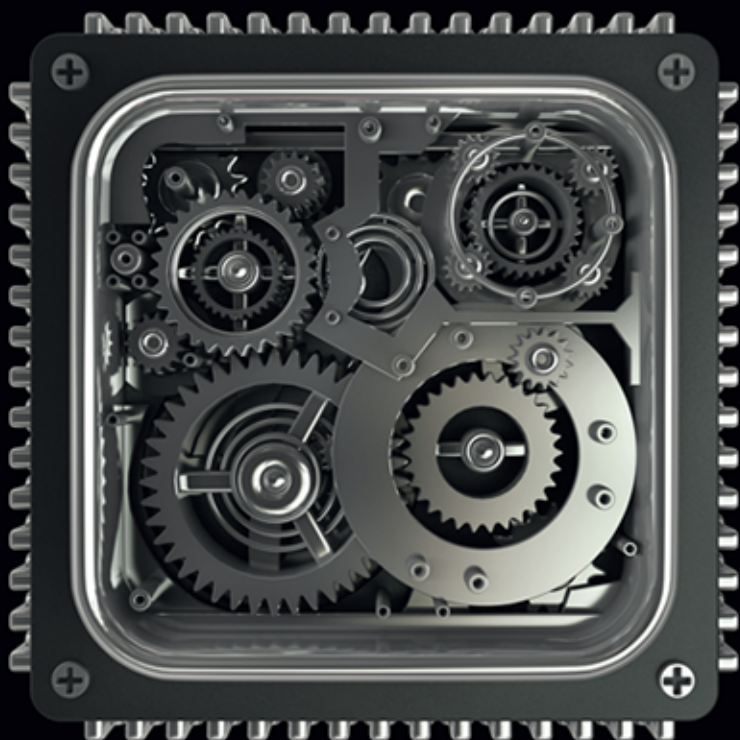


Tomasz Francuz

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



Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

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

Copyright © Helion 2013

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Rozdział 1. Wstęp, czyli nowi członkowie rodziny AVR	11
Przykłady do książki	12
Mikrokontrolery XMEGA	12
Errata	14
XMEGA a inne AVR-y	15
Kompilator	15
Nowe peryferia	16
Rozdział 2. Jak zacząć, czyli instalacja środowiska	21
Xplained jako płyta rozwojowa	22
Instalacja Xplained w systemie	22
Instalacja XMEGA z bootloaderem klasy DFU	25
Programowanie układów XMEGA	25
Wczytywanie firmware za pomocą FLIP	26
Bootloader firmy Atmel	28
Podstawowe opcje programu batchisp	28
Interfejs JTAG	32
Programowanie z wykorzystaniem interfejsu PDI	34
Programatory	34
AVRISP mkII	35
JTAGICE mkII	35
JTAGICEIII	36
AVROne!	36
AVR Dragon	36
Programowanie i debugowanie procesorów XMEGA	43
Odsprzęganie zasilania	44
Instalacja potrzebnego oprogramowania	44
Atmel Studio 6 — podstawy	45
Programowanie z poziomu AS6	53
— dlaczego warto korzystać z plików w formacie elf	53
Rozdział 3. Organizacja i zarządzanie projektem	55
Komentarze	55
Nazewnictwo	57
Literały	59
Podział kodu na pliki	61
Widoczność funkcji	66
Widoczność zmiennych	66

Podkatalogi	67
Podział funkcji	69
Zmienne globalne	70
Systemy wersjonowania	74
Serwer SVN na własnym komputerze	75
Jak korzystać z SVN	78
Praca z SVN	81
Uwagi końcowe	84
Inne ułatwienia	85
Jak pisać lepsze programy?	86
Rozdział 4. Jak tworzyć lepszy i efektywniejszy kod	89
Włączenie optymalizacji	90
Długość typów	91
Zmienne globalne i zmienne lokalne	93
Modyfikator register	94
Użycie globalnych rejestrów ogólnego przeznaczenia	97
Wykorzystanie innych rejestrów jako GPIOR	99
Inicjalizacja zmiennych globalnych	99
Położenie zmiennych w pamięci	99
Optymalizacja pętli	100
Optymalizacja funkcji	101
Optymalizacja funkcji main	101
Optymalizacja prologów i epilogów funkcji	101
Optymalizacja przekazywania parametrów funkcji	102
Optymalizacja zwracania wyników funkcji	105
Dodatkowe atrybuty funkcji	106
Inne	110
Eliminacja martwego kodu	111
Konstruktory i destruktory	114
Rozdział 5. Jak uporządkować chaos, czyli złożone typy danych i listy	117
Tablice	118
Struktury	119
Deep copy i shallow copy	123
Wskaźniki i struktury	126
Struktury anonimowe	127
Inicjalizacja pól struktury	128
Unie	129
Pola bitowe	130
Struktury a optymalizacja programu	131
Listy	135
Listy jednokierunkowe	136
Listy dwukierunkowe	140
Listy XOR	142
Bufory	148
Bufor pierścieniowy	149
Kolejki FIFO/LIFO	152
Stos	152
Rozdział 6. Koniec bałaganu, czyli o nazwach rejestrów i układów peryferyjnych	155
Struktury opisujące mikrokontrolery XMEGA	159
Nazwy rejestrów	162
Nazwy bitów	162

Rozdział 7. Lepiej i prościej, czyli porty IO procesora na sterydach	167
Piny wejściowe i wyjściowe	168
Konwersja poziomów logicznych pomiędzy układami pracującymi z różnymi napięciami zasilającymi	171
Łączenie wyjścia procesora z układem pracującym z napięciem 5 V	171
Zastosowanie aktywnego konwertera	174
Konwersja z napięcia wyższego na niższe	174
Dzielnik rezystorowy	176
Użycie do konwersji napięć buforów scalonych	177
Bufor dwukierunkowy stosowany w magistralach typu open drain	178
Kontrola nad portami procesora	181
Piny wejściowe i wyjściowe	182
Synchronizator	184
Konfiguracja sterownika pinu	186
Konfiguracja totem-pole	186
Konfiguracja Pull up/down	187
Konfiguracja bus keeper	188
Konfiguracja wired-AND	189
Konfiguracja wired-OR	191
Odwracanie wyjść IO	192
Kontrola szybkości opadania i narastania zboczy	193
Kontrola zdarzeń związanych z pinem	193
Rejestr kontrolny portu	194
Atomowa modyfikacja stanu pinów i wsparcie dla RMW	196
Alternatywne funkcje pinu	199
Porty wirtualne	200
Przekazywanie rejestru jako parametru funkcji	202
Remapowanie wyjść IO	203
 Rozdział 8. Kontroler NVM — jak prosto i przyjemnie dobrać się do pamięci	205
Pamięć EEPROM i związane z nią operacje	206
Dostęp do EEPROM z poziomu AVR-libc	208
Dostęp do EEPROM za pomocą bezpośredniego dostępu do kontrolera NVM	213
Techniki wear leveling	219
Rozdzielenie kasowania i zapisu pamięci	221
Dostęp do EEPROM z wykorzystaniem tokenów	224
EEPROM i awaria zasilania	232
Problem atomowości przy dostępie do EEPROM	244
Zapis do EEPROM z użyciem przerwań	244
Zapobieganie uszkodzeniu zawartości pamięci EEPROM	246
Dostęp do pamięci FLASH	246
Typy danych związane z pamięcią FLASH	248
Odczyt danych z pamięci FLASH	250
Dostęp do FLASH w kompilatorze avr-gcc 4.7 i wyższych	
— named address spaces	252
Wskaźniki wykorzystujące przestrzeń adresową	254
Typy 24-bitowe	258
 Rozdział 9. Potrzebuję więcej mocy — słów kilka o konfiguracji zegara	259
Rejestry kontrolne zegarów	260
Konfiguracja zegara	262
Źródła zegara	263
Odblokowywanie źródła zegara	270
Układ PLL	270

DFLL	272
Układ monitorowania zegara zewnętrznego	275
Zmiana źródła zegara i jego częstotliwości	277
Blokowanie ustawień zegara	277
Preskalery zegara	278
Uwagi	281
Kondensatory odsprzęgające	282
Rozdział 10. Przerwania i kontroler przerwań	285
Przerwania	285
Czym są przerwania?	285
Przerwania maskowalne	286
Przerwania niemaskowalne	287
Źródła przerwań	287
Konfiguracja i obsługa przerwań	288
Czas odpowiedzi na żądanie przerwania	288
Funkcja obsługi przerwania	289
Wektory przerwań	290
Puste wektory przerwań	292
Puste przerwania	293
Współdzielenie kodu przez przerwania	294
Atrybut naked i obsługa przerwań w assemblerze	295
Poziomy przerwań	297
Przerywanie przerwań	300
Priorytety przerwań	302
Priorytet dynamiczny	302
Globalna flaga zezwolenia na przerwanie	303
Przerwanie niemaskowalne	304
Rejestr stanu kontrolera przerwań	304
Modyfikator volatile	305
Zmiana kolejności instrukcji	307
Atomowość dostępu do danych	308
Instrukcje atomowej modyfikacji pamięci	311
Dostęp do wielobajtowych rejestrów IO	314
Funkcje reentrant	314
Rejestry IO ogólnego przeznaczenia	316
Rozdział 11. System zdarzeń	319
Rejestr multiplexera kanału zdarzeń	320
Zaawansowane funkcje kanału zdarzeń	322
Filtr cyfrowy	322
Dekoder kwadraturowy	322
Enkoder kwadraturowy z indeksem	331
Programowe sterowanie zdarzeniami	332
Zdarzenia jako sygnały sterujące układami zewnętrznymi	333
Inne funkcje rejestru	334
Rozdział 12. Timery i liczniki	337
Co to jest licznik?	337
Źródła zegara i preskaler	338
Typy i funkcje liczników	340
Licznik typu 0/1	341
Piny wyjściowe licznika	350
Licznik typu 2	351
Timery typu 4/5	353

