Wokół języka C

Rejestry I/O

W poprzednich numerach EdW porównywaliśmy działanie mikroprocesora rodziny ATmega do biura, w którym pracuje słabo rozgarnięty, ale bardzo skrupulatny i pracowity urzędnik. Po pierwsze chodziło o wskazanie, że mikroprocesor „sam z siebie umie niewiele”, a cała „mądrość” to dobrze napisany program. Po drugie, tamten materiał miał pokazać, dlaczego tak ważne jest zrozumienie kluczowej roli pamięci RAM (SRAM).

Przypomnijmy, że komórki RAM o niższych adresach to nie jest „zwykła pamięć”. Owszem, można je obsługiwać jak klasyczne ośmiobitowe komórki pamięci ulotnej RAM. Jednak nieprzypadkowo komórki te nazywane są też rejestrami. Pierwszych 32 komórek to rejestry ściśle związane z „kalkulatorem” CPU; są one używane do realizacji wielu rozkazów związanych z przetwarzaniem danych. Niektóre rozkazy mogą bezpośrednio „sięgać” nie tylko do całych tych rejestrów, ale do poszczególnych bitów rejestrów. W kontekście kursu C ta sprawa nas niewiele interesuje. Natomiast bardzo interesują nas „następne” komórki RAM.

Otóż następne 64 komórki pamięci to tak zwane rejestry wejścia/wyjścia (I/O memory), zwane też rejestrami specjalnymi (o specjalnej roli). Co bardzo ważne, to właśnie za pomocą tych rejestrów obsługiwane są różne układy peryferyjne, choćby porty wejścia/wyjścia.

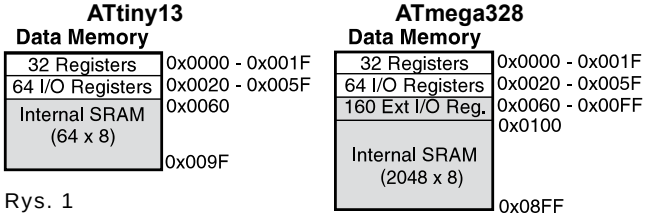
Rozwój rodziny AVR spowodował, że kolejne coraz „większe” mikrokontrolery mają coraz więcej urządzeń peryferyjnych, a przewidziane 64 specjalne rejestry I/O nie wystarczają do ich obsługi. Dlatego w tych „większych” mikrokontrolerach przewidziano dodatkową, rozszerzoną pamięć do obsługi peryferiów (extended I/O memory). Dopiero komórki pamięci o adresach wyższych są „zwykłą” pamięcią RAM (SRAM). **Rysunek 1** pokazuje strukturę pamięci RAM „malutkiego” procesora ATtiny13 i większego ATmega328 (podobnie, choć nie identycznie, podzielona jest pamięć danych RAM jeszcze większych procesorów AVR rodziny XMEGA).

Różnice między „podstawową” i rozszerzoną (extended) przestrzenią rejestrów specjalnych polegają m.in. na tym, że istnieją rozkazy (kodu maszynowego i asemblera), które „sięgają” wprost do rejestrów I/O, ale tylko tych 64 „podstawowych”. Warto o tym wiedzieć, ponieważ pozwala to wyjaśnić zagadkę straszącą początkujących. Otóż pamiętamy, że komórki pamięci o adresach 0...31 (szesnastkowo 0x00...0x1F) to rejestry związane z CPU. A więc pierwszy z 64 rejestrów I/O ma w pamięci RAM adres 32 (czyli 0x20), a ostatni rejestr „podstawowej” przestrzeni I/O ma adres 95 (0x5F). Tymczasem w niektórych bardzo często wykorzystywanych rozkazach (kodu maszynowego i asemblera) zawarty jest *numer kolejny* tego rejestru I/O, czyli liczba 0...63 (0x00...0x3F), a nie numer komórki RAM, wynoszący 32...95 (0x20...0x5F). Dlatego w kartach katalogowych spotykamy „podwójną” numerację obszaru I/O, jak pokazuje **rysunek 2**, będący fragmentem karty katalogowej Atmega328. W pierwszej kolumnie (***Adress***) w nawiasach podane są numery komórek RAM, a bez nawiasów (kolor niebieski) numery kolejne 64 rejestrów „podstawowych”.

