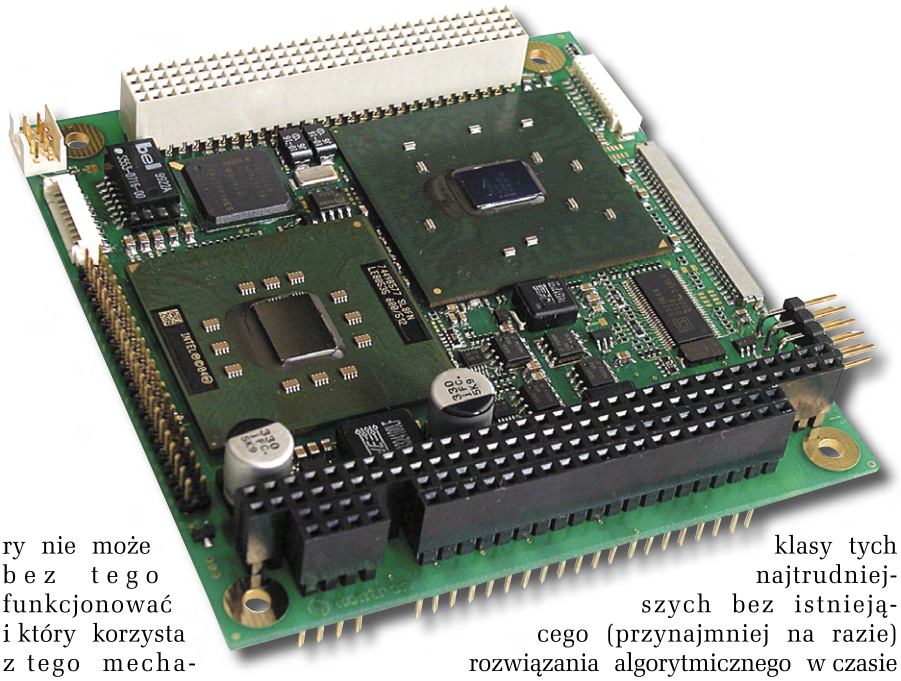
Dynamiczne zarządzanie pamięcią w małych systemach *embedded*

Oprogramowanie typu embedded staje się coraz bardziej złożone, zarówno pod względem realizowanych funkcji jak i budowy. Wiele bloków funkcjonalnych programu współpracuje ze sobą w sposób dynamiczny – zadania są tworzone i usuwane, struktury danych przez nie wykorzystywane „znikają" z pamięci po tym, jak przestają być potrzebne itd. Zwykle nie da się tego osiągnąć w prosty (tani) sposób bez mechanizmu dynamicznego zarządzania pamięcią. Możliwe rozwiązania przedstawiamy w artykule.

