerwanie, lecz wcale nie chcemy go obsługiwać. Dlaczego? Wywołanie przerwania ma zwykle jeden istotny skutek — wejście do funkcji jego obsługi kasuje flagę związaną z danym przerwaniem. I czasami skasowanie tej właśnie flagi jest istotne. Oczywistym rozwiązaniem jest zdefiniowanie funkcji obsługi niezawierającej żadnych instrukcji:

```
ISR(TCC0_CCA_vect) {};
```

Nie jest to jednak dobre rozwiązanie — kompilator dla tak zdefiniowanej funkcji obsługi wygeneruje minimalny prolog i epilog:

```
ISR(TCC0_CCA_vect) {};
 25c:   1f 92          push   r1
 25e:   0f 92          push   r0
 260:   0f b6          in   r0, 0x3f   ; 63
 262:   0f 92          push   r0
 264:   11 24          eor   r1, r1
 266:   0f 90          pop   r0
 268:   0f be          out   0x3f, r0   ; 63
 26a:   0f 90          pop   r0
 26c:   1f 90          pop   r1
 26e:   18 95          reti
```

Twórcy *AVR-libc* przewidzieli tę sytuację i zdefiniowali specjalne makro, które jako argument przyjmuje nazwę wektora:

```
EMPTY_INTERRUPT(TCC0_CCA_vect);
```

Tak wygenerowany kod ciągle nie jest optymalny, ale jest zdecydowanie lepszy niż poprzedni:

```
EMPTY_INTERRUPT(TCC0_CCA_vect);
 262:   18 95          reti
```

W tym przypadku obsługa przerwania składa się z dwóch instrukcji — `jmp`/`rjmp` umieszczonej pod adresem wektora oraz `reti` kończącej przerwanie. Dalsza optymalizacja wymagałaby bezpośredniej ingerencji w podstawowe funkcje biblioteki i zazwyczaj nie ma większego sensu. **Warto jednak pamiętać, że wykonanie nawet pustej funkcji obsługi przerwania zajmuje parę taktów procesora.**

Współdzielenie kodu przez przerwania

Czasami zdarza się, że ten sam kod obsługi przerwania możemy wykorzystać dla różnych przerwań. W takiej sytuacji zgodnie z duchem języka C powinniśmy współdzielony kod umieścić w zewnętrznej funkcji, którą będziemy wywoływać z obu funkcji obsługi przerwania, np.:

```
void shared_code() //Kod współdzielony przez dwie funkcje
{
}
ISR(TCC0_CCA_vect)
{
   shared_code();
};
ISR(TCC0_CCB_vect)
{
   shared_code();
};
```

Widzimy, że obie funkcje obsługi wykorzystują ten sam kod. Rozwiązanie to jest jak najbardziej poprawne, lecz niezbyt optymalne. Dlaczego? Po pierwsze, nasza funkcja `shared_code` może zostać osadzona w miejscu wywołania, co wydłuży wygenerowany kod. Jeśli z kolei nie zostanie osadzona, to zostanie wygenerowane jej wywołanie ze wszystkimi tego konsekwencjami (czas potrzebny na skoki, konieczność zachowania stanu niektórych rejestrów). W obu sytuacjach zostaną wygenerowane osobne prologi i epilogi funkcji, które jak widzieliśmy na poprzednich przykładach, bywają długie. Czyli obie sytuacje nie są optymalne. Tu znowu mamy rozwiązanie eleganckie. Zakładając, że mamy zdefiniowaną funkcję obsługi przerwania o wektorze `TCC0_CCA_vect`, a funkcja o wektorze `TCC0_CCB_vect` jest identyczna, możemy ją zdefiniować, wykorzystując możliwość tworzenia aliasów:

```
ISR(TCC0_CCB_vect, ISR_ALIASOF(TCC0_CCA_vect));
```

Alias powoduje, że dwa wektory przerwań wskazują na tę samą funkcję, nie są więc generowane oddzielne prologi i epilogi, a powstały kod jest optymalny. W naszym przypadku oba użyte wektory będą wskazywać na tę samą funkcję obsługi, dzięki umieszczeniu w tabeli wektorów przerwań instrukcji skoku do tego samego handlera:

```
  40:   0c 94 31 01     jmp   0x262   ; 0x262 <__vector_16>
  44:   0c 94 31 01     jmp   0x262   ; 0x262 <__vector_16>
```

Atrybut naked i obsługa przerwań w asemblerze

Procedura obsługi przerwania powinna być możliwie najkrótsza. Zapewnia to bezproblemową obsługę przerwań. Aby mieć pełną kontrolę nad kodem wygenerowanym przez kompilator, czasami trzeba napisać procedurę obsługi przerwania w asemblerze. Kod asemblerowy możemy dodać jako krótką wstawkę w procedurze obsługi przerwania napisanej w języku C lub możemy całą funkcję napisać w asemblerze. Najprościej w tym celu wykorzystać wbudowany w kompilator języka C asembler. Napisanie ciała funkcji w asemblerze nie daje nam jednak kontroli nad tworzonym automatycznie prologiem i epilogiem funkcji, którego celem jest zachowanie stanu wszystkich używanych rejestrów procesora, w tym szczególnie rejestru stanu procesora (ang. *Flags*). Możemy zabronić kompilatorowi tworzenia prologu i epilogu, stosując atrybut `ISR_NAKED`, który jest zdefiniowany następująco:

```
#  define ISR_NAKED      __attribute__((naked))
```

Po napotkaniu funkcji z atrybutem `naked` kompilator przestaje generować dla niej prolog i epilog, o zachowanie stanu rejestrów i ich odtworzenie musi więc zadbać programista. Konsekwencją braku epilogu jest także brak generowania instrukcji powrotu z przerwania — `reti`, musimy sami zadbać o jej dodanie.

Ponieważ pisząc w języku C, nie mamy kontroli nad tym, jakie rejestry wykorzysta kompilator po zdefiniowaniu tego atrybutu, mamy dwie opcje prawidłowego napisania funkcji obsługi przerwania:

1. Możemy w prologu zachować stan wszystkich rejestrów i w epilogu je odtworzyć. Nie daje nam to jednak żadnej przewagi nad pozostawieniem tego zadania kompilatorowi, a więc rozwiązania tego nigdy nie stosujemy.
2. Całe ciało funkcji będzie napisane w asemblerze, na stos odkładamy wyłącznie te rejestry, których stan zmodyfikowaliśmy.

Widzimy więc, że stosując funkcję z atrybutem `naked`, w praktyce musimy całe ciało funkcji napisać w asemblerze[2]. Generalnie warto pisać w asemblerze bardzo krótkie i proste funkcje, które są krytyczne czasowo. Dla dłuższych zysk pod postacią skrócenia kodu i skrócenia czasu wykonania funkcji zwykle jest nieistotny. Dla zilustrowania użycia atrybutu `naked` przeanalizujemy następujący kod:

```
uint8_t licznik;

ISR(TCC0_CCA_vect)
{
   ++licznik;
};
```

Stworzyliśmy prostą funkcję, której zadaniem jest po prostu inkrementacja zmiennej `licznik`. Jednak kompilator generuje dla niej stosunkowo długi kod:

[2] Jak od każdej reguły, także od tej istnieją wyjątki, kiedy wykorzystujemy proste funkcje niemodyfikujące rejestrów mikrokontrolera.

```
 262:   1f 92          push   r1
 264:   0f 92          push   r0
 266:   0f b6          in   r0, 0x3f   ; 63
 268:   0f 92          push   r0
 26a:   11 24          eor   r1, r1
 26c:   8f 93          push   r24
   ++licznik;
 26e:   80 91 05 20    lds   r24, 0x2005
 272:   8f 5f          subi   r24, 0xFF   ; 255
 274:   80 93 05 20    sts   0x2005, r24
};
 278:   8f 91          pop   r24
 27a:   0f 90          pop   r0
 27c:   0f be          out   0x3f, r0   ; 63
 27e:   0f 90          pop   r0
 280:   1f 90          pop   r1
 282:   18 95          reti
```

Dlaczego? Wynika to z założeń ABI kompilatora. My jednak możemy lepiej napisać tę funkcję. O ile sama inkrementacja zmiennej jest zrealizowana w sposób optymalny, to problemem jest prolog i epilog funkcji. Stosując atrybut `ISR_NAKED`, możemy całą funkcję napisać w asemblerze:

```
ISR(TCC0_CCA_vect, ISR_NAKED)
{
  asm volatile(
   "push r1 \n\t"
   "in   r1, 0x3f \n\t"
   "push   r24 \n\t"
   "lds   r24, licznik \n\t"
   "inc r24 \n\t"
   "sts   licznik, r24 \n\t"
   "pop   r24 \n\t"
   "out   0x3f, r1 \n\t"
   "pop   r1 \n\t"
   "reti \n\t"
   );
};
```

W tym przypadku napisaliśmy całą funkcję sami, co nieznacznie ją skróciło. Ale obsługę przerwania możemy napisać, nie tylko korzystając z wbudowanego asemblera; możemy ją umieścić w osobnym pliku z kodem asemblerowym (z rozszerzeniem *.S*). W tym celu wystarczy dodać do projektu nowy plik (wybieramy plik asemblera), po czym wpisujemy jego zawartość:

```
#include <avr/interrupt.h>

 .global TCC0_CCA_vect
TCC0_CCA_vect:
 push r1
 in   r1, 0x3f
 push   r24
 lds   r24, licznik
 inc r24
 sts   licznik, r24
 pop   r24
```

```
out   0x3f, r1
pop   r1
      reti
```

Jak widzimy, do naszego pliku należy dołączyć nagłówek *interrupt.h* zawierający definicję wektorów przerwań (opcjonalnie warto też dołączyć nagłówek *io.h*, co da nam możliwość odwoływania się do nazw symbolicznych rejestrów). Wektor przerwania, z którym chcemy powiązać naszą funkcję, nazywa się `TCC0_CCA_vect`, lecz nie wystarczy zdefiniować takiej etykiety. Aby była ona widoczna dla linkera, musimy uczynić ją etykietą publiczną, co zapewnia słowo `.global`. Nasz plik zostanie skompilowany z użyciem asemblera i automatycznie dołączony. Warto jednak zauważyć, że niezależnie od tego, czy kod generuje kompilator, czy sami go tworzymy, różnica często jest niewielka lub wręcz pomijalna. A im bardziej skomplikowana jest funkcja obsługi przerwania, tym zazwyczaj mniejszy zysk płynie z jej napisania w asemblerze.

