TMw09 Programowanie mikrokontrolerów

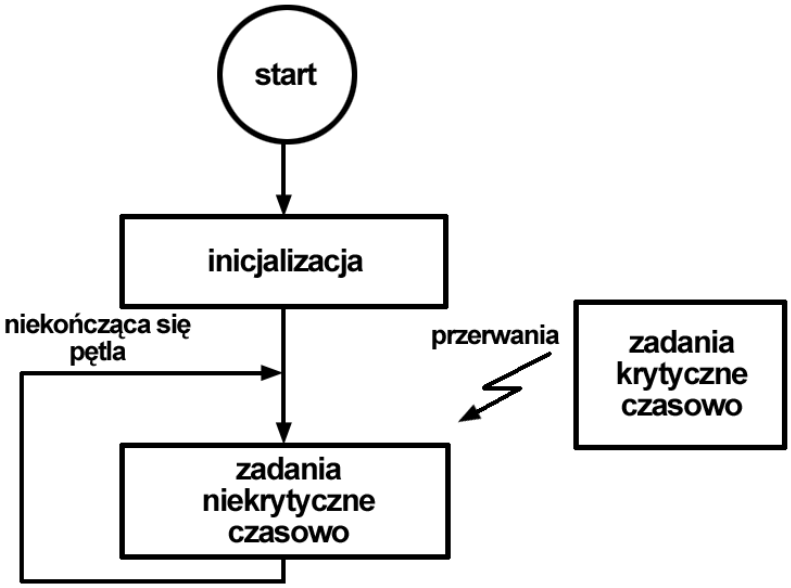
Programowanie mk jest ściśle związane z architekturą samego procesora i jego sprzętowego otoczenia. Tworzenie oprogramowania dla mse opartych na mk (i nie tylko) i ukierunkowanych na zadania pomiarowo-sterujące oraz komunikacyjne określa się w literaturze mianem **programowania zagnieżdżonego** (*embeded programming*). Można wyróżnić następujące cechy programów zagnieżdżonych:

- Program **jednoznacznie ustala funkcję mse**, tzn. użytkownik ma możliwość zmiany funkcji systemu zazwyczaj tylko w niewielkim zakresie przewidzianym przez program użytkowy. Ta właśnie cecha określana jest jako „**zagnieżdżenie**" programu.
- Działanie programu **musi spełniać określone wymagania czasowe** dotyczące przekraczania maksymalnego czasu reakcji na określone zdarzenia zewnętrzne oraz realizacji określonych zadań programowych w nieprzekraczalnym czasie. Ta cecha określana jest jako praca programu w „**czasie rzeczywistym**".
- Są to programy **działające na specyficznych zasobach sprzętowych** warunkowanych ukierunkowaniem budowy sprzętowej mse na konkretne zadanie.

Cykl tworzenia oprogramowania dla mk jest zbliżony do cyklu tworzenia oprogramowania dla komputerów osobistych. Cykl ten obejmuje trzy fazy:

- **napisanie kodu źródłowego**,
- **przetłumaczenie kodu źródłowego na kod maszynowy danego mk**,
- **uruchomienie programu w systemie docelowym (mse)**.

Faza pierwsza i druga wykonywana jest najczęściej przy zastosowaniu standardowego komputera osobistego. Komputer osobisty w tej funkcji określany jest jako **system rozwojowy** (*development system*). Faza trzecia realizowana jest na rzeczywistym mse zawierającym mk. System ten określany jest jako **system docelowy** (*target system*).
Tworzenie oprogramowania mk wymaga od programisty dysponowania odpowiednimi programami, tzw. programami narzędziowymi (*software tools*). Obecnie programy te łączone są w jeden pakiet. Umożliwia to realizację wszystkich etapów cyklu rozwoju oprogramowania dla mk w ramach jednego programu. Środowisko takie określa się skrótem **IDE** (*Integrated Development Enviroment*). Zawiera ono między innymi specjalizowany edytor tekstowy, kompilator, linker, symulator programowy, debuger, oprogramowanie sterujące programatorem sprzętowym służące do programowania mk.
Przed rozpoczęciem pisania kodu źródłowego (w edytorze tekstowym) należy wybrać określony język programowania. W praktyce stosuje się dwa rodzaje języków programowania mk**: język asemblera** lub jeden z języków wysokiego poziomu – **język C/C++** albo Java.
Ważna uwaga: poprawnie napisany program użytkownika musi działać w niekończącej się pętli. Jc po włączeniu napięcia zasilania bez przerwy pobiera z pamięci i wykonuje kolejne instrukcje. Nie może się zatem skończyć po wykonaniu ostatniego rozkazu. Zatem program użytkownika ma strukturę jak pokazano na rysunku poniżej.



Struktura programu użytkownika na mk

Po włączeniu napięcia zasilania w programie użytkownika musi się odbyć **jednorazowa inicjalizacja**, np. ustalenie trybu pracy urządzeń peryferyjnych. Właściwy program użytkownika jest realizowany częściowo w **niekończącej się pętli głównej**, a częściowo w **obsługach przerwa**ń.

Programowanie w języku asemblera

Asembler jest **językiem niższego poziomu**. Operuje on bezpośrednio na liście rozkazów (instrukcji) danej jc. Każdy rozkaz z tej listy ma przyporządkowaną krótką nazwę zwaną **mnemonikiem**. Za ich pomocą programista tworzy kod źródłowy. Następnie program tłumaczący tłumaczy mnemoniki na odpowiadający im kod maszynowy. Stąd określenie „język niższego poziomu". Program tłumaczący z języka asemblera określany jest jako **asembler** (*assembler*).

Programowanie w języku asemblera stosuje się w trzech przypadkach:

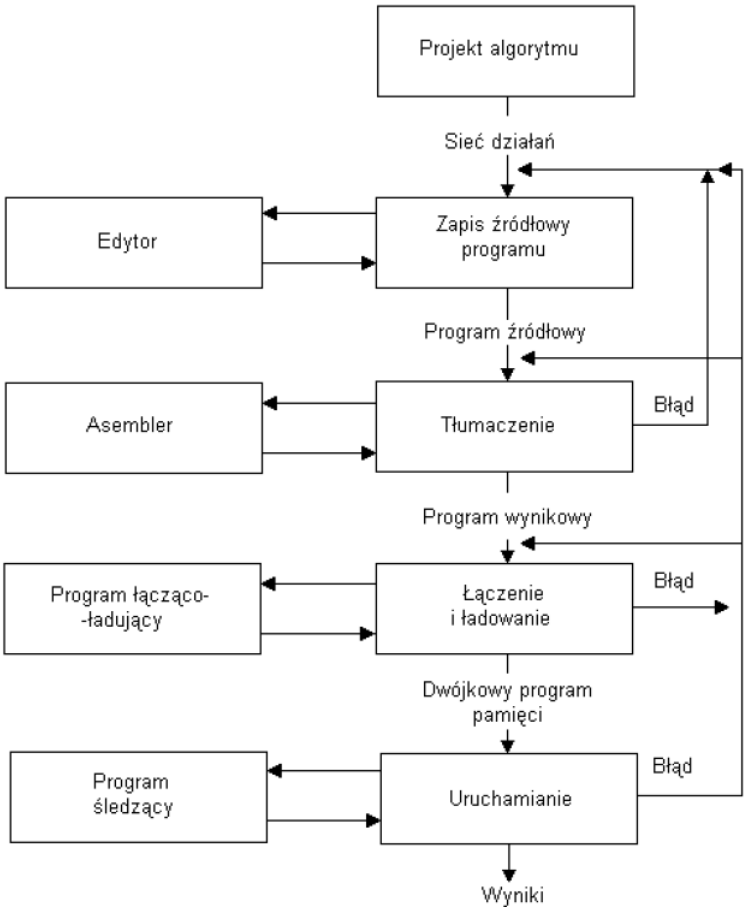
- gdy tworzona aplikacja jest bardzo prosta i zapis w języku asemblera nie sprawia kłopotu,
- gdy nie dysponujemy kompilatorem C/C++ lub jego koszt byłby niewspółmierny do skali projektu,
- gdy wymagania wobec szybkości działania programu i minimalizacji zajmowanej pamięci są szczególnie ostre.

Do zalet programowania w języku asemblera należy zaliczyć:

- Możliwość pełnego panowania nad zasobami systemu. Programista ma nieograniczony dostęp do wszystkich bloków na poziomie rejestrów i pojedynczych bitów. żadna z funkcji systemu nie jest ukryta, w szczególności możliwe jest dowolne, nawet nietypowe oddziaływanie na obszar stosu i mechanizm przerwań.
- Swobodne dysponowanie obszarem pamięci.
- Efektywny program wynikowy, szybszy i zajmujący na ogół znacznie mniej pamięci niż równoważny program zapisany w języku wysokiego poziomu.

- Możliwość swobodnego wyboru formatu danych i precyzji obliczeń. Programista może samodzielnie definiować wielobajtowe struktury danych do obliczeń o praktycznie dowolnej dokładności.
- Możliwość dopasowania algorytmu do indywidualnych cech architektury mk oraz optymalizacji programu wynikowego.

