Wear leveling, czyli równoważenie zużycia pamięci EEPROM

Przykłady oprogramowania dla mikrokontrolerów AVR

W wielu systemach procesorowych zachodzi potrzeba zapamiętywania parametrów, na przykład ustawień użytkownika również po wyłączeniu napięcia zasilającego. Idealnym rozwiązaniem w takim wypadku wydaje się użycie pamięci EEPROM, w którą często są wyposażone nowoczesne mikrokontrolery. To rozwiązanie, mimo iż jest nieskomplikowane, tanie i skuteczne ma jednak wadę wynikającą z cech pamięci tego typu i sposobu jej kasowania oraz zapisu. Pamięci EEPROM umożliwiają przeprowadzenie ograniczonej liczby zapisów (a w zasadzie kasowania) każdej ich komórki, po której wykonaniu nie ma pewności, iż przechowywane tam dane są poprawne. W artykule podano sposób przedłużenia czasu funkcjonowania takiej pamięci. Przykłady wykonano w języku C dla mikrokontrolerów AVR.

Współczesne pamięci EEPROM wytrzymują co najmniej 100 tysięcy operacji kasowania/zapisu. Z jednej strony nie wydaje się, by była to liczba zbyt mała w rozwiązaniach praktycznych, z drugiej zaś, istnieje wiele potencjalnych aplikacji, gdzie liczba 100 tysięcy cykli kasowania i zapisu może zostać osiągnięta dosyć szybko. Oczywiście, poprawnie skonstruowane urządzenie i przemyślana aplikacja użytkownika powinny zapewniać odpowiednie „obchodzenie" się z tym rodzajem pamięci, lecz łatwo sobie wyobrazić sytuację, dla której nie ma innej możliwości zapisywania bieżących danych, jak użycie wbudowanej pamięci EEPROM. Co oczywiste, istnieje wiele sposobów poradzenia sobie z tym problemem, jak choćby detekcja zaniku zasilania mikrokontrolera i natychmiastowy zapis interesujących nas wartości do pamięci EEPROM, lecz zwykle wymaga to zastosowania pewnych rozwiązań sprzętowych, jak choćby podtrzymanie zasilania na czas zapisu lub implementacji układu wykrywającego takie zjawisko, co nie zawsze jest pożądane. Można również każdorazowo zapisywać interesujący nas blok danych zaopatrując go w sumę kontrolną CRC8, której odczyt i porównanie (po restarcie mikrokontrolera) da pewność, czy zapisane dane są poprawne, jednak w wypadku błędu nie mamy pewności odnośnie do odczytania z pamięci jakichkolwiek poprawnych wartości, co trudno zaakceptować. Poza tym, operując ciągle na tym samym zestawie komórek pamięci EEPROM wcześniej, czy później możemy doprowadzić do przekroczenia dopuszczalnej liczby zapisów, gdyż nawet dla zapisu realizowanego z interwałem 1 minuty liczbę 100 tysięcy zapisów przekroczymy po niespełna 70 dniach ciągłego funkcjonowania urządzenia. Oczywiście, błędem byłoby traktowanie pamięci EEPROM, jako miejsca na przechowywanie bieżących zmiennych, bo przecież do tego celu mamy pamięć RAM, ale w ogólnym podejściu musimy wziąć pod uwagę to niekorzystne zjawisko. Jak w takim razie w prosty sposób rozwiązać ten problem? Stosujmy tak zwany Wear leveling, czyli równoważenie zużycia poszczególnych komórek pamięci poprzez wykorzystywanie różnych metod optymalizacji zapisu pozwalających na wydłużanie czasu, przez który dany nośnik danych jest nadal zdatny do użycia.

Technologia, o której mowa, jest znana dość dobrze producentom dysków SSD, gdyż bez jej stosowania, w zależności od typu komórek pamięci, taki dysk mógłby stać się bezużyteczny w całości lub w części nawet po wykonaniu 500 (!) cykli kasowania i zapisu (w przypadku pamięci TLC). W dyskach SSD cały proces jest nadzorowany sprzętowo przez odpowiedni sterownik i oprogramowanie oraz jest dla użytkownika niezauważalny. Krótko mówiąc, tematyka, którą chciałbym poruszyć nie jest niczym nowym, ale z moich obserwacji wynika, że jest to zagadnienie mało znane i rzadko stosowane. Czemu, więc, nie skorzystać z tego rodzaju mechanizmu, zwłaszcza, że jego podstawowa implementacja jest niezmiernie łatwa?