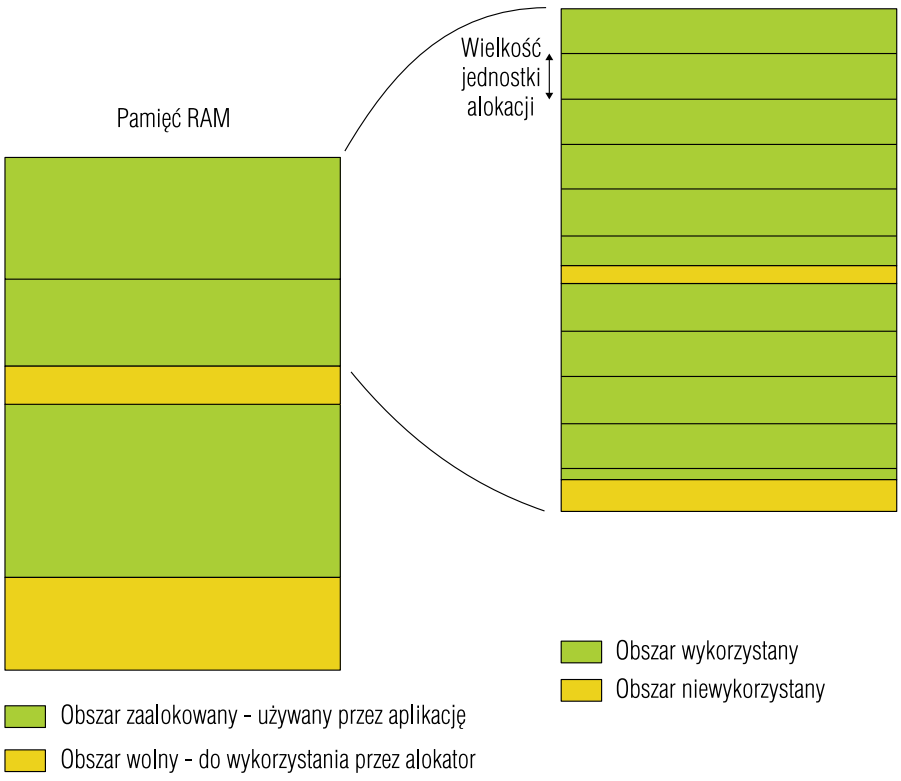


W bardziej złożonych aplikacjach zwykle nie można liczyć na komfort posiadania tak dużych zasobów pamięci, aby możliwe było wstępne utworzenie wszystkich obiektów programu przed jego uruchomieniem. Co więcej, byłoby to często bardzo trudne, albo wręcz niemożliwe ze względu na dynamikę działania aplikacji (skomplikowane powiązania) i ograniczone zasoby. W pełni statyczne podejście do alokacji pamięci mogłoby też być bardzo nieefektywne – po co na przykład utrzymywać w pamięci przez cały czas działania programu dane potrzebne tylko przy jego uruchamianiu. Innym elementem powodującym, że dynamiczne zarządzanie pamięcią staje się istotnym elementem świata systemów wbudowanych i jednocześnie systemów czasu rzeczywistego, jest fakt powstania specyfikacji i (działającej) implementacji odmiany języka Java w wersji RT (*real time*) – Java RTS. Wydaje się więc, że skoro rozwiązano problem dynamicznego zarządzania pamięcią dla języka, który nie może bez tego funkcjonować i który korzysta z tego mechanizmu w sposób bardzo intensywny, to tym bardziej można by to zrobić w języku C, gdzie tylko programista ma wpływ, czy i jak użyć sterty.