Jednak zastosowanie asemblera ma też wady: programowanie jest żmudne i zajmuje więcej czasu niż przy użyciu języków wysokiego poziomu. Program jest zatem droższy, trudniejszy do modyfikowania, bardziej podatny na błędy i mniej czytelny, nawet jeśli zastosowano obszerne komentarze.
Projektowanie oprogramowania w języku asemblera przebiega według schematu przedstawionego na rys. Do edycji programów źródłowych (standardowe rozszerzenie ASM) zapisanych w plikach tekstowych ASCII można użyć dowolnego, prostego edytora tekstowego. Następnie program źródłowy jest poddawany tłumaczeniu (czyli translacji lub asemblacji). W rezultacie najczęściej powstają dwa pliki. Pierwszy z nich to program wynikowy (*object code*) zapisany w pliku **OBJ**. Jest on **przemieszczalny (relokowalny** – *relocatable code*), tzn. może być przypisany do ustalonego później obszaru adresowego pamięci. Drugim jest plik tekstowy z raportem translacji i wydrukiem programu źródłowego z towarzyszącym mu kodem wynikowym (zazwyczaj w postaci szesnastkowej). Raport z translacji (*listing file* – stąd najczęstsze rozszerzenie LST) podaje ponadto numery i adresy kolejnych linii (instrukcji) programu, adresy zadeklarowanych zmiennych, zestawienie użytych nazw symbolicznych oraz ewentualne komunikaty o błędach.



Cykl projektowania w języku asemblera

Po uzyskaniu programu wynikowego wolnego od błędów translacji następuje faza łączenia programu (konsolidacji). W fazie tej wykonuje się połączenie modułów wynikowych (jeśli program został napisany w kilku plikach OBJ), dołączenie modułów bibliotecznych (jeśli zostały użyte w programie) oraz uzgodnienie adresów modułów. W rezultacie powstaje **nieprzemieszczalny** program binarny (*non-relocatable code*), który można załadować do pamięci mk i uruchomić. Format tego pliku zależy od typu procesora i użytego środowiska IDE. Jednym z najpopularniejszych formatów jest format **HEX** opracowany przez firmę Intel (Intel-HEX). Przechowuje on kody instrukcji i dane (stałe) w postaci ciągu rekordów zapisanych w kodzie ASCII. Każdy z rekordów ma stały format:

:llaaaatt[dd....]cc

przy czym:

: - znak dwukropka, identyfikator początku każdego nowego rekordu,
ll – długość pola danych (dd....) wyrażona w bajtach,
aaaa – 16-bitowy adres miejsca pamięci, do którego mają być załadowane dane zapisane w rekordzie,
tt – kod typu rekordu: 00 oznacza rekord danych, 01 oznacza rekord terminalny (kończący plik HEX),
dd – ciąg bajtów danych,
cc – suma kontrolna obliczana jako różnica: 00H-(suma wszystkich bajtów rekordu) mod 256.

Podczas ładowania programu z pliku HEX do pamięci systemu wykonywane jest rozpakowanie kolejnych rekordów, sprawdzanie sumy kontrolnej oraz przesłanie pola danych pod wskazany adres. Po załadowaniu program gotowy jest do uruchomienia. Testowanie programu jest wykonywane pod nadzorem **programu** ś**ledzącego** (symulator software'owy).

W początkowej fazie symulacji korzysta się z trybu pracy krokowej lub systemu **pułapek** (punktów zatrzymań) ulokowanych w zadanych miejscach programu. Śledzenie pracy programu polega na sprawdzeniu zawartości rejestrów i wybranych komórek pamięci w kolejnych fazach wykonywania programu. Jeśli zawartości te zmieniają się zgodnie z założeniami projektowymi, a przyłączone do systemu urządzenia peryferyjne pracują również zgodnie z oczekiwaniami, uruchamianie programu jest zakończone.
W ostatnim kroku program zostaje załadowany do pamięci mk znajdującego się w mse za pomocą odpowiedniego programatora (zakłada się, że mk posiada pamięć typu FLASH). Na tym etapie prac mse jest traktowany jako prototyp. Następnie odbywa się testowanie prototypu: sprawdzanie jego parametrów elektrycznych i czasowych. Gdy mse zachowuje się prawidłowo, to można uznać projekt za zakończony.

Programowanie w językach wyższego poziomu

Języki programowania wyższego poziomu są w zasadzie niezależne od listy instrukcji konkretnej jc. Programista operuje rozkazami wykonującymi bardziej złożone operacje niż język asemblera. Program tłumaczący z języka programowania wyższego rzędu na kod maszynowy nosi nazwę **kompilatora** (*compiler*).
W języku asemblera każda linia programu tłumaczona jest na odpowiedni rozkaz jc danego mk. W języku wyższego poziomu każda linia programu tłumaczona jest na kilka, kilkanaście, a czasami nawet na kilkadziesiąt rozkazów asemblera. Ten ostatni przypadek między innymi ma miejsce np. wówczas, gdy w programie zastosowane są działania na liczbach zmiennoprzecinkowych, zaś jc nie wspiera operacji zmiennoprzecinkowych.

Różne jc mają różne listy rozkazów, zatem program napisany w języku asemblera jest związany na stałe z daną jc. Program napisany w języku programowania wyższego poziomu jest w niewielkim stopniu związany z daną jc. W celu uruchomienia tego programu w mse zawierającym inny mk wystarczą na ogół niewielkie modyfikacje kodu źródłowego, a następnie przetłumaczenie tego kodu na kod maszynowy przy zastosowaniu narzędzi programowych właściwych dla danego mk. Zmiany w kodzie muszą być znacznie większe, jeżeli obydwa mk wykorzystują inne układy peryferyjne.

Dominującym językiem wyższego poziomu jest **język C**. Obecnie obowiązuje jego **standard ANSI C**. Jednak większość kompilatorów zawiera dodatkowe rozszerzenia tego języka związane ze specyficznymi wymaganiami przy programowaniu mk. Najczęściej są to elementy:

- wprowadzone są nowe typy zmiennych, np. **zmienne bitowe**,
- przy deklaracji zmiennych wprowadzane są mechanizmy umożliwiające umiejscowienie (**alokację**) zmiennej w określonym miejscu przestrzeni adresowej,
- wprowadzone są nowe słowa kluczowe języka C (np. słowo ***interrupt*** jest używane przy deklaracji funkcji wywoływanych przy wystąpieniu przerwania).

Zalecane jest aby pisząc program w języku C na mk stosować następujące rozwiązania:

- fragmenty kodu źródłowego związane z obsługą specyficznych układów peryferyjnych umieszczać w wydzielonych modułach,
- typy zmiennych deklarować nie bezpośrednio przy deklaracji zmiennych, ale poprzez słowo kluczowe *typdef*,
- alokacji zmiennych nie dokonuje się bezpośrednio przy deklaracji zmiennych, lecz za pomocą dyrektyw preprocesora, głównie przez użycie tzw. modelu pamięci,
- alokacja funkcji służących do obsługi przerwań (tzw. rozprowadzanie przerwań) odbywa się w wydzielonym zbiorze, często napisanym w języku asemblera.

Tworzenie programu w języku C przebiega według tego samego schematu, co cykl projektowy programu w języku asemblera. Pliki źródłowe programu zwykle oznacza się rozszerzeniem C.

Uruchamianie programu mk

Uruchamianie programów jest procesem **eliminacji błędów** i wprowadzania zmian do programu użytkowego w celu uzyskania bezbłędnej i zgodnej z założeniami pracy systemu w danym zastosowaniu.

Uruchamianie prototypowych mse z mk jest trudnym procesem. Wynika to z dwóch podstawowych przyczyn:

- mk mają wiele wbudowanych bloków funkcjonalnych (pamięci i układów peryferyjnych), do których dostęp w czasie pracy systemu jest utrudniony lub niemożliwy,
- sterowniki z wbudowanymi mk są przeznaczone do pracy w czasie rzeczywistym.

Testowanie systemu w czasie rzeczywistym, we współpracy z rzeczywistym otoczeniem, bez naruszenia relacji czasowych sygnałów, wymaga narzędzi diagnostycznych, które nie zakłócają pracy systemu.

Zatem opracowano wiele metod stosowanych w tym celu. Metody te dzielą się na dwie grupy:

- **bez wykorzystania systemu docelowego**,
- **z wykorzystaniem systemu docelowego**.

W ramach pierwszej grupy stosowana jest jedna metoda. Jest to wykorzystanie **symulatora** programowego mk (*simularot*). Symulatory umożliwiają symulację pracy jc, a bardziej rozbudowane także układów peryferyjnych mk. Zaletą tej metody jest to, iż nie trzeba dysponować fizycznym docelowym systemem do testowania oprogramowania oraz to, że na podstawie wyników testów (np. czasu wykonywania programu) można wprowadzać odpowiednie korekty do koncepcji budowy sprzętowej systemu docelowego jeszcze przed jego wykonaniem. Natomiast wadą tej metody jest to, że poprawne działanie programu z symulatorem nie daje całkowitej gwarancji bezbłędnej pracy w rzeczywistym systemie. Zatem ostateczna weryfikacja poprawności napisanego programu może nastąpić tylko podczas jego wykonywania w systemie docelowym.