Zasada działania prostego mechanizmu Wear leveling bazuje na fakcie, iż dla wybranej komórki danych rezerwujemy wiele kopii w innych obszarach pamięci EEPROM i każdego nowego zapisu dokonujemy w kolejnym, zarezerwowanym miejscu. Wynika z tego, że żywotność pamięci EEPROM w odniesieniu do naszej danej zwiększa się tyle razy, ile jej kopii zarezerwowano w innych obszarach EEPROM pod warunkiem, że każdy kolejny zapis jest dokonywany w nowym miejscu i po osiągnięciu ostatniej pozycji zaczynamy od nowa, czyli od miejsca o adresie podstawowym. Nasuwa się więc pytanie – skąd po zaniku zasilania, program aplikacji czerpie „wiedzę", od którego miejsca należy zacząć proces zapisywania danej, by zachować założenie równomiernego zużycia pamięci? Musimy wprowadzić drugą zmienną, a w zasadzie bufor kołowy, w którym program aplikacji będzie umieszczał w „sprytny sposób" wskaźnik na nową pozycję do zapisu. Może w tym momencie brzmi to dość mgliście, zatem przejdę do wyjaśnienia sposobu działania programu realizującego tę funkcjonalność.

Załóżmy, że mamy pewien zestaw zmiennych aplikacji użytkownika, które cyklicznie musimy zapisywać w pamięci EEPROM, a które to zgrupowano strukturze danych pokazanej na **listingu 1**. Aby wygodnie było indeksować taką strukturę, która to przecież może zawierać pola o różnej długości, zamieńmy ją na unię przechowującą nasze dane pamiętając, że wielkość pola indexu tejże unii musi

```
Listing 1. Przykładowa struktura danych
typedef struct
{
    uint8_t Size;
    uint16_t Value;
    uint32_t Price;
    uint8_t Data[2];
} configType;
```

```
//Listing 2. Zmodyfikowana struktura danych
typedef union
{
    struct
    {
        uint8_t Size;
        uint16_t Value;
        uint32_t Price;
        uint8_t Data[2];
    };
    uint8_t index[9];
} configType;
```

odpowiadać wielkości całej struktury. Nowy, zmodyfikowany w opisany powyżej sposób, typ danych pokazano na **listingu 2**. Teraz tworzymy zmienną w pamięci EEPROM, która będzie przechowywała nasze dane konfiguracyjne, ale od razu tworzymy wybraną przez nas, a zależną od wielkości dostępnej pamięci EEPROM i oczekiwanego poziomu zwiększenia jej żywotności liczbę kopii, jak na **listingu 3**. Jak pokazano, od razu utworzyliśmy dodatkową zmienną w pamięci EEPROM, a mianowicie zmienną *wlStatBuff*, która to jest buforem kołowym niezbędnym z punktu widzenia mechanizmu pozyskiwania informacji na temat lokalizacji w pamięci, w których należy dokonywać kolejnego zapisu struktury danych *wlConfigBuff* dla zachowania równomiernego zużycia pamięci. W tym miejscu warto zaznaczyć, że zarówno specyfikator EEMEM, który de facto informuje kompilator, że zmienne mają być umieszczone w pamięci EEPROM, jak i dalej używane funkcje „wbudowane" kompilatora AVR-GCC wymagają dołączenia pliku nagłówkowego *avr/eeprom.h* do projektu aplikacji.

W tej chwili zastanawiacie się zapewne, jak działa cały ten mechanizm? Otóż, jego działanie polega na wykonaniu 3 prostych kroków:

1. Znalezieniu miejsca w pamięci EEPROM, w którym należy zapisywać naszą strukturę danych, czyli miejsca, gdzie należy umieścić kolejną jej kopię.
2. Zapisaniu tejże kopii pod ustalonym powyżej adresem.
3. Aktualizacji wartości bufora stanu (zmiennej *wlStatBuff*) by następujące po sobie zapisy (po restarcie aplikacji) odbywały się kolejnych miejscach równoważąc tym samym zużycie pamięci EEPROM.

