STUDIA INFORMATICA 2008
Volume 29 Number 3A (78)

Dariusz CABAN
Politechnika Śląska, Instytut Informatyki

MINIMALIZOWANIE ROZMIARU KODU WYNIKOWEGO PROGRAMÓW W JĘZYKU C DLA MIKROSTEROWNIKÓW RODZINY 8051

Streszczenie. Kod wynikowy programu napisanego w języku C zajmuje z reguły więcej miejsca w pamięci niż kod wykorzystującego ten sam algorytm programu w języku asemblera. Różnicę tę można zmniejszyć przez stosowanie odpowiednich konstrukcji języka wysokiego poziomu.

Słowa kluczowe: mikrosterownik, język C, kod źródłowy, kod wynikowy

DECREASING SIZE OF OBJECT CODE OF C PROGRAMS FOR 8051 FAMILY MICROCONTROLLERS

Summary. The object code of C program usually occupies more memory space than object code of assembly program using the same algorithm. Application of appropriate constructs of the high-level language can decrease this difference.

Keywords: microcontroller, C language, source code, object code

1. Wprowadzenie

Przestrzeń adresowa pamięci programu w mikrosterownikach często nie jest zbyt duża. Stąd opinia, że programów dla mikrosterowników nie należy pisać w języku wysokiego poziomu, na przykład w C, tylko w języku asemblera. Kod wynikowy programów w języku C z reguły zajmuje więcej pamięci, ale nie musi to być aż kilka – kilkanaście razy więcej, jak podano np. w [6]. Nie można zdawać się wyłącznie na dokonywaną przez kompilator optymalizację, wiele zależy od użytych w programie konstrukcji języka wysokiego poziomu.

Mikrosterowniki rodziny 8051 nadal cieszą się sporą popularnością i są oferowane w wielu wersjach. Jednak ich architektura nie sprzyja zwięzłości kodu wynikowego programów pisanych w języku C. Wynika to m. in. z małej liczby rejestrów wskaźnikowych. Brakuje też trybu adresowania pośredniego z przesunięciem, który byłby przydatny przy odwołaniach do pól struktury, gdy dysponuje się jej adresem.

W artykule podano kilka sposobów na zmniejszenie rozmiaru kodu wynikowego programów w języku Cx51, dialekcie języka C opracowanym przez firmę Keil [3]. W trakcie badań wykorzystano kompilator w wersji 6.20. Kompilator dokonywał optymalizacji kodu według kryterium minimalnego rozmiaru (ang. Favor Code Size), przy ustawionym najwyższym jej poziomie.

Autor chciałby podziękować za cenne uwagi p. Dariuszowi Zonenberg z firmy AIUT Sp. z o.o. w Gliwicach.

2. Rozmieszczenie danych w pamięci

Mikrosterowniki rodziny 8051 różnią się m.in. pojemnością wewnętrznej pamięci RAM, jej maksymalna wartość wynosi 256 bajtów. Poza nielicznymi wyjątkami można do nich dołączać zewnętrzną pamięć RAM o maksymalnej pojemności 64 kB. Twórcy kompilatorów języka C dla mikrosterowników tej rodziny wprowadzili podział pamięci RAM na obszary w zależności od tego, które tryby adresowania są używane przy odwołaniach do danych. W języku Cx51 podział ten jest następujący:

- **data** – bajty wewnętrznej pamięci RAM o adresach 00h÷7Fh (tzw. dolny blok RAM), tryb adresowania bezpośredniego, w niektórych operacjach tryb adresowania pośredniego zawartością rejestru roboczego R0 lub R1,
- **idata** – cała wewnętrzna pamięć RAM, tryb adresowania pośredniego zawartością rejestru roboczego R0 lub R1,
- **bdata** – bajty wewnętrznej pamięci RAM o adresach 20h÷2Fh, każdy bit w tym obszarze jest dostępny indywidualnie, tryb adresowania bezpośredniego (adresy bitów 00h÷7Fh),
- **pdata** – 256-bajtowa strona zewnętrznej RAM, numer strony określa stan bitów portu P2 mikrosterownika, tryb adresowania pośredniego zawartością rejestru roboczego R0 lub R1,
- **xdata** – cała zewnętrzna pamięć RAM, tryb adresowania pośredniego zawartością rejestru DPTR.

W bajtach wewnętrznej RAM o adresach 00h÷1Fh mieszczą się cztery zestawy rejestrów roboczych R0÷R7. W dowolnej chwili mogą być używane rejestry tylko z jednego zestawu,

o numerze określonym przez stan dwóch bitów w rejestrze statusu PSW. Przy odwołaniach do znajdujących się w nich danych stosowany jest tryb adresowania rejestrowego.

W pamięci programu, w języku Cx51 oznaczonej symbolem **code**, mogą być przechowywane dane, których wartość nie zmienia się w trakcie działania programu, np. tablice konwersji. Pobranie bajtu z tej pamięci odbywa się w trybie adresowania pośredniego z indeksowaniem, adres jest sumą zawartości: rejestru DPTR i akumulatora.

Podane oznaczenia obszarów pamięci są zarazem słowami kluczowymi języka Cx51. Przy ich użyciu programista może wymusić inne, niż domyślne, rozmieszczenie deklarowanych zmiennych. Domyślny sposób rozmieszczenia określa tzw. model pamięci. W kompilatorze Keil do wyboru są trzy modele: small, compact i large, w których obszarami na dane są odpowiednio: **data**, **pdata** i **xdata**.

Tabela 1

Rozmiary kodu wynikowego instrukcji inkrementacji zmiennej bajtowej

operacja	typ zmiennej ***i***	położenie zmiennej ***i***	rozmiar kodu
`i++`	`unsigned char`	rejestr Rn	1
		data	2
		idata	3
		pdata	5
		xdata	6

Najmniej pamięci zajmuje kod wykonujący operację na zmiennych rejestrowych (tabela 1). Rozkazy z rejestrowym trybem adresowania są w większości jednobajtowe, numer rejestru stanowi część kodu takiego rozkazu. Jako zmienne rejestrowe można deklarować zmienne automatyczne (zmienne lokalne funkcji lub bloku) i powinno się z tej możliwości korzystać w przypadku intensywnie używanych zmiennych. Oczywiście, zostaną one rozmieszczone w rejestrach w miarę możliwości. Kompilator stara się też umieścić w rejestrach te zmienne automatyczne, których położenia nie podano w deklaracjach lub jest ono takie samo jak domyślne.

