Struktura programu w asemblerze mikrokontrolera 8051

Program w asemblerze, dający ten sam kod wynikowy, może być napisany na wiele sposobów. Źle napisany program po pewnym czasie (a być może już w czasie jego uruchamiania) będzie nieczytelny nawet dla samego autora, będzie więc programem "jednokrotnego użytku". Dobry program, napisany w sposób czytelny i zrozumiały, będzie łatwy do uruchomienia a jego modyfikacja lub rozbudowa będzie możliwa nawet po długim czasie. Będzie on mógł być również łatwo wykorzystany, w całości lub we fragmentach, przez innego programistę. Poniżej pokażemy na co warto zwrócić uwagę przy pisaniu programów w asemblerze mikrokontrolera 8051. Pokazane przykłady odnoszą się do programowania przy użyciu pakietu uVision firmy Keil.

Sposób podejścia do programowania zależy w pewnym stopniu od wielkości projektu. Stosowanie niektórych zasad jest ważniejsze przy większych projektach, jednak warto przyzwyczaić się do ich stosowania nawet przy pisaniu małych programów.

1. Zapis programu

Program w asemblerze składa się z instrukcji sterujących przebiegiem tłumaczenia programu (np. $INCLUDE), dyrektyw asemblera (np. END) oraz rozkazów procesora (zapisanych w postaci odpowiednich symboli mnemonicznych, np. MOV) z ewentualnymi parametrami. W programie mogą również występować komentarze, umieszczone po znaku średnika. Tekst po średniku jest ignorowany. Jednemu rozkazowi procesora odpowiada jeden wiersz programu w postaci:

```
[etykieta:] rozkaz    [parametr_1], [parametr_2], [parametr_3]          [; komentarz]
```

```
test:       CJNE    A, #10, skok        ; sprawdź czy A równe 10
```

Kody rozkazów jak też inne nazwy symboliczne w programie mogą być pisane małymi lub dużymi literami, zależy to tylko od osobistych przyzwyczajeń programisty. Dobrze jest jednak przyjąć pewne konwencje i stosować je konsekwentnie. Często przyjmuje się, ze dyrektywy asemblera pisane są dużymi literami, podobnie jak nazwy rejestrów (np. A, B, DPTR).

Zalecane jest wyrównanie programu w kolumnach. Kod rozkazu piszemy po znaku tabulacji, tak aby pozostawić miejsce na ewentualne etykiety. Jeśli często stosowane będą długie etykiety, to można użyć więcej niż jednej tabulacji. Jeśli wyjątkowo długa etykieta nie mieści się w pierwszym polu, to może ona tworzyć samodzielną linię. Komentarze powinny być również wyrównane w kolumnach. Warto również zastosować tabulację pomiędzy kodem rozkazu i ewentualnymi parametrami. Dzięki temu łatwiej będzie znaleźć konkretny rozkaz (zwykle przeglądając program poszukujemy określonego typu rozkazu, dopiero w drugiej kolejności sprawdzamy jego parametry).

Przykład:

Program nieczytelny:

```
MOV A,B ; komentarz 1
JNZ skok_1
dluga_etykieta: CJNE R7, #10, skok_2 ; komentarz 2
skok_1: INC zmienna
skok_2: INC DPTR
```

Ten sam program zapisany lepiej:

```
            MOV     A, B                ; komentarz 1
            JNZ     skok_1
dluga_etykieta:
            CJNE    R7, #10, skok_2     ; komentarz 2
skok_1:     INC     zmienna
skok_2:     INC     DPTR
```

Stosując nazwy symboliczne warto je dobierać tak, aby odzwierciedlały one funkcję pełnioną przez stałą lub zmienną którą oznaczają. Nazwy składające się z dwóch wyrazów można oddzielić znakiem podkreślenia dla polepszenia ich czytelności:

```
            bit1, bit2                  ; takie nazwy niewiele mówią
            zmienna1, zmienna2
            stala1, stala2

            stan_wejscia, status        ; zdecydowanie lepiej
            licznik_impulsow, adres
            opoznienie, okres
```