> Nie używaj wstawek w asemblerze, jeśli nie masz naprawdę dobrych powodów, by to robić. Utworzony kod jest mało czytelny i trudno przenieść go na inny mikrokontroler. Rzadko też zysk czasowy jest istotny w kontekście całej aplikacji.

Poziomy przerwań

XMEGA dysponuje trzema poziomami przerwań umożliwiającymi podzielenie generowanych przerwań ze względu na istotność. Dla każdego źródła przerwań może być określony jeden z trzech poziomów poprzez złożenie prefiksu określającego źródło przerwań z sufiksem podanym w tabeli 10.1. I tak dla przykładu: przerwania związane z błędem DMA mają prefiks `DMA_CH_ERR`, do którego dodajemy sufiks związany z wybranym poziomem przerwań, np. `INTLVL_LO_gc`, co daje nam `DMA_CH_ERRINTLVL_LO_gc`.

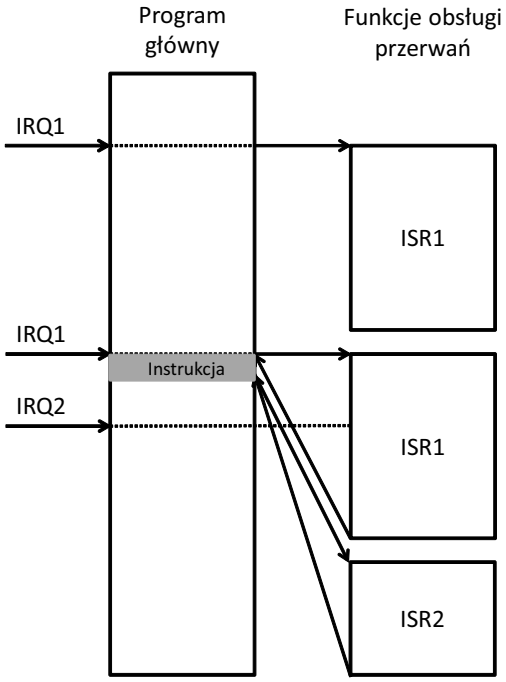
Tabela 10.1. *Poziomy przerwań mikrokontrolera XMEGA*

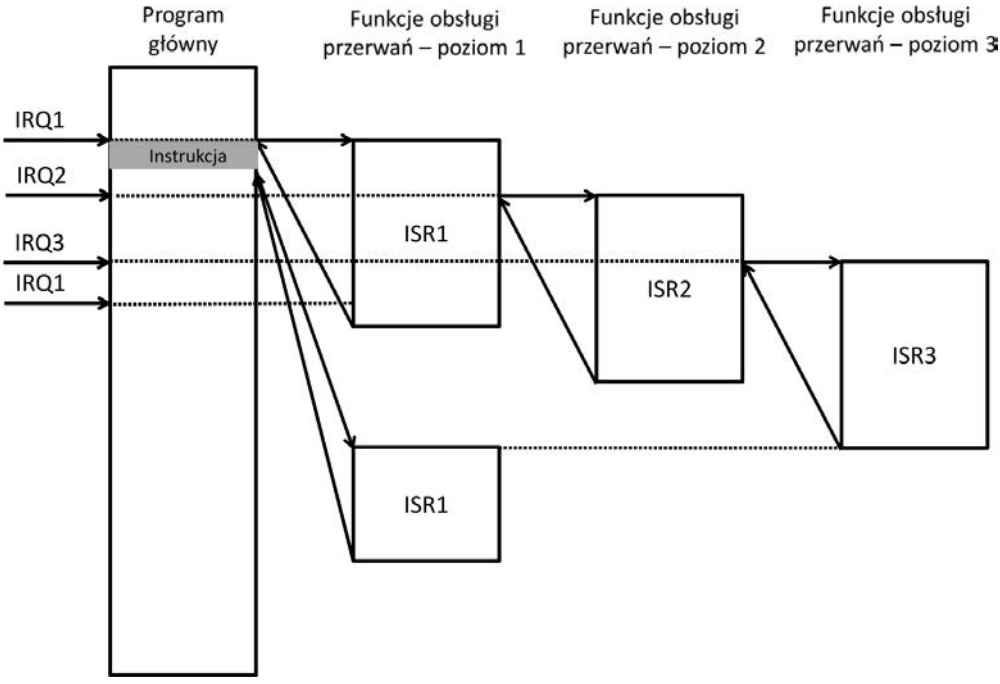
Poziom przerwania	Symbol	Opis
0	`INTLVL_OFF_gc`	Dane przerwanie jest wyłączone
1	`INTLVL_LO_gc`	Przerwanie o niskim priorytecie
2	`INTLVL_MED_gc`	Przerwanie o średnim priorytecie
3	`INTLVL_HI_gc`	Przerwanie o wysokim priorytecie

Przerwania tego samego poziomu nie mogą się wzajemnie przerywać (rysunek 10.4). Kolejne przerwanie tego samego poziomu zostanie obsłużone dopiero w chwili zakończenia aktualnie wykonywanego przerwania.

Przerwania o wyższym poziomie mogą przerywać przerwania o poziomie niższym (rysunek 10.5). Odwrotna sytuacja jest niemożliwa. To znaczy, że np. przerwanie o poziomie średnim może zostać przerwane przez przerwanie o poziomie wysokim, ale zgłoszone przerwanie o poziomie niskim, zanim zostanie obsłużone, musi poczekać na zakończenie obsługi aktualnie wykonywanego przerwania o poziomie średnim lub wysokim. Przerwania o wysokim poziomie nic nie jest w stanie przerwać — z wyjątkiem tzw. przerwania niemaskowalnego (*NMI*), o którym dowiesz się nieco później.

Rysunek 10.4. *Przerwania IRQ1 i IRQ2 są przerwaniami tego samego poziomu. Żądanie przerwania IRQ2 zostanie obsłużone dopiero w chwili zakończenia funkcji obsługi wcześniejszego przerwania (funkcja ISR1). Po zakończeniu obu funkcji obsługi (ISR1 i ISR2) wznowione zostanie wykonywanie głównego wątku*





Rysunek 10.5. *Poziomy przerwań. Przerwanie o wyższym poziomie (ISR2, ISR3) może przerwać wykonywanie przerwania o poziomie niższym. Odwrotna sytuacja jest niemożliwa i kolejne przerwanie ISR1 musi poczekać, aż zakończy się obsługa wcześniejszych przerwań*

Każdy podsystem XMEGA mogący generować przerwania posiada rejestr kontrolujący poziom przerwania, o nazwie `INTCTRL`. Jeśli układ peryferyjny może zgłaszać więcej przerwań, to rejestrów je kontrolujących jest więcej — zazwyczaj dwa, o nazwach `INTCTRLA` i `INTCTRLB`, co umożliwia teoretycznie przechowywanie informacji o poziomie 8 różnych przerwań. Rejestry te zawierają dwubitowe pola — wartość `0` oznacza zablokowanie możliwości zgłoszenia przerwania, a wartości `1 – 3` określają poziom przerwania (najwyższy poziom to `3`). Stąd też aby odblokować przerwania przepełnienia timera `C0` i nadać im najwyższy poziom, należy wpisać do rejestru:

```
TCC0.INTCTRLA=TC_OVFINTLVL_HI_gc;
```

Od tej chwili timer będzie zgłaszał przerwania nadmiaru, co nie znaczy, że będą one obsłużone — zależy to jeszcze od konfiguracji podsystemu PMIC. Z każdym przerwaniem związana jest flaga sygnalizująca jego żądanie. Flaga taka może być kasowana programowo lub sprzętowo w momencie obsłużenia (wejścia do funkcji obsługi) przerwania. Flagi przerwań związanych z danym układem peryferyjnym zgrupowane są zazwyczaj w jednym rejestrze układu peryferyjnego o nazwie `INTFLAGS`.

> Kiedy dane przerwanie nie jest obsługiwane (jest wyłączone), odpowiadająca mu flaga ciągle jest ustawiana w chwili, w której dane przerwanie mogłoby zostać wygenerowane, lecz nie jest kasowana. Stąd też flagi nieużywanych przerwań często mają wartość `1`.