Buforowanie	354
Kontrola nad licznikiem	357
Kaskadowe łączenie liczników	358
Rejestr tymczasowy TEMP	359
Wykorzystanie PWM do generowania sygnałów analogowych	361
Przykład — generowanie napięcia o zmiennej amplitudzie	363
Przykład — generowanie dowolnego przebiegu	364
PWM i przerwania	369
A może DMA?	371
Rozszerzenie zwiększające rozdzielczość	374
Tryb HiRes dla licznika typu 2	377
Rozdział 13. Kontroler DMA	379
Przesyłanie pamięć-pamięć	383
Odwracanie danych	384
Przesyłanie nakładających się bloków pamięci	385
Wypełnianie pamięci wzorcem	388
Przesyłanie pamięć-rejestr IO	389
Wyzwalacze	392
Praca buforowa	395
Priorytety kanałów DMA	396
Przerwania DMA	397
Błąd transmisji DMA	397
Przerwanie końca transakcji	398
Rozdział 14. LED-y — co z nich można wycisnąć?	399
Taśmy LED-owe	399
Trochę o właściwościach oka, czyli RGB w praktyce	403
Program sterujący	404
Wyświetlacze LED 7-segmentowe	407
Licznik LED	412
Matryce LED	416
Projekt PCB i zasilanie	421
Dobór napięcia zasilającego matrycę	422
Regulacja prądu diod	423
Układ z matrycą dwukolorową	423
Sterowanie matrycą	425
PWM inaczej, czyli jak uzyskać odcienie kolorów	433
Rozdział 15. Układy zegarowe w praktyce	437
RTC czy... RTC?	438
16-bitowy układ RTC	438
Synchronizacja dostępu do rejestrów RTC	441
Rejestry PER i COMP	442
RTC w trybie uśpienia	445
32-bitowy układ RTC	445
Generator sygnału zegarowego	445
Rejestry PER i COMP	445
Rejestr CNT	446
Przykładowy program kalendarzowy	446
Linuksowy marker czasowy	447
Konwersja czasu	447
Czas drogą radiową, czyli DCF77	451
Trochę więcej o DCF77	452
Dekodowanie danych	454

Moduł odbiornika DCF77	454
Przykład	455
Układ podtrzymywania zasilania	462
Dobór źródła zasilania awaryjnego	463
Wykorzystanie baterii	463
Superkondensatory	463
Akumulatory	464
Podtrzymanie zasilania dla całego procesora	465
Układ zapasowego zasilania bateryjnego	465

Rozdział 16. Budujemy zegar z budzikiem,	
czyli skończona maszyna stanów w praktyce	469
FSM oparta na switch/case	471
FSM oparta na tablicach	474
Zegar z alarmem	477
Jeszcze o maszynach stanu	489

Rozdział 17. Komunikacja na różne sposoby, czyli USART w praktyce	491
Elektryczna realizacja interfejsu USART	492
Format transmisji danych	494
Szybkość transmisji	496
Terminal	496
Podgląd transmisji danych	497
Wirtualny port szeregowy	499
Konfiguracja interfejsu	500
Konfiguracja pinów IO	501
Konfiguracja formatu ramki danych	502
Funkcje dodatkowe interfejsu	502
Ustawienie szybkości interfejsu	503
Kontrola poprawności danych	508
Transmisja danych	509
Realizacja transmisji przez pooling	509
Wykorzystanie przerw	512
Wykorzystanie DMA	517
Równoczesny dostęp do USART z wielu „wątków”	525
Dostęp do USART z wykorzystaniem strumieni	527
Metoda get	528
Metoda put	528
Otwieranie strumienia	529
Tryb MPCM	530

Rozdział 18. Wizualizacja danych	535
Atmel Data Visualizer	535
Format danych	537
Struktury wykorzystywane przez ADV	540
Ultradźwiękowy pomiar odległości	543
Moduły cyfrowe	544
Własny moduł	548
Budujemy analizator stanów logicznych	550
Sprzęt	551
Protokół komunikacji	553
Implementacja protokołu	555
Jak szybko próbkować?	560
Klient	564
Uwagi praktyczne	566

Rozdział 19. Wykorzystanie podczerwieni do transmisji danych	569
Modulacja IR	570
Porozmawiajmy z pilotem TV	572
Część sprzętowa, czyli odbiornik IR	572
Część programowa, czyli standardy kodowania	574
Standard NEC i pokrewne	575
RC5 i Motorola	581
Kod RC5	587
Inne standardy kodowania	592
Nadajnik IR	592
Inżynieria odwrotna — dekodujemy sygnał pilota aparatu Canon	595
Transmisja danych	597
Budujemy pakiet danych	600
Sprzętowy generator CRC	602
CRC liczone programowo	604
Transmisja pakietowa	605
Uniwersalny pilot	608
Interfejs IrDA i IRCOM	613
Skorowidz	615

Rozdział 10.

Przerwania i kontroler przerwania

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?

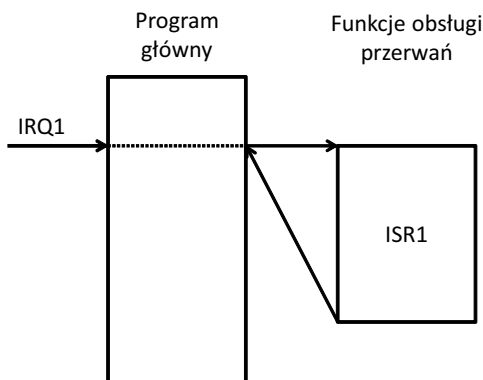
Na to pytanie najprościej odpowiedzieć, pokazując przykład z życia codziennego. Wyobraź sobie, że grasz w swoją ulubioną grę i nagle dzwoni telefon. W tym momencie istnieją dwie możliwości:

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 przerwania 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 wznowiany 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.



Wskazówka

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 asemlera SEI (w C jej odpowiednikiem jest `sei()`). Przerwania, które możemy blokować, nazywane są **przerwaniem maskowalnymi**.



Wskazówka

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 przerw.

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

Konfiguracja i obsługa przerw

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

Aby odblokować przerwy w XMEGA, musimy spełnić kilka warunków:

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

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

Czas odpowiedzi na żądanie przerwy

Pomiędzy zajściem zdarzenia wywołującego przerwy 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 przerwy 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 przerwy 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 przerwania, tzw. *handlers* (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 przerwania były wyróżnione atrybutem `interrupt`, wymagającym kilku parametrów, m.in. numeru wektora przerwania, 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_ACOSTATE_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 przerwania 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.



Wskazówka

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.

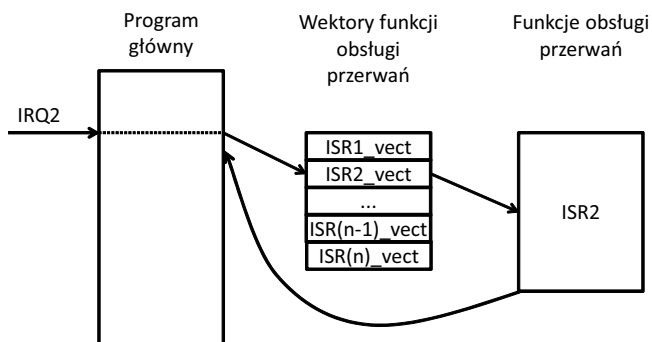
Jak widzimy, każda funkcja obsługi przerwania powiązana jest z tzw. wektorem przerwania. Każdy układ mogący generować przerwanie posiada przeznaczony wyłącznie dla niego wektor przerwania. Jeśli układ peryferyjny może generować kilka typów przerw, to zazwyczaj posiada oddzielne wektory dla każdego typu.