Argumenty funkcji są przekazywane przez wartość i mogą być one w niej używane jako inicjowane zmienne lokalne. Warto z tego korzystać, zwłaszcza wtedy, gdy funkcja nie jest funkcją współdzieloną (ang. *non-reentrant function*) i argumenty są jej przekazywane przez rejestry (sposób domyślny). Liczba tak przekazywanych argumentów wynosi od 1 do 3, zależy to od ich typu oraz kolejności ich występowania na liście [3].

Intensywnie używane zmienne zewnętrzne i statyczne powinny być przechowywane w pamięci wewnętrznej, o ile pozwoli na to ich rozmiar. Należy jednak pamiętać o tym, że w tej pamięci musi się jeszcze zmieścić stos.

3. Odpowiedni dobór typów danych

Typ danej określa sposób jej reprezentacji i rozmiar zajmowanej pamięci, co z kolei wyznacza zakres wartości, jakie dana może przyjmować. Lista rozkazów CPU mikrosterownika 8051 zawiera 111 rozkazów. Operandami 75 rozkazów są bajty, a 12 rozkazów pojedyncze bity. Także skoki warunkowe (13 rozkazów) są wykonywane w zależności od zawartości bajtu lub stanu bitu. Słowo 16-bitowe jest operandem 2 rozkazów, ale słowo to, przechowywane w rejestrze DPTR, reprezentuje adres bajtu w pamięci zewnętrznej. Należy zatem w miarę możliwości używać danych, zajmujących jeden bajt lub bit [1, 7]. Załóżmy, że mamy tablicę `N` liczników, gdzie $N \leq 60$:

```
unsigned int data timer[N];
```

używanych do odmierzania zadanych odcinków czasu. Zapis do licznika wartości niezerowej uruchamia odmierzanie, po czym jego zawartość jest dekrementowana w regularnych odstępach czasu. Zakończenie odmierzania następuje z chwilą wyzerowania licznika. Dekrementacja zawartości liczników może być wykonywana w pętli:

```
for(i = 0; i < N; i++)
  if (timer[i])
    --timer[i];
```

Programiści z reguły deklarują zmienną sterującą ***i*** jako zmienną typu `int`, chociaż będzie ona przyjmować tylko nieujemne wartości, a liczba obiegów często jest niewielka. Na rozmiar kodu wynikowego, obok typu zmiennej, ma wpływ również jej położenie w pamięci. Dlatego w tabeli 2 podano zakresy wartości rozmiaru kodu wynikowego pętli i wykonywanych w niej instrukcji, a nie pojedyncze wartości[1]. Skrajne wartości zakresów odpowiadają położeniu zmiennej odpowiednio w rejestrze roboczym i obszarze **xdata**, co stanowi najlepszy i najgorszy wariant jej położenia. Jak łatwo zauważyć, przez odpowiedni dobór typów rozmiar kodu wynikowego można zmniejszyć nawet ok. 1,6-krotnie.

Tabela 2

Rozmiary kodu wynikowego pętli do dekrementacji zawartości liczników

typ zmiennej sterującej	rozmiar kodu
`int` (2 bajty)	40÷81
`unsigned int` (2 bajty)	40÷81
`char` (1 bajt)	32÷51
`unsigned char` (1 bajt)	32÷49

Argumenty wyrażeń warunkowych powinny być tego samego typu. W kompilatorze Keil domyślnie włączona jest opcja **Enable ANSI integer promotions rules**. Gdy argumenty są

[1] Z tego samego powodu w dalszej części artykułu również kilkakrotnie podano zakresy wartości rozmiaru kodu wynikowego.

różnego typu, przed wykonaniem obliczeń argumenty 8-bitowe są przekształcane w argumenty typu `int`.

Do przechowywania danych przyjmujących tylko wartości: 0 i 1 należy używać zmiennych typu `bit`, umieszczanych w obszarze **bdata**. Jak wcześniej wspomniano, lista rozkazów mikrosterownika 8051 zawiera rozkazy do operacji na pojedynczych bitach. Rozkazy do ustawiania, zerowania i negacji bitu są 2-bajtowe. Przypisanie wartości do zmiennej typu bajtowego wymaga kodu o rozmiarze 2÷6 bajtów, a negacja jednego bitu w bajcie, przy wykorzystaniu operacji XOR, 2÷7 bajtów. W instrukcji

```
if (flag) ...
```

gdzie ***flag*** to zmienna bitowa, sprawdzenie warunku i przekazanie sterowania w odpowiednie miejsce wykonuje jeden 3-bajtowy rozkaz skoku warunkowego, jeśli rozmiar kodu instrukcji wykonywanych przy spełnieniu warunku nie przekracza 124 bajtów (chociaż zasięg skoku warunkowego w przód wynosi 127 bajtów). Przy zmiennej ***flag*** typu bajtowego wymaga to kodu zajmującego 4÷6 bajtów. Kod ten zajmie dodatkowe 3 bajty, kiedy rozmiar kodu wynikowego instrukcji wykonywanej przy spełnieniu warunku jest większy niż 124 bajty.

4. Negacja wartości zmiennej bitowej

Służą do tego operatory negacji: `!` lub `~`, można wykorzystać również operację XOR. Ostatni ze sposobów wymaga więcej pamięci – kod wynikowy instrukcji

```
flag ^= 1;
```

zajmuje 5 bajtów. W pozostałych przypadkach negację wykonuje 2-bajtowy rozkaz CPL.