Flagę powiązaną z nieobsługiwanym przerwaniem możemy skasować poprzez zapis wartości 1 do reprezentującego ją bitu. Po takiej operacji flaga pozostaje skasowana do momentu ponownego zajścia zdarzenia potencjalnie mogącego wygenerować przerwanie. Większość flag może być także kasowana w chwili realizacji dostępu do powiązanego z flagą układu przez kontroler DMA. **Lecz nie wszystkie flagi są przez kontroler DMA zmieniane, takim istotnym wyjątkiem jest flaga nadmiaru (`OVF`) timera**. Flagi mają jeszcze jedno zastosowanie. W chwili zajścia zdarzenia generującego przerwanie dana flaga jest ustawiana i pozostaje ustawiona do momentu jego obsługi lub programowego skasowania. Dzięki temu nawet jeśli dane przerwanie zostanie zablokowane i w efekcie nie będzie mogło zostać obsłużone, to ustawienie odpowiadającej mu flagi będzie sygnalizowało kontrolerowi przerwań żądanie jego obsługi. W efekcie gdy tylko dane przerwanie zostanie odblokowane, spowoduje to jego natychmiastowe zgłoszenie i wywołanie odpowiedniej funkcji obsługi. Stąd też funkcja obsługi przerwania może być wywołana w stosunku do pojawienia się żądania z dużym opóźnieniem. W efekcie jej wywołanie może stracić sens. W takiej sytuacji przed odblokowaniem danego przerwania warto wcześniej skasować odpowiadającą mu flagę.

Z faktu, że informacja o żądaniu przerwania zapisywana jest w postaci flagi, wynika jeszcze jedna ważna rzecz. Jeśli od momentu zgłoszenia pierwszego żądania do chwili jego obsługi pojawi się więcej żądań, to zostaną one zgubione, w efekcie mimo wielokrotnych żądań przerwanie zostanie obsłużone jednokrotnie — mikrokontroler nie ma możliwości stwierdzenia, ile razy zgłaszane było żądanie.

Jak widzimy, każdy układ peryferyjny ma własne rejestry kontrolne odpowiedzialne za możliwość odblokowania przerwań i określenia ich poziomu. Jednak aby przerwania

danego poziomu mogły być obsługiwane, należy taką możliwość odblokować w kontrolerze PMIC. Można tego dokonać, ustawiając jeden z bitów kontrolnych (HILVLEN, MEDLVLEN lub LOLVLEN) rejestru kontrolnego (PMIC.CTRL):

```
PMIC_CTRL|=PMIC_LOLVLEN_bm;    //Odblokuj przerwania o niskim poziomie
PMIC_CTRL|=PMIC_MEDLVLEN_bm;   //Odblokuj przerwania o średnim poziomie
PMIC_CTRL|=PMIC_HILVLEN_bm;    //Odblokuj przerwania o wysokim poziomie
```

Jeśli przerwanie danego poziomu w kontrolerze PMIC nie będzie odblokowane, to pomimo że zostanie ono zgłoszone przez układ peryferyjny, nie zostanie obsłużone.

Kontroler PMIC kontroluje wyłącznie poziom przerwania, nie jest dla niego istotne jego źródło. Stąd też jeśli odblokujemy kilka przerwań tego samego poziomu w różnych układach peryferyjnych, to wszystkie one będą pod kontrolą tego samego bitu kontrolera PMIC. Dzięki temu łatwo możemy odblokować lub zablokować zgłaszanie przerwań danego poziomu.

Przerywanie przerwań

Jak wcześniej wspomniano, przerwanie o wyższym poziomie może przerwać wykonywanie funkcji obsługi przerwania o poziomie niższym. W przeciwieństwie jednak do innych mikrokontrolerów AVR przerwania o tym samym poziomie nie mogą się nawzajem przerywać. Jest to związane z tym, że po wejściu w funkcję obsługi przerwania danego poziomu mikrokontroler ustawia odpowiednie bity rejestru stanu kontrolera PMIC (PMIC.STATUS — bity HILVLEX, MEDLVLEX lub LOLVLEX), które pozostają ustawione, dopóki nie zostanie wykonana instrukcja RETI związana z danym poziomem przerwań. Pośrednio takie zachowanie ma jeszcze jedną konsekwencję — makrodefinicja ISR przyjmuje jeden z opcjonalnych parametrów:

- ISR_NOBLOCK — powodujące po wejściu do funkcji obsługi przerwania natychmiastowe ich odblokowanie, co umożliwia przerwanie funkcji obsługi przerwania;
- ISR_BLOCK — jest domyślnym zachowaniem — funkcja obsługi przerwania jest wykonywana z zablokowanymi przerwaniami i dzięki temu sama jest nieprzerywalna.

Ze względu na działanie kontrolera PMIC w XMEGA makro ISR_NOBLOCK nie zadziała tak, jak byśmy tego oczekiwali, nie ma więc sensu używać tego rozszerzenia.

Na przykład funkcja:

```
ISR(TCC0_CCA_vect, ISR_NOBLOCK)
{
   y=TCC0_CCA;

};
```

zostanie skompilowana do postaci:

```
ISR(TCC0_CCA_vect, ISR_NOBLOCK)
{
 262:   78 94          sei
 264:   1f 92          push   r1
 266:   0f 92          push   r0
 268:   0f b6          in   r0, 0x3f   ; 63
...
...
```

Początkowa instrukcja `sei` w XMEGA jest w tym przypadku kompletnie bez znaczenia — zezwolenie na kolejne przerwania tego samego poziomu zależy od kontrolera PMIC, a nie od flagi `I` rejestru stanu mikrokontrolera. W efekcie jej wykonanie jest tylko stratą czasu.

System przerwań w XMEGA nie zmienia stanu bitu `I` rejestru stanu procesora, w związku z tym dodanie instrukcji `SEI` na początku funkcji obsługi przerwania nic nie zmienia. Podobną funkcjonalność (tj. możliwość przerywania funkcji obsługi przerwania przez przerwanie o tym samym poziomie) można uzyskać, stosując pewne kruczki. A mianowicie musimy zasymulować wykonanie instrukcji `RETI`, jednak w taki sposób, aby nie zakończyć bieżącej funkcji obsługi przerwania. Rozwiązaniem jest krótka wstawka asemblerowa:

```
asm volatile("RJMP L_2%=\n\t"
             "L_1%=: "
          "RETI\n\t"
          "L_2%=: "
             "RCALL L_1%=\n\t" ::);
```

Jej działanie jest następujące. Pierwsza instrukcja skoku (`RJMP`) omija kolejną instrukcję (`RETI`), następnie jest wykonywana instrukcja `RCALL` odkładająca na stosie adres powrotu — czyli kolejnej instrukcji programu — i następuje skok do instrukcji `RETI` — efektem jej działania jest skok do instrukcji po instrukcji `RCALL`, która ją wywołała, a efektem ubocznym oszukanie kontrolera PMIC. Sądzi on, że bieżąca funkcja obsługi przerwania się zakończyła (wykonana została instrukcja powrotu z przerwania `RETI`), więc kasuje odpowiednią flagę rejestru statusu kontrolera. Od tej chwili kolejne przerwania tego samego poziomu co bieżąco obsługiwane będą przyjmowane. Powyższy kod możemy umieścić w dowolnym miejscu funkcji obsługi przerwania, której chcemy dać możliwość przerywania, lecz ze względów praktycznych powinien to być sam początek jej obsługi, np.:

```
ISR(TCC1_OVF_vect)
{
   asm volatile("RJMP L_2%=\n\t"
                "L_1%=: "
             "RETI\n\t"
             "L_2%=: "
                "RCALL L_1%=\n\t" ::);
//Dalszy ciąg instrukcji obsługi przerwania
}
```

Jednak jak każde oszustwo także to przedstawione powyżej ma krótkie nogi. Problem pojawi się w momencie, kiedy powyższą sztuczkę zastosujemy w przypadku przerwań wyższego poziomu (poziomu średniego lub wysokiego). Powyższa wstawka spowoduje odblokowanie możliwości przyjmowania kolejnych przerwań tego samego poziomu, lecz jeśli przerwanie o poziomie średnim lub wysokim przerwało przerwanie o poziomie niskim, a odblokowaliśmy, korzystając z powyższego kruczka przerwania, to po powrocie z funkcji obsługi przerwania o wyższym priorytecie zostaną także odblokowane przerwania o priorytecie niższym — powodem będzie dwukrotne wykonanie rozkazu `RETI` — raz w naszej wstawce asemblerowej, a drugi raz podczas prawdziwego zakończenia funkcji obsługi przerwania o wyższym priorytecie. Oczywiście ten problem też możemy rozwiązać, pisząc własny epilog funkcji obsługi przerwania kończący się instrukcją `RET`, a nie `RETI`, lecz czyni to całe rozwiązanie mało eleganckim.

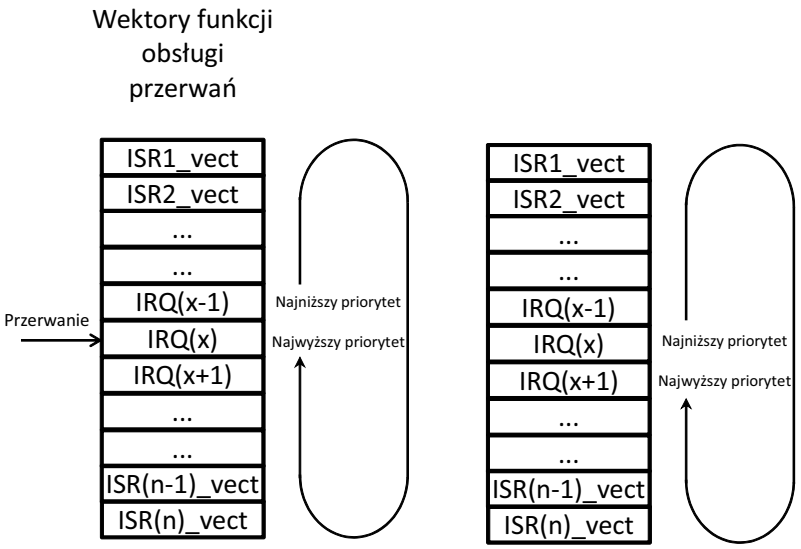
Z drugiej strony wielopoziomowy system przerwań dostępny w XMEGA powoduje, że stosowanie rozwiązań żywcem przeniesionych z innych rodzin AVR, gdzie przerwania były jednopoziomowe (co zazwyczaj poprawiano, stosując makro `NO_BLOCK`), jest w większości przypadków niepotrzebne. Zanim więc zaczniesz stosować pokazane powyżej sztuczki, dobrze się zastanów, czy jest to naprawdę niezbędne. Zazwyczaj w takiej sytuacji wystarczy rozbicie przerwań na kilka poziomów.