W skład drugiej grupy metod wchodzą:

- **metoda prób i błędów** - polega na wielokrotnym programowaniu mk i za każdym razem obserwacji działania programu w mse i jego korekcji na podstawie tych obserwacji, aż do uzyskania prawidłowego działania mk,
- zastosowanie **monitorów programowych** (*monitors*) i **programów** ś**ledzą**c**ych** (*debuggers*) – są najczęściej stosowane. Monitory są instalowane w pamięci programu mk i kontrolują wykonywanie właściwego programu użytkowego oraz komunikują się z systemem rozwojowym poprzez złącze szeregowe. Natomiast debugery pracują w przyłączonych do systemu komputerach PC,
- zastosowanie **emulatora sprzętowego** mk (ICE – *in-circuit emulators*) – polega to na umieszczeniu, na czas uruchamiania programu, w podstawce na mk sondy połączonej ze specjalnym układem sprzętowym, który emuluje działanie mk. Emulator wiernie odtwarza wszystkie właściwości mk łącznie z jego wszystkimi układami peryferyjnymi oraz pamięcią, wykorzystanie **specjalnych zasobów wewnętrznych mk** – niektóre mk zwłaszcza 32- bitowe posiadają specjalne zasoby sprzętowe przeznaczone do wspierania procesu uruchomiania programu. Zasoby te oferują w przybliżeniu wszystkie te możliwości co emulator sprzętowy, między innymi ustawienie pułapek oraz pracę krokową. Zasoby te komunikują się przez dedykowane wyprowadzenia mk. Są one dostępne wyłącznie na etapie uruchamiania programu, zatem nie są wykorzystywane przez normalny program użytkowy.

Sposoby programowania mk z pamiecią FLASH

Obecnie prawie wszystkie produkowane mk są dostępne w wersji z pamięcią programu typu FLASH. Zaletą pamięci FLASH (pamięć błyskowa) jest możliwość wielokrotnego jej programowania bezpośrednio w systemie docelowym (bez konieczności wyciągania mk z podstawki) – ISP (*In-System Programmable*).

Możemy wyróżnić dwa tryby programowania mk:

- **tryb równoległy** – wymagający umieszczenia mk w programatorze,
- **tryb szeregowy** (ISP).

Tryb programowania równoległego

Wadą trybu równoległego programowania jest konieczność wyciagnięcia mk z mse i umieszczenie go w programatorze. Zaletą, zwłaszcza dla mk firmy Atmel, jest to, iż mk nie dające się z jakichkolwiek przyczyn zaprogramować w trybie ISP, dadzą się programować w trybie równoległym.

Tryb programowania szeregowego (ISP)

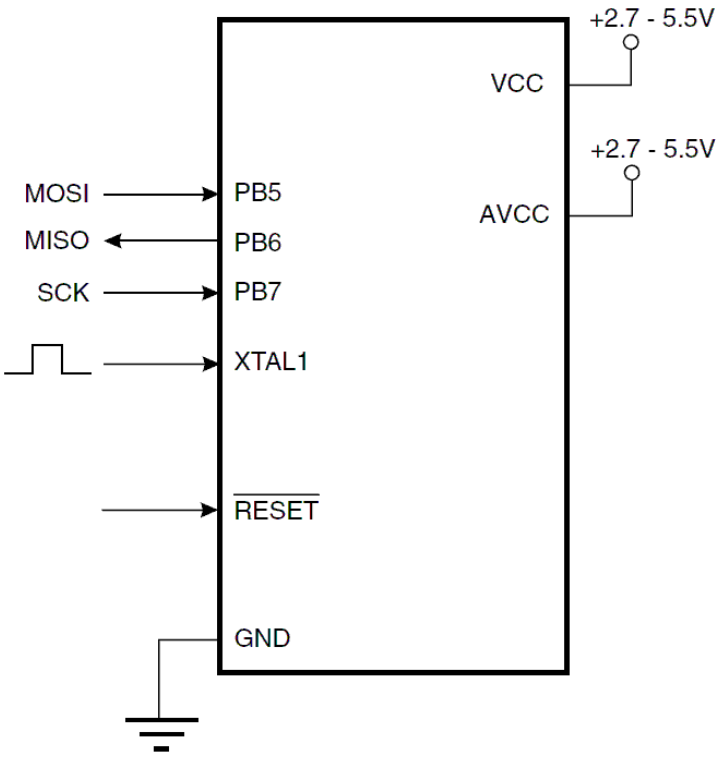
Jak wspomniano, zaletą programowania szeregowego jest możliwość programowania mk bezpośrednio w systemie docelowym (ISP).
Generalnie procedura programowania szeregowego mk jest następująca:

- Wprowadzenie mk w **tryb programowania**. Odbywa się to podczas resetu mk poprzez podanie na odpowiednie końcówki określonej sekwencji sygnałów lub odpowiednich poziomów napięć.
- Następnie poprzez dedykowany do programowania **interfejs szeregowy** wprowadza się szeregowo odpowiednie rozkazy sterujące procesem programowania pamięci. Część rozkazów dotyczy zapisu i odczytu danych do/ze specjalnych rejestrów konfiguracyjnych zawartych w pamięci FLASH odpowiedzialnych za konfigurację mk (tryb pracy oscylatora, zabezpieczenia przed odczytem, itd.). Rejestry te dostępne są wyłącznie w trybie programowania mk. Pozostałe rozkazy są związane z zapisem i odczytem danych z pamięci FLASH oraz z jej kasowaniem.
- Na zakończenie programowania następuje odpowiednia sekwencja sygnałów lub zdjęcie odpowiednich napięć powodujące wyjście z trybu programowania.

Możemy wyróżnić następujące sposoby programowania ISP:

- z wykorzystaniem interfejsu SPI (mk rodziny AVR i ST7),
- oparte na interfejsie JTAG (IEEE 1149.1) (mk ATmega16),
- bazując na dedykowanym interfejsie szeregowym,
- opierając się na interfejsie UART i wbudowanym programie ładującym (*loader*) zawartym w dedykowanej pamięci ROM mk,
- bazując na interfejsie USB i i wbudowanym programie ładującym zawartym w pamięci mk (32-bitowt mk rodziny AT91SAM7).

Pierwszy sposób, oparty na **interfejsie SPI**, zostanie przedstawiony na przykładzie mk ATmega16 firmy Atmel. Konfigurację pinów dla trybu programowania szeregowego bazującego na interfejsie SPI pokazano na rys.



Sygnały w trybie programowania szeregowego z wykorzystaniem interfejsu SPI

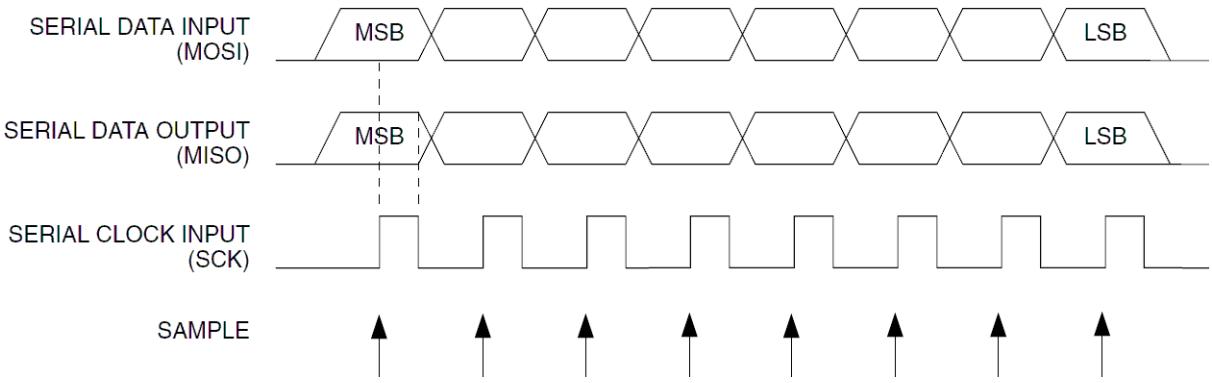
Jak widać z rys. do programowania wykorzystywany jest interfejs SPI (linie MOSI, MISO, SCK) oraz linia RESET. W trybie programowania mk musi być taktowany zewnętrznym sygnałem zegarowym podłączonym do linii XTAL1 o częstotliwości do12MHz lub musi być do mk podłączony oscylator kwarcowy. Sygnał zegarowy SCK taktujący transmisję musi być przynajmniej 3 razy wolniejszy niż częstotliwość na XTAL1.

Aby zaprogramować lub zweryfikować pamięć programu mk należy wykonać następującą sekwencję:

1. Włączyć napięcie zasilania, w tym czasie na linii RESET oraz SCK musi być stan niski.
2. Odczekać 20 ms i włączyć tryb szeregowego programowania wysyłając za pomocą interfejsu SPI komendę Programing Enable.
3. Wysłać odpowiednie komendy programujące i odczekiwać czasy potrzebne na zaprogramowanie lub komendy związaną z odczytem (lista wszystkich komend zawarta jest w tabeli).
4. Na zakończenie programowania lub weryfikacji układu ustawić sygnał RESET w stan wysoki, aby przejść w normalny tryb pracy mk.

