Emulacja niezawodnej, trwałej pamięci EEPROM w mikrokontrolerze 8-bitowym

Nowoczesne mikrokontrolery nie zawsze dysponują dużą ilością pamięci. Czasem jest ona silnie ograniczona przez producenta, by móc zaoferować komponent o najniższej cenie. Co zrobić, gdy wybierzemy taki układ, ale potrzebujemy odrobinę dodatkowej, wytrzymałej pamięci nieulotnej, nie chcąc przy tym montować zewnętrznego układu? Można emulować EEPROM z użyciem wbudowanej pamięci programu.

Niektóre mikrokontrolery, mimo że nie mają wbudowanej pamięci EEPROM, mogą zapisywać i odczytywać swoją pamięć programu (FPM – Flash Program Memory). Zazwyczaj nie jest ona duża, ale bywa, że wystarcza z zapasem. Problem w tym, że próba jej użycia jako dodatkowej pamięci na tymczasowe dane może grozić ich utratą. Ponieważ pamięć FPM nie jest przeznaczona do częstego zapisywania, jest wytwarzana w technologii, nadającej jej ograniczoną trwałość. W 8-bitowych mikrokontrolerach PIC żywotność tej pamięci wynosi ok. 10 tysięcy cykli. Choć jest to wartość zbliżona do wytrzymałości nowoczesnych, konsumenckich pamięci NAND FLASH, nie spełnia ona wymagań stawianych pamięciom HEF (High Endurance Flash). Jednakże korzystając z jej specyfiki można w łatwy sposób zwiększyć jej efektywną trwałość, choć wiąże się to z ograniczeniem ilości danych, które można w niej zmieścić.

Organizacja pamięci FPM

W mikrokontrolerach 8-bitowych PIC, takich jak np. PIC10F322, pamięć FPM składa się z określonej liczby rzędów, a te zawierają po 16 słów 14-bitowych. Przykładowo, układ PIC10F322 zawiera 512 słów pamięci FPM, a więc 32 rzędy. Przy 14 bitach na słowo oznacza to ponad 7000 bitów, które można w niej zmieścić. Jest to zarazem 7 razy więcej niż wbudowana w omawiany układ dodatkowa pamięć HEF o wytrzymałości 100 tysięcy cykli.

Do poszczególnych słów w rzędach można się odwoływać poprzez kolejne adresy, przy czym pierwsze słowo ma zawsze adres w formacie XX0h, a ostatnie (szesnaste) – XXFh. Starsze bity określają numer adresowanego wiersza.

Ponadto, zapis i czyszczenie pamięci odbywają się w dosyć specyficzny sposób. Mikrokontroler może czyścić pamięć FPM tylko całymi wierszami, co powoduje ustawienie każdego bitu w wierszu. Zapis polega więc na wyzerowaniu odpowiednich bitów w zapamiętywanych słowach. Korzystając z tych zależności można opracować optymalny sposób zapisywania i odczytywania pozostałej, wolnej pamięci, aby zmaksymalizować jej trwałość, a jednocześnie wciąż dało się w niej zmieścić trochę danych.