Komentarze powinny tłumaczyć co dzieje się w danym fragmencie programu. Nie powinny one powtarzać informacji która jest zawarta w samym zapisie rozkazu. Taki komentarz jak w przykładzie poniżej nie wnosi żadnej nowej informacji:

```
            MOV     R7, #0              ; wpisanie wartości 0 do rejestru R7
```

W poprawionej wersji komentarz wyjaśnia, że rejestr R7, pełniący funkcję licznika, jest zerowany (jego wartość będzie zapewne potem zwiększana przy zliczaniu impulsów):

```
            MOV     R7, #0              ; zerowanie licznika impulsów
```

Ważną sprawą jest używanie podprogramów (procedur). Podprogram powinien być dobrze opisany, tak aby można było go później użyć bez konieczności analizowania kodu. Dla podprogramów z parametrami należy podać sposób ich przekazywania lub zwracania. Warto również podać jakie rejestry są używane w podprogramie aby można było to uwzględnić przy wywołaniu podprogramu. Ilustruje to poniższy przykład:

```
;-------------------------------------------------------------------------------------------
;  Podprogram określa liczbę bitów niezerowych w bajcie
;  Wejście:         A - bajt w którym będą zliczane bity
;  Wyjście:         A - liczba bitów niezerowych
;  Używane:         R7
;-------------------------------------------------------------------------------------------
count_bits:
            MOV     R7, #0              ; zerowanie licznika bitów
loop:       CLR     C                   ; przeniesienie zastępuje najstarszy bit w A
            RRC     A                   ; CY - testowany bit
            JNC     zero                ; bit zerowy, licznik bez zmian
            INC     R7                  ; inkrementacja licznika jedynek
zero:       JNZ     loop                ; powtarzanie jeśli pozostały jeszcze bity niezerowe
            MOV     A, R7               ; licznik zwracany w A
            RET
```

W typowym przypadku podprogram jest fragmentem kodu wykorzystywanym wielokrotnie (być może z różnymi parametrami). Warto jednak używać podprogramów (nawet jeśli miałyby być one wywołane tylko jednokrotnie) dla polepszenia czytelności programu.

Szczególnym przypadkiem podprogramu jest obsługa przerwania. Powinna być ona zakończona rozkazem RETI (zamiast RET), należy również zadbać, aby wszystkie używane

rejestry zostały przechowane na stosie (dla rejestrów R0 - R7 można ewentualnie wykorzystać przełączanie banku rejestrów). Należy zwłaszcza pamiętać o przechowaniu rejestru PSW jeśli którykolwiek z jego bitów (najczęściej CY) będzie modyfikowany.

2. Wykorzystanie dyrektyw asemblera do organizacji kodu

W najprostszym przypadku jedyną niezbędną dyrektywą asemblera jest kończąca program dyrektywa END. W rzeczywistości program zostanie przetłumaczony nawet bez dyrektywy END, wystąpi jednak ostrzeżenie o jej braku. Jeśli w programie nie ma żadnych dyrektyw określających segment i położenie wewnątrz niego, to asembler przyjmuje domyślnie segment kodu a licznik położenia zostanie zainicjalizowany wartością 0000h (adres od którego procesor rozpoczyna wykonywanie programu po restarcie). Tak więc najprostszy wariant zapisu programu to:

```
MOV    A,R7          ; rozkaz ten umieszczony będzie w domyślnej pamięci kodu pod
                       adresem 0000h
END
```

Bardziej czytelny jest program, w którym używając dyrektywy ORG, ustalamy położenie rozkazu w pamięci kodu:

```
ORG    0             ; dyrektywa ORG ustala przesunięcie w domyślnej pamięci kodu
MOV    A, R7
END
```

Jeszcze lepszym wariantem jest użycie dyrektywy CSEG, dzięki której widać wprost, że program umieszczony jest w pamięci kodu, pod określonym adresem:

```
CSEG   AT 0          ; dyrektywa CSEG AT wybiera segment absolutny kodu
                       rozpoczynający się od adresu 0000h
MOV    A, R7
END
```