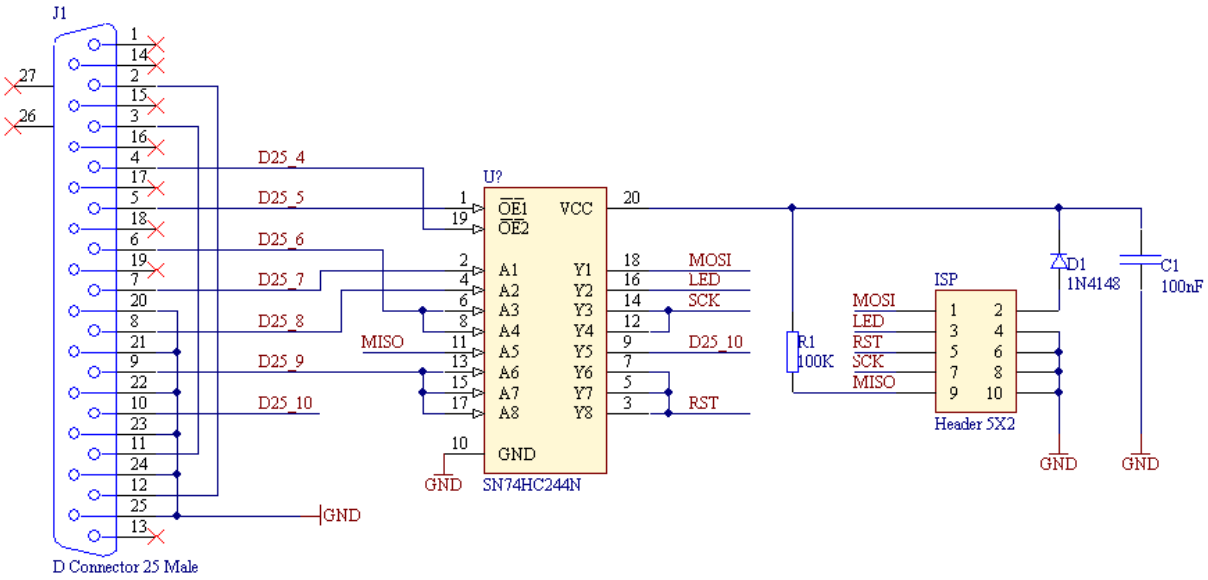
Tabela - Lista instrukcji obsługi mk ATmega w trybie programowania szeregowego

Instruction[1]/Operation	Instruction Format			
	Byte 1	Byte 2	Byte 3	Byte4
Programming Enable	$AC	$53	$00	$00
Chip Erase (Program Memory/EEPROM)	$AC	$80	$00	$00
Poll RDY/$\overline{BSY}$	$F0	$00	$00	data byte out
Load Instructions				
Load Extended Address byte[1]	$4D	$00	Extended adr	$00
Load Program Memory Page, High byte	$48	adr MSB	adr LSB	high data byte in
Load Program Memory Page, Low byte	$40	adr MSB	adr LSB	low data byte in
Load EEPROM Memory Page (page access)[1]	$C1	$00	adr LSB	data byte in
Read Instructions				
Read Program Memory, High byte	$28	adr MSB	adr LSB	high data byte out
Read Program Memory, Low byte	$20	adr MSB	adr LSB	low data byte out
Read EEPROM Memory	$A0	adr MSB	adr LSB	data byte out
Read Lock bits	$58	$00	$00	data byte out
Read Signature Byte	$30	$00	0000 000aa	data byte out
Read Fuse bits	$50	$00	$00	data byte out
Read Fuse High bits	$58	$08	$00	data byte out
Read Extended Fuse Bits	$50	$08	$00	data byte out
Read Calibration Byte	$38	$00	$0b00 000bb	data byte out
Write Instructions				
Write Program Memory Page	$4C	000a aaaa	aa00 0000	$00
Write EEPROM Memory	$C0	adr MSB	adr LSB	data byte in
Write EEPROM Memory Page (page access)[1]	$C2	adr MSB	adr LSB	$00
Write Lock bits	$AC	$E0	$00	data byte in
Write Fuse bits	$AC	$A0	$00	data byte in
Write Fuse High bits	$AC	$A8	$00	data byte in
Write Extended Fuse Bits	$AC	$A4	$00	data byte in



Przebiegi czasowe dla interfejsu SPI w trybie programowania

Producent udostępnia pełną dokumentację dotyczącą programowania mk rodziny AVR. Ponadto istnieje wiele darmowych aplikacji służących do programowania tych mk: AVRStudio, AVRisp, PonyProg, itd. Na rys. poniżej pokazano schemat ideowy programatora zgodnego ze standardem STK200 i obsługiwanego przez AVRisp oraz PonyProg.



Schemat ideowy programatora STK200

Programator ten podłączany jest do portu równoległego (portu LPT) komputera PC. Gdy komputer nie jest wyposażony w ten port, to należy korzystać z programatorów sterowanych za pomocą interfejsu USB, np. programatory standardu STK500.

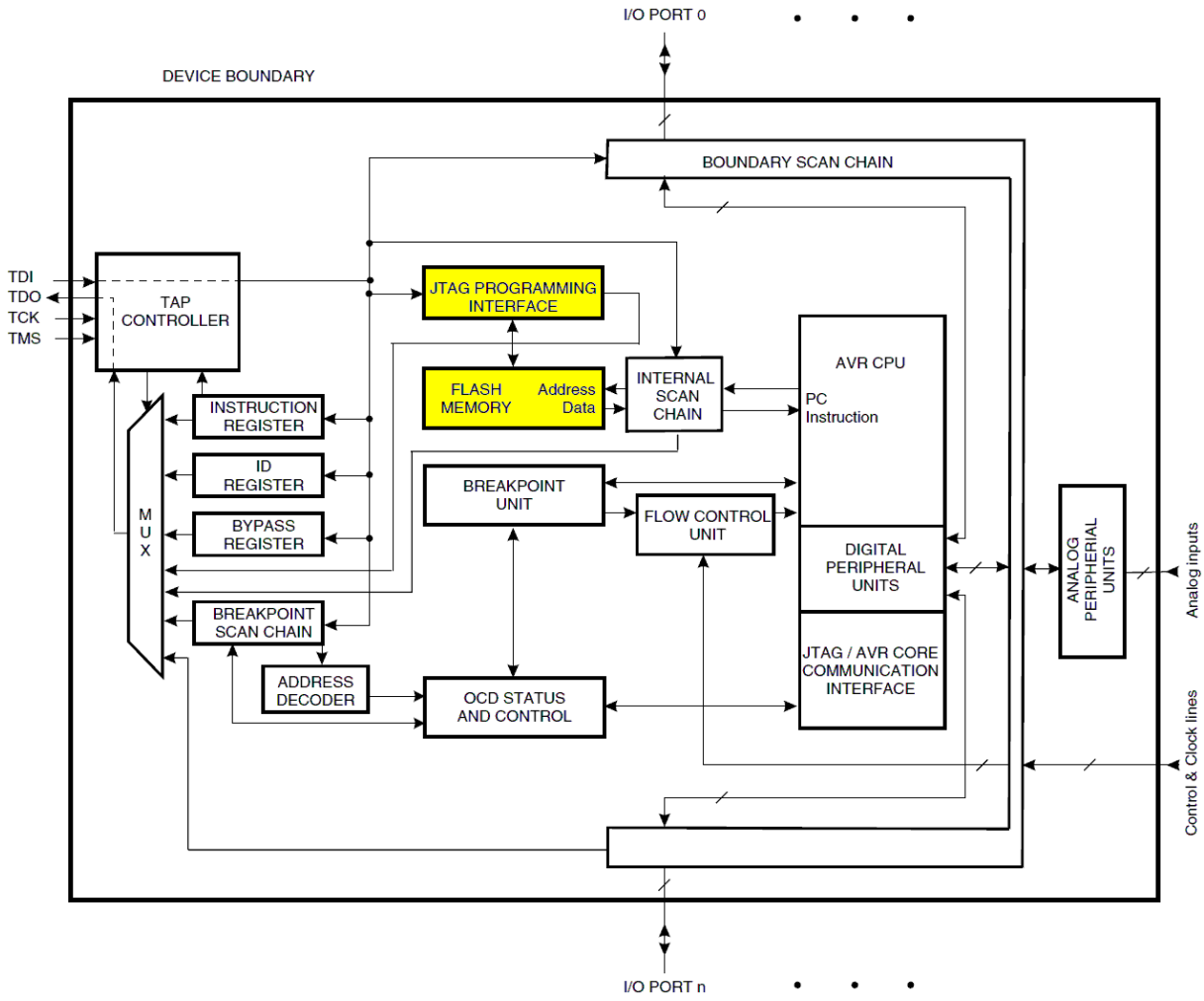
Kolejny sposób programowania szeregowego bazuje na **interfejsie JTAG (IEEE 1149.1)** przeznaczonym do:

- testowania poprawności połączeń na płytce drukowanej (oczywiście w obrębie układu scalonego jakim jest mk) bazując na ścieżce brzegowo-krawędziowej,
- programowania pamięci FLASH i EEPROM oraz bitów zabezpieczających,
- debugowania struktury wewnętrznej mk.