Priorytety przerwań

XMEGA dysponuje nie tylko poziomami, ale także tzw. priorytetami przerwań. Dla przerwań NMI oraz przerwań o poziomie wysokim i średnim zawsze używane są tzw. priorytety statyczne. To znaczy, że w sytuacji, w której jednocześnie zgłoszonych jest kilka przerwań (lub kilka przerwań oczekuje na swoją obsługę), najpierw obsługiwane jest przerwanie o niższym wektorze przerwania w tablicy wektorów przerwań. Dla przerwań o priorytecie niskim procesor może reagować w sposób bardziej elastyczny. Dla tego poziomu przerwań możemy wybrać priorytet statyczny, analogiczny do priorytetu statycznego przerwań wyższych poziomów, lub priorytet dynamiczny — o priorytecie przerwania w takiej sytuacji decyduje algorytm *round-robin*.

Priorytet dynamiczny

Priorytet statyczny jest prosty, ma jednak istotną wadę — często zgłaszane przerwanie o wysokim priorytecie może uniemożliwić obsługę przerwań o priorytetach niższych. Aby uniknąć takiej sytuacji, wprowadzono priorytet dynamiczny. W tym schemacie ostatnio obsłużone przerwanie otrzymuje najniższy priorytet, a przerwanie o kolejnym wektorze priorytet najwyższy. Po jego obsłużeniu najwyższy priorytet zyskuje przerwanie o kolejnym wektorze itd. W efekcie żadne pojedyncze przerwanie nie jest w stanie dominować — każde w pewnym momencie będzie miało najwyższy priorytet (rysunek 10.6).



Rysunek 10.6. *Zasadza działania priorytetów przerwań w trybie dynamicznym. Po odebraniu zgłoszenia przerwania przerwanie dostaje priorytet najniższy. Przerwanie o kolejnym wektorze zyskuje priorytet najwyższy, kolejne nieco niższy itd.*

Aby móc zrealizować priorytet dynamiczny, XMEGA dysponuje specjalnym rejestrem `INTPRI` (ang. *PMIC Priority Register*). Jego domyślna wartość to zero, natomiast w trybie z dynamicznymi priorytetami przechowuje on adres wektora ostatnio obsłużonego przerwania. W efekcie przerwania o wektorach wyższych niż wektor zawarty w `INTPRI` mają priorytet wyższy, a przerwania o wektorach mniejszych mają priorytet niższy. Rejestr ten możemy także zapisywać, co umożliwia programowe określenie priorytetu kolejnych przerwań. Jednak jest to możliwość raczej rzadko wykorzystywana. Potencjalnie ciekawsza jest inna możliwość. Mianowicie w przypadku wyłączenia priorytetów dynamicznych i przywrócenia priorytetów statycznych dla przerwań niskiego poziomu rejestr ten w pewnym stopniu określa priorytet przerwania. Analogicznie jak w trybie dynamicznym przerwania o wektorach wyższych niż wektor zawarty w tym rejestrze mają priorytet wyższy, a przerwania o wektorach niższych mają priorytet niższy. Różnica polega więc wyłącznie na tym, że w trybie statycznym wartość rejestru `INTPRI` nie jest automatycznie uaktualniana wartością wektora ostatnio obsłużonego przerwania. Stwarza to więc możliwość przynajmniej częściowej zmiany priorytetów przerwań.

Priorytet dynamiczny przerwań wybiera się poprzez ustawienie bitu `RREN` (ang. *Round-robin Scheduling Enable*) rejestru kontrolnego:

```
PMIC_CTRL|=PMIC_RREN_bm;        //Odblokuj dynamiczne priorytety przerwań
```

Globalna flaga zezwolenia na przerwanie

Podobnie jak inne AVR-y, także XMEGA dysponują globalną flagą zezwolenia na przerwania, znajdującą się w rejestrze stanu procesora (flaga `I`). Aby podsystem PMIC

mógł obsługiwać żądania przerwań, flaga ta musi zostać ustawiona instrukcją `sei()`, natomiast jej skasowanie instrukcją `cli()` powoduje zablokowanie wszystkich przerwań.

> Jednak w przeciwieństwie do innych mikrokontrolerów AVR wejście do funkcji obsługi przerwania nie powoduje ustawienia flagi `I` i blokowania przerwań. O tym, czy kolejne przerwania będą przyjęte, czy nie, decyduje podsystem PMIC.

W programie praktycznie nigdy nie będziemy musieli zmieniać stanu tej flagi. Ponieważ procesor po resecie ma tę flagę wyzerowaną, praktycznie jedynym miejscem, gdzie jawnie będziemy ją zmieniać, jest początek programu wykorzystującego przerwania — po inicjalizacji poszczególnych podsystemów przerwania będziemy odblokowywać instrukcją `sei()`. Instrukcji `cli()` właściwie nigdy nie powinniśmy stosować jawnie. Jest ona niejawnie wykonywana w trakcie realizacji tzw. sekcji atomowych, o których dowiesz się więcej z dalszej części rozdziału. W efekcie każdy program wykorzystujący przerwania inicjuje je w typowy sposób:

1. Konfiguracja przerwań w układach peryferyjnych.
2. Odblokowanie wymaganych poziomów przerwań w kontrolerze PMIC.
3. Wykonanie makra `sei()` odblokowującego przerwania globalnie.

Przerwanie niemaskowalne

Przerwania niemaskowalne (NMI, ang. *Non-maskable interrupts*) cechują się najwyższym priorytetem, nie można ich także zablokować (zamaskować) — stąd nazwa przerwania. Przerwania niemaskowalne służą do sygnalizacji pewnych krytycznych zdarzeń, jak np. zanik zasilania czy uszkodzenie jakiegoś ważnego elementu procesora lub układu elektronicznego. Przerwania te nie są typowo wykorzystywane — zamiast z nich powinniśmy korzystać ze zwykłych przerwań maskowalnych. XMEGA dysponują tylko jednym źródłem przerwań NMI — przerwanie takie jest generowane w chwili wykrycia uszkodzenia generatora kwarcowego, co prowadzi do wygenerowania przerwania o wektorze `OSCF_INT_vect`. Funkcja obsługi tego przerwania powinna zadbać o przełączenie generatorów zegara i ich ponowne skonfigurowanie oraz zasygnalizować użytkownikowi poważną awarię układu elektronicznego.

Rejestr stanu kontrolera przerwań

Kontroler przerwań ma własny rejestr stanu (`STATUS` — ang. *PMIC Status Register*). Zawiera on bity informacyjne informujące o aktualnym stanie przerwań, lecz w większości aplikacji prawdopodobnie płynące z tego informacje są zbędne. Ustawienie odpowiednich flag tego rejestru (`NMIEX`, `HILVLEX`, `MEDLVLEX` i `LOLVLEX`) sygnalizuje, że przerwania o odpowiadającym tym flagom poziomie są aktualnie wykonywane. Po zakończeniu ich obsługi stosowne flagi są kasowane (dokładniej są kasowane w chwili wykonania związanej z danym poziomem przerwań instrukcji `RETI`).

Modyfikator volatile

Słowo kluczowe `volatile` jest modyfikatorem, którego możemy użyć do poinstruowania kompilatora, jak ma traktować podaną zmienną. Jego użycie jest szczególnie częste w programowaniu systemów czasu rzeczywistego i mikrokontrolerów. Modyfikator ten przysparza często problemów osobom początkującym lub osobom, które wcześniej pracowały na większych komputerach, gdzie stosuje się go rzadko. Jego brak prowadzi do „dziwacznego" zachowania programu.

> Jeśli Twój program świetnie działa z wyłączoną optymalizacją (-00), a po jej włączeniu przestaje, to jedną z najczęstszych przyczyn takiego zachowania jest brak użycia słowa kluczowego `volatile` tam, gdzie jest to niezbędne.

Przeanalizujmy przykład:

```
char a;

ISR(TCC0_CCB_vect)
{
   a++;
}

int main()
{
   a=0;
   while(a==0) {};
}
```

Program ten po skompilowaniu z wyłączoną optymalizacją działa prawidłowo, po jej włączeniu ku naszemu zaskoczeniu program nigdy nie opuszcza pętli `while`, pomimo że zmienna `a` jest prawidłowo inkrementowana w procedurze obsługi timera. Problem leży właśnie w braku modyfikatora `volatile` dla zmiennej `a`. Jego dodanie powoduje, że zgodnie z naszymi oczekiwaniami za każdym razem zmienna `a` pobierana jest z pamięci i testowany jest warunek zakończenia pętli. Czym zatem jest słowo kluczowe, od którego tyle zależy? **Najogólniej pisząc, `volatile` wskazuje kompilatorowi, że zmienna zadeklarowana z tym modyfikatorem może być modyfikowana w sposób niewynikający bezpośrednio z otaczającego kodu.** Przy odwołaniu do takiej zmiennej kompilator zawsze będzie pobierał jej wartość z pamięci, unikając tworzenia jej kopii np. w rejestrach procesora. Efektem ubocznym jest wydłużenie kodu, gdyż zmienna zadeklarowana z tym atrybutem przy każdej operacji będzie odczytywana z pamięci, a następnie nowa wartość będzie natychmiast ponownie zapisywana.