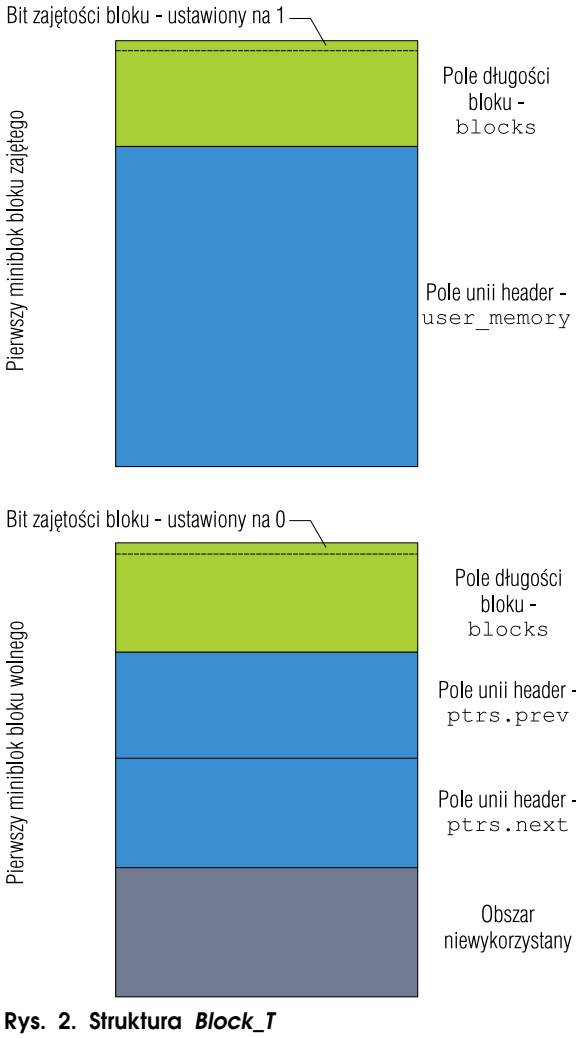
Jedną z trudności, na jakie napotykamy już na początku, jest problem mnogości algorytmów używanych przez funkcje zarządzające dynamiczną alokacją pamięci. W powszechnym przekonaniu fakt, iż jest ich tak dużo, świadczy o tym, że nie ma żadnego naprawdę dobrego rozwiązania i lepiej jest zaprojektować coś samemu, niż polegać na istniejących implementacjach. W efekcie większość projektów stara się unikać używania sterty w ogóle lub co najwyżej korzysta z mechanizmu puli bloków pamięci – *memory pool* (który zwykle jest sam w sobie bardzo dobry do wielu zastosowań, lecz nie jest w stanie zastąpić w pełni dynamicznego zarządzania pamięcią). Żeby rzecz uczynić jeszcze bardziej zagmatwaną, należy wspomnieć, że ogólne zagadnienie dynamicznej alokacji pamięci bez znajomości kolejności i parametrów żądań przydziału i zwalniania bloków jest jednym z wielu informatycznych problemów optymalizacji należących do klasy tych najtrudniejszych bez istniejącego (przynajmniej na razie) rozwiązania algorytmicznego w czasie wielomianowym (tj. do kategorii NP–zupełnych).

W celu niejakiego uporządkowania tematu warto określić, jakie wymagania powinien spełniać idealny alokator pamięci dynamicznej dla systemu czasu rzeczywistego typu *embedded*.

1. Musi być znany czas wykonania dowolnej operacji dla najgorszego przypadku. Niezależnie od aktualnego stanu systemu i żądanej usługi (przydział, zwolnienie) oraz jej parametrów wymagane jest, aby czas reakcji był zawsze określony od góry. Niestety, większość używanych algorytmów była projektowana z uwzględnieniem optymalizacji czasu wykonania dla typowego przypadku, a czas wykonania dla najgorszego przypadku może być w nich nawet o rzędy wielkości większy. Co więcej, maksymalny czas wykonania dla systemu idealnego nie powinien być zależny od struktur danych użytych w algorytmie, ani od wielkości obsługiwanej pamięci (złożoność $O(1)$).
2. Szybkość wykonania powinna być na tyle duża, aby spełnić wyma-