Wektory przerw

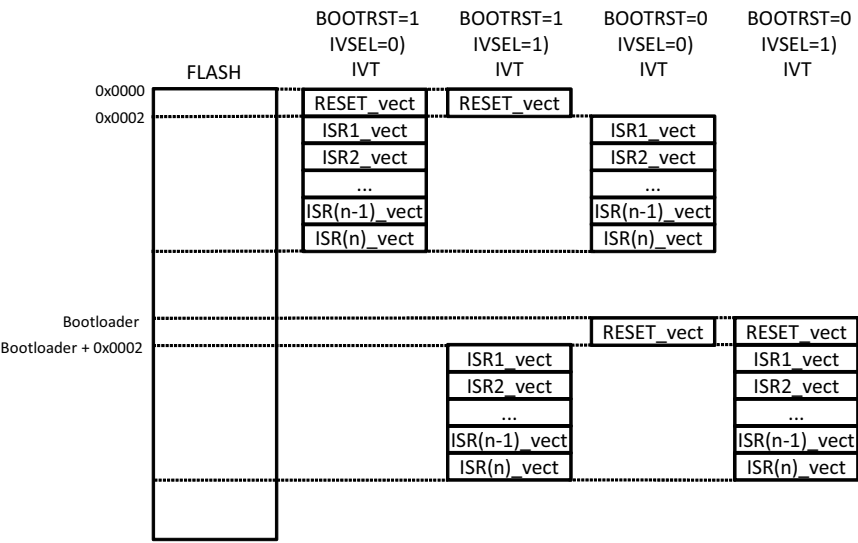
Układy peryferyjne posiadają przypisane im tzw. wektory obsługi przerw (ang. *ISR vectors*). Każdy podsystem posiada jeden lub więcej takich wektorów. Zgłoszenie przerwania powoduje wywołanie funkcji obsługi przerwania (ISR) powiązanej z jego wektorem (rysunek 10.2).

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 0x08 lub symbolicznie CCP_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 przerwania



Wskazówka

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 przerw. 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 przerw wskazywają 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 przerw 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 — `ret`, 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 asemblerze². 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 assemblerze:

```

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 assemblera; możemy ją umieścić w osobnym pliku z kodem assemblerowym (z rozszerzeniem `.S`). W tym celu wystarczy dodać do projektu nowy plik (wybieramy plik assemblera), 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

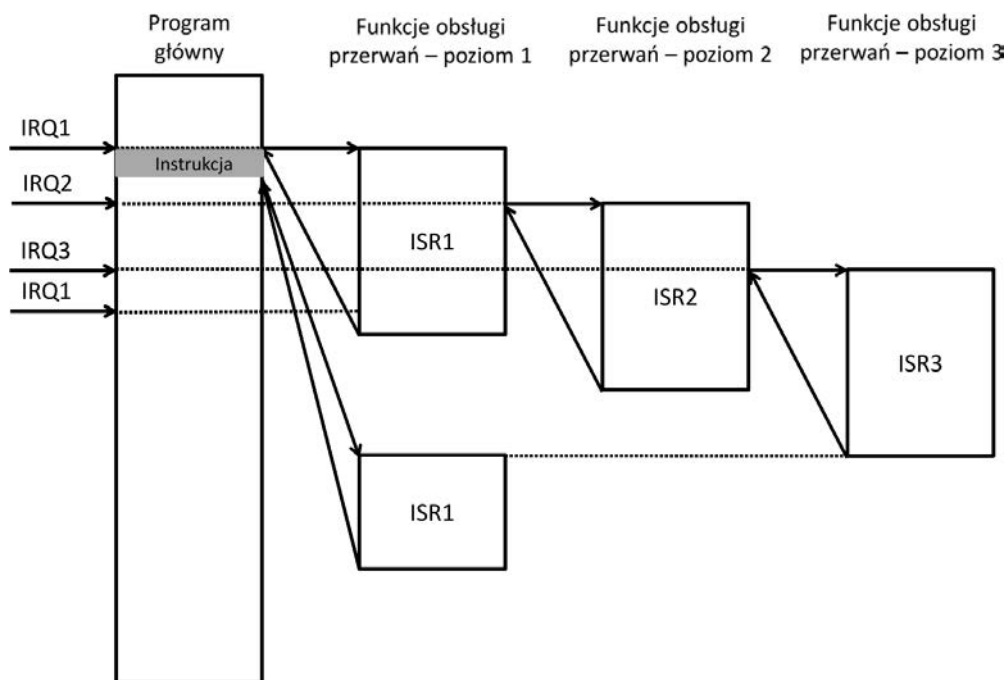
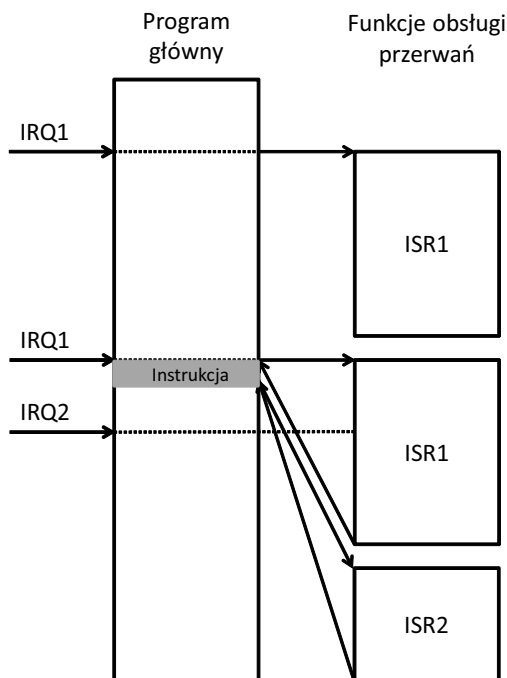
Poziom przerwania	Symbol	Opis
0	INTLVL_OFF_gc	Dane przerwanie jest wyłączone
1	INTLVL_LO_gc	Przerwanie o niskim priorytecie
2	INTLVL_MED_gc	Przerwanie o średnim priorytecie
3	INTLVL_HI_gc	Przerwanie o wysokim priorytecie

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 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 przerwaniem 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 obsługiwane — zależy to jeszcze od konfiguracji podsystemu PMIC. Z każdym przerwaniami 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.



Wskazówka

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 przerwaniami 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 obsługiwane, 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 obsługiwane 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 przerwania 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 przerwania danego poziomu.

Przerywanie przerwania

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 przerwania. 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 przerwaniem 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 przerwania wyższego poziomu (poziomu średniego lub wysokiego). Powyższa wstawka spowoduje odblokowanie możliwości przyjmowania kolejnych przerwania 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 przerwanie o priorytecie niższym — powodem będzie dwukrotne wykonanie rozkazu RETI — raz w naszej wstawce assemblerowej, 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.

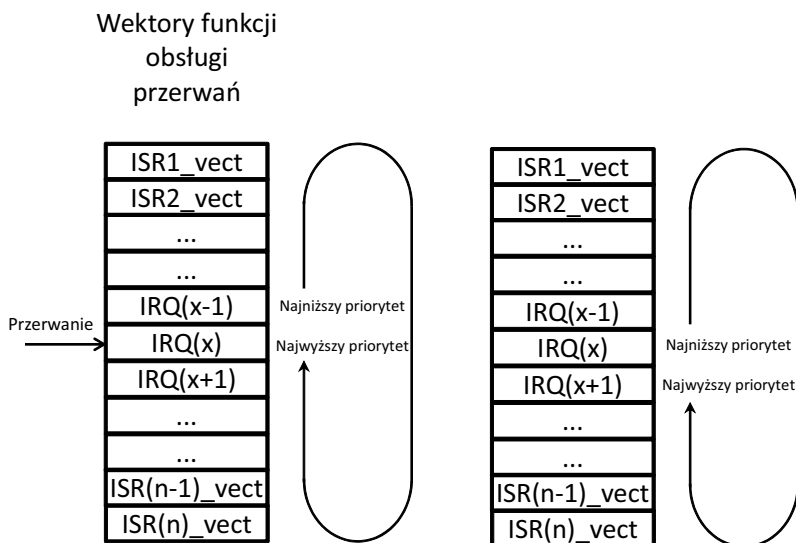
Z drugiej strony wielopoziomowy system przerwania dostępny w XMEGA powoduje, że stosowanie rozwiązań żywcem przeniesionych z innych rodzin AVR, gdzie przerwanie były jednopoziomowe (co zazwyczaj poprawiano, stosując makro NO_BLOCK), jest w większości przypadków niepotrzebne. Zanim więc zaczniesz stosować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

Priorytety przerwania

XMEGA dysponuje nie tylko poziomami, ale także tzw. priorytetami przerwania. Dla przerwania NMI oraz przerwania o poziomie wysokim i średnim zawsze używane są tzw. priorytety statyczne. To znaczy, że w sytuacji, w której jednocześnie zgłoszonych jest kilka przerwania (lub kilka przerwania oczekuje na swoją obsługę), najpierw obsługiwane jest przerwanie o niższym wektorze przerwania w tablicy wektorów przerwania. Dla przerwania o priorytecie niskim procesor może reagować w sposób bardziej elastyczny. Dla tego poziomu przerwania możemy wybrać priorytet statyczny, analogiczny do priorytetu statycznego przerwania 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 przerwania o priorytetach niższych. Aby uniknąć takiej sytuacji, wprowadzono priorytet dynamiczny. W tym schemacie ostatnio obsługiwane przerwanie otrzymuje najniższy priorytet, a przerwanie o kolejnym wektorze priorytet najwyższy. Po jego obsłudze 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. Zasada 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 obsługowanego przerwania. W efekcie przerwania o wektorach wyższych niż wektor zawarty w INTPRI mają priorytet wyższy, a przerwanie 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 przerwanie o wektorach wyższych niż wektor zawarty w tym rejestrze mają priorytet wyższy, a przerwanie 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 obsługowanego 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 przerwanie, znajdującą się w rejestrze stanu procesora (flaga I). Aby podsystem PMIC

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



Wskazówka

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

W programie praktycznie nigdy nie będziemy musieli zmieniać stanu tej flagi. Ponieważ procesor po resetie ma tę flagę wyzerowaną, praktycznie jedynym miejscem, gdzie jawnie będziemy ją zmieniać, jest początek programu wykorzystującego przerwy — po inicjalizacji poszczególnych podsystemów przerwy 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 przerwy inicjuje je w typowy sposób:

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

Przerwanie niemaskowalne

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

Rejestr stanu kontrolera przerw

Kontroler przerw ma własny rejestr stanu (`STATUS` — ang. *PMIC Status Register*). Zawiera on bity informacyjne o aktualnym stanie przerw, lecz w większości aplikacji prawdopodobnie płynące z tego informacje są zbędne. Ustawienie odpowiednich flag tego rejestru (`NMIEX`, `HILVLEX`, `MEDLVLEX` i `L0LVLEX`) sygnalizuje, że przerwy 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 przerw 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 „dziwaczego” zachowania programu.



Jeśli Twój program świetnie działa z wyłączoną optymalizacją (-O0), 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 jakiegokolwiek 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 tmp=x;
    tmp*=tmp;
    tmp<=1;
    x=tmp;
};
```