Dla osób programujących w asemblerze oznacza to konieczność pamiętania, której numeracji używać do jakich rozkazów: w przypadku popularnych rozkazów IN, OUT trzeba podać numer kolejny rejestru (0x00...0x3F), ale rozkazy te obsługują tylko „podstawowe” 64 rejestry, a na przykład dla asemblerowych rozkazów LD, ST obsługujących całą przestrzeń adresową, do numeru kolejnego rejestru trzeba dodać 32 (0x20), czyli trzeba podać jego „prawdziwy” numer komórki RAM.

Przy programowaniu w języku C takiego problemu nie ma. Nie trzeba pamiętać ani numerów kolejnych, ani adresów rejestrów. Wystarczy wiedzieć, jakie rejestry są wykorzystywane do obsługi poszczególnych urządzeń mikroprocesora i jakie są ich **nazwy**. Na rysunku 2, w drugiej kolumnie (***Name***) mamy skróty – nazwy rejestrów. Co równie ważne, indywidualne nazwy ma też większość bitów w tych rejestrach. Pisząc program w języku C podajemy albo nazwy rejestrów, albo nazwy pojedynczych zawartych w nich bitów. I kompilator wie, o co nam chodzi.

Konfiguracja i sterowanie poszczególnych obwodów mikroprocesora AVR są więc dziecinnie łatwe – wystarczy wpisywać odpowiednie stany logiczne



Rys. 1

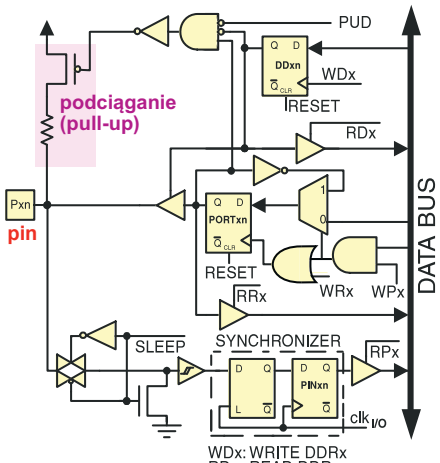
Rys. 2

		Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
		…			…	…		…			
	(102)	(0x66)	OSCCAL	Oscillator Calibration Register							
		(0x65)	Reserved	–	–	–	–	–	–	–	–
		(0x64)	PRR	PRTWI	PRTIM2	PRTIM0	–	PRTIM1	PRSPI	PRUSART0	PRADC
	(99)	(0x63)	Reserved	–	–	–	–	–	–	–	–
	(98)	(0x62)	Reserved	–	–	–	–	–	–	–	–
	(97)	(0x61)	CLKPR	CLKPCE	–	–	–	CLKPS3	CLKPS2	CLKPS1	CLKPS0
	(96)	(0x60)	WDTCSR	WDIF	WDIE	WDP3	WDCE	WDE	WDP2	WDP1	WDP0
63	(95)	0x3F (0x5F)	SREG	I	T	H	S	V	N	Z	C
62	(94)	0x3E (0x5E)	SPH	–	–	–	–	–	(SP10)	SP9	SP8
61	(93)	0x3D (0x5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0
…	…				…	…		…			
11	(43)	0x0B (0x2B)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0
10	(42)	0x0A (0x2A)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0
9	(41)	0x09 (0x29)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0
8	(40)	0x08 (0x28)	PORTC	–	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0
7	(39)	0x07 (0x27)	DDRC	–	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0
6	(38)	0x06 (0x26)	PINC	–	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0
5	(37)	0x05 (0x25)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0
4	(36)	0x04 (0x24)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0
3	(35)	0x03 (0x23)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0
2	(34)	0x02 (0x22)	Reserved	–	–	–	–	–	–	–	–
1	(33)	0x01 (0x21)	Reserved	–	–	–	–	–	–	–	–
0	(32)	0x0 (0x20)	Reserved	–	–	–	–	–	–	–	–