5. Pola

Dane, do których reprezentacji wystarcza kilka bitów, można przechowywać w polach. Jest to jednak postępowanie zalecane wtedy, gdy trzeba oszczędzać pamięć danych. W mikrosterownikach rodziny 8051 ma ona organizację bajtową, tymczasem pola mogą być umieszczane wyłącznie w słowach [4]. Nie ma to wpływu na rozmiar kodu wynikowego instrukcji realizujących operacje na zawartości pola, jeżeli nie przekracza ono granicy bajtu. W przeciwnym razie rozmiar kodu wzrasta. Załóżmy, że mamy zmienną z trzema polami: ***a***, ***b*** i ***c***, każde o rozmiarze 3 bitów, pole ***c*** przekracza granicę bajtu. Kod wynikowy instrukcji

przypisania wartości do pola ***a*** zajmuje 11÷20 bajtów, a do pola ***c*** 20÷28 bajtów[2]. Przesunięcie pola ***c*** w granice następnego bajtu spowoduje, że rozmiar kodu wynikowego w obu przypadkach będzie identyczny. Aby wymusić to przesunięcie, wystarczy rozdzielić pola: ***b*** i ***c*** 2-bitowym polem bez nazwy.

6. Odwołania do elementów tablic

Do elementów tablic można się odwoływać przy użyciu indeksów lub wskaźników. Indeks to dowolne wyrażenie o wartości całkowitej, które może zawierać zmienne typu całkowitego i stałe całkowite. Wskaźnik to zmienna z adresem innej zmiennej lub funkcji [4].

Gdy indeks jest stałą, adres elementu tablicy jest obliczany w trakcie konsolidacji programu. Przy odwołaniach do elementów tablicy umieszczonej w obszarze **data** wykorzystywany jest tryb adresowania bezpośredniego. Instrukcja przypisania

```
timer[2] = 100;
```

jest wtedy tłumaczona na dwa 3-bajtowe rozkazy. W pozostałych przypadkach używany jest tryb adresowania pośredniego, kod wynikowy tej instrukcji zajmuje maksymalnie 9 bajtów.

Gdy indeks zawiera zmienne, adres elementu tablicy jest obliczany w trakcie wykonania programu, a odwołania są realizowane w trybie adresowania pośredniego. Na rozmiar kodu wynikowego wpływ ma rozlokowanie w pamięci tablicy oraz występujących w indeksie zmiennych, czasem także ich typy. Kod wynikowy instrukcji przypisania

```
timer[i] = 100;
```

zajmie 11 bajtów, gdy tablica ***timer*** będzie umieszczona w pamięci wewnętrznej, a zmienna ***i***, typu bajtowego, w rejestrze roboczym. Przy obu zmiennych w obszarze **xdata** kod wynikowy zajmuje już 21 bajtów. Z takiego sposobu indeksowania korzysta się z reguły przy przetwarzaniu kolejnych elementów tablic.

W języku Cx51 dostępne są dwa rodzaje wskaźników: dedykowane (ang. *memory–specific pointers*) oraz uniwersalne (ang. *generic pointer*). W deklaracji wskaźnika dedykowanego podaje się, w jakim obszarze pamięci mieści się wskazywana dana, zajmuje on 1 lub 2 bajty pamięci (1 – dla danych w obszarach: **data**, **idata** i **pdata**, 2 – dla danych w obszarach: **xdata** i **code**). Wskaźnik uniwersalny zawiera adres zmiennej umieszczonej w dowolnym obszarze i zajmuje 3 bajty pamięci (kolejno: identyfikator obszaru, starszy bajt

[2] Wartości skrajne zakresów odpowiadają rozmieszczeniu zmiennej odpowiednio w obszarach: **data** oraz **xdata**, nie udało się wymusić jej umieszczenia w rejestrach roboczych.

adresu, młodszy bajt adresu). Wskaźnikową wersję rozpatrywanej już pętli do dekrementacji zawartości liczników można zapisać np. tak:

```
ptr = timer;
for(i = 0; i < N; i++)
{
  if (*ptr)
    --*ptr;
  ptr++;
}
```

W tabeli 3 podano zakresy rozmiaru kodu wynikowego pętli w zależności od sposobu odwołań do liczników i położenia tablicy liczników w pamięci, przy zmiennej sterującej ***i*** typu `unsigned char`. Jak widać, gdy decydujemy się na użycie wskaźników, należy używać dedykowanych, a nie uniwersalnych. Gdyby wskaźnik dedykowany oraz zmienna sterująca pętli musiały być przechowywane w obszarze **xdata**, lepiej wykorzystać indeksowanie.

Tabela 3

Rozmiary kodu wynikowego pętli przy różnych sposobach odwołań do liczników

położenie tablicy	rozmiar kodu		
	1.	2.	3.
data	32÷49	28÷46	167÷196
idata	30÷47	27÷45	167÷196
pdata	35÷52	34÷50	167÷196
xdata	61÷78	61÷85	167÷196

Oznaczenie kolumn:
1. wersja z indeksowaniem,
2. wersja ze wskaźnikiem dedykowanym,
3. wersja ze wskaźnikiem uniwersalnym.

7. Odwołania do rejestrów zewnętrznych układów wejścia-wyjścia

W obszarach: **pdata** i **xdata** mogą się też znajdować rejestry zewnętrznych układów wejścia-wyjścia wyposażonych w magistralę równoległą. Popularnymi układami tego typu są wyświetlacze LCD ze sterownikiem HD44780 firmy Hitachi.

Tabela 4

Przyjęte adresy rejestrów wyświetlacza w obszarze **xdata**

adres	rejestr	operacja
8000h	sterujący	zapis rozkazu
8001h	sterujący	odczyt statusu
8002h	danych	zapis danej
8003h	danych	odczyt danej

Załóżmy, że komunikacja z wyświetlaczem odbywa się przez odwołania do komórek o adresach podanych w tabeli 4. Odwołania można realizować przy użyciu wskaźników de-

dykowanych, zapisanych w pamięci programu. Deklaracje takich wskaźników muszą mieć postać:

```
unsigned char xdata * code command = 0x8000;
unsigned char xdata * code status  = 0x8001;
unsigned char xdata * code dataWR  = 0x8002;
unsigned char xdata * code dataRD  = 0x8003;
```

nie jest to jednak sposób oszczędny. Przechowanie czterech takich wskaźników wymaga 8 bajtów pamięci. Gdy nie zostanie ustawiony odpowiedni poziom prowadzonej przez kompilator optymalizacji kodu, kod wynikowy każdej instrukcji przypisania np.

```
*command = CLEAR_DISPLAY;    /* CLEAR_DISPLAY == 0x02 */
```

zajmie 16 bajtów. Z tego 13 bajtów przypada na ciąg rozkazów, który realizuje przepisanie adresu ze wskaźnika do rejestru DPTR. Kompilator może zoptymalizować kod przez ujęcie powtarzających się ciągów rozkazów w podprogram. Kod wynikowy powyższej instrukcji przypisania zajmie wtedy 9 bajtów, z czego 6 przypadnie na przygotowanie argumentu i wywołanie podprogramu; podprogram zajmie 11 bajtów.