Interfejs JTAG składa się z czterech linii:

- TMS (Test Mode Select) służy do sterowania kontrolerem TAP interfejsu JTAG,
- TCK (Test Clock) synchronizuje transmisje danych szeregowych,
- TDI (Test Data In) wejście danych szeregowych wprowadzanych do rejestru instrukcji lub rejestru danych (łańcucha ścieżki brzegowo-sterującej),
- TDO (Test Data Out) szeregowe wyście danych z rejestru instrukcji lub rejestru danych.

Aby móc programować pamięć mk za pomocą interfejsu JTAG, bit konfiguracyjny JTAGEN musi być ustawiony, a bit JTD w rejestrze MCUCSR wyzerowany. Wówczas obsługa programowania mk realizowana jest za pomocą 4-bitowych poleceń interfejsu JTAG.



Schemat blokowy układu interfejsu JTAG

Ostatnia metoda wykorzystuje **interfejs USB** do nawiązania komunikacji między mk a komputerem PC i rozpoczęcia programowania mk (np. mk AT91SAM7S firmy Atmel z 32-bitowym rdzeniem ARM7 TDMI). W tym przypadku, w dodatkowej pamięci programu mk znajduje się program bootujący. Po inicjalizacji mk konfiguruje on szeregowy port Debug Unit (DBGU) i port USB. Następnie uruchamiany jest program SAM-BA Boot, który oczekuje na polecenia z portu DBGU lub z interfejsu USB.

Na rys. poniżej pokazano sposób wejścia w tryb programowania z wykorzystaniem portu USB. Jak widać z rys. aby wprowadzić mk w tryb programowania wymagana jest odpowiednia sekwencja znaków wprowadzanych do interfejsu USB.

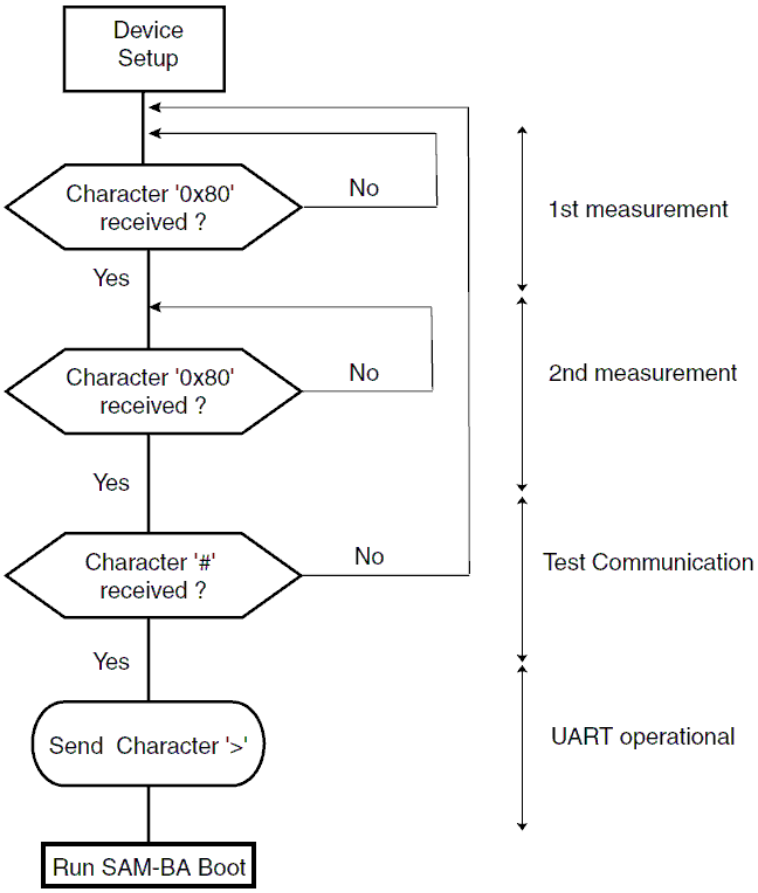


Diagram przedstawiający sposób wejścia w tryb programowania mk za pomocą interfejsu USB

Elementy asemblera.

Translator asemblera może operować na plikach tekstowych, których wiersze składają się z: **etykiet, mnemonicznych nazw instrukcji, dyrektyw preprocesora, dyrektyw asemblera oraz komentarzy**.

Translator nie rozpoznaje różnic pomiędzy dużymi i małymi literami w nazwach etykiet, symboli, instrukcji i dyrektyw asemblera (przeciwnie w przypadku dyrektyw preprocesora). Długość wiersza nie powinna przekraczać 120 znaków – możliwe jest przedłużenie po znaku kontynuacji „\\” na końcu linii.

Możliwa budowa wiersza pliku źródłowego:

[etykieta:] .dyrektywa [argument] [;komentarz]

[etykieta:] instrukcja [argument] [;komentarz]

[#dyrektywa_preprocesora] [argument] [//komentarz]

[;komentarz]

Komentarze dozwolone są w dowolnym miejscu, a ich treść jest ignorowana. Oznaczane są przez „;” lub podobnie jak w języku C przez „//” oraz „/*” i „*/”. Ograniczone są długością wiersza (do 120 znaków).

Etykiety mogą poprzedzać każdą instrukcję lub dyrektywę asemblera, ich nazwa nie może przekroczyć 31 znaków i kończy się dwukropkiem. Translator przypisuje etykiecie adres instrukcji w pamięci programu przed którą etykietę ustawiono lub adres zmiennej w pamięci danych oraz stałych w pamięci programu lub EEPROM.

Wyrażenia, na których operują instrukcje i dyrektywy mogą być budowane z elementów prostych, operatorów i funkcji.

Elementami prostymi mogą być liczby o rozdzielczości nie większej niż 64 bity i mogą je stanowić:

- wartości całkowite wprowadzane w czterech postaciach oraz znaki ASCII,
- wartości zmiennoprzecinkowe,
- etykiety, którym przyporządkowano adres w pamięci programu, danych lub EEPROM,
- etykiety zdefiniowane przez dyrektywy SET i EQU,
- skrót PC, któremu jest przyporządkowany adres w pamięci programu instrukcji, której PC jest argumentem.

Operatory umożliwiają tworzenie wyrażeń będących opisem operacji, które translator przeprowadza na argumentach.

Funkcje pozwalają na przeprowadzenie operacji jednoargumentowych.

Przykłady zastosowania operatorów i funkcji do budowy wyrażeń:

Poczatek:

ldi R16, low(500) *; ładuj R16 młodszym bajtem wartości 500*

*ldi R17, high(Poczatek*2)* *; ładuj R17 starszym bajtem dwukrotnie zwiększonego*

; adresu poprzedniej instrukcji

ldi R18, (1<<2)|(1<<5) *; ładuj R18 wartością 0b00100100*

Podstawowe dyrektywy asemblera

Służą do informowania programu asemblującego o typie docelowego mikrokontrolera, sposobie zagospodarowania pamięci oraz wspomagają pisanie programu dzięki makrom i mechanizmom translacji warunkowej. Nazwą dyrektywy poprzedza się kropką i ewentualnie etykietą.

.DEVICE informuje translator o typie układu dla którego pisany jest program, np.

.DEVICE ATmega32

Biblioteki (tj. fragmenty programu, np. podprogramy, definicje i makra) przechowywane mogą być w osobnych plikach, dołączanych dyrektywą **.INCLUDE**, której argumentem jest nazwa pliku zewnętrznego. Program asemblujący w pierwszym etapie translacji zastępuje omawianą dyrektywę zawartością wybranego pliku.

.INCLUDE ”nazwa.plk”

Dyrektywy **.LIST, .NOLIST, .LISTMAC** sterują generowaniem listingów z przebiegu translacji.

Dyrektywa **.EXIT** powoduje zakończenie translacji pliku źródłowego, pozostałe instrukcje i dyrektywy są ignorowane. Jeśli dyrektywa ta wystąpi w pliku dołączanym to translator powróci do przetwarzania pliku głównego.

Definicje

.EQU i **.SET** to dyrektywy umożliwiające przypisanie określonej wartości liczbowej do definiowanej etykiety. .EQU ma działanie jednorazowe – raz zdefiniowana etykieta nie może być deklarowana ponownie. .SET pozwala natomiast na wielokrotne przedefiniowanie danej etykiety.

Przykład:

.EQU sala = 9 *; „stala” ma wartość 9 na przestrzeni całego programu*
.SET zmienna = 10
…… *; tutaj „zmienna” ma wartość 10*

.SET zmienna = 11

…… ; tutaj „zmienna" ma wartość 11

Dyrektywa **.DEF** umożliwia przypisanie rejestrom nazw symbolicznych, co można odwołać przez dyrektywę **.UNDEF** i ponowną definicję.

Przykład:

```
.DEF    akumulator = R16
…
.UNDEF  akumulator
.DEF    akumulator = R17
….
```

Segmenty programu

Poza tłumaczeniem języka mnemonicznego na kod maszynowy translator ułatwia gospodarowanie pamięcią danych i EEPROM. Dyrektywami określającymi obszar, na którym w danym momencie operujemy, są: **.CSEG** (pamięć programu – segment domyślny), **.DSEG** (pamięć danych) oraz **.ESEG** (pamięć EEPROM).

Liczniki adresowe translatora działają automatycznie; można to zmienić dyrektywą **.ORG**.