> Dodanie modyfikatora `volatile` jest niezbędne w sytuacji, kiedy zmienna globalna jest modyfikowana w procedurze obsługi przerwania i jakiejkolwiek innej części kodu lub kiedy jej wartość może zmieniać się w sposób niewynikający bezpośrednio z wykonywanego kodu, jak to ma miejsce w przypadku rejestrów układów peryferyjnych.

Ten drugi przypadek obejmuje głównie zmienne, które odzwierciedlają rejestry sprzętowe, np. rejestry IO procesora. Dla przykładu stan rejestru `PORT.IN` procesora zależy od części sprzętowej układu, a nie od wykonywanego programu w języku C. Każdorazowo przy odwołaniu do takiego portu kompilator powinien wygenerować kod pobierający jego nową wartość, a nie używać wartości poprzednio odczytanej. Stąd też dostęp do rejestrów procesora odbywa się za pomocą predefiniowanych makr, które są rozwijane do definicji podobnej do przedstawionej poniżej:

```
(*(volatile uint8_t *)(mem_addr))
```

Powoduje ona, że zmienna traktowana jest jako wskaźnik na zasób sprzętowy, do którego chcemy uzyskać dostęp. Modyfikator `volatile` powoduje, że kompilator, napotykając taką konstrukcję, każdorazowo będzie pobierał nową wartość z rejestru IO lub wskazanego miejsca w pamięci.

Modyfikator ten możemy dodawać także do złożonych typów danych. W przypadku struktury zadeklarowanej z modyfikatorem `volatile` automatycznie każde jej pole zachowuje się tak, jakby było zdefiniowane z tym modyfikatorem. Jeśli wiemy, że zmienne o danym typie zawsze będą wymagały atrybutu `volatile`, to lepiej umieścić go w deklaracji struktury. Uchroni to programistę przed przypadkowym opuszczeniem słowa kluczowego `volatile`, co spowodowałoby nieprawidłowe działanie programu. Ma to jednak wadę polegającą na niemożności zadeklarowania zmiennej o podanym typie bez modyfikatora `volatile`, co — jak za chwilę zobaczymy — czasami bywa przydatne. Możemy więc zadeklarować strukturę normalnie, pamiętając, aby przy zmiennych tego wymagających dodawać modyfikator `volatile`. Możemy także zadeklarować dwie wersje struktury, jedną bez modyfikatorem `volatile`, a drugą z modyfikatorem:

```
struct IO
{
//Pola struktury
};
typedef volatile struct IO IO_v;
IO zmienna_nievolatile;
IO_v zmienna_volatile;
```

W powyższym przykładzie zadeklarowaliśmy dwa typy: jeden `IO`, a drugi `IO_v`, będący taką samą strukturą, ale zmienna zadeklarowana za jej pomocą będzie miała dodatkowo modyfikator `volatile`.

Modyfikator `volatile` „wyłącza” optymalizację dotyczącą zmiennej zadeklarowanej za jego pomocą, a więc odwołania do niej stają się bardzo kosztowne, zarówno pod względem czasu egzekucji programu, jak i liczby potrzebnych do tego instrukcji (a więc objętości generowanego kodu). W procedurze obsługi przerwania lub w bloku krytycznym, w którym przerwania są zablokowane i mamy pewność, że nic nie zmodyfikuje naszej zmiennej poza głównym ciągiem instrukcji, moglibyśmy chwilowo „wyłączyć” działanie modyfikatora `volatile`. Niestety, standard języka C nie przewiduje takiej możliwości. Jednak łatwo możemy tego dokonać, stosując sztuczkę polegającą na przypisaniu wartości zmiennej zadeklarowanej z modyfikatorem `volatile` do innej zmiennej, bez tego modyfikatora. Następnie operacje przeprowadzamy na tak utworzonej zmiennej pomocniczej, której ostateczną wartość z powrotem przepisujemy do pierwszej zmiennej. Ilustruje to przykład:

```
volatile int x;

ATOMIC_BLOCK(ATOMIC_RESTORE_STATE)
{
 int xtmp=x;
 tmpx*=tmpx;
 tmpx<<=1;
 x=tmpx;
};
```

W powyższym przykładzie zmienna x jest zmienną o atrybucie volatile. Przeprowadzone na niej operacje są w sekcji krytycznej, w której przerwania są zablokowane. Mamy więc gwarancję, że nic poza ciągiem instrukcji w tej sekcji nie zmodyfikuje zmiennej x. Stąd też w celu wygenerowania optymalnego kodu przepisaliśmy wartość zmiennej x do zmiennej tymczasowej, tmpx, na której prowadzimy dalsze operacje, a ich wynik z powrotem przepisujemy do zmiennej x. Zazwyczaj utworzenie takiej „sztucznej" zmiennej służącej wyłącznie do oszukania kompilatora jest wykrywane przez optymalizator, w efekcie żadna dodatkowa zmienna nie jest tworzona, a po prostu odwołania do zmiennej z atrybutem volatile traktowane są tak, jakby tego atrybutu nie było. **Oprócz sekcji krytycznych miejscem, gdzie możemy bezpiecznie posłużyć się powyższą sztuczką, są procedury obsługi przerwań, o ile zmienna używana jest wyłącznie w przerwaniach tego samego poziomu**. Domyślnie procedury obsługi przerwań tego samego poziomu nawzajem się blokują, nie ma więc obawy, że zmienna zmieni swój stan w sposób niewynikający z realizowanego kodu. Jeśli zmienna volatile wskazuje na zasób sprzętowy mogący ulec zmianie w trakcie wykonywania kodu programu, to oczywiście jej przypisanie do zmiennej pomocniczej powoduje „zamrożenie" stanu tego zasobu w zmiennej pomocniczej.

Zmiana kolejności instrukcji

Modyfikator volatile ma jeszcze jedno zastosowanie. Normalnie optymalizator może przestawiać wyrażenia, aby zapewnić optymalny przebieg programu, np. sekwencja:

```
int x=10;
x=y+2;
x=s+2;
```

wcale nie musi zostać wykonana w kolejności, w jakiej ją zapisaliśmy. Wynik programu będzie dokładnie taki sam, niezależnie, czy najpierw wykonamy operację x=y+2, czy x=s+2. W związku z tym optymalizator przestawi kolejność tych wyrażeń tak, aby otrzymać krótszy lub szybszy kod wynikowy. Modyfikator volatile uniemożliwia optymalizatorowi zmianę kolejności operacji, dzięki czemu wykonywane są one w takiej kolejności, w jakiej pojawiają się w kompilowanym programie. Dobrym przykładem wykorzystującym tę właściwość są rejestry IO procesora. Ich definicje zawierają modyfikator volatile, dzięki czemu optymalizator nie zmienia kolejności podanych instrukcji. W rezultacie ciąg instrukcji:

```
PORTA.OUT=0;
PORTA.OUT=0xFF;
PORTA.OUT=0;
```

zawsze zostanie wykonany w podanej kolejności. Bez tego modyfikatora optymalizator stwierdziłby, że powyższe trzy instrukcje można zredukować do ostatniej, a więc na porcie nie pojawiłaby się przejściowo wartość `0xFF`.

Atomowość dostępu do danych

W paragrafie poświęconym modyfikatorowi `volatile` rozważaliśmy działanie następującego kodu:

```
volatile char a;

ISR(TCC0_CCB_vect)
{
 a++;
}

int main()
{
 a=0;
 while(a==0) {};
}
```

Zastanówmy się, czy gdyby zmienić typ zmiennej `a` z typu `char` na np. typ `int`, to czy program również wykonywałby się prawidłowo? Pozornie tak, ale zobaczmy, jak w takim przypadku jest realizowana pętla `while(a==0) {};`. Okazuje się, że porównanie to realizowane jest „na raty”:

```
 while(a==0) {};
  d2: 80 91 02 01  lds r24, 0x2002
  d6: 90 91 03 01  lds r25, 0x2003
  da: 89 2b        or r24, r25
  dc: d1 f3        breq .-12       ; 0xd2 <volatile_test+0x8>
```

Najpierw odczytywana jest starsza część 16-bitowej zmiennej, a następnie młodsza. Co jeśli przerwanie timera nastąpi pomiędzy tymi operacjami? W przypadku kiedy `a` zawierało `0` i pętla powinna zostać przerwana, kolejne przerwanie może zmienić jej stan na 1 i warunek nie będzie spełniony. Podobnie w poniższym przykładzie:

```
int a;

ISR(TCC0_CCB_vect)
{
 a++;
}

int main()
{
 a=0;
 while(1)
 {
  a++;
 };
}
```

inkrementacja zmiennej a czasami będzie prowadziła do nieprawidłowych wyników. Aby temu zapobiec, musimy zapewnić, aby operacje przeprowadzone na zmiennej a były atomowe, to znaczy aby niemożliwe było ich przerwanie. *AVR-libc* dostarcza wygodnych narzędzi do realizacji atomowego dostępu do zmiennych czy zamykania całych bloków instrukcji w sekcje atomowe. Funkcje tego typu znajdują się w pliku nagłówkowym <*util\atomic.h*>. W pliku tym znajduje się makrodefinicja `ATOMIC_BLOCK(typ)`, która przyjmuje dodatkowy parametr określający sposób jej działania. Parametr ten przybiera następujące postaci:

- `ATOMIC_FORCEON`
- `ATOMIC_RESTORESTATE`
- `NONATOMIC_FORCEOFF`

Najczęściej wykorzystujemy opcję `ATOMIC_RESTORESTATE`, powodującą zapamiętanie przed wejściem do sekcji krytycznej globalnej flagi zezwolenia na przerwania, zablokowanie przerwań, wykonanie kodu z sekcji i na końcu przywrócenie stanu flagi zezwolenia na przerwanie. W związku z tym jeśli przed wejściem do sekcji krytycznej przerwania są zablokowane, to po jej opuszczeniu również zostaną zablokowane. Jeśli były włączone, to po opuszczeniu sekcji zostaną włączone. W niektórych sytuacjach wiemy jednak, jaki jest zastany stan flagi zezwolenia na przerwania, nie musimy więc go zapamiętywać. W takiej sytuacji możemy zastosować parametr `ATOMIC_FORCEON`, powodujący, że po opuszczeniu sekcji krytycznej przerwania będą odblokowane, lub rzadziej `NONATOMIC_FORCEOFF`, powodujący, że przerwania po opuszczeniu sekcji pozostaną zablokowane.