Numery kolejne „podstawowych” rejestrów I/O

w nawiasach (NN) - „prawdziwe” adresy komórek RAM

dziesiętnie / szesnastkowo



DDxn	PORTxn	PUD (MCUCR)	I/O	Pull-up	
0	0	X	Input	No	**Wysoka impedancja**
0	1	0	Input	Yes	**Wejście „podciągnięte”**
0	1	1	Input	No	**Wysoka impedancja**
1	0	X	Output	No	**Wyjście - stan niski**
1	1	X	Output	No	**Wyjście - stan wysoki**

Rys. 3

do odpowiednich rejestrów w przestrzeni I/O pamięci RAM.

Przykładowo, gdy chcemy wszystkie osiem pinów portu B skonfigurować jako wyjścia, musimy do rejestru o nazwie **DDRB** (Data Direction Register B) wpisać osiem jedynek. Nie musimy pamiętać numeru rejestru w pamięci – wystarczy, że wykorzystamy nazwę DDRB. Gdy potem zechcemy ustawić na pinach tego portu jakąś kombinację stanów logicznych, to tę kombinację stanów wpiszemy do rejestru o nazwie **PORTB** (Data Register). Gdy zechcemy zmienić stan jednego tylko pinu portu B, możemy wpisać potrzebny stan logiczny tylko do jednego bitu rejestru (PORTB0...PORTB7). Znów nie musimy znać numeru rejestru w pamięci.

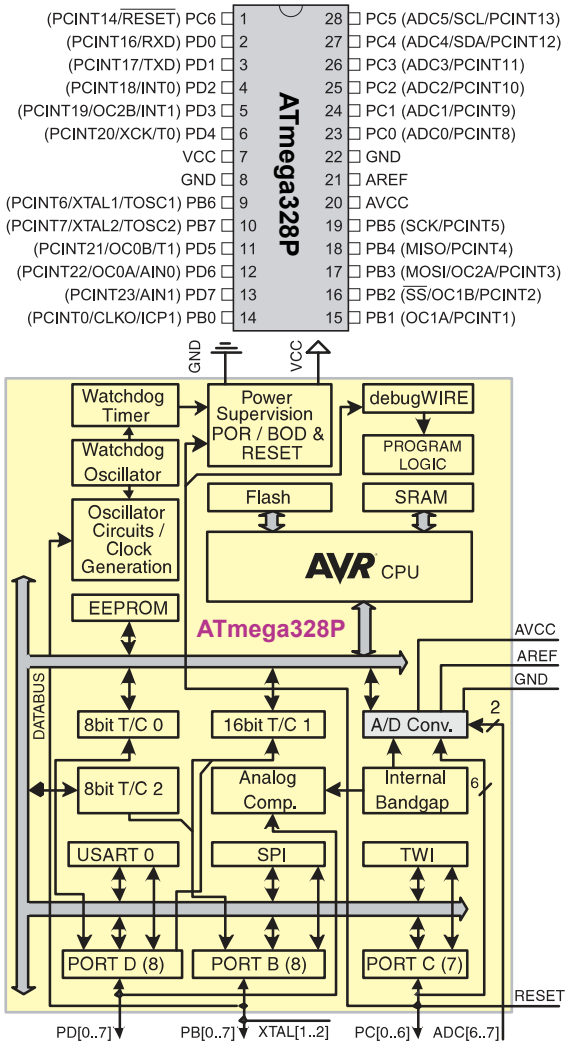
Gdy natomiast wszystkie piny portu B mają być wejściami, w rejestrze DDRB mają być zera. Możemy dać rozkaz zerowania, ale po resecie (podaniu zasilania) rejestr ten, jak większość innych – jest zerowany automatycznie. Gdy dany pin pracuje jako wejście, ma on ogromną oporność (impedancję) wejściową, ale możemy włączyć „podciąganie” (pull-up), by w spoczynku miał stan wysoki (rezystor dołączony do plusa zasilania) i by stan niski pojawiał się po zwarciu wejścia do masy, np. przy współpracy z przyciskami. Podciąganie włączamy, wpisując jedynkę do odpowiedniego bitu rejestru PORTB. Rzeczywisty, aktualny stan danego pinu możemy w każdej chwili odczytać z rejestru **PINB**, i to zarówno przy pracy w roli wejścia, jak też wyjścia(!), co też ma sens. Budowa wewnętrzna pinu, który może pracować jako wejście i wyjście, przedstawiona jest na **rysunku 3**. Te dość skomplikowane obwody obsługiwane są w prosty sposób właśnie przez wpisywanie i odczytywanie stanu trzech bitów w trzech rejestrach: DDRxn, PORTxn, PINxn. Dodatkowo ustawienie (1) bitu PUD (PullUp Disable) w rejestrze o nazwie MCUCR (MCU Control Register) blokuje obwody podciągania we wszystkich portach mikroprocesora.