To wszystko! Jak widać, kluczowym mechanizmem tego, całego procesu, jest algorytm uzupełniania bufora stanu umieszczonego w pamięci EEPROM, by w dalszych krokach program aplikacji każdorazowo po restarcie systemu „wiedział", od którego miejsca należy zacząć zapisywanie kolejnych kopii przykładowej struktury danych. Co należy szczególnie podkreślić, do poprawnej pracy algorytmu jest niezbędne, aby cały dostępny obszar zajęty przez bufor stanu (zmienną *wlStatBuff*) był zainicjowany na samym początku, czyli wyłącznie podczas pierwszego uruchomienia aplikacji, identycznymi wartościami, np. 0xFF (wartość dla skasowanej/nieużywanej dotychczas pamięci EEPROM). Inicjalizację taką można wykonać manualnie, lub w przypadku mikrokontrolerów AVR skorzystać z odpowiedniego bitu konfiguracyjnego, a mianowicie EESAVE. Jego wyzerowanie (w nomenklaturze firmy Atmel ustawienie tam logicznej „1"), będące jednocześnie ustawieniem standardowym (produkcyjnym) wszystkich układów rodziny AVR, powoduje skasowanie całej pamięci EEPROM podczas programowania (a dokładnie kasowania) pamięci Flash mikrokontrolera, czyli podczas wgrywania programu aplikacji. To najprostsze i zarazem skuteczne rozwiązanie nadania używanej wcześniej pamięci EEPROM wartości początkowych (0xFF).

Przejdźmy zatem do mechanizmu wyszukiwania bieżącego miejsca w pamięci EEPROM, pod które należy zapisywać naszą strukturę

```
Listing 3. Deklaracje zmiennych dla realizacji mechanizmu Wear leveling
#define WL_BUFF_SIZE 32  //Liczba kopii struktury danych i jednocześnie rozmiar bufora stanu
configType wlConfigBuff[WL_BUFF_SIZE] EEMEM;
uint8_t wlStatBuff[WL_BUFF_SIZE] EEMEM;
```

```
//Listing 4. Funkcja findStatBuffAddr()
uint8_t findStatBuffAddr(void)
{
    register uint8_t i, prevCellValue, currCellValue;
    for(i=0; i<WL_BUFF_SIZE; ++i)
    {
        prevCellValue = eeprom_read_byte(&wlStatBuff[(i-1) & (WL_BUFF_SIZE-1)])+1;
        currCellValue = eeprom_read_byte(&wlStatBuff[i]);
        if(prevCellValue != currCellValue) break;
    }
    return i;
}
```

```
Listing 5. Funkcja updateStatBuffer()
void updateStatBuffer(void)
{
    register uint8_t prevCellValue, currCellNumber;
    //Ustalamy bieżącą pozycję zapisu w buforze Stanu
    currCellNumber = findCurrBuffAddr();
    //Ustalamy wartość poprzedniego elementu w buforze Stanu
    prevCellValue = eeprom_read_byte(&wlStatBuff[(currCellNumber-1) & (WL_BUFF_SIZE-1)]);
    //Zapisujemy wartość prevCell+1 na bieżącej pozycji bufora Stanu
    eeprom_write_byte(&wlStatBuff[currCellNumber], prevCellValue+1);
}
```

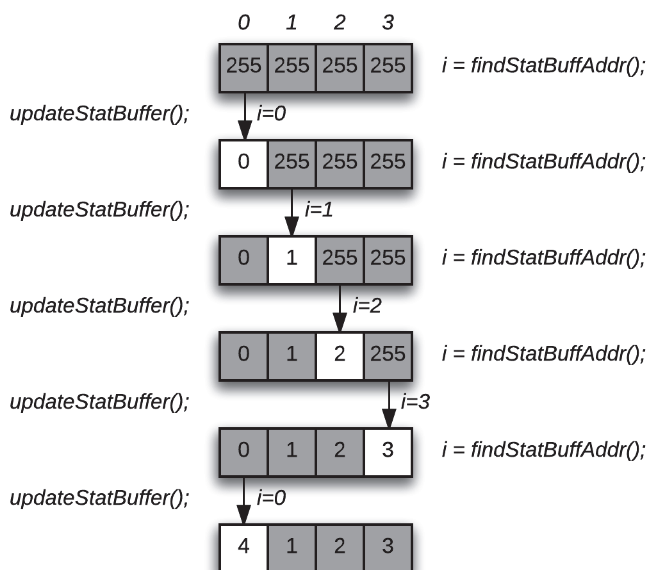
```
Listing 6. Sposób użycia mechanizmu Wear leveling w aplikacji użytkownika
//Ustalamy bieżącą pozycję zapisu w buforze Stanu
uint8_t Addr = findCurrBuffAddr();
//Pod bieżącą pozycją zapisujemy kolejną kopię struktury danych
eeprom_update_block(Config.index, &wlConfigBuff[Addr].index, sizeof(configType));
//Aktualizujemy bufor stanu
updateStatBuffer();
```

```
Listing 7. Defninicje obsługi przerwania EEPROM Ready Interrupt
#define EEPROM_INTR_ENABLE EECR |= (1<<EERIE)  //EEPROM Ready Interrupt Enable
#define EEPROM_INTR_DISABLE EECR &= ~(1<<EERIE)  //EEPROM Ready Interrupt Disable
```