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, `tmp`, 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 przedstawiać 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 przerwanie, 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 przerwanie, 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 assembler 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 assemblera `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 blokowane³. 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 realizującą 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 semafor. 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 musiałby 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 *Przykłady/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.



Wskazówka

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

Niemniej jeśli próbujemy 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 zadziały 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 flags=(IO*)&GPIO1;
```

Zmienna `flags` 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*)&GPIO0;
```

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

- adres
 - 16-bitowy, 250
 - 24-bitowy, 251
- adresowanie struktur, 135
- AES, Advanced Encryption Standard, 16
- akcja, action, 470
- akumulatory, 464
- algorytm
 - eliminacji błędów, 582
 - round-robin, 302
 - zmiany wartości licznika, 413
- alokacja dynamiczna, 145
- alokator pamięci, 525
- alternatywa wykluczająca, exclusive or, 143
- analizator stanów logicznych, 550, 566
- ANSI C, 122
- ASCIIZ, 225
- assembler, 90
- ASF, Atmel software framework, 44
- Atmel Data Visualizer, 535
 - pakiety, 540
 - wykorzystywane struktury, 540
- Atmel Studio 6, 12, 44
 - biblioteki, 49
 - dodawanie pliku, 46
 - kompilacja, 52
 - nowy projekt, 45
 - opcje kompilacji, 48
 - opcje projektu, 47
 - szablony, 45
 - toolchain, 45
- atomowa modyfikacja stanu pinów, 196
- atrybut
 - aligned, 238
 - always_inline, 108, 313
 - const, 107

- deprecated, 113
- flatten, 109
- ISR_NAKED, 242, 296
- naked, 295
- noclone, 109
- noinline, 108
- nonnull, 109
- noreturn, 108
- optimize, 110
- progmem, 254, 249
- pure, 107
- automat stanów, 470
- automatyczne
 - dołączanie pliku, 165
 - eliminowanie funkcji, 111
- autouzupełnianie, 164
- AVR Studio, 44
- AVR Toolchain 8 Bit, 311
- awaria zasilania, 232

B

- batchisp, 26–32
 - device, 29
 - hardware, 29
 - operation, 29
 - operation blankcheck, 30
 - operation erase, 30
 - operation loadbuffer, 30
 - operation read, 30
 - port COM10, 32
- bateria CR2032, 463
- biblioteka
 - AVR-libc, 15, 165, 208
 - libc.a, 49
 - libm.a, 49
 - libobjc.a, 49
 - libprintf_flt.a, 50

biblioteka

- libprintf_min.a, 50
- libscanfflt.a, 50
- libscanf_min.a, 50

biblioteki zewnętrzne, 49

bit

- DMA_CH_REPEAT_bm, 561
- DREIF, 523
- DWEN, 42
- Enable, 445
- ERRIF, 331
- INVEN, 192, 594
- inwersji, 194
- IVSEL, 291
- JTAGEN, 34
- LEDBLINKMARKER, 415
- NVM_EEMAPEN_bm, 231
- QDEN, 323
- QDIRM, 331
- RREN, 303
- SRLEN, 193
- startu, 494
- stopu, 495
- TCn_DIR_bm, 357
- TCn_LUPD_bm, 357
- USART_MPCM_bm, 531
- VBAT_HIGHERS_bm, 466
- XOSCSEL, 466

bity konfiguracyjne, fusebity, 260

blok pamięci, 227, 245

blokowanie

- JTAG, 34
- pinu, 194
- ustawień zegara, 277

błąd

- deklarowania zmiennej, 209
- dereferencji wskaźnika, 254
- kompilacji, 165, 249
- kontrolera NVM, 206
- odczytu liczników, 321
- parzystości, Parity Error, 494, 508
- przekroczenia adresu, 254
- przepełnienia bufora, 511
- ramki, Frame Error, 495
- transferu danych, 386
- transmisji, 605
- transmisji DMA, 397
- zmiany wartości wskaźnika, 217

BOD, Brown-out Detector, 246

bootloader, 25, 27, 218

bootloader preprogramowany, 28

budowanie matryc, 421

bufor, 148, 354

- IC1A, 361
- IR_Buffer, 611
- pierścieniowy nadajnika, 521

bufory

- dwukierunkowe, 178
- pierścieniowe, 149
- scalone, 177
- typu open-drain, 173

BUFOVF, Buffer Overflow, 511

burst transfer, 380

C

CCP, Configuration Change Protection Register, 270

CMYK, 404

commit, 79, 81, 82

COMP, Counter Compare Register, 442

CRC, Cyclic redundancy check, 17, 600, 607

- generator sprzętowy, 602
- pakietów, 508
- wyliczanie programowe, 604

CTRL, Oscillator Control Register, 260

CTRL, System clock control register, 277

czas

- filtrowania sygnału, 327
- latencji, 286
- letni, 454
- odpowiedzi na żądanie przerwania, 288
- podtrzymania zasilania, 464
- propagacji zdarzenia, 321
- trwania półbitu, 590
- wykonania pętli, 100
- zimowy, 454

czasy narastania zboczy, 191

częstotliwość

- generowanych impulsów, 596
- modulacji, 596
- odświeżania matrycy, 431, 433
- oscylacji, 265
- pracy timera, 366
- próbkowania, 554, 561
- przebiegu PWM, 348
- przebiegu taktującego timer, 340
- przerwań, 412
- PWM, 362
- rezonatora, 268
- taktowania procesora, 48
- taktowania rdzenia procesora, 340

D

DAC, Digital to analog converter, 18
dane
 dla matrycy, 429
 dla zatrasku, 429
DCF77, 451
 dekodowanie danych, 454
 dekodowanie ramki, 457
 ramka, 453, 460
debuger, 36
debuger JTAG, 43
debugowanie, 41, 90
debugowanie własnych programów, 552
definicja
 EEMEM, 208
 F_CPU, 49
definiowanie zasobów, 157
dekoder
 kwadraturowy, 322
 sygnału kodowanego, 574
dekodowanie
 Manchester, 582
 RC5, 589
 struktury pakietu, 607
 sygnału pilota, 595
dereferencja wskaźnika, 231, 254
DES, Data Encryption Standard, 16
destruktor, 114
detekcja awarii zasilania, 235, 237
Device Monitoring Studio, 498
DFLL, Digital frequency locked loop, 272
dioda Schottky'ego, 235
diody
 na podczerwień, 595
 zabezpieczające, 169
DIR, Data Direction Register, 162
długość
 pola, 130
 rekordu, 225
DMA, Direct Memory Access, 16, 357, 371, 379–398, 517
dobór dzielnika, 235
dodatek Naggy, 85
dodawanie
 elementu listy, 137, 144
 plików, 46, 68, 82
 węzła, 141
dostęp do
 EEPROM, 206, 213, 224, 244
 FLASH, 246, 252
 instancji modułu, 156
 pamięci, *Patrz* DMA
 pliku, 84

 rejestrów, 163
 rejestrów RTC, 441
 rejestrów 16-bitowych, 360
 USART, 525
drżania styków, 330
drzewo projektu, 63
drżenie, jitter, 567
duty cycle, 417
dynamiczna alokacja pamięci, 123
dyrektywa
 #define, 59, 160
 #endif, 64
 #include, 64, 68
działanie
 modułu SRF05, 545
 preskalera, 505
 rejestru MPCMASK, 196
 układu zbierania próbek, 560
dziedziczenie, 122, 538
dziedziczenie struktur, 540
dzielnik rezystorowy, 169, 176, 235