Nasz przykładowy kod wykorzystujący makrodefinicję `ATOMIC_BLOCK` wygląda więc następująco:

```
int a;

ISR(TCC0_CCB_vect)
{
 a++;
}

int main()
{
...inicjalizacja PMIC i odblokowanie przerwań
 sei();
 a=0;
 while(1)
 {
  ATOMIC_BLOCK(ATOMIC_FORCEON)
  {
  a++;
  };
 };
}
```

Wykorzystaliśmy tu parametr `ATOMIC_FORCEON`, gdyż stan flagi zezwolenia na przerwanie jest znany.

> Makrodefinicja `sei()` powinna być wykorzystywana tylko raz w programie, do odblokowania przerwań. Nigdy nie powinna zajść sytuacja, w której konieczne byłoby wykorzystanie makrodefinicji `cli()`.

Gdy testujemy warunek logiczny pętli, tak jak w pierwszym przykładzie, nie jest możliwe wykorzystanie makrodefinicji `ATOMIC_BLOCK`. Aby taki program działał poprawnie, musimy w sposób atomowy przepisać zawartość licznika do zmiennej pomocniczej, a następnie w pętli testować tę zmienną:

```
volatile char a;

ISR(TCC0_CCB_vect)
{
 a++;
}

int main()
{
 a=0;
 int tmpa=0;
 while(tmpa==0)
{
 ATOMIC_BLOCK(ATOMIC_FORCEON)
 {
  tmpa=a;
 };
};
}
```

W tym przypadku operację porównania wykonujemy na kopii zmiennej `a`, nie zachodzi więc obawa, że zostanie ona niespodziewanie zmodyfikowana. Zauważmy, że operacja przypisania zmiennej `tmpa` wartości zmiennej `a` nie jest w mikroprocesorach AVR operacją atomową, stąd zachodzi konieczność zastosowania makrodefinicji `ATOMIC_BLOCK`.

> Osoby znające asembler mikroprocesorów AVR z pewnością w tym momencie zaprotestują. Niektóre architektury AVR mają zaimplementowaną wykonywaną atomowo instrukcję `MOVW Rd,Rr`. Niestety, pisząc w języku wysokiego poziomu, jakim jest język C, nie mamy gwarancji, że instrukcja `tmpa=a` zostanie zamieniona na wykonywaną atomowo instrukcję asemblera `MOVW`.

W sytuacji, kiedy w dużej sekcji krytycznej chcemy czasowo zezwolić na nieatomowe wykonywanie niektórych instrukcji, użyteczna może okazać się makrodefinicja `NONATOMIC_BLOCK(`*typ*`)`. Jej parametry są identyczne jak w przypadku makra `ATOMIC_BLOCK`.

Makrodefinicja `ATOMIC_BLOCK` ingeruje w globalną flagę zezwolenia na przerwanie, stąd też w sekcjach atomowych wszystkie przerwania są blokowane[3]. Nie zawsze jest to wymagane i pożądane. Jeśli na przykład do jakiejś zmiennej dostęp następuje w kodzie aplikacji i kodzie przerwania określonego poziomu, to wystarczy, że na czas dostępu

[3] Z wyjątkiem przerwań NMI.

do niej zablokujemy wyłącznie dany poziom przerwań lub wręcz dane źródło przerwań. W efekcie procesor będzie mógł obsługiwać przerwania pozostałych poziomów w sposób niezakłócony. Niestety makrodefinicje dostarczane przez AVR-libc nie udostępniają takiej funkcjonalności, z drugiej strony wykorzystywanie takich sztuczek prowadzi do powstania kodu bardzo wrażliwego na błędy.

Instrukcje atomowej modyfikacji pamięci

Pod tym nieco enigmatycznym pojęciem pokazane zostaną nowe instrukcje asemblera, jakimi dysponuje rdzeń XMEGA należących do drugiej generacji tych procesorów. W stosunku do XMEGA pierwszej generacji (serie A oraz D) rdzeń XMEGA drugiej generacji (serie AU, B, C, E) realizuje cztery nowe instrukcje:

- `XCH` realizującą instrukcję wymiany pomiędzy rejestrem procesora a komórką pamięci adresowaną przez rejestr indeksowy `Z`;
- `LAS` realizującą instrukcję ustawiania wskazanych bitów w komórce pamięci adresowanej przez rejestr `Z`, przy czym poprzednia wartość tej komórki zostaje przepisana do wskazanego rejestru CPU;
- `LAC` realizująca operację podobną do `LAS`, z tym że wskazany bit jest kasowany;
- `LAT` realizującą podobne operacje, z tym że dany bit ulega zmianie na przeciwny.

Dla przykładu instrukcja `LAS` w sposób atomowy realizuje następującą sekwencję poleceń:

```
LAS Z, Rx:
 Tmp=(Z)
 (Z)= Tmp | Rx
 Rx=Tmp
```

w efekcie `Rx` zawiera poprzednią wartość komórki pamięci wskazywanej przez `(Z)`, a komórka pamięci wskazywana przez `(Z)` zawiera sumę bitową swojej poprzedniej wartości i `Rx`.

Aby móc wykorzystać te instrukcje, oprócz odpowiedniego rdzenia CPU musimy wykorzystać w miarę nową wersję *toolchaina* — są one dostępne, począwszy od wersji `avr-gcc` 4.6.2 udostępnionej przez firmę Atmel (*AVR Toolchain 8 Bit, Version: 3.4.1.830 — GCC 4.6.2*).

Do czego te instrukcje możemy wykorzystać? Przede wszystkim realizowana przez nie operacja jest operacją atomową, co natychmiast czyni je obiektem zainteresowania we wszystkich zastosowaniach, w których w sposób atomowy musimy zmieniać stan jakiejś flagi. Ma to zastosowanie przy programowaniu współbieżnym, gdzie dostęp do danego zasobu może obywać się w kilku wątkach. Instrukcje te umożliwiają więc stworzenie tzw. semaforów binarnych i ułatwiają tworzenie ich rozwinięcia — semaforów zliczających. Stąd też konstrukcja typu:

```
if(!(flagi & 1)) //Czy dostęp do zasobu jest wolny? (flaga wyzerowana)
{
   flagi|=1;     //Zakaż dostępu do zasobu innym procesom
  ... ciąg dalszy instrukcji wykonywanych, jeśli flaga nie była ustawiona...
}
```

którą należy wykonywać atomowo, co wymaga najczęściej zablokowania przerwań, może zostać znacznie uproszczona — można ją zrealizować za pomocą jednej instrukcji asemblera — w tym przypadku LAS.

> Ponieważ język C nie dysponuje składnią umożliwiającą implementację tego typu zadań, a jednocześnie gwarantującą wykorzystanie atomowego dostępu do danych, odpowiednie przykłady zaimplementujemy przynajmniej częściowo w asemblerze.

Nasze wstawki zaimplementujemy jako krótkie funkcje języka C, które w łatwy sposób udostępnią funkcjonalność nowych instrukcji dla programów napisanych w tym języku. Dla przykładu zajmijmy się instrukcją LAS (implementacja LAC i LAT jest taka sama). Instrukcja ta operuje na komórce pamięci wskazywanej przez rejestr indeksowy Z. Jako drugi argument występuje dowolny rejestr (R0 – R31), który zawiera maskę bitową. Po wykonaniu operacji rejestr ten będzie zawierał poprzednią wartość komórki wskazywanej przez Z. Informacje te umożliwią nam stworzenie stosownej funkcji:

```
uint8_t Asm_LAS(volatile uint8_t *flag, uint8_t val)
{
 asm volatile ("las %a1, %0" : "+r" (val) : "e" (flag));
 return val;
}
```

Nasza funkcja jako argumenty przyjmuje adres komórki, którą w sposób atomowy chcemy zmodyfikować, oraz maskę bitową, z którą sumować będziemy wartość komórki przechowującej semafory. Funkcja ta zwraca poprzednią wartość tej komórki pamięci. Jak pamiętamy, zapis "e" oznacza, że zmienna flag znajduje się w jednym z rejestrów indeksowych. Ponieważ LAS/LAC/LAT wymaga, aby był to rejestr Z, musimy to wymusić poprzez zapis las %a1 — %a1 spowoduje, że drugi argument z listy argumentów słowa kluczowego asm (a więc zmienna flag) zostanie umieszczony w rejestrze Z. Ponieważ instrukcje te modyfikują swój drugi operand — argument funkcji — zmienną flag zadeklarowano z modyfikatorem volatile. Dzięki temu kompilator wie, że wartość komórki pamięci wskazywanej przez flag może ulec zmianie i jeśli jest ona przechowywana w rejestrach w czasie wykonania funkcji, to zostanie odczytana ponownie. Takie rozwiązanie jest efektywniejsze niż tworzenie listy modyfikowanych wartości, która musiałaby zawierać "memory", w efekcie czego wszelkie wartości odczytane z pamięci, a umieszczone w rejestrach musiałyby zostać ponownie odczytane. Jednak instrukcje te modyfikują także swój drugi argument — przekazywany rejestr, stąd też jest on na liście operandów wyjściowych z określeniem "+r" (val), co jak wiemy, oznacza, że operand taki jest umieszczony w dowolnym rejestrze, przy czym instrukcje asemblera nie tylko go odczytują, ale także modyfikują. Jest to istotne, gdyż bez informacji o modyfikacji tego operandu kompilator w ramach optymalizacji nie przeładowywałby rejestru reprezentującego wartość val pomiędzy kolejnymi instrukcjami, w efekcie czego powstałby nieprawidłowy kod.