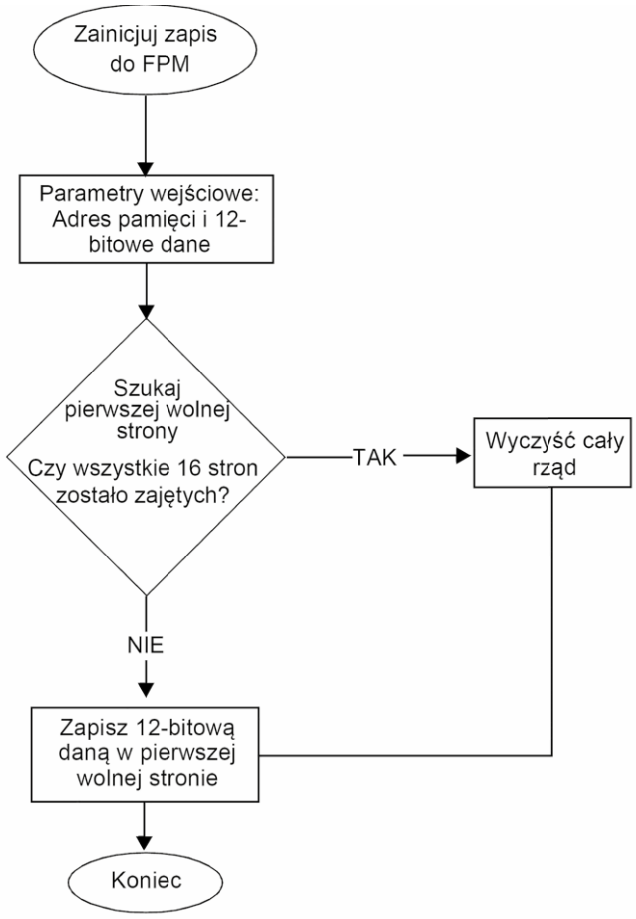
Zapis stronami

Trwałość pamięci programu odnosi się do liczby cykli zapisu i odczytu, i jest podana dla pojedynczej komórki, a więc dotyczy każdego bitu. Gdyby jednak tę pamięć podzielić na strony i zapisywać dane cyklicznie na kolejnych stronach, wtedy efektywna żywotność pamięci znacznie by wzrosła – tyle razy, na ile stron podzielilibyśmy dane. Ze względu na „kasowanie" pamięci FPM całymi rzędami i możliwość zapisywania tylko zer na wybranych pozycjach adresowanego słowa, optymalne będzie podzielenie wierszy na strony. Skoro każdy wiersz składa się z 16 słów 14-bitowych, które łatwo zapisać pojedynczym poleceniem, dobrym pomysłem będzie podział wiersza na 16 stron. Każda ze stron w danym wierszu będzie adresowana przez użytkownika w ten sam sposób (adresem całego rzędu), a dopiero wewnętrzna funkcja zapisująca będzie tłumaczyła podany adres rzędu na adres konkretnego słowa i zapisywała lub odczytywała z niego dane. Pozwoli to 16-krotnie zwiększyć żywotność takiej pamięci, gdyż statystycznie, każde słowo pamięci będzie zapisywane 16-krotnie rzadziej. Niestety wiąże się to z 16-krotnym zmniejszeniem dostępnej dla użytkownika pojemności pamięci FPM.

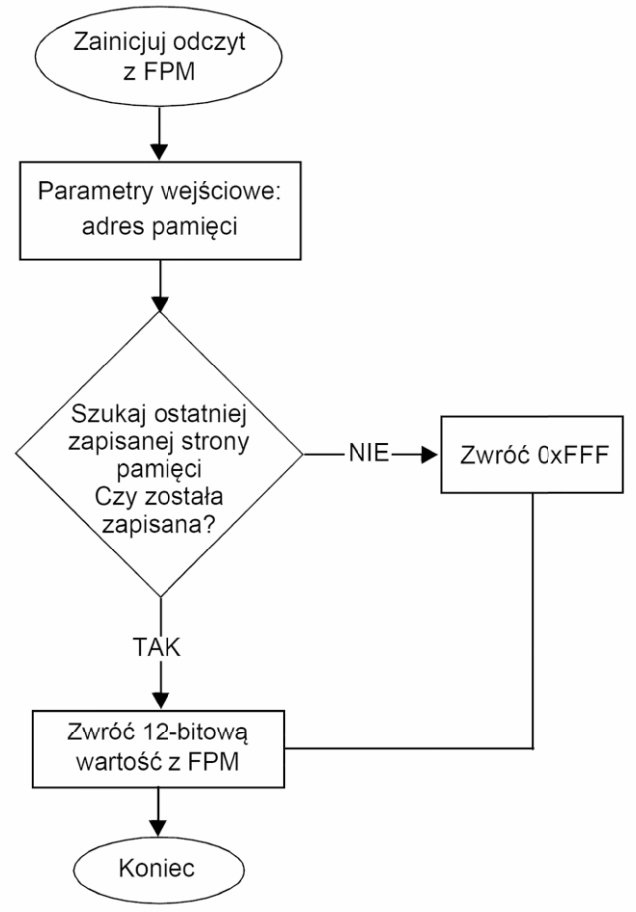
Proces zapisu

Funkcja zapisu tak przygotowanych danych jest całkiem prosta. Jako jej argumenty wystarczy podać adres wiersza oraz daną do zapisania. Jednakże trzeba w jakiś sposób rozpoznawać, która strona w rzędzie została ostatnio zapisana.

Po wyczyszczeniu rzędu, wszystkie bity są ustawione na jedynki. Funkcja może wtedy zaczynać od zapisu przekazanej przez użytkownika danej na pierwszym słowie. By przy kolejnym zapisie funkcja „wiedziała", że ma skorzystać z drugiego słowa, w trakcie pierwszego zapisu zerowany jest jeden z najstarszych bitów słowa 14-bitowego i służy on do rozpoznawania,



Rysunek 1. Algorytm zapisu w emulowanej pamięci



Rysunek 2. Algorytm odczytu z emulowanej pamięci

że dana strona została już wykorzystana i trzeba użyć kolejnej. Oznacza to zarazem, że długość zapisywanych słów musi zostać ograniczona choćby do 13 bitów, choć w praktyce wygodniejsze będzie posługiwanie się liczbami 12-bitowymi.