E

EDMA, Enhanced DMA, 16
EEPROM, 206, *Patrz także* pamięć
 deklaracje danych, 208
 dostęp do zmiennych, 209
 funkcje dostępowe, 210
 liczba zapisów, 233
 mapowanie, 213
 rozdzielenie kasowania, 221
 tokeny, 224
 zapobieganie uszkodzeniu, 246
elementy listy, 136
emulacja RS232, 500
emulator portu szeregowego, 499
enkoder, 413, 415
 kwadraturowy z indeksem, 331
 obrotowy, 323
Epoch, 447

F

FIFO, First in, first out, 151
filtr
 cyfrowy, 322
 dolnoprzepustowy RC, 361
 RC, 367
firmware, 26
flaga
 BUFOVF, 511
 DMA_CH_CHBUSY_bm, 521
 EERIF, 398

- flaga
 - globalna, 303
 - I, 286, 304
 - nadmiaru, OVF, 299
 - NVM_NVMBUSY_bm, 231
 - RXCIF, 472
 - SYNCBUSY, 442
 - TimestampUpdate, 460
 - TxFlag, 515
 - USART_RXCIF_bm, 510
 - VBAT_BBORF_bm, 465
 - VBAT_BBORF_bm, 465
 - VBAT_BBPWR_bm, 465
 - VBAT_XOSCFAIL_bm, 466
- FLASH, 246, *Patrz także* pamięć
- funkcje_far, 251
- typy danych, 248
- FLIP, FLexible In-system Programmer, 26
- format
 - ASCII, 497
 - Epoch, 460
 - ramki RC5, 587
 - ramki USART, 502
 - transmisji danych, 494
- framework
 - .NET, 536
 - ASF, 44, 447, 500
- FSM, 471, 474
- funkcja
 - _crc_xmodem_update, 604
 - AC_init(), 241
 - ASM_ROL, 429
 - calendar_date_to_timestamp, 460, 483
 - calendar_timestamp_to_date, 449
 - cb_Send_Add, 522
 - ChgFunc_PrgBtn1Pressed, 486
 - ChgFunc_PrgSetDateBtn2Pressed, 488
 - ChgFunc_PrgSetTimeStart, 486
 - ChgFunc_TimeDateBtn1Pressed, 485
 - ChgFunc_TimeDateDispChg, 485
 - dead_code, 111
 - DMA_InitTransfer, 521
 - eeeprom_busy_wait(), 212
 - eeeprom_is_ready(), 212
 - eeeprom_update_byte, 483
 - EEPROM_write_int, 245
 - eeeprom_write_word, 220
 - EEPROMReg_AddRegEntry, 231
 - EEPROMReg_CreateRegEntry, 230
 - EEPROMReg_DelRegEntry, 232
 - EEPROMReg_GetFirstAvailPos(), 229
 - EEPROMReg_GetRegEntry, 232
 - EEPROMReg_GetRegItem, 229
 - EEPROMReg_MoveEEPROMBlock, 227
 - fdevopen(), 527
 - free, 126, 525
 - FSM_DispatchEvent, 481
 - get_char, 258
 - GetTimestamp(), 461
 - GetToken, 516
 - init, 115
 - main, 101
 - malloc, 145, 522
 - malloc_re, 525
 - memcpy, 215, 241
 - memset, 459
 - obl2, 106
 - obsługi przerwania, 289
 - pgm_read_byte, 410, 432, 510
 - pgm_read_word, 484
 - pop, 153
 - push, 153
 - rect, 103
 - rect2, 103
 - rect3, 104
 - reentrant, 525
 - rtc32_is_busy(), 446
 - Sample_init(), 561
 - SendPkt, 606
 - sprintf, 215
 - stack_init, 153
 - strcmp_P, 229
 - strcpy_P, 522
 - TranslateCommand(), 516
 - USART_init, 514
 - USART_send_block, 547
 - USART_send_buf_F, 522
 - usart_set_baudrate, 510
 - write_out, 203
 - write_out_struct, 203
- funkcje
 - _dealy_us, 597
 - _far, 251
 - analizatora, 553
 - biblioteki AVR-libc, 315, 316
 - dodatkowe licznika, 337
 - dodatkowe USART, 502
 - eliminujące drgania styków, 329
 - kanalu zdarzeń, 322
 - kopiujące, 125
 - liczników, 340
 - operujące na czasie, 450
 - operujące na EEPROM, 212
 - pinu, 199
 - reentrant, 314
 - rejestru, 334

G

generator
 32 kHz, 263
 frakcyjny, 505
 kwarcowy, 265
 kwarcowy 32 768 Hz, 269
 RC 2 MHz, 264
 RC 32 768 kHz, 275
 RC 32 MHz, 264
 RTC, 283
 sprzętowy CRC, 602
 sygnału zegarowego, 445
generatory
 RC, 273
 wzorcowe, 273
 zegarowe zewnętrzne, 265
generowanie
 dowolnego przebiegu, 364
 napięcia o zmiennej amplitudzie, 363
 nośnej, 612
 programowe zdarzeń, 333
 przebiegu PWM, 348, 434
 przerwań, 330
 sinusoidy, 368
 sygnałów analogowych, 361
 zmodulowanego przebiegu, 594
głębia koloru, 405
główna gałąź projektu, 85

H

handlery, 289
historia zmian, 84

I

IDE, Integrated development environment, 44
implementacja protokołu SUMP, 555
inicjacja DMA, 610
inicjalizacja
 interfejsu USART, 509
 pól struktury, 128
 timera, 370
 transferu DMA, 393
 zmiennych, 99
instalacja
 serwera VisualSVN, 76
 XMEGA, 25
 Xplained, 22
instrukcja
 CLI, 286
 cli(), 304

LAC, 311
LAS, 311
LAT, 311
LDS, 197
lpm, 254
MOVW Rd,Rr, 310
NOP, 185
RETI, 301, 304
sei(), 304
STD, 133
STS, 133, 197
XCH, 311
instrukcje
 asemblera, 197, 200, 311
 opóźniające, 185
interfejs
 debugWIRE, 41
 IRCOM, 613
 IrDA, 613
 ISP, 43
 JTAG, 32, 43
 pamięci zewnętrznej, 19
 PDI, 34, 43
 RS232, 29, 497
 RS232-TTL, 492
 SPI, 419, 427
 UART, 43
 USART, 427, 430, 491–533
interfejsy
 IO, 19
 szeregowe, 491
interpretacja danych, 607
inwersja, 192
inżynieria odwrotna, reverse engineering, 498, 595
IRCOM, IR Communication Module, 614
IrDA, Infrared Data Association, 19, 613
ISC, Input/Sense Configuration, 193
ISR, Interrupt handler/service routine, 286
IVT, Interrupt vector table, 291

J

język C, 15
język UML, 469

K

kable
 proste, straight cable, 493
 skrosowane, crossed cable, 493
kalibracja
 generatora, 275
 układu RTC, 438

- kanal
 - CHnCTRL, 322
 - DMA, 372, 520
 - kontrolera DMA, 158
 - zdarzeń, 320
- kasowanie
 - bajtów strony, 223
 - klucza, 232
 - pamięci, 221
 - rekordu, 232
- kierunek
 - pinu, 182
 - zliczania, 357
- klient LogicSniffer, 551, 564
- klucz, 232
- kod
 - assemblerowy, 90, 132, 134
 - Graya, 329
 - Manchester, 570, 581
 - martwy, 111
 - Motorola, 582, 584
 - RC5, 587
 - reentrant, 314
- kodowanie
 - bifazowe, 570, 574
 - bitów w NEC, 575
 - koloru, 435
 - odległości impulsu, 570
 - szerokości impulsu, 570
- kody przycisków, 580
- kolejki
 - FIFO, 151
 - LIFO, 151
- kolejność
 - danych, 386
 - instrukcji, 307
- komendy NVM, 222
- komentarze, 55
- komparator analogowy, 19
- kompilacja, 52
- kompilator
 - avr-gcc, 12, 15
 - avr-gcc 4.7, 252
 - gcc, 15, 101
- komunikacja
 - master – slave, 601
 - w podczerwieni, 569
 - z klientem, 553
- kondensator odsprężający, 44, 282
- konfiguracja
 - AVR Dragon, 33
 - bus keeper, 188
 - DMA, 371, 426, 523
 - interfejsu USART, 500
 - kanalu, 396
 - kanalu DMA, 385
 - kanalu zdarzeń, 345
 - klienta LogicSniffer, 564
 - pinu IO, 193
 - portów mikrokontrolera, 181
 - preskalerów, 278, 281
 - przerwań, 288
 - Pull up/down, 187
 - serwera SVN, 80
 - timera, 347, 427, 546, 612
 - totem-pole, 186
 - USART, 547
 - wired-AND, 189
 - wired-OR, 191
- konfigurowanie
 - systemu zdarzeń, 332
 - typu transferu, 394
 - wyzwalaczy, 558, 565
 - zegara, 262
- konstruktor, 114
- kontrola przesyłanych informacji, 600
- kontroler
 - DMA, 379–398
 - EDMA, 341
 - NVM, 205, 213
 - przerwań, 17, 285, 304
- konwersja
 - czasu, 447
 - daty, 450
 - markera czasowego, 449
 - napięcia, 171, 174, 177
 - poziomów logicznych, 172
 - poziomów napięć, 180
- konwertery dwukierunkowe, 174
- kopiowanie
 - bloków pamięci, 227
 - elementów tablicy, 388
 - głębokie, deep copy, 125
 - płytkie, shallow copy, 124
 - struktury, 125
 - tablicy, 390