Rys. 1. Fragmentacja zewnętrzna i wewnętrzna



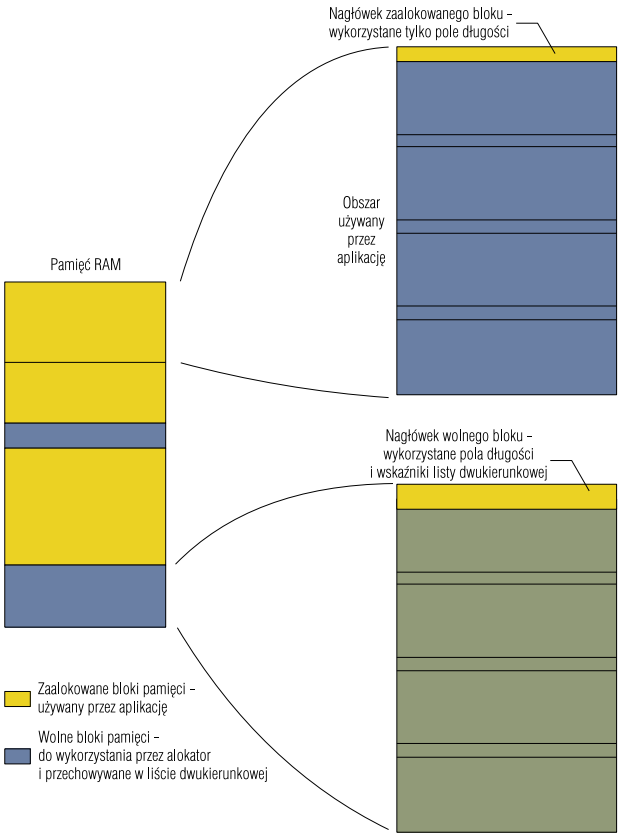
Rys. 2. Struktura *Block_T*

gania systemu czasu rzeczywistego. Jest oczywiste, że sama znajomość maksymalnego czasu wykonania operacji nie jest wystarczającym kryterium użycia danego algorytmu do zastosowań RT. Jeżeli czas wykonania przekracza wymagany czas reakcji systemu RT, to alokator jest po prostu nieprzydatny.

3. Żądanie przydziału pamięci musi być zawsze spełnione. Jest to wprawdzie wymaganie dla systemu idealnego bez ograniczonych zasobów, dlatego w przypadku systemu osadzonego należy po prostu dążyć do maksymalizacji prawdopodobieństwa, że przydział pamięci się powiedzie. Nie trzeba chyba dodawać, że ze względu na specyfikę systemów typu *embedded* efekt nieudanej alokacji pamięci może być o wiele bardziej destrukcyjny w porównaniu do systemów „biurkowych". Niestety, ta sama specyfika powoduje też, że algorytmy muszą pracować w o wiele „trudniejszych" warunkach, a to choćby z tego względu, iż przez całe miesiące musi być zapewniona nieprzerwania praca urządzenia, i to przy dosyć ograniczonych zasobach pamięci RAM. Wynika z tego na przykład to, że system dynamicznej alokacji pamięci dla oprogramowania typu *embedded* musi być bardziej odporny na efekt fragmentacji sterty. Chodzi tu zarówno o fragmentację wewnętrzną, czyli spowodowaną wewnętrzną budową alokatora (np. minimalna wielkość przydzielanych bloków, „granulacja" wielkości bloków itd.), jak i o fragmentację zewnętrzną, czyli spowodowaną kolejnością przedzielania i zwalniania bloków pamięci przez konkretną aplikację (np. z trzech kolejnych bloków następuje zwolnienie tylko dwóch „skrajnych").

W celu spełnienia powyższych wymagań używane są przeróżne strategie, algorytmy i struktury danych. Stosuje się na przykład alokację bloków tylko o stałej wielkości lub o wielkości będącej tylko potęgą liczby 2. Używa się zwykle dosyć złożonych struktur danych, jak zbiory list, posegregowane zestawy list, bitmapy, drzewa typu AVL itd. Przy rozwiązywaniu problemu fragmentacji stosuje się różne strategie znajdowania wolnych bloków, które nadają się do powtórnego wykorzystania; próbuje się np. wyszukiwania wolnego bloku najlepiej pasującego rozmiarem do tego, który ma być zaalokowany (*best fit*), czasem robi się wręcz odwrotnie, tj. używany jest pierwszy wolny blok, który ma wystarczająco dużą wielkość (*first fit*), niekiedy zaś jest to strategia pośrednia (np. *good fit*). Samo wyszukiwanie wśród zwolnionych bloków jest zwykle niewystarczające i algorytmy próbują unikać fragmentacji przez scalanie ze sobą tych wolnych bloków, które są ułożone sekwencyjnie (w pamięci fizycznej). Robi się to również na wiele sposobów, np. scalanie (a w zasadzie jego próba) następuje natychmiast po zwolnieniu danego bloku, czasem dzieje się to z opóźnieniem (np. przy przekroczeniu pewnej liczby wolnych bloków), a zdarza się, że scalanie w ogóle nie jest przeprowadzane.