danych. Mechanizm ten bazuje na pewnych założeniach, dla funkcji zarządzającej buforem stanu, czyli aktualizującej ten bufor po wykonaniu zapisu kopii struktury. Funkcja ta działa w taki sposób, iż pod adresem bieżącego miejsca w buforze stanu, a więc bieżącego miejsca, w którym zapisano ostatnią kopię struktury konfiguracyjnej, zapisuje wartość, która znajduje się pod adresem poprzednim powiększoną o 1. Wynika z tego, że funkcja, której zadaniem jest znalezienie bieżącego, wolnego miejsca, w którym należy zapamiętać kolejną kopię struktury danych, przegląda bufor kołowy (zmienną *wlStatBuff*) w poszukiwaniu indeksu (adresu) miejsca, dla którego wartość elementu poprzedniego plus 1 jest różna od wartości elementu bieżącego (bieżącego indeksu). Obraz jest wart więcej niż 1000 słów, więc warto spojrzeć na **rysunek 1** pokazujący sposób działania mechanizmu obsługi bufora stanu (dla uproszczenia założono 4-elementową wielkość bufora stanu).

Jak wspomniano, na początku zakładamy, że wszystkie komórki bufora stanu mają tę samą wartość – w naszym wypadku 0xFF. W kolejnym kroku, rozpoczyna działanie funkcja *findStatBuffAddr()*, której zadaniem jest znalezienie bieżącego adresu, pod którym należy zapisywać kolejną kopię struktury danych. Funkcja ta każdorazowo przeszukuje cały bufor (od elementu nr 0 do n-1, porównując element poprzedni z bieżącym), więc pierwsze jej porównanie jest wykonywane na elementach numer 3 i numer 0. Wartość elementu numer 3 (0xFF) powiększona o 1 (0x00) jest różna od wartości elementu numer 0 (0xFF), funkcja zwraca adres 0x00. Program aplikacji powinien w tym miejscu zapisać kopię struktury danych pod adresem *wlConfigBuff[0]*, a następnie wywołać funkcję *updateStatBuffer()*, której zadaniem jest aktualizacja bufora stanu, by kolejne przeszukiwania zakończone były wskazaniem poprawnego elementu. Ta aktualizacja polega na tym, że w miejscu bieżącego adresu w buforze stanu (w przypadku pierwszego kroku to element *wlStatBuff[0]*) jest zapisywana wartość poprzedniego elementu (w wypadku pierwszego kroku to element *wlStatBuff[3]*, czyli wartość 0xFF) powiększona o jeden – w tym wypadku będzie to 0x00. Kolejne, analogiczne kroki pokazane na rys. 1 towarzyszące następnym zapisom kopii struktury danych powodują wskazanie kolejnych elementów bufora stanu i ich stosowną aktualizację, przy czym należy zauważyć, że po aktualizacji elementu o numerze 3 funkcja przechodzi z powrotem do elementu nr 0 i tak dalej.

Pora na przedstawienie wspomnianych wcześniej funkcji. Funkcję odpowiedzialną za znalezienie bieżącego adresu, pod którym należy zapisywać kolejną kopię struktury danych, pokazano na **listingu 4**, natomiast na **listingu 5** funkcję, której zadaniem jest aktualizowanie bufora stanu.