L

- LED-y, 399
- liczba
 - elementów listy, 146
 - elementów w buforze, 150
 - kanałów zdarzeń, 320
- liczby magiczne, magic numbers, 59
- licznik, 20, 337
 - CNT, 456
 - dekodera kwadraturowego, 327

- enkodera, 329
- LED, 412
- sekund, 448, 458
- Timestamp, 547
- typu 0/1, 341
- typu 2, 351, 377
- LIFO, Last in, first out, 151
- linia LA, 424
- linuksowy marker czasowy, 447
- lista sll, 140
- listy, 135
 - dodawanie elementu, 137
 - dwukierunkowe, 140
 - jednokierunkowe, 136
 - usuwanie elementu, 138
 - wielowymiarowe, 147
 - XOR, 142
 - zwracanie elementu, 139, 146
- literały, 59
- literały złożone, 255
- lokalna kopia
 - projektu, 81
 - repozytorium, 82

Ł

- łączenie
 - liczników, 358
 - układów I2C, 179

M

- magistrala I2C, 178
- magistrale typu open drain, 178
- majority voting, 505
- makrodefinicje
 - ATOMIC_BLOCK, 198, 309, 460
 - fdev_setup_stream, 527
 - ISR, 289, 300
 - pgm_get_far_address, 251
 - pgm_read_byte, 250, 253
 - PGM_STR, 256
 - PROGMEM, 248
 - PSTR, 255
 - strażnik, 64
- maksymalna długość kabla, 494
- mapowanie EEPROM, 213
- marker czasowy zdarzenia, 450, 577
- maska bitowa, 163, 201
- maszyna
 - stanów, 470–472, 478, 490, 576
 - wirtualna Java, 551

- matryce
 - dwukolorowe, 423
 - LED, 416–436
- MD5 pakietu, 540
- menedżer serwera SVN, 77
- metoda
 - get, 528
 - put, 528
- metody kodowania danych, 571
- mierzenie odległości, 548
- mikrokontroler
 - XMEGA128A1, 608
 - XMEGA256A3BU, 477, 608
- mikrokontrolery XMEGA, 11–16, 167
- model OSI, 614
- modulacja
 - fali świetlnej, 569
 - IR, 570
 - sygnału, 594
- moduł
 - calendar, 448
 - CRC, 17, 604
 - DMA, 159
 - HiRes, 376
 - IRCOM, 614
 - kryptograficzny, 16
 - MOD-11.Z Xmega eXplore, 25
 - odbiornika DCF77, 454
 - przechwytyjący, Capture unit, 456
 - SRF05, 544
 - SRF-05, 549
 - USART, 160
 - Xplained, 240, 537
- moduły projektu, 61
- modyfikacja
 - danych, 212
 - rejestrów kontrolnego, 232
- modyfikator
 - const, 98, 218, 247
 - extern, 66
 - register, 94
 - static, 66, 73, 66, 73
 - volatile, 127, 185, 202, 238, 305
- modyfikowanie stanu pinów, 196
- monitorowanie
 - wymiany danych, 498
 - zegara zewnętrznego, 275
- MPCM, Multi-processor Communication Mode, 530

N

- nadajnik IR, 592
- najmłodszy bajt, 130
- najstarszy bajt, 130

napięcie
 na pinie IO, 170
 tętnień, ripple voltage, 236
 zasilania matrycy, 422
 zasilania procesora, 171
 nazewnictwo, 57
 nazwy
 bitów, 162
 modułów, 161
 pól bitowych, 164
 rejestrów, 162
 NMI, Non-maskable interrupts, 304
 NVM, non-volatile memory, 205

O

obliczanie odległości, 547
 obsługa
 dziedziczenia, 539
 EEPROM, 208
 enkodera obrotowego, 328
 enkoderów, 17
 pamięci, 205
 pamięci dynamicznych, 19
 paneli pojemnościowych, 17
 przerwania DMA, 524
 przerwania timera, 459, 526
 przerwań, 17, 208, 241, 288, 514, 532, 562
 przerwań w assemblerze, 295
 przycisku, 597
 RTC32, 481
 stosu, 152
 wyświetlania, 414
 zdarzeń, 18
 ochrona
 pamięci, 246
 rejestrów IO, 270
 odbiornik
 DCF77, 451
 IR, 572
 podczerwieni TFMS1380, 596
 ultradźwięków, 549
 odbiór
 bitów, 586
 danych, 603
 zakodowanych sygnałów, 599
 odblokowywanie
 kanału DMA, 611
 źródła zegara, 270
 odcienie kolorów, 433
 odczyt
 stanu licznika, 327
 pamięci EEPROM, 210, 214
 pamięci FLASH, 250, 522

odgałęzienie, branch, 78
 odmierzanie czasu, 416, 451
 odrzucanie ramek, 605
 odsprzęganie zasilania, 44
 odwracanie
 danych, 384
 wyjść IO, 192
 zmian, 83
 opcja -fms-extensions, 539
 opcje
 kompilacji, 48
 programu batchisp, 28
 operacje
 bitowe, 163, 313
 logiczne, 163
 operator
 pobierania adresu, &, 251
 referencji, 121
 sizeof, 606
 wyboru pola, 160
 opóźnienie, 185
 optymalizacja, 48, 90
 funkcji main, 101
 kodu, 89
 pętli, 100
 programu, 131
 prologów i epilogów, 101
 przekazywania parametrów, 102
 zwracania wyników, 105
 organizacja adresu, 216
 oscylator, 262
 OSI, Open System Interconnection, 614
 otwieranie strumienia, 529

P

pakiet
 ADV_PKT_CONFIG_END, 540
 ADV_PKT_CONFIG_FIELD, 540
 ADV_PKT_CONFIG_START, 540
 ADV_PKT_CONFIG_STREAM, 540
 ADV_PKT_DATA, 540
 pakiety danych, 601
 pamięci zewnętrzne, 233
 pamięć
 EEPROM, 206
 FLASH, 246
 FRAM, 233
 SDRAM, 51
 SRAM, 209
 parametr
 -fms-extensions, 122
 -mcall-prologues, 101
 -mrelax, 110

- mshort-calls, 110
- mstrict-X, 110, 135
- ATOMIC_FORCEON, 309
- ATOMIC_RESTORESTATE, 309
- ISR_BLOCK, 300
- ISR_NOBLOCK, 300
- minimum burst length, 597
- NONATOMIC_FORCEOFF, 309
- regname, 226
- parametry próbkowania, 565
- PER, Counter Period Register, 442
- PER, Period register, 457
- pętla, 100
 - nieskończona, 489
 - sprzężenia, 264
- pilot
 - RC-1, 595
 - TV, 572
 - uniwersalny, 608
- pin
 - IO, 186
 - P1, 455
 - RxD, 614
 - VBAT, 462, 467
- piny
 - aktywujące bootloader, 28
 - wejściowe, 168, 182
 - wyjściowe, 168, 182
 - wyjściowe licznika, 350
- planista, scheduler, 526
- plik
 - atomic.h, 309
 - Atxmega128a1.xml, 27
 - avr_cdc.inf, 23
 - bufusart.c, 513
 - calendar.c, 447
 - calendar.h, 447
 - delay.h, 455
 - eeeprom.h, 205, 208
 - EEPROMToken.c, 227
 - EEPROMToken.h, 226
 - elf, 217
 - flash.hex, 31
 - io.h, 159, 165
 - Main.c, 65
 - pgmspace.h, 205, 248, 525
 - Pierwszy_Projekt.hex, 32
- pliki
 - .eep, 217
 - elf, 12, 41, 53
 - Intel HEX, 12
 - Makefile, 12, 62
 - nagłówkowe, 63–65
 - produkcyjne, Production files, 54
 - źródłowe, 68
- plywająca bramka, 206
- PMIC, 288
- podczerwień, 569
- podgląd transmisji danych, 497
- podkatalogi, 67
- podłączanie
 - enkodera, 325
 - kwarcu zegarkowego, 440
 - odbiornika IR, 573
 - taśmy LED, 401
- podsystem PMIC, 288
- podtrzymanie zasilania, 235, 465
- podtrzymanie zasilania układu RTC, 467
- podwójne buforowanie, Double buffering, 395
- podział
 - frakcyjny, 506
 - funkcji, 69
 - projektu, 62–65
- pojemność
 - bramki, 402
 - pasożytnicza, 175
 - wejścia XTAL, 266
- pola
 - bitowe, 130
 - struktury, 128
- pole
 - ISC, 193, 195
 - Len, 225
 - OPC, 195
- polecenie
 - FLASH, 517
 - FLASH start end, 513
 - Set Divider, 554
 - Set Read & Delay Count, 554
 - Set Trigger Mask, 554
 - svnadmin dump, 78
 - svnadmin hotcopy, 78
- połączenie, merge, 85
 - diod LED, 391
 - LCD z mikrokontrolerem, 326
 - matrycy ze sterownikami, 424
 - mikrokontrolera, 493
 - mikrokontrolera z matrycą, 424
- pomiar
 - odległości, 543
 - okresu, 344, 354
 - szerokości impulsu, 344, 354, 456, 546
- pooling, 509, 516, 556
- porównanie CCA z CNT, 373
- port
 - COM3, 537
 - REMAP, 204
 - szeregowy USARTC0, 579