Jako przykład efektywnego sposobu dynamicznego zarządzania



Rys. 3. Struktura pamięci w stercie

pamięcią (stertą) zostanie przedstawiony algorytm wykorzystujący prostą listę dwukierunkową i granulację pamięci sterty o stałej wielkości. Od razu należy zaznaczyć, że maksymalny czas wykonania operacji przydziału pamięci nie jest stały w sensie zależności od parametrów systemu. Operacja zwalniania pamięci (*free*) jest zawsze wykonywana w stałym czasie (złożoność $O(1)$). Funkcja przydziału pamięci potrzebuje w najgorszym razie czasu, który jest proporcjonalny do liczby bloków używanych przez algorytm jako minimalna jednostka alokacji (złożoność $O(n)$, np. n=8192/32=256). Z tego też względu nie jest to dobry wybór dla mikrokontrolerów, które mają do dyspozycji megabajty RAM–u, natomiast dla małych ARM–ów czy AVR–ów algorytm sprawdza się znakomicie. Ponieważ jednostka alokacji jest ustalana przed kompilacją systemu i nie zmienia się w trakcie jego działania, spełnione jest zatem wymaganie, że maksymalny czas wykonania może być wyznaczony *a priori*.

Do kluczowych cech prezentowanego rozwiązania należą:

– Użycie listy dwukierunkowej do przechowywania listy zwolnionych bloków, dzięki czemu wstawianie i usuwanie elementów takiej listy odbywa się w stałym czasie – w szczególności nie zależy od jej długości.

– Dystrybucja (rozkład liczby bloków w funkcji ich wielkości) wolnych bloków staje się z czasem działania systemu zbliżona do dystrybucji żądań, co w efekcie pozwala na dobre „wpasowywanie" się wielkości przydzielanych bloków w wielkość bloków zwolnionych. Zaletą takiego zachowania się dystrybucji wolnych bloków jest to, że można użyć najprostszego algorytmu znajdowania wolnego bloku typu *first fit* (tj. pierwszego, który jest wystarczająco duży).

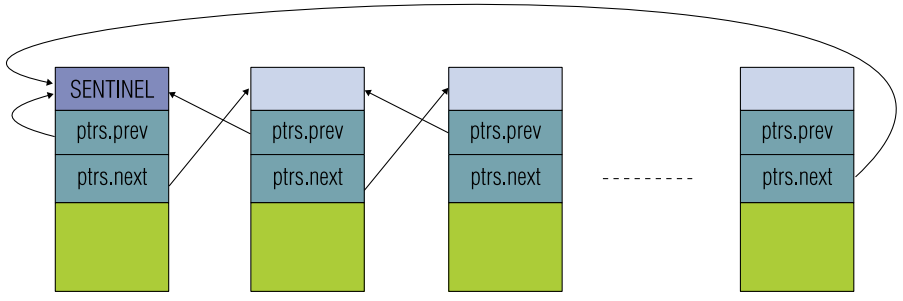
– Stała wielkość najmniejszej jednostki alokacji – minibloku. Wszelkie żądania przydziału pamięci są zaokrąglane w górę do najbliższej wielokrotności rozmiaru minibloku. Warto też zauważyć, że wielkość minibloku jest jednym z najważniejszych parametrów pozwalających na dostrojenie algorytmu do konkretnego systemu (rozmiar słowa procesora, wielkość RAM–u). Z jednej strony mniejsze minibloki powodują mniejszą fragmentację wewnętrzną, z drugiej zaś wymagają dłuższego przetwarzania (wydłuża się maksymalna długość listy wolnych bloków).

Struktura bloku pamięci jest stosunkowo prosta – w pierwszym minibloku wchodzącym w jego skład przechowywana jest informacja o długości całego bloku, jego stanie (zaalokowany lub zwolniony) i jego pozycji w liście (jeżeli jest to blok wolny). Do tego celu wykorzystuje się typ *Block_T*, który jest strukturą (**list. 1**).