Oznaczając w ten sposób kolejne słowa jako używane wystarczy, by funkcja zapisująca szukała kolejnego wolnego słowa i tam umieściła dane, zerując przy tym odpowiedni bit. Gdy okaże się, że po przeszukaniu całego wiersza, wszystkie słowa są już zajęte, można ponownie go wyczyścić (a więc ustawić jedynki) i zapisać podaną daną na pierwszej pozycji. Algorytm realizujący ten mechanizm przedstawiono na **rysunku 1**.

Proces odczytu

Odczyt danych przebiega bardzo podobnie z tym, że kierunek poszukiwania aktualnej strony jest przeciwny. Funkcja zaczyna czytać słowa w podanym wierszu poczynając od ostatniego (XXFh) do momentu aż znajdzie słowo, w którym najstarszy bit jest wyzerowany. Następnie je zwraca. Jeśli dojdzie do początku wiersza (XX0h) i nie znajdzie wyzerowanych bitów, to oznacza, że żadna wartość nie została zapisana w tym wierszu. Algorytm realizujący ten mechanizm przedstawiono na **rysunku 2**.

Przykładowa implementacja

Na **listingu 1** pokazano przykładową implementację emulacji pamięci wysokiej wytrzymałości w pamięci programu. Korzystanie z przygotowanej, prostej biblioteki sprowadza się do używania jednej funkcji *uint16_t ReadWrite_HEFlash(uint8_t rw, uint16_t data, uint8_t rowstartaddr)*. Pierwszym argumentem jest parametr określający rodzaj operacji („1” to zapis a „0” to odczyt), drugim 12-bitowe dane do zapisu, a trzecim adres rzędu, w którym dane mają być zapisywane, lub z którego będą odczytywane. Naturalnie, funkcja ignoruje drugi argument, jeśli prowadzony ma być odczyt. W adresie natomiast zawsze jest ignorowany drugi (czyli mniej znaczący) z bajtów, by wskazywać na cały rząd, a nie jego wybrane słowo.

W podanej implementacji stan strony jest zapisywany na dwóch najstarszych bitach każdego słowa w następujący sposób:

- Jeśli bity te <13:12> mają wartość 0b11 to znaczy, że dana strona jest gotowa do zapisu.
- Jeśli bity te <13:12> mają wartość 0b10 to znaczy, że dana strona jest zajęta.

Oczywiście wyczyszczenie rzędu powoduje sprawienie, że wszystkie strony stają się gotowe do zapisu. W przypadku próby odczytu wartości z pustego rzędu, funkcja zwraca wartość FFFh.

Ograniczenia i uwagi

Dokonując zapisów w pamięci programu w trakcie, gdy program jest wykonywany, należy upewnić się, że przypadkiem nie zostanie nadpisany obszar przez niego zajmowany. Pamięć zajmowaną przez skompilowany program można określić zaglądając do mapy pamięci w środowisku kompilatora (w tym przypadku MPLAB X) po kompilacji. Ponadto należy upewnić się, że nie zostaną nadpisane wektory resetu i przerwań, zlokalizowane najczęściej pomiędzy adresami 0000h i 0004h.

Jeśli przygotowany program korzysta z wskazanej biblioteki oraz z przerwań, należy upewnić się, że przerwania nie zakłócą

pracy mechanizmu zapisywania na kolejnych stronach. Dlatego w kodzie na list. 1 należy odkomentować linie „save_INTERRUPT();" i „INTCON = SaveInt;". Jeśli program nie obsługuje przerwań, funkcje te można pozostawić wykomentowane.

Korzystanie z pamięci FPM na dane może również sprawiać problemy podczas aktualizacji oprogramowania. Środowiska takie jak MPLAB X domyślnie czyszczą najpierw całą pamięć FPM przed jej zapisem. Jeśli użytkownik chce zachować wcześniej zapisane dane, musi włączyć opcję ochrony określonego zakresu adresów pamięci. W MPLAB X można to łatwo zrobić modyfikując ustawienia projektu. Po włączeniu okna ustawień należy znaleźć pozycję odpowiadającą wybranemu programatorowi (np. PICkit 3, REAL ICE lub ICD3), po czym na liście opcji odszukać polecenia „Preserve Program Memory". Po włączeniu tej opcji możliwe jest podanie zakresu adresów pamięci, która ma być chroniona (**rysunek 3**).