Ponieważ nasza funkcja jest krótka — składa się zaledwie z jednej instrukcji (oraz opcjonalnie instrukcji generowanych przez kompilator służących do załadowania wartości val i adresu zmiennej flag), warto zadbać, aby nie było tworzone wywołanie do niej, lecz była ona osadzana w miejscu wywołania:

```
uint8_t Asm_LAS(volatile uint8_t *flag, uint8_t val)
__attribute__((always_inline));
```

Jak widzimy, zadba o to atrybut `always_inline`.

Spróbujmy zobaczyć, jak nasze instrukcje działają w praktyce — kod znajduje się w katalogu *Przyklady/Przerwania/LAx*.

Prosty kod ilustrujący wykorzystanie tych funkcji może wyglądać tak:

```
Asm_LAS(&flagi, 1);
Asm_LAS(&flagi, 1);
Asm_LAC(&flagi, 1);
Asm_LAT(&flagi, 3);
```

i spowoduje on wygenerowanie następującego kodu asemblerowego:

```
   asm volatile ("las %a1, %0" : "+r" (val) : "e" (flag));
 25c:   e0 e0          ldi   r30, 0x00   ; 0
 25e:   f0 e2          ldi   r31, 0x20   ; 32
 260:   81 e0          ldi   r24, 0x01   ; 1
 262:   98 2f          mov   r25, r24
 264:   95 93          las   Z, r25
 266:   98 2f          mov   r25, r24
 268:   95 93          las   Z, r25
   return val;
}
uint8_t Asm_LAC(volatile uint8_t *flag, uint8_t val)
{
   asm volatile ("lac %a1, %0" : "+r" (val) : "e" (flag));
 26a:   86 93          lac   Z, r24
   return val;
}
uint8_t Asm_LAT(volatile uint8_t *flag, uint8_t val)
{
   asm volatile ("lat %a1, %0" : "+r" (val) : "e" (flag));
 26c:   83 e0          ldi   r24, 0x03   ; 3
 26e:   87 93          lat   Z, r24
```

Warto jako samodzielne ćwiczenie prześledzić działanie tego kodu w symulatorze, a także sprawdzić, jak zmiana typu operandów w instrukcji `asm` wpływa na poprawność generowanego kodu asemblerowego.

Instrukcje tego typu możemy wykorzystać także jako szybsze i krótsze alternatywy dla operacji bitowych — `&`, `|` i `^`. W takiej sytuacji nie wykorzystujemy zwracanej przez pokazane funkcje wartości. Co nam to daje? Typowe operacje bitowe są operacjami RMW, w efekcie prosta operacja:

```
X|=3;
```

ustawiająca dwa najmłodsze bity zmiennej `X` wymaga jej załadowania do rejestru, wykonania operacji sumy logicznej i przeładowania wyniku do komórki pamięci, w której zmienna ta jest przechowywana. Dzięki zastosowaniu instrukcji `LAx` cała operacja jest krótsza i szybsza.

Dostęp do wielobajtowych rejestrów IO

Mikrokontrolery XMEGA podobnie jak wcześniejsze wersje mikrokontrolerów AVR ze względu na swoją 8-bitową budowę nie są w stanie w sposób atomowy uzyskać dostępu do danych dłuższych niż 8 bitów. W przypadku wielobajtowych zmiennych możemy ten problem rozwiązać, zamykając instrukcje realizujące dostęp do zmiennych w bloki atomowe. Warto pamiętać, że podobnie sytuacja ma się w przypadku dostępu do wielobajtowych rejestrów IO. Dostęp do wszelkich rejestrów o długości większej niż 8 bitów odbywa się sekwencyjnie bajt po bajcie. Rodzi to pewne problemy. Aby zapis do całego rejestru wykonywał się w tym samym czasie, co jest istotne w przypadku np. rejestrów `CNT` timerów, wprowadzono rejestry tymczasowe, z których wartość przepisywana jest do właściwego rejestru w chwili zapisu najstarszego bajtu rejestru. Rejestr tymczasowy jest zawsze wspólny dla wszystkich wielobajtowych rejestrów IO danej instancji modułu. To znaczy, że np. timery `C0` i `C1` posiadają oddzielne rejestry tymczasowe, lecz w ramach instancji modułu, np. timera `C0`, wszystkie jego 16-bitowe rejestry współdzielą ten sam rejestr tymczasowy. Jakie to niesie konsekwencje, zostało szerzej omówione w rozdziale 12. „Timery i liczniki". Warto jednak pamiętać, że te same problemy występują dla wszystkich układów peryferyjnych posiadających wielobajtowe rejestry IO.

> W efekcie należy zachować szczególną ostrożność, jeśli do danego układu peryferyjnego odwołujemy się w aplikacji i w funkcji obsługi przerwania. W takiej sytuacji odwołania do wszystkich rejestrów dłuższych niż 8-bitowe muszą być realizowane w sposób atomowy — np. ujęte w blok `ATOMIC_BLOCK`.

Nieprzestrzeganie tej zasady doprowadzi do błędnych zapisów i (lub) odczytów z danego układu peryferyjnego i w konsekwencji nieprzewidywalnego działania aplikacji.

Funkcje reentrant

Kod *reentrant* to kod, który może być wykonywany jednocześnie z dwóch lub więcej wątków. Co prawda kojarzy się to z systemami wielozadaniowymi, ale w pewnych warunkach sytuacja taka może występować także podczas programowania na AVR bez obecności systemu operacyjnego. Funkcja może zostać ponownie wywołana przed zakończeniem jej działania w dwóch sytuacjach:

- W przypadku funkcji rekurencyjnych — jednak w tym wypadku miejsce przerwania funkcji jest dokładnie znane.
- W przypadku funkcji wywoływanych w procedurze obsługi przerwań — w tym wypadku miejsce przerwania funkcji przez przerwanie jest nieprzewidywalne, co stwarza pewne kłopoty.

Podczas wykonywania procedury obsługi przerwania istnieje możliwość wywołania innych funkcji, może więc dojść do sytuacji, w której wykonywana funkcja jest przerywana, a następnie wywoływana ponownie z procedury obsługi przerwania. Musimy

sobie zdawać sprawę z tego, że nie każda funkcja może być bezpiecznie wywoływana w sytuacji, w której jej poprzednie wywołanie nie zostało zakończone. Problem funkcji *reentrant* zwykle nie dotyka funkcji, które operują wyłącznie na zmiennych lokalnych. Zmienne takie tworzone są na stosie i każde wywołanie funkcji operuje na swojej lokalnej kopii zmiennych. Wyjątkiem od tej reguły są zmienne lokalne zadeklarowane z modyfikatorem `static`. W tym przypadku istnieje tylko jedna kopia zmiennej, która z punktu widzenia funkcji zachowuje się jak zmienna globalna. Zdecydowanie „niebezpieczne" są funkcje operujące na danych zewnętrznych — zmiennych globalnych lub wskaźnikach do współdzielonych zasobów lub danych statycznych. Z pewnością kłopoty sprawią funkcje, które odwołują się bezpośrednio do zasobów sprzętowych: funkcje wysyłające dane przez magistrale procesora, realizujące transmisje I2C, 1-wire, SPI itd. Aby mogły być bezpiecznie wywoływane z procedury obsługi przerwania, muszą być napisane w specjalny sposób.

Zdecydowana większość funkcji w *AVR-libc* jest *reentrant*, wyjątkiem są funkcje pokazane w tabeli 10.2.