Pole `blocks` zawiera w sobie informację o długości całego bloku pamięci i jest ustawiane na liczbę minibloków, które wchodzą w skład tegoż bloku (np. alokacja 100 bajtów przy minibloku wielkości 16 bajtów spowoduje umieszczenie tu liczby 7). Najstarszy bit tego pola jest wykorzystany jako znacznik określający status bloku – zajęty czy wolny (ogranicza to maksymalną długość bloku do wielkości 2^N`*L-sizeof(Block_Size_T)`, gdzie `N` to liczba bitów typu `Block_Size_T`, a `L` to wielkość minibloku). Następna część struktury – unia `header` – jest używana dwojako. W blokach zajętych stanowi po prostu początek danych aplikacji, a w blokach zwolnionych umieszczone są tam wskaźniki wykorzystywane przez listę dwukierunkową.

List. 1.

```
01 typedef struct Block_Tag
02 {
03   BlockSize_T blocks;
04   union
05   {
06      uint8_t user_memory[BLOCK_SIZE - sizeof(BlockSize_T)];
07      struct
08      {
09         struct Block_Tag *prev;
10         struct Block_Tag *next;
11      } ptrs;
12   } header;
13 } Block_T;
```



Rys. 4. Struktura listy dwukierunkowej

List. 2.
```
1 void free(void *ptr)
2 {
3   Block_T *top;
4   Block_T *base_area;
5   MEM_DECLARE_CRITICAL;
6
7   if (NULL == ptr)
8   {
9     return;
10    }
11
12    top = mem_pool + number_of_blocks;
13    base_area = TO_BLOCK_PTR(ptr);
14
15    if ((base_area < mem_pool) || (base_area >= top))
16    {
17      return;
18    }
19
20    MEM_ENTER_CRITICAL();
21    base_area->blocks &= ~RESERVED_BIT;
22    insert_after(base_area);
23    MEM_EXIT_CRITICAL();
24  }
```

System zarządzania pamięcią dynamiczną przechowuje zaskakująco mało „prywatnych" informacji. Właściwie jest to tylko wskaźnik do listy wolnych (nieużywanych przez aplikację) bloków pamięci, który jest jednocześnie znacznikiem końca sterty. Wspomniana lista zawiera na początku tylko dwa bloki: początkowy znacznikowy i drugi blok o rozmiarze całej sterty. Jak już wspomniano, lista jest dwukierunkowa, czyli każdy jej element zawiera w sobie informacje (wskaźniki) o poprzedniku i następcy.

Operacja zwolnienia pamięci jest bardzo prosta, a przez to efektywna. Wskaźnik będący parametrem funkcji *free* jest adiustowany (linia 13, **list. 2**), tak aby wskazywał na początek minibloku (wystarczy „cofnąć" go o długość pola `blocks`). Po tym zabiegu można już bezpośrednio operować na polach struktury `Block_T`. Najpierw jest sprawdzane, czy podany wskaźnik nie jest „pusty" (`NULL`) i czy mieści się w zakresie sterty (linie 7...18, list. 2). Jeżeli te warunki są spełnione, to kasowany jest bit zajętości bloku (linia 21, list. 2), a cały blok jest wstawiany na początek listy wolnych bloków (linia 22, list. 2). Jak można zauważyć, operacja jest wykonywana zawsze w tym samym czasie.

List. 3.
```
1 void *malloc(size_t size)
2 {
3   BlockSize_T blocks_required;
4   Block_T *block;
5   MEM_DECLARE_CRITICAL;
6
7   blocks_required =
8    ((BlockSizeAddOp_T) size +
9     (BlockSizeAddOp_T) sizeof(BlockSize_T) +
10       (BlockSizeAddOp_T) sizeof(Block_T) - 1) / (BlockSizeAddOp_T) BLOCK_
SIZE;
11
12    MEM_ENTER_CRITICAL();
13
14    if (NULL == sentinel)
15    {
16      init_mem_pool();
17    }
18
19    for (block = sentinel->header.ptrs.next;
20       block != sentinel;
21       block = block->header.ptrs.next)
22    {
23      merge_blocks(block);
24      if (block->blocks >= blocks_required)
25      {
26        break;
27      }
28    }
29
30    if (block == sentinel)
31    {
32      MEM_EXIT_CRITICAL();
33      return NULL;
34    }
35
36    if (block->blocks > blocks_required)
37    {
38      split_block(block, blocks_required);
39    }
40
41    unlink(block);
42    block->blocks |= RESERVED_BIT;
43
44    MEM_EXIT_CRITICAL();
45
46    return block->header.user_memory;
47  }
```