Podsumowanie

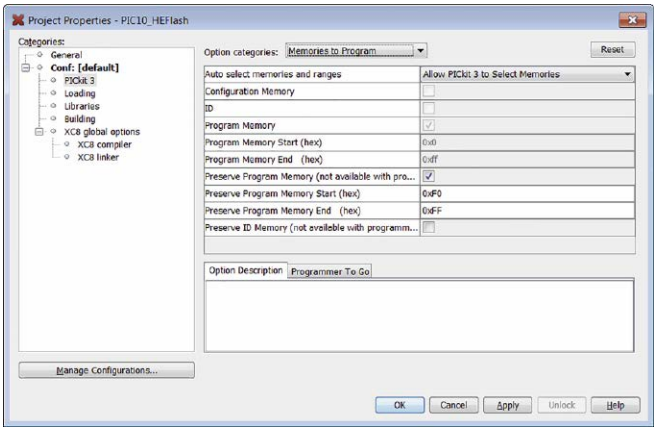
Użycie pamięci FPM w omówiony sposób nie pozwala uzyskać dużych zasobów, ale za to wytrzymałość przygotowanych komórek jest bardzo duża, gdyż statystycznie wynosi 160 tysięcy cykli, a więc ponad półtorakrotnie tyle, co wytrzymałość wbudowane w układy PIC pamięci HEF. Warto dodać, że trwałość ta jest zachowywana w szerokim zakresie temperatury pracy układu, tj. od –40°C do +85°C. W przykładowym mikrokontrolerze PIC10F322, przy założeniu, że połowa jego pamięci FPM została zajęta przez program, pozostaje 256 wolnych słów, zorganizowanych w 16 rzędów. Oznacza to, że dla użytkownika korzystającego z zaprezentowanej biblioteki pozostaje miejsce na 16 zmiennych 12-bitowych, które można trwale przechowywać. Są to 192 bity, a więc ok. 25% bitów dostępnych do zapisu w pamięci HEF wbudowanej w ten układ. Ta sama biblioteka będzie także poprawnie działać dla innych układów z serii PIC10 i PIC12.

MARCIN KARBOWNICZEK, EP

Artykuł przygotowano na podstawie noty aplikacyjnej, napisanej przez Willema J. Smita z firmy Microchip Technology Inc.

Listing 1. Funkcje omawianej biblioteki

```
uint16_t ReadWrite_HEFlash(uint8_t rw, uint16_t data, uint8_t rowstartaddr) {
  uint8_t addr;
  Save_INTERRUPT();
  if (rw == 1) {
    addr = rowstartaddr;
  } else {
    addr = (rowstartaddr + 0xF);
  }

  uint8_t cnt;
  //Poszukiwanie pierwszej wolnej strony w wierszu
  for (cnt = 0x10; cnt != 0x0; cnt--) {
    Read_FLASH(addr); //Odczytaj pamięć
    if (rw == 1) { //Jeśli prowadzony jest zapis
      if (PMDATH == 0x3F) { //Sprawdź, czy strona jest dostępna (czy jej treść zawiera 14 jedynek
        PMDAT = data; //Przygotuj dane użytkownika do zapisu
        asm(„bsf PMDATH, 5"); //Ustaw flagę zapisu fragmentu pamięci FPM
        asm(„bcf PMDATH, 4");
        Write_FLASH(); //Zapisz pamięć
        break;
      }
      addr++;//przejdź do kolejnej strony
      //Sprawdź czy cały wiersz został już zapisany
      if (cnt <= 0x1) {
        //Jeśli został zapisany cały wiersz, zostanie on teraz wyczyszczony
        PMCON1 = 0b00010100;
        PMADR = rowstartaddr;
        Unlock_FLASH();
        //Zapis danych użytkownika po wyczyszczeniu
        PMDAT = data; //Przygotuj dane użytkownika do zapisu
        asm(„bsf PMDATH, 5"); //Ustaw flagę zapisu fragmentu pamięci FPM
        asm(„bcf PMDATH, 4");
        Write_FLASH(); //Zapisz pamięć
      }
    } else { //W przypadku odczytu odnajdź ostatnią zapisaną stronę
      if ((PMDATH & 0x20) && (PMDATH != 0x3F)) { //Sprawdź czy strona jest zapisana
        data = (PMDAT & 0xFFF); //Odczytaj dane ze strony
        break;
      }
      addr--;//przejdź do wcześniej strony
    }
    INTCON = SaveInt;
  }
  return data;
}

void Save_INTERRUPT(void) {//Wstrzymywanie obsługi przerwań
  SaveInt = INTCON;
  GIE = 0;
}

void Read_FLASH(uint8_t address) {
  PMADR = address;
  PMCON1 = 0b00000001; //Ustawianie operacji na odczyt pamięci Flash
  while (RD);
}

void Unlock_FLASH(void) {
  PMCON2 = 0x55; //Przygotowywanie pamięci do czyszczenia
  PMCON2 = 0xAA;
  WR = 1;
  while (WR);
  WREN = 0;
}

void Write_FLASH(void) {
  //PMCON1 = 0b00100100;
  //Unlock_FLASH();
  PMCON1 = 0b00000100;
  Unlock_FLASH();
}
```



Rysunek 3. Opcje ochrony fragmentu pamięci FPM przed zapisem w środowisku Microchip MPLAB X