Sterowanie portami I/O jest więc dziecinnie łatwe. Ale jak pokazuje górna część **rysunku 4**, poszczególne piny portów procesora mogą pełnić jedną z kilku alternatywnych funkcji. Obwody elektryczne poszczególnych pinów portów są więc znacznie bardziej skomplikowane, niż pokazuje rysunek 3. Nie trzeba się jednak bać. Można w pierwszym przybliżeniu przyjąć, że te dodatkowe rozmaite funkcje portów domyślnie są wyłączone (dlatego zwykle najprościej jest wykorzystać pin jako „zwyczajny” port wejścia/wyjścia). Ale gdy trzeba, można bardzo łatwo włączyć te dodatkowe obwody peryferyjne i wykorzystać je, oczywiście przez ustawienie odpowiednich stanów logicznych w odpowiednich rejestrach, dla nich przeznaczonych.

I tak na przykład nóżka 14, która zasadniczo jest najmłodszym pinem portu B (PB0), może też pełnić rolę wejścia dla przerwania (PCINT0 – Pin Change Interrupt Request 0). Natomiast nóżki 4, 5, będące pinami portu D (PD2, PD3) mogą służyć nie tylko jako wejścia pokrewnych przerwań zewnętrznych (PCINT18, PCINT19), ale też „bogatszych” przerwań zewnętrznych INT0, INT1. Nie będziemy się wgłębiać w szczegóły pięciu dostępnych przerwań zewnętrznych, z których dwa (INT0, INT1), są konfigurowane i obsługiwane za pomocą rejestrów EICRA – External Interrupt Control Register A o adresie (0x69), EIMSK – External Interrupt Mask Register (0x3D), EIFR – External Interrupt Flag Register (0x3C). Pozostałe trzy „zbiorcze” przerwania (PCINT0...7, PCINT8....14, PCINT16...23) są obsługiwane za pomocą rejestrów o nazwach i adresach: PCICR – Pin Change Interrupt Control Register (0x68), PCIFR – Pin Change Interrupt Flag Register (0x3B), PCMSK2 – Pin Change Mask Register 2 (0x6D), PCMSK1 – Pin Change Mask Register 1 (0x6C), PCMSK0 – Pin Change Mask Register 0 (0x6B). Ponadto, aby zezwolić na przerwania, trzeba też ustawić bit oznaczony I w rejestrze stanu SREG – Status REGister. Jak widać na rysunku 2, bardzo ważny rejestr SREG na numer kolejny 63 = 0x3F i adres (95) = (0x5F), czyli jest to ostatni z rejestrów „podstawowych”. W karcie katalogowej znajdziesz dokładniejszy opis poszczególnych bitów rejestrów. Początek opisu rejestru SREG znajdziesz na **rysunku 5**.