- porty
 - IO, 20, 157, 167
 - wirtualne, 200
- postinkrementacja, 100
- poziom
 - optymalizacji, 90
 - przerwania INTCTRL, 299
- poziomy
 - napięć wejściowych, 172
 - przerwań, 297, 298
- półduplex, half-duplex, 598
- prawo Ohma, 169
- prąd
 - samorozladowania, 464
 - szczytowy diod, 417
- predekrementacja, 100
- preskaler, 338
- preskalery zegara, 278, 440
- priorytet
 - dynamiczny, 303
 - statyczny, 302
- priorytety
 - kanałów DMA, 396
 - przerwań, 302
- procesor
 - XMEGA, 25, 43, 171, 277
 - XMEGA 128A1, 22
- program
 - Atmel Data Visualizer, 536
 - AVRDude, 26
 - batchisp, 26–32
 - Device Monitoring Studio, 498
 - Doxygen, 57
 - FLIP, 24, 26
 - kalendarzowy, 446
 - Naggy, 86
 - objcopy, 53
 - RealTerm, 497, 532
 - Subversion, 74
- programator
 - AVR Dragon, 33, 36–41
 - AVRISP mkII, 35
 - AVROne!, 36
 - JTAG, 43
 - JTAGICE mkII, 35
 - JTAGICEIII, 36
 - USBasp, 26
- programatory do XMEGA, 34
- programowanie, 86
- programowanie układów XMEGA, 25
- protokoły
 - komunikacji, 553
 - transmisji, 540
- protokół
 - HTTPS, 76
 - RC5, 597
 - SUMP, 553
- próbkowanie, 567
- próg zadziałania BOD, 246
- przebieg sygnału prostokątnego, 178
- przechwytywanie
 - sygnału pilota, 610
 - zdarzeń, 345, 348
- przejście, transition, 470
- przejściówka USB-RS232, 499
- przekazywanie parametrów, 102
 - przez referencję, 103
 - przez wartość, 103
- przekazywanie rejestru IO, 202
- przeladowywanie rejestrów indeksowych, 133
- przenoszenie kodu, 55
- przepełnienie
 - bufora, 148, 511
 - licznika, 341, 457
 - licznika timera, 611
- przerwania, 195, 244, 285, 369, 512
 - DMA, 397, 524
 - maskowalne, 286, 304
 - niemaskowalne, 287
 - o wysokim poziomie, 297
 - RTC, 483
 - timera, 526
- przerwanie
 - końca transakcji, 398
 - odbioru transmisji, 591
 - przepełnienia licznika, 444
 - RTC_OVF_vect, 443
 - USARTC0_TXC_vect, 428
 - USARTxx_DRE_vect, 512
- przerywanie przerwań, 300
- przestrzeń barw, 404
- przestrzeń adresowa
 - __flash, 254, 256
 - __memx, 257
- przesunięcie, offset, 156, 218
- przesunięcie czasowe, 449
- przesuwający się napis, 433
- przesyłanie
 - bloków pamięci, 385
 - danych, 251
 - pamięć-pamięć, 383
 - pamięć-rejestr IO, 389
- przetwornik analogowo-cyfrowy, 19
- przygotowywanie pakietu, 607
- przypisanie kanałów timera, 353
- przywracanie stanu timera, 587
- puste wektory przerwań, 292
- PWM, 361, 368, 433

Q

QDEN, Quadrature Decode Enable, 323

R

ramka

DCF77, 453, 457

RC5, 588

UART, 494

ramki

adresu, 530

danych, 453, 530, 588

referencje, 112

regulacja

intensywności kolorów, 406

jasności, 425

prądu diod, 423

rejestr

ACOMUXCTRL, 162

ADDRCTRL, 381

BACKUP0, 467

BAUDCTRLB, 431, 496

CCA, 547

CCABUF, 372

CCn, 456

CCP, 270

CCxBUF, 354

CH0CTRL, 327

CHECKSUM, 603

CHnMUX, 320

CLKEVOUT, 334

CLR, 199

CNT, 338, 341, 405, 446

COMP, 441, 445

CTRL, 260, 277

CTRLA, 339, 561

CTRLB, 231, 348, 531

CTRLF, 357

CTRLFCLR, 357

CTRLFSET, 357

DATA, 332, 510, 512

DATAIN, 603

DESTADDR0, 380

DIR, 182

IN, 184

INTCTRL, 163

INTPRI, 303

MPCMASK, 195, 326

OUT, 157, 182

PER, 331, 342, 356, 441, 457, 562

PERBUF, 354

PERH, 378

PERL, 378

PINnCTRL, 192, 194

PMIC_CTRL, 291, 292

PORTE_OUT, 198, 203

PORTE_OUTTGL, 198, 203

REMAP, 203

REPCNT, 381, 398, 431

RTCCTRL, 439

SET, 199

SRCADDR, 525

SRCADDR0, 380

STATUS, 231, 261, 304

STROBE, 332

TEMP, 359

TGL, 199

TRFCNT, 158, 372, 380

VBAT.CTRL, 466

X, 135

XOSCCTRL, 267

rejestry

buforowe, 354, 356

danych, 205

GPIO, 97

IO ogólnego przeznaczenia, 316

konfiguracji przerwań, 205

kontrolne, 205

kontrolne portu, 194

kontrolne zegarów, 260

licznika, 341

multiplexera kanału zdarzeń, 320

przechowujące zmienne programu, 99

stanu, 205

rekord DCF77, 454

remapowanie wyjść

IO, 203

timerów, 350

repozytorium, 77

resetowanie procesora, 37, 246

rezystor podciągający, Pull up, 173

RGB, 403

RMW, read-modif-write, 196

rozdzielczość przebiegu PWM, 374

rozdzielenie kasowania pamięci, 221

rozgałęzianie, branch, 85

rozkład sygnałów

interfejsów ISP, 43

interfejsów PDI, 43

złącza JTAG, 37

rozkład wyprowadzeń

interfejsu HV, 39

interfejsu ISP, 39

interfejsu JTAG, 39

rozmieszczanie kwarcu, 267

rozszerzenie HiRes, 375

RREN, Round-robin Scheduling Enable, 303

RTC, Real-time counter, 17, 437, 462

rzutowanie, 218, 241

S

- schemat podłączenia przycisków, 183
- schemat układu RTC, 439
- SDRAM, 19
- segmenty wyświetlacza, 409
- serwer SVN, 74–84
 - opcja Add to Subversion, 79
 - opcja Commit Project Changes, 82
 - opcja Remove, 82
 - opcja Revert, 83
 - opcja Update Project to Latest Version, 81
 - opcja View History, 84
- sesja debugera, 243
- skaler, 241
- skasowanie bootloadera, 33, 40
- skończona maszyna stanów, 469
- skrypt Makefile, 62
- słowo kluczowe
 - static, 66, 91
 - struct, 120
 - typedef, 239, 248
 - volatile, 305
- sprawdzanie
 - CRC, 17, 600, 607
 - ramek, 460
- stała
 - czasowa, 234
 - OLS_SAMPLE_BUF, 561
- stan, state, 470
 - Bat_FirstTime, 482
 - Bat_Normal, 482
 - IR_NEC_FirstBit, 579
 - IR_NEC_Trailer, 579
 - OLS_St_Cmd_Received, 556
 - OLS_St_Rec5byteCmd, 556
 - OLS_St_Waiting, 556
- standard NEC, 575
- sterowanie
 - jasnością LED, 434
 - matrycą LED, 418, 425, 430
 - multiplexowe wyświetlaczem, 408
 - taśmą LED, 404
 - taśmą RGB, 406
 - układami zewnętrznymi, 333
 - zdarzeniami, 332
- sterownik
 - kolumn, 420
 - matrycy, 421
- stopień podziału zegara, 281
- stos, 101, 152
- strona pamięci EEPROM, 216
- struktura, 119, 126, 131
 - bitfield, 130
 - bufora pierścieniowego, 149
 - calendar_date, 448
 - ciezarówka, 121
 - clendar_date, 450
 - DaneSRAM, 243
 - DCF77Record, 461
 - FILE, 528
 - kolo, 128
 - listy dwukierunkowej, 140
 - lit, 256
 - matrycy LED, 417
 - node, 147
 - PORT_struct, 156
 - PORT_t, 157, 181
 - projektu, 69
 - prostokat, 539
 - punkt, 124, 538
 - RTC_t, 162
 - usartout, 529
- struktury anonimowe, 127
- strumienie, streams, 527, 537
- strumienie języka C, 530
- strumień usartout, 530
- sufiks
 - _flash, 253
 - _bm, 163
 - _bp, 164
 - _far, 251
 - _gc, 162
 - _gm, 163
 - _gp, 164
 - _P, 254
 - BUF, 354
 - CLR, 197
 - H, 162
 - L, 162
 - SET, 197
 - TGL, 197
- suma kontrolna pakietów, 508
- superkondensatory, 462
- sygnał
 - DCF77, 452
 - LA, 427
 - pilota, 610
- symbol ^, 143
- symulator debugera, 88
- synchronizator, 184
- synchronizowanie
 - odczytu rejestrów, 441
 - wersji projektu, 81
- system zdarzeń, Event system, 319
- szerokość impulsu, 343, 376
- szerokość impulsu PWM, 373