Kolejnym przypadkiem (poza początkiem programu), w którym konieczne jest umieszczenie fragmentu kodu pod określonym adresem jest obsługa przerwań. Sygnał przerwania (jeśli jest ono odblokowane) powoduje automatyczne wywołanie podprogramu umieszczonego pod określonym adresem, różnym dla każdego ze źródeł przerwań. Pierwszy adres przerwania (przerwanie zewnętrzne EX0) to 0003h, kolejne rozmieszczone są w odstępie 8 bajtów. Zwykle cała obsługa przerwania nie mieści się w 8 bajtach, dlatego pod zadanym adresem umieszcza się tylko skok do obsługi przerwania umieszczonej w innym miejscu:

```
CSEG   AT 0          ; początek programu głównego pod adresem 0000h
LJMP   start         ; skok do początku programu (ominięcie obsługi przerwań)

CSEG   AT 03h        ; pod adresem 03h początek obsługi przerwania EX0
LJMP   int_ex_0      ; skok do właściwej obsługi przerwania

CSEG   AT 0Bh        ; pod adresem 0Bh początek obsługi przerwania Timera 0
LJMP   int_timer_0   ; skok do właściwej obsługi przerwania
```

Jeśli dysponujemy ciągłą przestrzenią pamięci kodu, to właściwie cały program mógłby być umieszczony w segmencie absolutnym, rozpoczynającym się od adresu 0000h. Warto jednak wprowadzić dodatkowe segmenty relokowalne (a jeśli to konieczne absolutne), w których będą umieszczane pewne podobne funkcjonalnie fragmenty kodu. Można na przykład wprowadzić osobne segmenty dla programu głównego, obsługi przerwań czy też podprogramów.

W przypadku typowej architektury sprzętu taki podział służy jedynie do uporządkowania kodu i ułatwia analizę programu. W bardziej złożonych przypadkach, na przykład przy stosowaniu przełączanych banków pamięci, może on być konieczny. Można sobie wyobrazić taką sytuację, że pewne fragmenty programu (np. zawierające obsługę przerwań lub podprogramów) będą musiały być dostępne zawsze, podczas gdy inne fragmenty kodu będą odczytywane z przełączanych banków pamięci.

W poniższym przykładzie cały program umieszczony jest w czterech segmentach:

- początek programu w segmencie absolutnym pod adresem 0000h,
- skoki do obsługi przerwań w segmencie absolutnym pod adresami 0003h i 000Bh,
- dalsza część programu głównego w segmencie relokowalnym PROG,
- obsługa przerwań w segmencie relokowalnym INT,
- procedury w segmencie relokowalnym PROC,
- stałe w segmencie relokowalnym CONST.

```
PROG    SEGMENT   CODE      ; deklaracja segmentu programu głównego
INT     SEGMENT   CODE      ; deklaracja segmentu obsługi przerwań
PROC    SEGMENT   CODE      ; deklaracja segmentu podprogramów
CONST   SEGMENT   CODE      ; deklaracja segmentu stałych

        CSEG    AT  0       ; początek programu głównego pod adresem 0000h
        LJMP    start       ; skok do kodu programu (ominięcie obsługi przerwań)

        CSEG    AT  03h     ; pod adresem 0003h początek obsługi przerwania EX0
        LJMP    int_ex_0    ; skok do właściwej obsługi przerwania

        CSEG    AT  0Bh     ; pod adresem 000Bh początek obsługi przerwania Timera 0
        LJMP    int_timer_0 ; skok do właściwej obsługi przerwania

        RSEG    PROG        ; wybór segmentu programu głównego
start:
        LCALL   delay       ; wywołanie podprogramu

        SJMP    $           ; pętla kończąca program

        RSEG    INT         ; wybór segmentu obsługi przerwań
int_ex_0:
        ...                 ; kod obsługi przerwania EX0
        RETI                ; powrót z obsługi przerwania
int_timer_0:
        ...                 ; kod obsługi przerwania Timera 0
        RETI                ; powrót z obsługi przerwania

        RSEG    PROC        ; wybór segmentu podprogramów
delay:  ...                 ; kod obsługi podprogramu delay
        RET                 ; powrót z podprogramu

        RSEG    CONST       ; wybór segmentu stałych
const_8:  DB    10          ; stała bajtowa
const_16: DW    1234h       ; stała dwubajtowa
table:  DB      1, 2, 3, 4  ; tablica 4 stałych bajtowych
text:   DB      'Napis', 0  ; stały tekst zakończony znakiem 0

        END                 ; koniec programu
```