Na rysunku 2 widać też inne rejestry o rozmaitym przeznaczeniu. I tak para rejestrów SPL (0x5D), SPH (0x5E) to tak zwany wskaźnik stosu (Stack Pointer), rejestr WDTCSR – Watchdog Timer Control Register (0x60) konfiguruje układ „czuwaka” – watchdoga, CLKPR – Clock Prescale Register (0x61) konfiguruje dzielnik częstotliwości zegara

Rys. 4



systemowego, a zawartość rejestru OSCCAL – Oscillator Calibration Register (0x66) może posłużyć do dokładniejszej kalibracji wewnętrznego generatora taktującego RC.

A teraz przejdźmy do nóżek 9, 10 procesora, które według rysunku 4 zasadniczo są „najwyższymi" pinami portu B. Można jednak do nich dołączyć rezonator kwarcowy (XTAL). Wcześniej mówiliśmy, że konfigurację poszczególnych nóżek realizujemy za pomocą rejestrów w przestrzeni I/O, czyli komórek pamięci RAM o adresach poniżej 0xFF. Tak, ale dotyczy to spraw „bieżących". Natomiast w tym przypadku jest inaczej. Chodzi bowiem o jedno z podstawowych ustawień: sygnałem z jakiego źródła ma być taktowany procesor podczas pracy?

Należy to ustawić jednorazowo podczas wpisywania programu do procesora i realizują to „fusebajty", zrealizowane jako komórki pamięci FLASH. Spośród trzech „fusów" dostępnych w ATmega 328 w „niższym" (Fuse Byte Low) trzy najmłodsze bity to CKSEL2...CKSEL0 określają źródło sygnału zegarowego. Podobnie najstarszy bit RSTDISBL (External Reset Disable) „wyższego fusebajtu" określa rolę nóżki 1: niezaprogramowany – domyślnie konfiguruje nóżkę 1 jako wejście RESET, a zaprogramowany czyni z niej pin 6 portu D (PD6). Z kolei zaprogramowanie bitu WDTON (Watchdog Timer Always On) tego „wyższego" fusebajtu włącza „najbardziej radykalny" sposób działania watchdoga.

Podkreślmy, że fusebity są zapisywane podczas programowania procesora w specjalnie wydzielonych, „dodatkowych" komórkach pamięci nieulotnej FLASH. Natomiast konfiguracja i bieżąca obsługa procesora polega na zapisywaniu lub odczytywaniu rejestrów o adresach (0x20)...(0xFF) w pamięci ulotnej RAM. Po dołączeniu zasilania (i przy resecie z innych powodów) rejestry I/O są „czyszczone". Potem zaczyna pracę program wpisany do pamięci FLASH i pierwsze rozkazy tego programu powinny skonfigurować mikroprocesor przez wpisanie do rejestrów I/O zawartych w ulotnej pamięci RAM wszystkich potrzebnych ustawień. W przypadku mikrokontrolerów jednoukładowych działanie programu w większości polega na czynnym korzystaniu z peryferii, a to polega właśnie na zapisie i odczycie rejestrów I/O.

Autor programu musi znać budowę i możliwości danego procesora. Nie musi znać adresów rejestrów I/O, ale powinien wiedzieć jakie peryferia są dostępne w danym procesorze i jak można je wykorzystać. Przy programowaniu w asemblerze trzeba poznać i uwzględnić wszystkie szczegóły. Przy programowaniu w języku C też można stworzyć od zera cały program, ale istnieją gotowe „kawałki" – funkcje, pozwalające w prostszy sposób obsługiwać peryferie procesora.