Oszczędniejszym rozwiązaniem jest użycie zmiennych absolutnych:

```
unsigned char xdata command _at_ 0x8000;
unsigned char xdata status  _at_ 0x8001;
unsigned char xdata dataWR  _at_ 0x8002;
unsigned char xdata dataRD  _at_ 0x8003;
```

lub makroinstrukcji `XBYTE` z pliku `absacc.h`. Kod wynikowy instrukcji zapisu rozkazu do sterownika wyświetlacza będzie liczył 6 bajtów.

8. Inicjowanie zmiennych

Zasady inicjowania zmiennych są podane w pracy [4]. Jeżeli deklarowanym zmiennym nie nadano wartości początkowych, to:

- zawartość zmiennych automatycznych i rejestrowych będzie przypadkowa,
- zmienne zewnętrzne i statyczne otrzymują wartość 0.

Wartości początkowe jawnie inicjowanym zmiennym automatycznym i rejestrowym są nadawane przy każdym wejściu do funkcji lub bloku, natomiast zmiennym zewnętrznym i statycznym tylko raz, przed uruchomieniem funkcji `main()`, co realizuje tzw. procedura startowa (ang. *startup code*).

Procedura startowa wypełnia najpierw pamięć zerami, przez co zmienne nieinicjowane otrzymują wartość 0. Ale uwaga: domyślnie używana procedura, zawarta w bibliotece, zeruje **zawsze tylko dolny blok wewnętrznej RAM**! Tych 128 bajtów pamięci zawiera niemal każdy mikrosterownik 8051 – niemal, gdyż produkowane są też jego wersje z 64 bajtami. Zatem

w razie potrzeby programista musi jawnie zainicjować zerami zmienne położone w innych obszarach, chociaż te z obszaru **idata** fizycznie mogą się znaleźć w dolnym bloku RAM. Powoduje to jednak znaczne zwiększenie rozmiaru kodu wynikowego procedury startowej. Przy braku inicjowanych zmiennych rozmiar tego kodu jest minimalny i wynosi 15 bajtów. W przeciwnym przypadku procedura zajmuje 143 bajty pamięci, dodatkowo w pamięci zapisane są tablice z wartościami początkowymi. Struktura takiej tablicy jest następująca:

- 1÷2 bajty – kod obszaru (2 bity) + informacja o liczbie bitów przeznaczonych na zakodowanie liczby bajtów z danymi (1 bit) + liczba bajtów z danymi (5 lub 13 bitów),
- 1÷2 bajty – adres początkowy (1 bajt – gdy tablica zawiera wartości początkowe dla zmiennych w obszarach: **data**, **idata**, **pdata**, 2 bajty – dla zmiennych w obszarze **xdata**),
- n bajtów – wartości początkowe,
- 1 bajt – znacznik końca, wartość 0.

Jawne zainicjowanie już tylko jednej zmiennej typu np. bajtowego spowoduje prawie 10-krotne zwiększenie rozmiaru kodu wynikowego domyślnej procedury startowej. Jeżeli w programie nie występują statyczne zmienne lokalne funkcji, przypisywanie w funkcji `main()` wartości początkowych innym zmiennym statycznym i zmiennym zewnętrznym może dać oszczędniejszy kod wynikowy, ale to programista musi ustalić drogą eksperymentalną.

Kod domyślnej procedury startowej dostępny jest też w postaci źródłowej, w plikach asemblerowych: `startup.a51` oraz `init.a51`. Można ją zmodyfikować i użyć zamiast procedury bibliotecznej. Wystarczą niewielkie zmiany w pliku `startup.a51`, aby była zerowana cała pamięć danych w systemie. Przypisanie wartości początkowych zmiennym inicjowanym wykonuje kod zawarty w pliku `init.a51`. Zmiany w nim powinny się ograniczyć do uzupełnienia makroinstrukcji odświeżania układu nadzorcy (ang. *watchdog*). Potrzeba taka może zajść wówczas, gdy w procedurze startowej uruchamiany jest układ nadzorcy, a liczba inicjowanych zmiennych jest duża. Jeżeli w systemie brak zewnętrznej RAM, można wykorzystać kod z pliku `init_tny.a51`, zaoszczędzi się 39 bajtów pamięci programu.

W postaci źródłowej dostępne są także procedury startowe przeznaczone dla pewnych wersji mikrosterownika 8051, np. serii LPC900 i 80C75x firmy Philips.

9. Pętle o zadanej liczbie obiegów

Każda z instrukcji pętli w języku C umożliwia wykonanie pewnych operacji zadaną liczbę razy. Do tej pory w artykule używano tylko pętli `for`, w praktyce stosowanej najczęściej. Podawano przy tym sumaryczne rozmiary kodu wynikowego pętli i zawartych w niej instrukcji. Teraz rozpatrywany będzie tylko kod wynikowy pętli.