3. Wykorzystanie dyrektyw asemblera do organizacji danych

Rozpatrzmy fragment programu, w którym używane są zmienne bitowe oraz komórki pamięci (wewnętrznej i zewnętrznej). Teoretycznie program mógłby wyglądać następująco:

```
        SETB    0
        MOV     A, 21h
        CJNE    A, 22h, skok
        MOV     DPTR, #8001h
        MOVX    @DPTR, A
skok:   CPL     1
        INC     21H
        MOV     DPTR, #8000h
        MOVX    A, @DPTR
```

Widać, że w tym programie używane są bity 0 i 1, komórki pamięci wewnętrznej o adresach 21h i 22h oraz komórka pamięci zewnętrznej o adresie 8000h. Program jest jednak zupełnie nieczytelny, trudno się zorientować co przechowywane jest w pamięci. W przypadku modyfikacji układu zmiennych w pamięci wymagane są zmiany w kodzie programu (często w wielu miejscach). Program stanie się zdecydowanie bardziej przejrzysty jeśli użyjemy dyrektywy EQU do zdefiniowania zmiennych:

```
bit_0   EQU     0
bit_1   EQU     1
var_0   EQU     21h
var_1   EQU     22h
xvar_0  EQU     8000h
xvar_1  EQU     8001h

        SETB    bit_0
        MOV     A, var_0
        CJNE    A, var_1, skok
        MOV     DPTR, #xvar_1
        MOVX    @DPTR, A
skok:   CPL     bit_1
        INC     var_0
        MOV     DPTR, #xvar_0
        MOVX    A, @DPTR
```

Zauważmy, że adresy zmiennych w dyrektywach EQU zostały podane indywidualnie dla każdej zmiennej. Jeśli zdecydujemy się przenieść cały blok zmiennych w inny obszar pamięci, to trzeba będzie wykonać wiele modyfikacji. Dlatego lepiej jest zdefiniować adresy bazowe poszczególnych bloków zmiennych, adresy zmiennych będą wówczas określone poprzez przesunięcie wewnątrz bloku:

```
base_bits   EQU     0
bit_0       EQU     (base_bits + 0)
bit_1       EQU     (base_bits + 1)

base_vars   EQU     21h
var_0       EQU     (base_vars + 0)
var_1       EQU     (base_vars + 1)

base_xvars  EQU     8000h
xvar_0      EQU     (base_xvars + 0)
xvar_1      EQU     (base_xvars + 1)
```

W przypadku tak zapisanych zmiennych, jeśli zajdzie konieczność przeniesienia całych bloków zmiennych, wystarczy zmienić definicje adresów bazowych (base_bits, base_vars,

base_xvars) a adresy poszczególnych zmiennych zostaną wtedy zmodyfikowane automatycznie.

Zdefiniowanie zmiennych w pokazany sposób nie daje jednak zabezpieczenia przed użyciem symbolu zmiennej w niewłaściwym kontekście. Asembler widzi symbol jako stałą lub adres, nie jest jednak w stanie stwierdzić czy jest to adres bitu, komórki pamięci wewnętrznej, zewnętrznej czy też adres w pamięci kodu. Nie wystąpi więc błąd w sytuacjach nieprawidłowego użycia symbolu takich jak przedstawione poniżej:

```
MOV   A, bit_0      ; zmienna bitowa użyta zamiast zmiennej 8-bitowej
SETB  var_0         ; zmienna 8-bitowa użyta zamiast zmiennej bitowej
LCALL xvar_0        ; adres w pamięci zewnętrznej użyty jako adres podprogramu (w
                      pamięci kodu)
```