Przykład:

```
.DSEG                ; dalsza część programu dotyczy pamięci danych
.ORG   0x0060        ; ustal licznik adresu w p. danych na początek pamięci SRAM
Zmienna1:
.BYTE 2              ; zadeklaruj zmienną w pamięci danych o długości 2 bajtów
                     ; etykiecie „Zmienna1" odpowiada wartość 0x0060
Zmienna2:
.BYTE 1              ; zadeklaruj zmienną w pamięci danych o długości 1 bajtu
                     ; etykiecie „Zmienna2" odpowiada wartość 0x0062
.ESEG                ; dalsza część programu dotyczy pamięci EEPROM
Stale_E:     .DB 134, 35, -30, 0xFF, 034
                     ; zadeklaruj ciąg stałych w pamięci EEPROM
                     ; etykiecie „Stale_E" odpowiada adres stałej 134
                     ; który translator dobrał automatycznie (na 0)
.CSEG                ; dalsza część programu dotyczy pamięci programu
.ORG  0x0000         ; ustal licznik adresu w p. programu na wektor zerowania
rjmp  Zerowanie      ; skocz do instrukcji następnej po etykiecie „Zerowanie"

.ORG 50              ; ustal licznik adresu w p. programu na 50
Stale_P:  .DW  0xABCD, 0xFFAA, 2346
                     ; zadeklaruj ciąg stałych w pamięci programu
                     ; etykiecie „Stale_P" odpowiada wartość 50, która jest
                     ; adresem stałej 0xABCD w pamięci programu
Zerowanie:           ; Uwaga: adresacja słów 16-bitowych w pamięci programu
```

Stałe w pamięci programu i EEPROM

Deklaruje się z dyrektywą **.DB** (8 bitów), **.DW** (16 bitów), **.DD** (32 bity), **.DQ** (64 bity).

Deklaracja **łańcuchów znakowych** możliwa jest z użyciem dyrektywy .DB. Kod ASCII podajemy w cudzysłowie.

Przykład:

```
.ESEG
tekst:  .DB     ”tekst zakończony zerem”, 0
```

Makra

Makrami nazywa się struktury predefiniujące bloki programowe o zmiennych parametrach. Umieszczenie wywołania danego makra (jego nazwy i argumentów) w głównej części programu sprawia, że translator zastępuje nazwę makra jego rozwinięciem, z uwzględnieniem zadanych parametrów.

Makra asemblera definiuje się na początku programu. Zawartość każdego z nich umieszcza się między dyrektywami **.MACRO** (z argumentem będącym nazwą makra) i **.ENDMACRO** (lub **.ENDM**). W ich wnętrzu można wykorzystać maksymalnie 10 parametrów, których wartość reprezentowana jest przez złożenie znaku „@” z numerem parametru.

Przykład:

```
.MACRO dodaj_16b   ; początek definicji makra
add  @0, @2        ; dodaj parametr drugi do zerowego
adc  @1, @3        ; dodaj z przeniesieniem parametr trzeci do pierwszego
.ENDMACRO          ; koniec definicji makra

dodaj_16b R0, R1, R16, R17      ; wywołanie makra – para rejestrów R17:R16
                                ; zostanie dodana do pary R1:R0
```

Translacja warunkowa

Asemblacja warunkowa pozwala np. przygotować wersje programów bibliotecznych dla różnych procesorów. Wykorzystywane są dyrektywy **.IF**, **.ENDIF**, **.ELSE**, **.ELIF**.

W trakcie translacji istnieje możliwość wyświetlania komunikatów, co bywa przydatne szczególnie przy wykorzystywaniu mechanizmów warunkowych. Służy do tego dyrektywa **.MESSAGE**, której argumentem powinien być ujęty w cudzysłów komunikat. Dyrektywa **.ERROR** wywołuje natomiast błąd translacji, a jej argumentem musi być komunikat błędu.

Przykład translacji warunkowej i komunikatów:

```
.EQU  mikrokontroler = 2      ; zdefiniuj etykietę „mikrokontroler” o wartości równej 2,
                              ; czemu odpowiadać będzie wybór mikrokontrolera ATmega16
.SET  komunikaty = 1          ; zdefiniuj etykietę o nazwie „komunikaty”
                              ; (fakt istnienia tej definicji powoduje wyświetlanie komunikatów)
.IF mikrokontroler == 1
.DEVICE ATMega32
        .IFDEF komunikaty
                .MESSAGE   ”Mikrokontroler ATmega32”
        .ENDIF
        .ELIF  mikrokontroler == 2
                .DEVICE ATmega16
        .IFDEF komunikaty
                .MESSAGE   ”Mikrokontroler ATmega16”
        .ENDIF
.ELSE
        .IFDEF komunikaty
                .MESSAGE   ”Mikrokontroler ATmega32”
        .ENDIF
.ENDIF
```

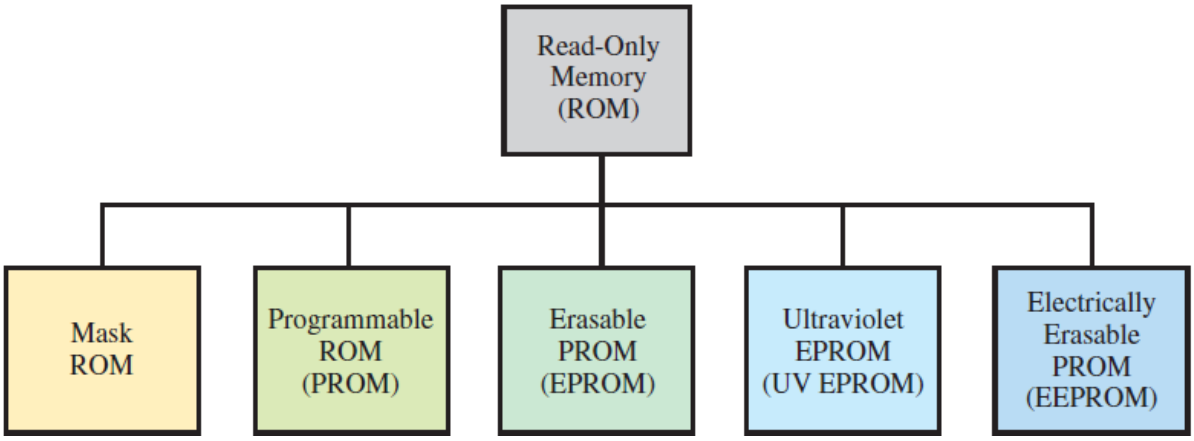
Pliki nagłówkowe

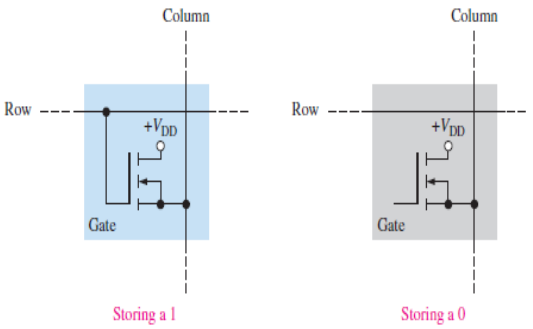
Wraz z programem asemblującym dostarczane są pliki nagłówkowe dla wszystkich produkowanych układów AVR. Mają one rozszerzenie **.inc**, np. m32def.inc.

Każdy taki plik zawiera odpowiednią dyrektywę .DEVICE oraz definicje (.EQU) nazw symbolicznych następujących wartości:

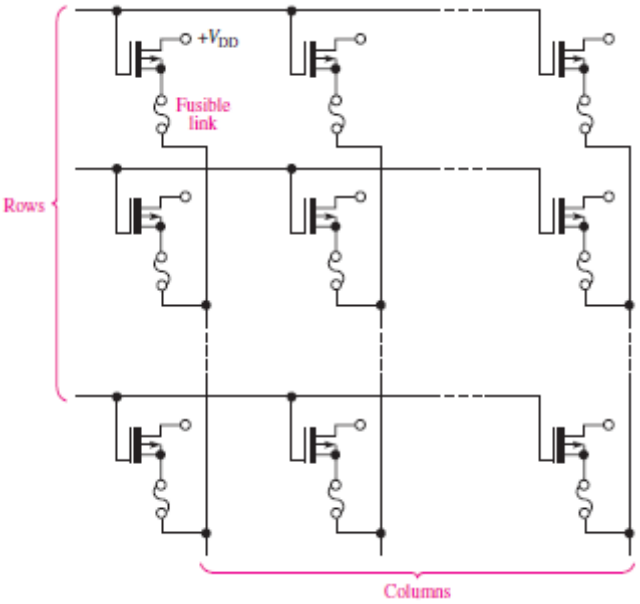
- adresów rejestrów funkcyjnych (np. SREG=0x3F),
- numerów bitów w rejestrach funkcyjnych (np. PINA0=0),
- adresów końca obszarów pamięci (np. RAMEND=0x45F),
- wektorów przerwań (np. INT0addr=0x002).