Gdy przetwarzane są kolejne elementy tablicy, zmienna sterująca pętli może służyć zarówno do odliczania obiegów, jak i do indeksowania. Poniżej zostaną rozpatrzone przypadki, w których przyjmuje ona wartości narastające, począwszy od 0. Pętlę z przykładu dotyczącego dekrementacji zawartości liczników można napisać następująco:

```
1. for(i = 0; i < N; i++) .....;
2. i = 0; while(i < N) { .....; i++;}
3. i = 0; do { .....; i++;} while(i < N);
4. i = 0; do .....; while(++i < N);
5. i = 0; do .....; while(i++ < N - 1);
```

W tabeli 5 podano rozmiary kodu wynikowego pętli przy zmiennej sterującej ***i*** typu `unsigned char`. Pętle 1 i 2 są tu równoważne ([4]), kompilator Keil w obu przypadkach generuje identyczny kod wynikowy. Przy zmiennej sterującej w rejestrze roboczym lub obszarze **data** warunek kontynuacji jest sprawdzany nie na początku, ale na końcu obiegu i wykonuje to 3-bajtowy rozkaz CJNE. Sprawdzenie warunku przy zmiennej w innych obszarach oraz w pętlach 3÷5 wymaga wyznaczenia różnicy porównywanych wielkości, decyzję o kontynuacji lub zatrzymaniu pętli podejmuje się na podstawie stanu znacznika przeniesienia CY. Po modyfikacji zawartości zmiennej sterującej w pętlach 1÷2 musi wówczas nastąpić skok na początek pętli, co wykonuje 2-bajtowy rozkaz krótkiego skoku bezwarunkowego.

Tabela 5

Rozmiary kodu wynikowego pętli 1÷5

położenie zmiennej ***i***	rozmiar kodu				
	1.	2.	3.	4.	5.
rejestr Rn	6	6	9	9	11
data	10	10	12	12	13
idata	18	18	13	13	13
pdata	20	20	15	14	16
xdata	23	23	17	16	18

Gdy odwołania do elementów tablicy będą wykonywane poprzez wskaźnik, zmienna sterująca posłuży tylko do odliczania obiegów pętli. Odliczanie może następować w górę lub w dół. Kolejne warianty pętli do dekrementacji liczników to (pominięto instrukcję warunkową i warianty z pętlą `while`):

```
1. for(i = 0; i++ < N; ) .....;
2. for(i = 0; ++i < N + 1; ) .....;
3. for(i = N; i; i--) .....;
4. for(i = N; i--; ) .....;
5. for(i = N + 1; --i; ) .....;
6. i = N; do { .....; --i;} while(i);
7. i = N; do .....; while(--i);
8. i = N - 1; do .....; while(i--);
```

W tabeli 6 podano rozmiary kodu wynikowego pętli 1÷2 oraz 6÷13, typem zmiennej ***i*** jest `unsigned char`. Pętle 3÷5 pominięto, gdyż ich kod wynikowy pozostał niezmieniony. Do organizacji pętli 1÷2 wykorzystywany jest zawsze rozkaz CJNE, stąd zmniejszenie rozmiaru

ich kodu wynikowego w trzech przypadkach. Korzystne jest odliczanie obiegów w dół. Zatrzymanie pętli następuje wtedy po osiągnięciu przez zmienną sterującą wartości 0. Pętle: 8, 11 i 12, przy zmiennej sterującej w rejestrze roboczym lub obszarze **data**, realizuje się przy użyciu rozkazu DJNZ. W pozostałych przypadkach podjęcie decyzji o kontynuacji lub zatrzymaniu pętli wymaga tylko przesłania zawartości zmiennej sterującej do akumulatora i wykonania rozkazu JZ lub JNZ.

Tabela 6

Rozmiary kodu wynikowego pętli 1÷2 oraz 6÷13

położenie zmiennej ***i***	rozmiar kodu									
	1.	2.	6.	7.	8.	9.	10.	11.	12.	13.
rejestr Rn	6	6	13	11	4	11	8	4	4	8
data	10	10	15	14	6	12	11	6	6	10
idata	11	11	15	15	10	12	12	10	10	10
pdata	13	13	18	16	13	16	14	13	12	14
xdata	15	15	20	18	15	18	16	15	14	16

Niekiedy liczba obiegów pętli zostaje określona dopiero w trakcie wykonywania programu. Ma to miejsce np. przy przetwarzaniu ramek i komunikatów protokołu komunikacyjnego. Z wieloma urządzeniami pomiarowo – kontrolnymi można komunikować się przy użyciu protokołu Modbus, w którym długość ramek i komunikatów zależy m. in. od rodzaju transakcji [5]. Jeżeli urządzenie posiada zbiór 16-bitowych rejestrów oznaczonych symbolami 4xxxx, to gdy otrzyma komunikat z poleceniem odczytu zawartości `n` takich rejestrów i argumenty polecenia są poprawne, powinno odesłać komunikat z danymi liczący 3 + 2∗`n` bajtów. Jednorazowo można odczytać zawartość od 1 do 125 rejestrów. Ograniczenie górne wynika z tego, że komunikaty protokołu Modbus mogą zawierać co najwyżej 254 bajty. Pętla, w której zawartość rejestrów jest kopiowana do bufora z komunikatem, może być zapisana następująco (pominięto niektóre warianty z pętlą `while`, w zmiennej ***len*** zapisana jest liczba odczytywanych rejestrów):

```
1. for(i = 0; i < len; i++) { .....; }
2. i = 0; do { .....; i++; } while(i < len);
3. i = 0; do { .....; } while(++i < len);
4. for(i = 0; i++ < len; ) { .....; }
5. for( ; len; len--) { .....; }
6. while(len--) { .....; }
7. do { .....; len--; } while(len);
8. do { .....; } while(--len);
```

Pętle 14÷16 trzeba stosować wówczas, gdy wartość zmiennej ***i*** jest używana wewnątrz pętli. W pętli `do - while` warunek kontynuacji sprawdzany jest na końcu, przez co zapisane w niej instrukcje zostaną wykonane co najmniej raz. Polecenie odczytu zawartości 0 rejestrów jest jednak niewykonalne i wejście w tę pętlę nie nastąpi. W tabeli 7 zamieszczono zakresy roz-

miaru[3] kodu wynikowego powyższych pętli przy zmiennych typu `unsigned char`, w zupełności wystarczających. Znowu korzystne okazało się odliczanie obiegów w dół, zwłaszcza przy sprawdzaniu warunku kontynuacji na końcu pętli. Gdy zmienna sterująca znajduje się w rejestrze roboczym, pętla `do - while` jest tłumaczona na jeden 2-bajtowy rozkaz DJNZ.