Problem ten może być rozwiązany przez zastąpienie dyrektyw EQU dyrektywami takimi jak BIT, CODE, DATA, IDATA, XDATA. Służą one do przyporządkowania symbolowi adresu w obszarze określonego typu. Dzięki temu asembler może wykryć użycie adresu w niewłaściwym kontekście. Rozpatrywany przykład będzie wyglądał następująco:

```
base_bits   EQU    0
bit_0       BIT    (base_bits + 0)
bit_1       BIT    (base_bits + 1)

base_vars   EQU    21h
var_0       DATA   (base_vars + 0)
var_1       DATA   (base_vars + 1)

base_xvars  EQU    8000h
xvar_0      XDATA  (base_xvars + 0)
xvar_1      XDATA  (base_xvars + 1)
```

Najbardziej zaawansowany sposób organizacji zmiennych w pamięci to wykorzystanie segmentów. W tym przypadku nie jest konieczne przyporządkowywanie zmiennym konkretnych adresów pamięci a jedynie zdefiniowanie zmiennej w segmencie odpowiedniego typu. Rozmieszczeniem segmentów w pamięci zajmuje się program łączący (linker) który dba o to, aby nie zachodziło nakładanie się obszarów zmiennych (co przy "ręcznym" rozmieszczaniu zmiennych jest całkiem prawdopodobne). W rozpatrywanym przykładzie wariant z użyciem segmentów ma następującą postać:

```
B_DATA SEGMENT   BIT     ; deklaracja segmentu danych bitowych
D_DATA SEGMENT   DATA    ; deklaracja segmentu danych w pamięci wewnętrznej
X_DATA SEGMENT   XDATA   ; deklaracja segmentu danych w pamięci zewnętrznej

       RSEG  B_DATA      ; wybór segmentu danych bitowych
bit_0: DBIT  1           ; zarezerwowanie miejsca na dwie zmienne bitowe
bit_1: DBIT  1

       RSEG  D_DATA      ; wybór segmentu danych w pamięci wewnętrznej
var_0: DS    1           ; zarezerwowanie miejsca na dwie zmienne
var_1: DS    1

       RSEG  X_DATA      ; wybór segmentu danych w pamięci zewnętrznej
xvar_0: DS   1           ; zarezerwowanie miejsca na dwie zmienne
xvar_1: DS   1
```

Jeśli zależy nam aby segment zmiennych umieszczony był pod określonym adresem, to można zamiast segmentu relokowalnego użyć segmentu absolutnego. Załóżmy, że w powyższym przykładzie pamięć zewnętrzna dostępna jest od adresu 8000h. Musimy wówczas zadbać o to aby zmienne xvar_0 i xvar_1 były umieszczone od adresu 8000h (ewentualnie

wyżej). Możemy wówczas zrezygnować z deklaracji segmentu relokowalnego i zastosować segment absolutny w następujący sposób.

```
            XSEG  AT  8000h    ; segment absolutny od adresu 8000h
xvar_0:     DS    1            ; zarezerwowanie miejsca na dwie zmienne
xvar_1:     DS    1
```

Użycie segmentu absolutnego pod zadanym adresem może być również konieczne, jeśli w obszarze pamięci zewnętrznej zostały umieszczone rejestry układów stosowanych do obsługi urządzeń zewnętrznych (np. wyświetlacza, zegara RTC). Ich adres jest wtedy ściśle określony przez sprzętowe układy dekodowania adresów, na przykład:

```
            XSEG  AT  0FF2Ch  ; adres bazowy rejestrów kontrolera LCD
lcd_control:  DS  1           ; 0FF2Ch - adres rejestru sterującego
lcd_data_wr:  DS  1           ; 0FF2Dh - adres rejestru danych (zapis)
lcd_status:   DS  1           ; 0FF2Eh - adres rejestru statusu
lcd_data_rd:  DS  1           ; 0FF3Fh - adres rejestru danych (odczyt)
```

Jeśli położenie poszczególnych zmiennych w pamięci jest nieistotne a jedynym ograniczeniem jest brak fizycznej pamięci w pewnych obszarach (np. zewnętrzna pamięć danych dostępna jest nie od adresu 0000h ale od 8000h), to możliwy jest też wariant z używaniem tylko segmentów relokowalnych ale konieczna jest wówczas odpowiednia konfiguracja linkera, aby rozpoczął on umieszczanie segmentów od wskazanego adresu.