EEPROM pamięć danych

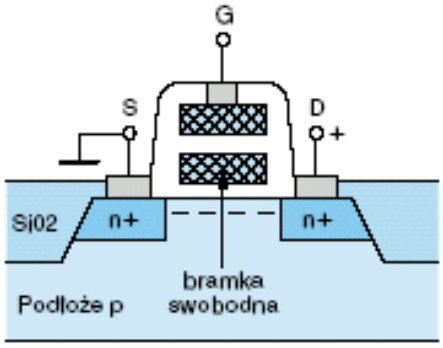




Komórki pamięci ROM



Komórki pamięci PROM



Komórka pamięci EEPROM

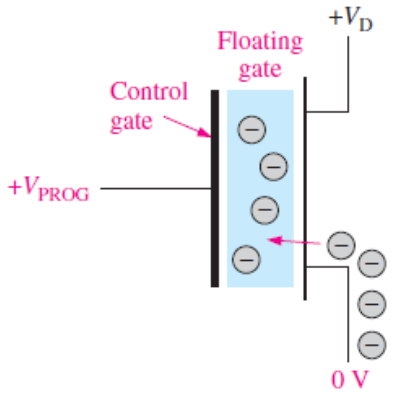
•EPROM (Electrically Programmable Read-Only Memory) lub (Erasable PROM) – jest pamięcią stałą, która umożliwia kasowanie danych za pomocą ładunków elektrycznych. Jest to kasowalna pamięć tylko do odczytu – to rodzaj nieulotnej pamięci typu ROM zawartej w układzie scalonym, który może być programowany i przeprogramowywany za pomocą specjalnego urządzenia elektronicznego tzw. programatora. Mikroukłady EPROM

umieszczane są w przeźroczystej obudowie, tak aby jej zawartość mogła być kasowana poprzez naświetlanie promieniami ultrafioletu. Mikroukłady EPROM są najczęściej stosowane do przechowywania danych, które najprawdopodobniej nie będą już nigdy zmieniane, na przykład BIOS.

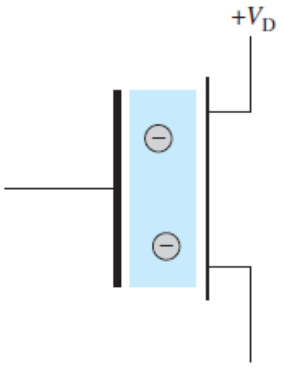
• EEPROM (Electrically Erasable Programmable Read-Only Memory) – jest to elektrycznie kasowalna i programowalna pamięć tylko do odczytu - rodzaj stałej pamięci której zawartość można wymazać i ponownie zaprogramować przez przyłożenie napięcia elektrycznego do układów pamięciowych, a następnie wpisanie w nie nowych instrukcji .

Programowanie komórek EEPROM.

Wstępnie zawartość wszystkich komórek jest kasowana przez usunięcie ładunków z obszarów pływających bramek. Oznacza to, że wszystkie komórki pamiętają stan 1. Następnie w komórkach, które mają pamiętać stan 0 wprowadza się ładunek ujemny (elektrony) w obszarze pływających bramek.

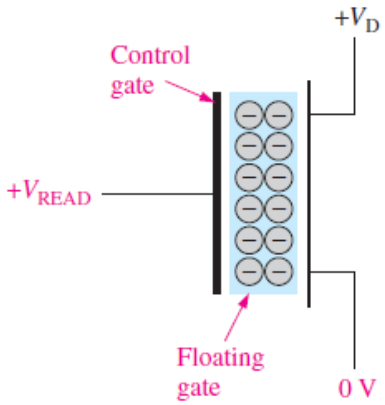


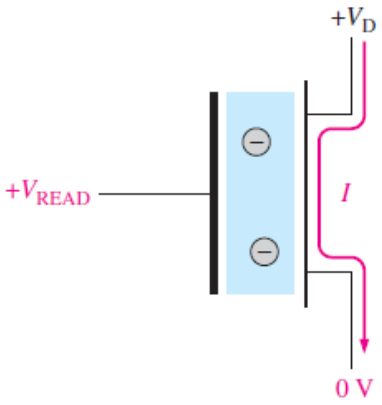
Komórka w stanie 0



Komórka w stanie 1

Operacja odczytu zawartości komórki EEPROM





Podczas operacji odczytu tranzystor pozostaje zatkany, jeśli komórka pamięta 0 lub przewodzi, jeśli komórka pamięta 1.

ATmega256 zawiera 4k bajty pamięci danych EEPROM. Jest ona zorganizowana jako oddzielna przestrzeń pamięci, w której oddzielny bajt może być zapisywany lub odczytywany. EEPROM wytrzymuje ponad 100 000 cykli zapisu i wymazania. Pamięć EEPROM może być zapisywana także wraz z pamięcią programu, nawet za pomocą programatora ISP. Można w niej przechowywać np. dane pomiarowe, zmienne konfiguracyjne, które nie mogą ulec skasowaniu po wyłączeniu zasilania lub w trakcie resetu mikrokontrolera.
Dostęp pomiędzy EEPROM i jednostką centralną jest opisany poniżej z wyszczególnieniem Rejestru Adresowego EEPROM, Rejestru Danych EEPROM i Rejestru Sterowania EEPROM. Rejestry te są dostępne w przestrzeni adresowej I/O procesora.

Operacje odczytu/zapisu EEPROM

Odczyt z pamięci EEPROM przebiega według następującej procedury:

- do rejestrów adresowych wpisuje się adres bajtu w pamięci EEPROM, spod którego chcemy pobrać daną,
- ustawiamy odpowiedni bit uruchamiający proces odczytu w rejestrze sterującym,
- czekamy, aż ustawi się odpowiednia flaga informująca o zakończeniu odczytu,
- w rejestrze danych znajduje się nasza dana.

Natomiast zapis danej do pamięci EEPROM przebiega według następującej procedury:

- do rejestrów adresowych wpisuje się adres bajtu w pamięci EEPROM, do którego chcemy wpisać daną,
- do rejestru danych wprowadzamy naszą daną,
- ustawiamy odpowiedni bit uruchamiający proces zapisu do EEPROM (często należy również ustawić bity odblokowujące zapis),
- czekamy, aż ustawi się odpowiednia flaga informująca o zakończeniu zapisu (zapis trwa od 2 do 4ms).
- Najczęściej procedury zapisu danych do pamięci EEPROM najpierw kasują zawartość docelowej komórki, a następnie wpisują do niej naszą daną. Gdy tak nie jest to należy pamiętać, aby przed zapisem do EEPROM skasować zawartość tej komórki (poprzez ustawienie odpowiednich bitów w rejestrze sterującym). Kasowanie pamięci EEPROM (podobnie jak FLASH) polega na ustawieniu w ich poszczególnych komórkach samych jedynek.

Podczas odczytu z EEPROM jednostka centralna jest zatrzymywana na cztery cykle maszynowe zanim następny rozkaz zostanie wykonany. Kiedy EEPROM jest zapisywany jednostka centralna jest zatrzymywana na dwa cykle maszynowe przed wykonaniem następnej instrukcji.

Opis rejestrów związanych z obsługą pamięci EEPROM.

Rejestr Adresu EEPROM – EEARH, EEARL

EEARH	0x22	-	-	-	-	EEAR11	EEAR10	EEAR9	EEAR8
EEARL	0x21	EEAR7	EEAR6	EEAR5	EEAR4	EEAR3	EEAR2	EEAR1	EEAR0

- bity15..12 – Res: Bity zarezerwowane

Te bity są zarezerwowane i zawsze będą odczytywane jako 0. Podczas zapisu pod te adresy należy przypisać tym bitom zera dla kompatybilności z przyszłymi urządzeniami.

- bity 11..0 – **EEAR11..0**: Adresy EEPROM

Rejestry adresu EEPROM – EEARL, EEARH – specyfikują adres w 4k przestrzeni adresów EEPROM. Bajty danych w EEPROM są ułożone linearnie pomiędzy 0 i 4096. Początkowa wartość EEAR jest niezdefiniowana. Jednak odpowiednia wartość musi zostać wpisana nim EEPROM będzie dostępny.

Rejestr danych EEPROM – EEDR

EEDR	0x20	EEDR7	EEDR6	EEDR5	EEDR4	EEDR3	EEDR2	EEDR1	EEDR0
		MSB							LSB

Wszystkie bity są zarówno do zapisu jak i odczytu. Wszystkie bity są inicjowane wartości logiczną 0.

- bity 7..0 **EEDR7..0** : dane EEPROM

Dla operacji zapisu w EEPROM, rejestr EEDR zawiera dane, które mają zostać zapisane pod adres podany w rejestrze EEAR. Przy operacji odczytu z EEPROM rejestr EEDR zawiera odczytaną daną z pod adresu podanego przez rejestr EEAR.

Rejestr kontroli EEPROM – EECR

EECR	0x1F	-	-	EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE

- bity 7..6 – Res: bity zarezerwowane

Te bity są bitami zarezerwowanymi i będą zawsze odczytywane jako zero.

- bity 5..4 – **EEPM1 i EEPM0**: EEPROM Programming Mode

EEPROM Mode Bits

EEPM1	**EEPM0**	**Programming Time**	**Operation**
0	0	3.4ms	Erase and Write in one operation (Atomic Operation)
0	1	1.8ms	Erase only
1	0	1.8ms	Write only
1	1	–	Reserved for future use

- bit 3 – **EERIE**: **EEPROM Ready Interrupt Enable**

Odblokowanie zgłoszenie przerwania po gotowości pamięci EEPROM.
Wpisanie jedynki umożliwia obsługę przerwania jeżeli I-bit w SREG jest ustawiony. Wpisanie zera maskuje przerwania. Bit gotowości przerwania EEPROM generuje stałe przerwanie kiedy EEPE jest wyzerowane.