Bardziej złożona i ciekawa jest operacja przydziału pamięci. Najpierw jest sprawdzane, czy podsystem dynamicznego zarządzania pamięcią jest zainicjowany; jeżeli nie, to funkcja wywołuje odpowiednią procedurę (linie 14...17, **list. 3**). Uwaga – jeżeli jest taka potrzeba, to inicjację można też wykonać *explicite* w kodzie startowym programu. Potem algorytm przegląda listę wolnych bloków, a jednocześnie dla każdego z nich uruchamiana jest procedura scalania bloków (linie 19...28, list. 3). W zależności od rozmiaru wolnego bloku (po scaleniu) podejmowane są różne działania.

Jeżeli dany blok jest większy niż potrzeba, tzn. większy niż wymagany przez parametr funkcji `malloc`, to ów blok jest dzielony na dwa nowe bloki – pierwszy z nich ma wielkość żądanej pamięci i jest zwracany jako wynik operacji `malloc`, drugi zaś pozostaje na liście wolnych bloków (linie 19...28, list. 3).

Jeżeli blok ma wielkość dokładnie wymaganą przez `malloc`, to jest po prostu zwracany jako wynik funkcji.

W obu tych sytuacjach przed wyjściem z funkcji `malloc` ustawiany jest bit zajętości bloku, a blok jest usuwany z listy wolnych bloków (linie 41...42, list. 3) – wówczas wskaźnik zwracany przez funkcję (linia 46) wskazuje na obszar pamięci zaczynający się nie na początku minibloku, lecz na miejscu bezpośrednio po polu `blocks`.

Jeżeli przeszukanie całej listy nie pozwoliło na znalezienie bloku o odpowiedniej długości (lub nie ma żadnych wolnych bloków), to funkcja `malloc` zwraca wynik `NULL` (linie 30...34, list. 3).

Operacja łączenia bloków jest stosunkowo prosta. Począwszy od zadanego (jako parametr) bloku bazowego znajdowany jest początek następnego bloku w fizycznej pamięci RAM (linie 5 i 11, **list. 4**). Jeże-

R E K L A M A

μ's
MICROS sp.j.
Hurtownia podzespołów elektronicznych
Kraków, ul. Godlewskiego 38
tel. (012) 636 95 66
fax. (012) 636 93 99
e-mail: biuro@micros.com.pl
Szeroki wybór podzespołów elektronicznych. Prowadzimy obsługę sklepów, zakładów produkcyjnych oraz innych podmiotów gospodarczych.
szczegóły w katalogu internetowym:
http://www.micros.com.pl

- PRZETWORNIKI PIEZOAKUSTYCZNE
Z GENERATOREM LUB BEZ GENERATORA
W OBUDOWIE LUB BEZ OBUDOWY ("BLASZKI")
JEDNOTONOWE LUB WIELOTONOWE
- PRZETWORNIKI ELEKTROMAGNETYCZNE
Z GENERATOREM LUB BEZ GENERATORA
DOSTĘPNE TAKŻE W WERSJI SMD
- MIKROFONY POJEMNOŚCIOWE
- SYRENY

li tenże blok (następnik) jest wolny (tj. ma nieustawiony bit zajętości), to usuwany jest z listy bloków wolnych (linia 9, list. 4) i przyłączany do bloku bazowego przez wydłużenie jego długości o długość (linia 10) bloku następnika. Operacja jest powtarzana, aż kolejny następnik nadaje się do scalenia (pętla w liniach 7...12, list. 4).