Rysunek 1. Sposób działania mechanizmu obsługi bufora stanu

Wszystko gotowe! Pora na przedstawienie krótkiego listingu, który prezentuje sposób zastosowania w aplikacji mechanizmu Wear leveling do zapisywania naszej przykładowej struktury danych. Cały proces zaprezentowano na **listingu 6**. Zmienna *Config* typu *configType* jest wejściową strukturą danych w pamięci RAM, której kopię będziemy przechowywali w pamięci EEPROM. Co ważne, nie bez powodu zamiast funkcji bibliotecznej środowiska AVR-GCC *eeprom_write_block()* zastosowałem funkcję *eeprom_update_block()*. Jak łatwo się domyślić, różnica jest taka, że druga z funkcji (update) sprawdza zawartość komórek pamięci EEPROM, na których ma przeprowadzać operację kasowania/zapisu i wykonuje ją wyłącznie wtedy, gdy nowa wartość jest inna, niż odczytana z docelowego adresu, co przyczynia się do jeszcze mniejszego zużycia pamięci EEPROM oraz skraca czas zapisu, który to, o czym musimy pamiętać, wynosi od 3,3 do 8,5 ms dla mikrokontrolerów z rodziny AVR (dla operacji kasowania i zapisu bajta lub strony pamięci). Oczywiście, wbudowane funkcje zapisu nie wstrzymują działania programu aplikacji użytkownika na wspomniany czas programowania, gdyż nie oczekują na jego koniec, jednak samo programowanie kolejnego bajta/strony jest możliwe po upłynięciu wspomnianego wcześniej czasu. Dla ułatwienia, plik nagłówkowy *avr/eeprom.h* zawiera funkcję, za której pomocą możemy sprawdzić czy kontroler pamięci EEPROM gotowy jest na wykonanie kolejnego zapisu. Mowa o funkcji *eeprom_is_ready()* zwracającej logiczną „1", jeśli kontroler pamięci EEPROM jest gotowy na wykonanie operacji odczytu lub zapisu.

Wszystko to bardzo łatwe, prawda? Jest jednak jeden mały „haczyk". Wbudowane w środowisko AVR-GCC funkcje obsługujące pamięć EEPROM zakładają, że w czasie ich działania (dotyczy to głównie funkcji zapisujących, ale w szczególnych przypadkach nie tylko) są zablokowane przerwania systemowe. Oczywiście, pierwsza rzecz, która przychodzi do głowy to myśl, że nie powinno to być wielkim problemem, bo przecież na czas ich działania możemy zablokować przerwania. Czasami jest to jakieś rozwiązanie, ale innym razem zupełnie niedopuszczalne. Wszystko zależy od konkretnej aplikacji, lecz trzeba mieć świadomość, że wykorzystanie funkcji *eeprom_write_block()* lub *eeprom_update_block()* w taki sposób, iż na czas jej wykonania blokujemy przerwania systemowe może przy większych blokach danych uniemożliwić obsłużenie tychże przerwań przez czas kilkunastu/kilkudziesięciu ms. Przy zapisywaniu bajta, czas ten byłby znacznie krótszy i można rozważyć takie rozwiązanie, jednak dla bloku danych i wcześniej wspomnianych funkcji lepiej przemyśleć rozwiązanie docelowe. Oczywiście, cały czas mowa o aplikacji użytkownika, która korzysta z przerwań systemowych.

Zapewne zastanawiacie się, skąd takie założenia Twórców tychże funkcji. Wynika to ze sposobu inicjowania operacji zapisu pamięci EEPROM w mikrokontrolerach z rodziny AVR. Otóż proces ten zakłada pewną kolejność ustawiania wybranych bitów rejestru kontrolnego kontrolera pamięci EEPROM, która to nie może zostać zakłócona przez przerwanie systemowe. Proces ten, według dokumentacji, zakłada wykonanie poszczególnych, następujących po sobie kroków:

1. Odczekanie, aż bit EEPE w rejestrze EECR zostanie wyzerowany.
2. Nadanie adresu komórki przeznaczonej do zapisu w rejestrze EEAR.
3. Nadanie wartości do zapisania w adresowanej komórce pamięci w rejestrze EEDR.
4. Ustawienie bitu EEMPE i jednoczesne wyzerowanie bitu EEPE w rejestrze EECR.
5. Ustawienie bitu EEPE w rejestrze EECR w czasie do 4 taktów zegara CPU od wykonania kroku 4.

Wystąpienie przerwania systemowego w czasie pomiędzy wykonaniem punku 4 i 5 oraz jego obsłużenie z pewnością będzie trwało znacznie dłużej, aniżeli 4 takty zegara, co w efekcie uniemożliwi zainicjowany wcześniej zapis do pamięci EEPROM. Jak, w takim razie, rozwiązać ten nietypowy problem i jednocześnie nie blokować