Tabela 7

Rozmiary kodu wynikowego pętli 14÷21

14.	15.	16.	17.	18.	19.	20.	21.
10÷19	8÷20	8÷16	12÷20	6÷10	8÷12	2÷9	2÷8

Jeżeli instrukcja zmiany wartości zmiennej sterującej jest jedną z kilku instrukcji w części modyfikującej pętli `for`, najlepiej zapisać ją na końcu ciągu. Tak samo warto postępować wtedy, gdy zmiana następuje wewnątrz pozostałych pętli.

10. Rozwijanie pętli

Niekiedy korzystniej jest powielić instrukcję, zamiast wykorzystywać pętlę. Jest to nazywane rozwijaniem pętli (ang. *unrolling loops*) [7]. Gdy do zainicjowania wartościami niezerowymi tablicy

```
unsigned char data array[3]
```

zostanie użyta pętla 12 i wskaźnik dedykowany, kod wynikowy całej pętli zajmie 9÷18 bajtów pamięci. Kod wynikowy instrukcji:

```
array[0] = array[1] = array[2] = 10;
```

zajmuje 9 bajtów.

11. Zliczanie zadanej liczby zdarzeń

Niekiedy pewne operacje wykonuje się po zliczeniu zadanej liczby zdarzeń, np. przerwań od zegara sprzętowego. Można to zaprogramować następująco:

```
1. counter++; if (counter > N - 1) ...
2. if (++counter > N - 1) ...
3. counter++; if (counter == N) ...
4. if (++counter == N) ...
5. counter--; if (!counter) ...
6. if (!--counter) ...
```

[3] Obie zmienne w pętlach 14÷17 w najlepszym przypadku znajdują się w rejestrach roboczych, w najgorszym w obszarze **xdata**.

gdzie `N` jest zadaną liczbą zdarzeń, początkowa zawartość zmiennej ***counter*** jest równa: 0 przy zliczaniu w górę, `N` przy zliczaniu w dół. W tabeli 8 podano rozmiary kodu wynikowego powyższych instrukcji przy zmiennej ***counter*** typu `unsigned char`. Gdy zmienna ta znajduje się w rejestrze roboczym lub obszarze **data**, dekrementację i porównanie realizuje jeden rozkaz DJNZ.

Tabela 8
Rozmiary kodu wynikowego instrukcji do zliczania zdarzeń

położenie licznika	rozmiar kodu					
	1.	2.	3.	4.	5.	6.
rejestr Rn	7	7	4	4	2	2
data	9	9	7	7	3	3
idata	9	9	7	7	6	6
pdata	11	10	9	8	8	7
xdata	12	11	10	9	9	8

12. Podejmowanie decyzji wielowariantowych

Rozpatrywany tu będzie przypadek, gdy warianty są wybierane stałymi wartościami typu całkowitego lub znakowego. Wybór taki umożliwia instrukcja `switch`. Sposób tłumaczenia tej instrukcji przez kompilator Keil zależy od kilku czynników. Jeśli wyrażenie wybierające daje w wyniku wartość typu `unsigned char`, przy liczbie wariantów ≤ 6 (poza `default`) dla dokonania wyboru wykonuje się operacje dodawania i porównania z 0. W przeciwnym razie wykorzystywana jest tablica zawierająca stałe i adresy skoków, przekazanie sterowania w odpowiednie miejsce realizuje funkcja biblioteczna.

Tabela 9
Rozmiary kodu wynikowego instrukcji wyboru wielowariantowego

kolejność stałych	rozmiar kodu	
	`switch`	`ci¹g if ... else if ...`
7, 3, 11, 5	17÷20	12÷20
3, 7, 5, 11	17÷20	12÷20
0, 7, 11, 5	17÷20	12÷19
7, 0, 11, 5	15÷18	12÷19
0, 1, 2, 3	14÷17	12÷19
3, 2, 1, 0	12÷15	12÷19
0, 1, 2, 3, 4, 5, 6	66÷69	21÷34
6, 5, 4, 3, 2, 1, 0	66÷69	21÷34

W tabeli 9 podano wartości rozmiaru kodu wynikowego instrukcji `switch` dla przykładowych stałych, przy takiej kolejności ich zapisu w instrukcji i wartości wyrażenia wybierającego typu `unsigned char`. Kolejność zapisu stałych nie ma żadnego wpływu na rozmiar kodu,

jeśli nie ma wśród nich 0. Jeżeli jednak jest, to rozmiar kodu nieco zmniejszymy, gdy nie umieścimy jej na pierwszej pozycji. Gdy stałe tworzą ciąg arytmetyczny o różnicy równej 1, kompilator optymalizuje jeszcze operacje dodawania. Przy $N > 6$ wariantach tablica ze stałymi i adresami skoków zajmuje $3*N + 4$ bajty, funkcja biblioteczna 37 bajtów, reszta przypada na rozkazy realizujące przygotowanie argumentu dla tej funkcji oraz jej wywołanie.

Zamiast instrukcji `switch` można wykorzystać ciąg instrukcji `if … else if …` Przy $N \le 6$ wariantach w większości przypadków kod wynikowy zajmie jednak więcej miejsca. Ale np. przy 7 wariantach uzyska się – w najgorszym razie – nieco ponad dwukrotne zmniejszenie rozmiaru tego kodu.

13. Tablice wskaźników do funkcji

Gdy trzeba wywołać jedną z N funkcji o takich samych nagłówkach, niekiedy lepiej robić to za pośrednictwem wskaźników do funkcji, a nie instrukcji decyzyjnych. Zależy to od kilku czynników: liczby funkcji, liczby i typów ich argumentów, czy zwracane przez funkcje wartości są wykorzystywane oraz zestawu wartości wybierających funkcje. Duże znaczenie ma także sposób zadeklarowana tablicy wskaźników – najlepiej, jeśli będzie to tablica ze wskaźnikami dedykowanymi, umieszczona w pamięci programu (inaczej wzrośnie rozmiar procedury startowej). W tabeli 10 podano rozmiary kodu wynikowego fragmentu programu, w którym następuje wybór i wywołanie funkcji.