- bit 2 – **EEMPE**: **EEPROM Master Programming Enable**

Bit ten determinuje czy ustawienie EEWE na jeden powoduje zapis do EEPROM. Jeżeli EEMWE jest ustawione na jeden to zapis jedynki do EEWE w przeciągu czterech cykli maszynowych spowoduje zapis danej do EEPROM pod wybranym adresem. Jeżeli EEMPE jest wyzerowane to wpisanie jedynki do EEWE nie będzie miało żadnego efektu. Wpisanie

jedynki do EEMPE przez oprogramowanie, sprzęt wyzeruje tą wartość po czterech cyklach maszynowych.

- bit 1 – **EEWE**: Bit umożliwiający zapis do EEPROM

Kiedy adres i dane są dobrze ustawione bit ten musi być ustawiony aby zapisać wartość do EEPROM. EEMWE musi być ustawione kiedy logiczna jedynka jest wpisywana do EEWE, w przeciwnym wypadku nie nastąpi zapis do EEPROM. Następująca procedura powinna być przestrzegana podczas zapisu do EEPROM:

1. Czekać dopóki EEWE stanie się zerem
2. Czekać, aż SPMEN w SPMCR stanie się zerem
3. Zapisać nowy adres do EEAR – opcjonalne
4. Zapisać nową daną do EEDR – opcjonalne
5. Zapisać logiczną jedynkę do EEMPE podczas zapisu zera do EEPE w EECR
6. W czasie czterech cykli maszynowych po ustawieniu EEMPE zapisać logiczną jedynkę do EEPE.

EEPROM nie może być programowany podczas zapisu CPU do pamięci Flash. Oprogramowanie musi sprawdzić czy programowanie Flash jest skończone nim zacznie przygotowywać zapis do EEPROM. Krok 2 jest zależny od tego czy oprogramowanie zawiera bootloadera pozwalającego jednostce centralnej programować Flash. Jeżeli Flash nigdy nie jest aktualizowane przez CPU krok 2 może zostać pominięty.

UWAGA: Przerwanie między 5 i 6 krokiem spowoduję, że zapis będzie nieudany, ponieważ upłynie czas dla wzorcowego bitu umożliwiającego zapis. Jeśli obsługa przerwania mająca dostęp do EEPROM przerywa inny proces mający dostęp do EEPROM rejestry EEAR i EEDR zostaną zmodyfikowane powodując błąd przy dostępie do EEPROM. Zaleca się, aby globalną flagę przerwań zerować przy wykonywaniu czterech ostatnich kroków w celu uniknięcia tych problemów.
Kiedy czas na zapis upłynął bit EEWE jest zerowany przez sprzęt. Oprogramowanie użytkownika może sprawdzać ten bit w oczekiwaniu pojawienia się tam zera przed zapisem następnego bajtu. Kiedy EEWE jest ustawiony, jednostka centralna jest wstrzymywana na dwa cykle i nim następna instrukcja zostanie wykonana.

- bit 0 – **EERE**: bit umożliwiający odczyt z EEPROM

Kiedy odpowiedni adres jest ustawiony w rejestrze EEAR bit EERE musi zostać ustawiony na logiczne jeden aby spowodować odczyt z EEPROM. Odczyt z EEPROM zajmuje jedną instrukcję i żądana dana jest dostępna od razu. Kiedy następuje odczyt z EEPROM jednostka centralna jest wstrzymywana na cztery cykle nim zostanie wykonana następna instrukcja.
Użytkownik powinien sprawdzać bit EEPE przed rozpoczęciem operacji odczytu. Jeśli jest wykonywana instrukcja zapisu, to nie można ani odczytać EEPROM, ani zmienić danych w rejestrze EEAR.

Następujące przykłady kodu pokazują funkcje zapisujące do EEPROM. Przykłady zakładają, że przerwania są kontrolowane tak, że nie zajdzie żadne przerwanie podczas wykonywania tych funkcji. Przykłady te zakładaj ą również, że nie ma flash boot loadera obecnego w oprogramowaniu. Jeśli taki kod jest obecny, funkcje zapisu muszą również oczekiwać na rozkaz zakończenia z SPM.

Kod w asemblerze:

EEPROM_write:

```
;czekaj, aż zostanie zakończony zapis poprzednich danych
        sbic EECR, EEPE
        rjmp EEPROM_write
;ustaw adresy r18:r17 w rejestrze adresów
        out EEARH ,r18
        out EEARL, r17
; zapisz dane do rejestru r16
        out EEDR, r16
; ustaw EEMPE na jeden
        sbi EECR, EEMPE
;rozpocznij zapis
        sbi EECR, EEPE
        ret
```

Kod w języku C:

```
void EEPROM_write(unsigned int uiAddress, unsigned char ucData)
{
        while(EECR & (1<<EEPE ))
        EEAR = uiAddress;
        EEDR =ucData;
        EECR |= (1<<EEMPE);
        EECR |= (1<<EEPE);
}
```

Następne przykłady kodu przedstawiają funkcje odczytujące z EEPROM. Przykłady te zakładają, że przerwania są kontrolowane i że nie zajdzie żadne podczas obsługi tych funkcji.

Kod w Asemblerze:

EEPROM_read;

```
// czekaj na zakończenie wcześniejszego zapisu
        sbic EECR, EEWE
        rjmp EEPROM_read
// ustaw rejestr adresów
        out EEARH, r18
        out EEARL, r17
//rozpocznij odczyt
        sbi EECR, EERE
// odczytaj dane z rejestru danych
        in r16, EEDR
        ret
```

Kod w C:

```
unsigned char EEPROM_read(unsigned char uiAddress)
{
while(EECR & (1<<EEPE))
;
EEAR= uiAddress;
        EECR |= (1<<EERE);
        return EEDR;
}
```

Zapobieganie zniekształceniom EEPROM

W okresach kiedy jest niski stan napięcia dane w EEPROM mogą ulec zniekształceniu ponieważ CPU i EEPROM mogą działać niepoprawnie. Normalna sekwencja zapisu do EEPROM wymaga minimalnego napięcia aby przebiegać poprawnie. Po drugie, jednostka centralna sama w sobie może wykonywać rozkazy niepoprawnie przy zbyt niskim napięciu.

Błędów w danych zapisanych na EEPROM można łatwo uniknąć postępując w następujący sposób:

Należy utrzymywać AVR RESET w stanie aktywnym (stan niski) w okresach niewystarczającego zasobu mocy. To może zostać zapewnione przez odblokowanie wewnętrznego detektora zużycia energii elektrycznej (BOD). Jeżeli poziom wykrycia nie odpowiada wymaganemu poziomowi zewnętrzny obwód ochrony przed resetem może zostać wykorzystany. Jeżeli podczas operacji zapisu procesor zostanie zresetowany to operacja zapisu zostanie dokończona po zapewnieniu odpowiedniego zasobu napięcia.

Programowanie pamięci EEPROM w Arduino

Warunkiem korzystania z pamięci EEPROM w szkicach jest wywołanie biblioteki EEPROM (dołączonej do środowiska Arduino IDE) za pomocą następującego wyrażenia:

```
#include <EEPROM.h>
```

Zapisanie wartości w pamięci EEPROM wymaga już tylko jednego prostego wyrażenia:

```
EEPROM.write(a, b);
```

Parametr a reprezentuje pozycję (z przedziału od 0 do 1023), na której dana wartość ma zostać zapisana, natomiast parametr b to zmienna zawierająca bajt danych, który ma być zapisany w pamięci EEPROM na wskazanej pozycji.
Aby uzyskać dane z pamięci EEPROM, należy użyć wyrażenia w postaci:

```
value = EEPROM.read(position);
```

Powyższe wyrażenie odczytuje dane zapisane w pamięci EEPROM na pozycji position i zapisuje je w zmiennej value.
Żywotność pamięci EEPROM jest dość ograniczona, zatem pamięć po pewnej liczbie operacji zapisu przestanie działać! Według deklaracji producenta (firmy Atmel) pamięć EEPROM platformy Arduino obsługuje maksymalnie 100 tys. cykli zapisu i odczytu na każdej pozycji. Operacje odczytu nie wpływają na czas działania tej pamięci.

```
// Listing 1. Szkic demonstrujący działanie pamięci EEPROM
#include <EEPROM.h>
int zz;
void setup()
{
 Serial.begin(9600);
 randomSeed(analogRead(0));
 Serial.println("Zapisywanie liczb losowych...");
 for (int i = 0; i < 1024; i++)
 {
  zz = random(255);
  EEPROM.write(i, zz);
 }
 Serial.println();
 for (int a = 0; a < 1024; a++)
 {
  zz = EEPROM.read(a);
  Serial.print("Pozycja w pamięci EEPROM: ");
  Serial.print(a);
  Serial.print(" zawiera wartość ");
  Serial.println(zz);
  delay(25);
 }
}
void loop() {}
```