szybkość

- interfejsu USART, 503
- narastania zbocza, 173, 193
- próbkowania sygnału, 559
- transmisji, 494, 506, 597
- transmisji SPI, 431, 435
- USART, 503

szyfrowanie

- 3DES, 16
- AES, 16
- DES, 16

Ś

śledzenie transmisji RS232, 499

środowisko programistyczne, IDE, 164

T

tablica, 118, 474

- Bufor, 395
- cmd, 557
- dst, 384
- filltbl, 389
- przejąć, 324
- src, 384
- Stack, 153
- TransTable, 475
- wave, 367, 372
- wektorów przerwań, 291

taśma

- LED, 399
- RGB, 406

technika

- RMW, 196
- wear leveling, 219, 233

terminal, 496

terminal RealTerm, 497

testowanie komunikacji IR, 599

timer, 337

- TCC0, 289, 526
- XMEGA, 339

timery

- tryb generowania PWM, 346
- tryb przechwytywania, 346
- typu 4/5, 353

tłumienie filtra dolnoprzepustowego, 363

tokeny, 224

toolchain, 45

transakcja, 379

transceiver, 492

transfer DMA, 374

translacja poziomów napięć, 174

transmisja

- full-duplex, 493
- IR, 570
- pakietowa, 605
- RC5, 588
- RS232, 495
- SPI, 431

transmisje danych, 497

- DMA, 517
- jednoczesne wysyłanie danych, 526
- pooling, 509
- przerwania, 512
- w podczerwieni, 597

tranzystor

- MOS, 206
- MOSFET, 402
- N-MOSFET, 179

trawersowanie, 144

tryb

- DMA, 383
- HiRes, 377
- IRCOM, 614
- MPCM, 530, 531
- PDI, 34
- PWM, 346, 349
- single slope, 355
- uśpienia RTC, 445

tryby pracy licznika

- bajtowy, 351
- dzielony, 351
- dual slope, 348
- generowanie przebiegu PWM, 348
- podstawowy, 342
- pomiar okresu, 343
- single slope, 348
- szerokość impulsu, 343

twierdzenie Kotelnikowa-Shannona, 551

tworzenie

- kopii katalogów, 78
- lokalnej kopii projektu, 81
- nazw pól bitowych, 164
- pakietu danych, 600
- protokołu komunikacji, 598
- konfiguracja pola danych, 542
- konfiguracja strumienia, 541
- koniec konfiguracji, 543
- pakiet danych, 543
- początek konfiguracji, 541

typ

- prog_char, 248
- uint8_t, 202
- wyliczeniowy
 - AC_MUXPOS_enum, 162
 - RTC_COMPINTLVL_t, 163

typ

- wyliczeniowy
 - RTC_OVFINTLVL_t, 163
 - RTC_PRESCALER_t, 162

typy

- 16-bitowe, 92
- 24-bitowe, 258
- 8-bitowe, 91
- modułów, 161
- przerwań, 512
- transferu, 394
- złożone, 117
- zmiennopozycyjne, 51

U

uaktualnienie projektu, 82

układ

- 74LVC08, 177
- ADC, 19
- AVR Dragon, 25
- AWeX, 338
- BOD, 242, 246
- DAC, 18
- DFLL, 272
- DMA, 319, 562
- MAX232, 492, 549
- monitorowania napięcia, 465
- monitorujący, 275
- PLL, 259, 270
- podtrzymywania zasilania, 462
- RTC, 17, 437, 462
 - 16-bitowy, 438
 - 32-bitowy, 445
- RTC32, 482
- SCT2024, 422, 425
- SPI, 431
- TFMS1380, 572
- USART, 259
- Watchdog, 242
- XMEGA128A1, 531
- zapasowego zasilania, 465

układy

- nadajników IR, 593
- peryferyjne, 155
- ultradźwiękowy pomiar odległości, 543
- UML, Unified Modeling Language, 469

unie, 129

urządzenie

- master, 533, 600
- slave, 599

USART, 19, 491–533, 602

- funkcje dodatkowe, 502
- konfiguracja formatu ramki danych, 502

- konfiguracja pinów IO, 501
- kontrola poprawności danych, 508
- pinów IO, 492
- strumienie, streams, 527
- szybkość transmisji, 496
- transmisja danych, 509
- tryb IRCOM, 614
- tryb pracy MPCM, 530
- ustawianie szybkości, 503

ustawienia zegara, 277

usuwanie

- elementów listy, 138, 145
- końca, 111
- nieużywanych funkcji, 114
- plików, 82

uszkodzenie

- pamięci, 246
- procesora, 468

W

wartość

- klucza, 226
- NULL, 149
- preskalera, 441

wczytywanie firmware, 26

wektor

- OSCF_INT_vect, 304
- RESET, 31, 291
- TCC0_CCA_vect, 297
- TCC0_CCB_vect, 294
- TCC1_CCA_vect, 456
- TCx_OVF_vect, 457
- TCxn_CCy_vect, 356

wektory obsługi przerwań, 290

wersjonowanie plików, 74, 78, 83

węzeł listy dwukierunkowej, 141

widoczność

- funkcji, 66
- zmiennych, 66

wielkość strony, 206

wielobajtowe rejestry IO, 314

wielomian CRC, 602

wirtualny port szeregowy, 24, 499

wizualizacja danych, 535

wskaźnik, 123, 126

difptr, 143

do bufora pierścieniowego, 607

do pamięci, 607

na typ char, 254

na wskaźnik, 138

ptr, 254

współczynnik odwzorowania barw, CRI, 403

wtyczka AnkhSVN, 75, 79

wybór źródła zegara, 271
 wyciek pamięci, memory leak, 522
 wydajność prądowa pinów IO, 170
 wyłączanie modulacji, 613
 wypełnianie pamięci wzorcem, 388
 wyświetlacze LED, 407
 wyświetlanie
 cyfr, 409, 413, 480
 daty, 487
 wywołanie funkcji, 105
 wyzwalacze, Triggers, 392
 wyzwalacze równoległe, 565
 wyzwalanie transferu DMA, 394
 wzory płytek PCB, 20

X

XMEGA, 602
 XOSCCTRL, XOSC Control Register, 267

Z

zakłócenia sygnału, 364
 zapis
 czasu, 447
 daty, 447
 do EEPROM, 210, 211, 244
 do rejestru CCx, 356
 markera czasowego, 347
 strony pamięci EEPROM, 222
 zarządzanie
 mocą, 18
 projektem, 55
 zegarem, 18
 zasada projektowania PCB, 267
 zatrzymanie DMA, 611
 zawartość
 EEPROM, 243
 pamięci mikrokontrolera, 518
 zdarzenie, 321
 Action_Btn1Pressed, 488
 Action_Start, 485
 Action_TimerTick, 485
 DMA_CH_TRIGSRC_TCC0_OVF_gc, 372
 generowane programowo, 332
 overflow, 429
 update, 348

zegar, 259
 CLKper, 279, 322, 339, 375
 RTC, 162
 z alarmem, 477
 z budzikiem, 469
 zestawy Xplained, 22
 zewnętrzne źródło zegara, 268
 zgłoszenie przerwania, 290
 zjawisko
 under-shoot, 193
 zatrzaśnięcia, latch up, 552
 złożenie przebiegów, 204
 zmiana
 aktualnego czasu, 479
 danych w EEPROM, 210, 212
 routingu pinu RxD, 614
 stanu, 484
 zmienna
 cmdrec, 516
 date, 450
 halfbittime, 586
 pEEPROM, 219
 timestamp, 448, 451, 460
 zmienne
 globalne, 67, 70, 99
 GPOR, 317
 lokalne, 93
 wielobajtowe, 314
 znacznik końca transmisji, 575
 zwalnianie pamięci, 125
 zwiększanie licznika sekund, 444
 zwracanie wyników funkcji, 106

Ż

źródło
 danych, 603
 przerwań, 287
 zasilania awaryjnego, 463
 zdarzeń, Event source, 319
 zegara, 263, 268, 277, 338, 439

Ż

żądanie przerwania, 286

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA WYDAWNICZA

 **Helion SA**

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!**

helion.pl
księgarnia
internetowa

Nr katalogowy: 14452



Księgarnia internetowa:

<http://helion.pl>



Zamówienia telefoniczne:

0 801 339900



0 601 339900



Helion

Sprawdź najnowsze promocje:

• <http://helion.pl/promocje>

Książki najchętniej czytane:

• <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

• <http://helion.pl/nowości>

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

sięgnij po **WIĘCEJ**



KOD KORZYŚCI

ISBN 978-83-246-7877-8



Cena: 99,00 zł

Informatyka w najlepszym wydaniu