Tabela 10

Rozmiary kodu wynikowego instrukcji wywołania 1 z N funkcji

wartości wybierające	s. w. f.	rozmiar kodu przy n argumentach		
		n = 0	n = 1	n = 2
3, 2, 1, 0	1.	24÷27	32÷35	40÷43
	2.	24÷32	32÷39	40÷48
	3.	38÷41	37÷40	39÷42
6, 5, 4, 3, 2, 1, 0	1.	87÷90	101÷104	115÷118
	2.	42÷55	56÷69	70÷80
	3.	44÷47	43÷46	45÷48

Oznaczenie kolumn:
s. w. f. – sposób wyboru funkcji.
Oznaczenie wierszy:
1. wybór przy użyciu instrukcji `switch`,
2. wybór przy użyciu ciągu instrukcji `if else if`,
3. wybór przy użyciu tablicy wskaźników dedykowanych.

Wyniki te uzyskano w przypadku, gdy: wartość wybierająca, przechowywana w zmiennej, oraz argumenty funkcji były typu `unsigned char`, funkcjom przekazywano stałe i zwracane przez nie wartości były ignorowane, tablica wskaźników dedykowanych była zapisana

w pamięci programu. Nie sprawdzano, czy wartość wybierająca mieści się w dopuszczalnym zakresie.

Rozmiar tablicy ze wskaźnikami do N funkcji wynosi $2*N$ bajtów, jeśli funkcje są wybierane kolejnymi wartościami. W przeciwnym razie tablica zajmie $2*(n_{max} - n_{min} + 1)$ bajtów, gdzie n_{max} i n_{min} to maksymalna i minimalna wartość wybierająca, a część jej komórek wskaźnik do funkcji wywoływanej przy pozostałych wartościach z tego zakresu.

14. Przekazywanie argumentów do funkcji

Sposób przekazywania argumentów do funkcji, które nie są funkcjami współdzielonymi ustala się przy użyciu dwóch dyrektyw kompilatora: NOREGPARMS i REGPARMS. Pierwsza z nich wymusza przekazywanie wszystkich argumentów przez przydzielony funkcji blok pamięci. Druga nakazuje kompilatorowi, aby do 3 argumentów funkcja otrzymywała za pośrednictwem rejestrów, resztę przez blok pamięci. Jest to sposób używany domyślnie. W tabeli 11 przedstawiono stosowaną przez kompilator Keil zasadę przydziału rejestrów poszczególnym argumentom funkcji [3].

Tabela 11

Zasada przydziału rejestrów argumentom funkcji

p. a.	typ argumentu			
	a.	b.	c.	d.
1	R7	R6 – R7	R4 – R7	R1 – R3
2	R5	R4 – R5	R4 – R7	R1 – R3
3	R3	R2 – R3		R1 – R3

Oznaczenia kolumn:
p. a. – pozycja argumentu na liście,
a. argument typu `char` lub 1-bajtowy wskaźnik dedykowany,
b. argument typu `int` lub 2-bajtowy wskaźnik dedykowany,
c. argument typu `long` lub `float`,
d. wskaźnik uniwersalny.

Jak wiadomo, argumenty funkcji mogą być w niej traktowane jako inicjowane zmienne lokalne, a kod wykonujący operacje na zmiennych umieszczonych w rejestrach zajmuje najmniej pamięci. Warto zatem nie przekraczać liczby 3 argumentów i przekazywać je przez rejestry. Ważna jest też ich kolejność na liście. Na przykład funkcję o nagłówku:

```
void foo(char a, int b, int *c)
```

wszystkie argumenty będą przekazywane przez rejestry, zgodnie z przyjętą zasadą będą to odpowiednio: R7, R4 – R5 oraz R1 – R3. Natomiast przy takim nagłówku:

```
void foo(int *c, int b, char a)
```

przez rejestry będą przekazywane tylko argumenty `c` i `b`, a ostatni przez blok pamięci [1].

Argumenty typu `bit` należy wymieniać na końcu listy. W przeciwnym razie wymienione po nich argumenty zawsze będą przekazywane przez blok pamięci [3].

Przy dużej liczbie argumentów można wypróbować podany w artykule [1] pomysł zgrupowania argumentów w strukturę i przekazywania funkcji adresu struktury. Do realizacji odwołań do elementów struktury użyteczny byłby tryb adresowania pośredniego z przesunięciem, niestety w mikrosterownikach rodziny 8051 takiego trybu nie ma. Przy wskaźniku dedykowanym obliczenie adresu elementu wymaga kilku rozkazów, przy uniwersalnym funkcji bibliotecznych. Może okazać się, że kod wynikowy funkcji zajmie więcej miejsca, niż gdyby argumenty były przekazywane przez blok pamięci.

Argumenty dla funkcji współdzielonych (ang. *reentrant functions*) są w miarę możliwości przekazywane przez rejestry, zgodnie z zasadami podanymi w tabeli 11. Funkcja korzysta jednak z ich kopii, które przechowuje na stosie programowym. Położenie tego stosu określa przyjęty model pamięci. Lokowane na nim są też argumenty, których nie można było przekazać w rejestrach, oraz zmienne lokalne funkcji. Operacje z użyciem stosu wprowadzają dość spory narzut w kodzie, dlatego nie należy pisać funkcji współdzielonych bez uzasadnionej potrzeby.

15. Funkcje biblioteczne

Możliwość korzystania z funkcji bibliotecznych to udogodnienie dla programisty, ale kod wynikowy niektórych z nich zajmuje dość sporo pamięci. Na przykład funkcja `strcmp()`, dokonująca porównania tekstów, zajmuje aż 421 bajtów. Powodów tego jest kilka. Argumentami tej funkcji są wskaźniki do tekstów. Tekst może być umieszczony w jednym z 5 obszarów pamięci, a ponieważ są 4 tryby adresowania pośredniego (jeden wspólny dla obszarów: **data** oraz **idata**), daje to 16 różnych zestawów adresów. W każdym przypadku porównanie realizuje inny ciąg rozkazów. Wskaźniki są wskaźnikami uniwersalnymi, przekazywanymi przez rejestry (zasady przekazywania argumentów dla funkcji bibliotecznych są inne niż dla funkcji pisanych przez użytkownika kompilatora). Przygotowanie argumentów oraz wywołanie funkcji to dalszych co najmniej 15 bajtów kodu.