To jest dobra wiadomość, ale w sumie sytuacja wcale nie jest wesoła. Już sama objętość kart katalogowych mikrokontrolerów ATmega może przyprawić o ból głowy. Karta katalogowa procesora ATmega328 ma 660 stron! Ale dla porównania: karta małego ATtiny13A ma „tylko" 176 stron.

Na szczęście w kartach katalogowych mnóstwo informacji się powtarza, więc w sumie te ponad 600 stron można byłoby „ścisnąć" kilkakrotnie. Ponadto procesory rodziny AVR (ATtiny, ATmega, a nawet XMEGA) mają mnóstwo wspólnych cech i jednakowe zasady działania. Różnią się głównie ilością pamięci i peryferiów. Ale faktem jest, że trudno ogarnąć i zapamiętać wszystkie szczegóły.

Objętość materiału oraz liczba różnych szczegółów i opcji jest ogromna. Nie powinna jednak przytłaczać, ponieważ można zacząć od rozwiązań najprostszych, a potem stopniowo „wgryzać się" w temat. Nie ulega wątpliwości, że ***jeśli ktoś chce w pełni wykorzystywać współczesne mikroprocesory – musi się wiele nauczyć***. Nie ma dobrej drogi „na skróty"! Trzeba na to poświęcić dużo czasu i włożyć mnóstwo wysiłku. Cykl w EdW może pomóc w rozpoczęciu przygody z programowaniem AVR-ów w języku C, ale to Ty musisz stopniowo nabyć mnóstwo wiedzy i umiejętności. Mam delikatną sugestię: może dobrze byłoby założyć zeszyt (gruby) i notować w nim swoje spostrzeżenia i „odkrycia"? Ponadto w związku z tym proponuję ważne...

Zadanie domowe:

Ściągnij karty katalogowe mikroprocesorów ATmega328P (http://goo.gl/YpF4YV) oraz Attiny13 (http://goo.gl/9L949K). Porównaj budowę wewnętrzną – zwłaszcza dostępne peryferie (rysunki odpowiednio na stronach 6 i 4). Potem w końcowej części każdej karty znajdziesz *Register Summary* – pełną tabelę z adresami i nazwami rejestrów (odpowiednio strony 612 i 158). Po pierwsze sprawdź, czy adresy rejestrów o takich samych nazwach są jednakowe w tych procesorach AVR? Po drugie, w tabelach z prawej strony znajdziesz odnośniki – hiperłącza do stron, gdzie są omawiane szczegóły. Niezależnie od tego na ile znasz elementarne podstawy angielskiego spróbuj choć trochę poznać rolę i wykorzystanie poszczególnych rejestrów.

Piotr Górecki

Rys. 5

7.3.1 SREG – AVR Status Register

The AVR Status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

- **Bit 7 – I: Global Interrupt Enable**

The Global Interrupt Enable bit must be set for the interrupts to be enabled. The individual interrupt enable control is then performed in separate control registers. If the Global Interrupt Enable Register is cleared, none of the interrupts are enabled independent of the individual interrupt enable settings. The I-bit is cleared by hardware after an interrupt has occurred, and is set by the RETI instruction to enable subsequent interrupts. The I-bit can also be set and cleared by the application with the SEI and CLI instructions, as described in the instruction set reference.

- **Bit 6 – T: Bit Copy Storage**

The Bit Copy instructions BLD (Bit LoaD) and BST (Bit STore) use the T-bit as source or destination for the operated bit. A bit from a register in the Register File can be copied into T by the BST instruction, and a bit in T can be copied into a bit in a register in the Register File by the BLD instruction.

- **Bit 5 – H: Half Carry Flag**

The Half Carry Flag H indicates a Half Carry in some arithmetic operations. Half Carry Is useful in BCD arithmetic. See the "Instruction Set Description" for detailed information.