Ponieważ występują tutaj dwie zagnieżdżone pętle (każda o potencjalnej długości maksymalnej liczby minibloków w systemie), może się wydawać, że maksymalny czas (najgorszy przypadek, kiedy cała sterta jest pofragmentowana do pojedynczych minibloków) wykonania operacji przydziału pamięci jest proporcjonalny do kwadratu maksymalnej liczby minibloków (złożoność $O(n^2)$). Zewnętrzna pętla to pętla w funkcji `malloc`, a pętla wewnętrzna jest pętlą w funkcji `merge_blocks`. Należy tu jednak uwzględnić fakt, że w operacji łączenia pętla wykona się tylko wtedy, gdy nastąpi przynajmniej jedno scalenie bloków, co zaś powoduje automatyczne skrócenie pętli zewnętrznej.

List. 4.
```
1 static void merge_blocks(Block_T *block)
2 {
3   Block_T *successor;
4
5   successor = block + block->blocks;
6
7   while (0 == (successor->blocks & RESERVED_BIT))
8   {
9     unlink(successor);
10      block->blocks += successor->blocks;
11      successor = block + block->blocks;
12    }
13  }
```

Czytelnik zapewne zauważył w prezentowanym kodzie takie wyrażenia, jak: `MEM_ENTER_CRITICAL` i `MEM_EXIT_CRITICAL`. Są to operacje niezbędne w systemach wielozadaniowych i ich rzeczywista implementacja zależy od użytego RTOS–a. W najprostszym przypadku może to być odpowiednie zablokowanie (z zapamiętaniem statusu) przerwań i ich odblokowanie (odtworzenie ze statusu), jednak jest to mało efektywne i często niedopuszczalne (podsystem pamięci blokuje całe urządzenie). Zdecydowanie lepszym rozwiązaniem jest użycie prostego mechanizmu synchronizacyjnego, jak np. *mutex*.

Na koniec chciałbym wspomnieć, że istnieją co najmniej dwa algorytmy *open source* służące do dynamicznego zarządzania pamięcią (*Half–Fit* i TLSF), których maksymalny czas wykonania jest stały zarówno dla operacji przydziału, jak i zwalniania pamięci.

Artur Lipowski
Delphi Poland SA

Jeżeli temat zainteresuje Czytelników, zajmiemy się nim w niedalekiej przyszłości.

R E K L A M A

ACS ELEKTRONIK
SZYDŁOWIEC 26-500 ul. Kolejowa 11
e-mail: acs@acs.ats.pl tel./fax. 048 617-60-00
WWW.ACS.ATS.PL
PROFESJONALNE URZĄDZENIA LABORATORYJNE

OSCYLOSKOPY CYFROWE
ADS220
- pasmo 60MHz
- sampling 2 x 200MSPS
- rozdzielczość 8bit
- 2 kanały + EXT
- zakres 5mV - 5V
- analiza FTT, pomiary: freq, okres, pk-pk, RMS, średnia...
- interpolacja sin(x)/x, kalibracja 24bit
- z notebookiem mobilne stanowisko pomiarowe

PROGRAMATORY PAMIECI ACS
VI-LAB
ERICA
PS32
- wirtualne laboratorium - 3 funkcje programator, emulator RT, tester
- podstawka ZIF 48Pin 0,3"- 0,6"
- emulacja pamięci w czasie rzeczywistym 27xxx, 62xxx, 24cxx, 93cxx, 25/95xxx
- możliwość dopisywania własnych układów

PROGRAMATORY PAMIECI XELTEK
SP3000U
- obsługa ponad 20,000 układów
- możliwość pracy bez komputera
- wbudowany LCD, klawiatura, pamięć CF-256MB
- komunikacja port USB
- podstawka ZIF 48Pin 0,3"- 0,6"
- praca z układami 100pin
- adaptery 1:1
- tester TTL,CMOS, PLD, SRAM, DRAM, MCU