Część funkcji bibliotecznych, w tym `strcmp()`, ma nagłówki takie, jak to określono w standardzie ANSI C. Funkcja `strcmp()` zwraca więc daną typu `char`, której wartość informuje o tym, czy tekst pierwszy jest leksykograficznie: mniejszy (< 0), równy (== 0) czy większy (> 0) od drugiego z porównywanych tekstów. Wartość tę funkcja oblicza odejmując od siebie kody znaków na pierwszej pozycji, na której teksty się różnią. Tymczasem zwykle interesuje nas tylko informacja o zgodności lub niezgodności tekstów.

Warto napisać własny wariant funkcji do porównywania tekstów, gdy będą one rozmieszczone zawsze w tych samych obszarach pamięci. Można wtedy użyć wskaźników dedykowanych. Kod źródłowy poniższej funkcji:

```
bit my_strcmp(char code *s, char xdata *t)
{
  for( ; *s == *t; s++, t++)
    if (*s == '\0')
      return(1);      // jednakowe teksty

  return(0);          // różne teksty
}
```

jest wzorowany na kodzie źródłowym funkcji `strcmp()`, podanym w pracy [4]. Jej kod wynikowy zajmuje tylko 33 bajty pamięci, gdy argumenty są przekazywane przez rejestry. W takim przypadku narzut w kodzie związany z przygotowaniem argumentów i wywołaniem funkcji wynosi minimum 11 bajtów.

Użycie pętli o zadanej liczbie powtórzeń do przesłania tekstu np. do wyświetlacza LCD wymaga obliczenia długości tekstu. Obliczenie takie wykonuje funkcja biblioteczna `strlen()`, zajmująca 42 bajty pamięci. Tekst to ciąg znaków zakończony znakiem `'\0'` (znak o kodzie 0). Wysyłanie kolejnych znaków tekstu można zatem prowadzić do napotkania takiego znaku. Kod wynikowy pętli

```
while(*s) ...
```

gdzie ***s*** jest dedykowanym wskaźnikiem na znak zajmie 10÷17 bajtów, gdy tekst będzie znajdować się w obszarze **code**.

16. Makroinstrukcje lub powielanie kodu

Jeżeli narzut w kodzie związany z przygotowaniem argumentów i wywołaniem funkcji jest większy niż kod samej funkcji, należy ją zastąpić makroinstrukcją lub powielić jej kod [7].

17. Funkcje obsługi przerwań

Funkcja obsługi przerwania przechowuje na stosie zawartość rejestrów: ACC, B, DPTR i PSW, gdy są one przez nią używane. Tak samo zawartość rejestrów roboczych, jeśli programista nie określi, z którego ich zestawu funkcja ma korzystać. W najgorszym przypadku wprowadza to narzut równy 55 bajtów. Ma to miejsce na przykład wtedy, gdy wywoływana jest w niej inna funkcja, której kod, również źródłowy, jest zapisany w innym pliku. Wydzie-

lenie banku rejestrów na potrzeby funkcji obsługi przerwania zmniejsza ten narzut maksymalnie do 23 bajtów.

18. Zakończenie

Pamięć programu bywa zasobem dość ograniczonym, wtedy trzeba nią oszczędnie gospodarować. Łatwiej pisać i modyfikować później program w języku C, ale jego kod wynikowy z reguły zajmuje więcej pamięci niż kod wykorzystującego **ten sam algorytm** programu w języku asemblera. Różnicę tę można zmniejszyć przez stosowanie odpowiednich konstrukcji języka wysokiego poziomu. Efekt przynosi już użycie zmiennych bajtowych i bitowych tam, gdzie to wystarcza. Warto zauważyć, że oszczędza się przy tym pamięć danych. Najkorzystniej jest przechowywać dane w pamięci wewnętrznej, a jej pojemność jest niewielka.

Stosowanie zmiennych bajtowych i bitowych oraz odpowiednie rozmieszczenie danych w pamięci przyniesie zamierzony efekt także wtedy, gdy program będzie tłumaczony przy użyciu kompilatora innego producenta. „Skuteczność" pozostałych z przedstawionych sposobów trzeba jednak wtedy sprawdzić doświadczalnie.

LITERATURA

1. Jones N.: Efficient C Code for Eight-Bit MCUs. Embedded Systems Programming, November 1998.
2. Jones N.: 'Optimal' C Constructs for 8051 Microprocessors. Embedded Systems Programming, October 2002.
3. Keil Software: Cx51 Compiler. Optimizing C Compiler and Library Reference for Classic and Extended 8051 Microcontrollers. 2001.
4. Kernighan B. W., Ritchie D. M.: *Język C*. WNT, Warszawa 1988.
5. Modbus Application Protocol Specification V1.1b. Modbus-IDA, December 2006, http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b.pdf.
6. Pełka R.: Mikrokontrolery. Architektura, programowanie, zastosowanie. WKi£, Warszawa 1999.
7. Schleich T. M.: Optimizing C for Embedded Systems. Proceedings of the Embedded Systems Conference, San Francisco 2005.

Recenzent: Prof. dr hab. inż. Ryszard Pełka

Wpłynęło do Redakcji 3 marca 2008 r.

Abstract

Program memory space can be limited resource. C program is simpler to write and modify than assembly program. When the same algorithm is used in both programs, object code of C program is usually larger. It is possible to decrease this difference by using appropriate constructs of the high-level language. Some of such constructs of Cx51 language, the dialect of C language for 8051 family microcontrollers, are discussed in this article. The Cx51 compiler was worked out by Keil. Version 6.20 of the compiler was used during examinations. Some constructs bring effects independently from the compiler, for example, applying byte and bit variables, where it is enough. It is worthwhile to notice that it also saves data memory space. Saving variables in internal data memory is most advantageous, but it is very limited resource.

Adres

Dariusz Caban: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16,
44-100 Gliwice, Polska, Dariusz.Caban@polsl.pl.