Tabela 10.2. *Funkcje biblioteki AVR-libc, które nie są reentrant*

Funkcja	Problem	Rozwiązanie
`rand()`, `random()`	Używają zmiennych globalnych	Użyj wersji reentrant `rand_r()`, `random_r()`
`strtod()`, `strtol()`, `strtoul()`	Używają zmiennej globalnej `errno`	Funkcje są reentrant, jeśli aplikacja nie korzysta z kodów błędów zwracanych w errno lub są one chronione makrem `ATOMIC_BLOCK`
`malloc()`, `realloc()`, `calloc()`, `free()`	Używają stosu i innych struktur globalnych do zarządzania pamięcią	Ochrona za pomocą `ATOMIC_BLOCK` lub napisanie własnych funkcji zarządzających pamięcią
`fdevopen()`, `fclose()`	Używają funkcji `calloc()` i `free()`	jw.
`eeprom_*()`, `boot_*()`	Używają rejestrów IO	jw.
`printf()`, `printf_P()`, `vprintf()`, `vprintf_P()`, `puts()`, `puts_P()`	Zmieniają globalny strumień `stdout`	jw. Użycie innych podobnych funkcji, jak `sprintf()`, `sprintf_P()`, `snprintf()`, `snprintf_P()`, `vsprintf()`, `vsprintf_P()`, `vsnprintf()`, `vsnprintf_P()`, jest bezpieczne
`fprintf()`, `fprintf_P()`, `vfprintf()`, `vfprintf_P()`, `fputs()`, `fputs_P()`	Problem występuje tylko wtedy, gdy jako argument podany jest ten sam strumień o typie `FILE`	Strumień podany jako argument wywołania musi być różny dla kolejnych wywołań
`assert()`	Wywołuje funkcję `fprintf()`	jw.
`clearerr()`	Zmienia argument `FILE`	jw.

Tabela 10.2. *Funkcje biblioteki AVR-libc, które nie są reentrant (ciąg dalszy)*

Funkcja	Problem	Rozwiązanie
`getchar()`, `gets()`	Korzysta ze strumienia globalnego `stdin`	jw. Aczkolwiek czytanie ze `STDIN` w różnych wątkach nie ma sensu.
`fgetc()`, `ungetc()`, `fgets()`, `scanf()`, `scanf_P()`, `fscanf()`, `fscanf_P()`, `vscanf()`, `vfscanf()`, `vfscanf_P()`, `fread()`	Problem występuje tylko wtedy, gdy jako argument podany jest ten sam strumień o typie `FILE`	jw.

Jak widać, w wielu przypadkach rozwiązaniem problemu funkcji *reentrant* jest wywoływanie ich jako operacji atomowych, np.:

```
ATOMIC_BLOCK(ATOMIC_RESTORESTATE)
{
 void *ptr=malloc(100);
}
```

W ten sposób mamy gwarancję, że funkcja `malloc` nie zostanie przerwana i wywołana ponownie np. w procedurze obsługi przerwania. Rozwiązanie to jest efektywne, jeśli możemy pozwolić sobie na zablokowanie obsługi przerwań na czas wywołania takiej funkcji.

Najbezpieczniej jest jednak unikać wywołania w procedurze obsługi przerwania funkcji, co do których nie jesteśmy pewni, że są one funkcjami *reentrant*.

Rejestry IO ogólnego przeznaczenia

Podobnie jak inne modele AVR, także XMEGA jest wyposażona w tzw. rejestry IO ogólnego przeznaczenia (ang. *General Purpose IO Registers*), w skrócie `GPIOR`. Rejestry te nie są powiązane z żadnym układem peryferyjnym i nie pełnią żadnej specjalnej funkcji. W XMEGA znajdują się one w przestrzeni IO, począwszy od adresu `0x0000` aż do adresu `0x000F`, stąd też mamy dostęp do 16 ośmiobitowych rejestrów. Ich szczególna przydatność wynika z istnienia w asemblerze AVR specjalnych instrukcji `SBI/CBI` umożliwiających atomowy, zajmujący 1 takt zegara dostęp do poszczególnych bitów tych rejestrów. Jest to szczególnie cenne, gdy chcemy przekazywać jakieś informacje lub dane pomiędzy programem a funkcjami obsługi przerwań. W tym celu wygodnie jest się posłużyć rejestrami `GPIOR`. Dostęp do tych rejestrów odbywa się poprzez ich nazwy (`GPIOR0` – `GPIORF`) w taki sam sposób jak do zmiennych lub innych rejestrów IO, np.:

```
GPIOR1=10;
```

powoduje przypisanie rejestrowi GPIOR1 wartości 10. Stąd też rejestry GPIOR możemy traktować jako 16 predefiniowanych 8-bajtowych zmiennych. Korzyści ze stosowania tych rejestrów w pełni ujawniają się przy dostępie do poszczególnych ich bitów, dla przykładu ustawienie bitu:

```
GPIOR1|=1;
```

Może wygenerować prostą instrukcję:

```
   GPIOR1|=1;
 3cc:   08 9a          sbi   0x01, 0   ; 1
```

> Aby to było możliwe, musi być włączona optymalizacja. Pamiętaj też, że kompilator może wygenerować optymalny kod i zazwyczaj w tym przypadku go generuje, jednak ponieważ piszemy w języku wysokiego poziomu, nie ma gwarancji co do sposobu, w jaki kompilator wygeneruje kod asemblerowy.

Niemniej jeśli spróbujemy ustawić jednocześnie 2 bity:

```
GPIOR1|=3;
```

to zostanie wygenerowana normalna sekwencja instrukcji:

```
 3ce:   81 b1          in    r24, 0x01   ; 1
 3d0:   83 60          ori   r24, 0x03   ; 3
 3d2:   81 b9          out   0x01, r24   ; 1
```

Dlaczego? Wynika to z faktu, że instrukcje CBI/SBI mogą zmieniać stan wyłącznie jednego bitu. Jeśli chcemy zmienić naraz kilka bitów, należy użyć omówionych wcześniej instrukcji atomowej modyfikacji pamięci (o ile użyta XMEGA nimi dysponuje). Ich użycie:

```
Asm_LAS(&GPIOR1, 3);
```

co prawda wygeneruje ciąg instrukcji, ale sama modyfikacja rejestru przebiegnie w sposób atomowy.

Zmienne GPIOR możemy grupować, tworząc zmienne 2-, 4- i wielobajtowe. W tym celu wystarczy użyć zwykłego rzutowania typów, np.:

```
volatile uint16_t *wsk=(uint16_t*)&GPIOR2;
*wsk=0xaaff;
```

> Powyższe zadziała poprawnie z kompilatorem avr-gcc w wersji 4.7.2 i wyższych. Niższe wersje zawierają błąd powodujący wygenerowanie niepoprawnego kodu.

Rejestry GPIOR często traktuje się jako zbiór flag — w takiej sytuacji odwołujemy się do poszczególnych bitów rejestru, więc wygodnie jest zdefiniować odpowiednią strukturę:

```
typedef struct
{
   bool b0 : 1;
   bool b1 : 1;
   bool b2 : 1;
```

```
    bool b3 : 1;
    bool b4 : 1;
    bool b5 : 1;
    bool b6 : 1;
    bool b7 : 1;
} volatile IO;
```

Posiadając taką definicję struktury, możemy ją przypisać dowolnemu rejestrowi:

```
IO * const flagi=(IO*)&GPIO1;
```

Zmienna `flagi` będzie zawierała 8 flag, do których dostęp odbywać się będzie atomowo. Ale nasza struktura nie musi zawierać wyłącznie flag, możemy umieścić w niej dowolne pola:

```
struct
{
    bool b0 : 1;
    bool b1 : 1;
    bool b2 : 1;
    uint8_t licznik;
    uint16_t adres;
} volatile * const GPIO=(void*)&GPIOR0;
```

Zmienna `GPIO` zostanie automatycznie zainicjowana tak, aby wskazywać na początek obszaru `GPIO`, w efekcie maksymalna długość struktury może wynosić aż 16 bajtów. Pokazana struktura zajmuje 3 bajty, gdyż pole `licznik` jest wyrównane do granicy bajtu.

Skorowidz

A

B

C

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

Q

R

S

Ś

T

U

W

X

Z

Ź

Ż

Układy AVR przebojem wdarły się na rynek mikrokontrolerów. Szybko zdobyły serca hobbystów i profesjonalistów — zdecydowały o tym niska cena, ogromne możliwości i wydajność obliczeniowa oraz prostota używania i dostępność darmowych narzędzi, które ułatwiają programowanie tych układów. Szczególnie dużym uznaniem wśród użytkowników mikrokontrolerów cieszą się układy należące do rodziny XMEGA, jednak do tej pory brak było na polskim rynku publikacji opisujących ich architekturę, programowanie i zastosowanie.

Lukę tę doskonale wypełnia książka *AVR. Praktyczne projekty*, której autor postawił sobie za cel szczegółowe omówienie problemów związanych z programowaniem mikrokontrolerów XMEGA, z uwzględnieniem modułów, które nie występują w układach należących do innych rodzin AVR. Jeśli posiadasz już podstawowe umiejętności w zakresie programowania mikrokontrolerów, pozycja ta umożliwi Ci praktyczne rozwinięcie talentów w tej dziedzinie — samodzielnie zrealizujesz m.in. projekt zegara, analizatora logicznego i wiele innych!

To doskonała pozycja zarówno dla czytelników pierwszej książki Tomasza Francuza *Język C dla mikrokontrolerów AVR. Od podstaw do zaawansowanych aplikacji*, jak i dla osób, które opanowały podstawy programowania mikrokontrolerów we własnym zakresie.

- Architektura układów AVR XMEGA
- Warsztat pracy programisty mikrokontrolerów
- Tworzenie projektów i zarządzanie nimi
- Sposoby pisania efektywnego kodu
- Korzystanie z różnych typów danych
- Wykonywanie operacji wejścia-wyjścia
- Zarządzanie pamięcią mikrokontrolera
- Sterowanie pracą zegara
- Obsługa przerwań i system zdarzeń
- Używanie timerów i liczników
- Sterowanie urządzeniami zewnętrznymi
- Praktyczne przykłady zastosowań mikrokontrolerów

Dowiedz się, co można osiągnąć za pomocą języka C i układów AVR!

ISBN 978-83-246-7877-8
9 788324 678778
Cena: 99,00 zł