Listing 8. Funkcja obsługi przerwania EEPROM Ready Interrupt

```
ISR(EE_READY_vect)
{
    register uint8_t prevValue;
    static uint8_t Index, whereToWrite;
    if(Index < sizeof(configType))  //Bajty struktury danych: 0...n-1
    {
        //Najpierw ustalamy miejsce w pamięci EEPROM, pod które należy zapisać kopię struktury danych
        if(Index == 0) whereToWrite = findCurrBuffAddr();
        //Zapisujemy bajt po bajcie tejże struktury danych
        EEAR = (uint16_t) &wlConfigBuff[whereToWrite].index[Index];  //Adres bajta
        EEDR = Config.index[Index];  //Wartość bajta z pamięci RAM
        EECR |= (1<<EEMPE);  //EEPROM Master Programming Enable
        EECR |= (1<<EEPE);  //EEPROM Programming Enable
        Index++;
    }
    else  //Bajt bufora Statusu dla obsługi mechanizmu Wear leveling
    {
        //Kopia struktury danych została już zapisana, więc musimy zaktualizować bufor Statusu - Wear leveling
        //Odczytujemy wartość z poprzedniej pozycji bufora Statusu i pod pozycją bieżącą zapisujemy tą wartość+1
        prevValue = eeprom_read_byte(&wlStatBuff[(whereToWrite-1) & (WL_BUFF_SIZE-1)]);
        EEAR = (uint16_t) &wlStatBuff[whereToWrite];  //Adres bajta
        EEDR = prevValue+1;  //Wartość bajta
        EECR |= (1<<EEMPE);  //EEPROM Master Programming Enable
        EECR |= (1<<EEPE);  //EEPROM Programming Enable
        //Wszystko mamy już zrobione, więc wyłączamy przerwanie EE_READY_vect i zerujemy Index
        Index = 0;
        EEPROM_INTR_DISABLE;
    }
}
```

przerwań systemowych? Rozwiązaniem bardzo skutecznym, a jednocześnie maksymalnie wydajnym z punktu widzenia aplikacji użytkownika jest wykorzystanie przerwania kontrolera pamięci EEPROM, które to po uruchomieniu jest zgłaszane za każdym razem, gdy kontroler pamięci jest gotowy do wykonania operacji zapisu/odczytu (przerwanie *EEPROM Ready Interrupt*). Uruchomienie tego przerwania powinno następować w programie głównym w chwili, gdy istnieje potrzeba zapisania kopii struktury danych, zaś sam zapis struktury oraz aktualizacja bufora stanu powinny być wykonywane w samej procedurze obsługi przerwania kontrolera pamięci EEPROM, zwalniając program główny aplikacji z potrzeby podejmowania jakichkolwiek działań w tej materii. Efektywne, wygodne i do tego skuteczne. Przejdźmy zatem do rozwiązań programowych.

W pierwszym kroku wprowadzimy dwie definicje, które służą odblokowaniu i zablokowaniu przerwania *EEPROM Ready Interrupt*, a które to pokazano na **listingu 7**. Funkcję obsługi przerwania *EEPROM Ready Interrupt* pokazano na **listingu 8**. Cały proces zapisu kopii struktury danych i aktualizacji bufora stanu uproszczono do maksimum, a do tego w całości odbywa się on sprzętowo, w procedurze obsługi przerwania kontrolera pamięci EEPROM, co powoduje, iż mamy pewność, że inne procedury obsługi przerwań systemowych nie spowodują, iż zainicjowany wcześniej proces zapisu pamięci EEPROM nie dojdzie do skutku. Sposób inicjowania operacji zapisu nowej kopii struktury danych oraz aktualizacji bufora stanu ogranicza się on do uruchomienia przerwania *EEPROM Ready Interrupt*, które to inicjuje całą procedurę za pomocą polecenia *EEPROM_INTR_ENABLE*.

To wszystko odnośnie do sposobu przedłużania funkcjonowania pamięci EEPROM w zakresie stosowania mechanizmu Wear leveling i jego praktycznego zastosowania. Mam nadzieję, że ten w gruncie rzeczy nieskomplikowany mechanizm przyda się w zastosowaniach praktycznych, a artykuł zachęci do jego implementacji również dla innych rodzin mikrokontrolerów.

Robert Wołgajew, EP

REKLAMA

THE ULTIMATE
Networking
HANDBOOK
Podłącz wszystkie swoje urządzenia i ciesz się perfekcyjną siecią domową
172 STRONY
Porady dla użytkowników systemów Windows, Mac OS i Linux
Wydanie cyfrowe dostępne na www.ulubionykiosk.pl
ULUBIONY KIOSK.PL
PRZEJRZYSZ I KUPISZ NA WWW.ULUBIONYKIOSK.PL