
KA
MAMI
TECHDAYS

Programowanie mikrokontrolerów w języku C

Grzegorz Mazur

Plan warsztatów

- Język C – standardy, pojęcia
- Typy i atrybuty obiektów
- Zestaw do ćwiczeń: płytki STM32L476 Nucleo64 i KA-Nucleo-multisensor – prezentacja
- Zapoznanie ze środowiskiem programowym STM32CubeIDE
- Obserwacja działania i modyfikacja programów dla µC – projekty programowe

Język C – historia, koncepcja

- Zaprojektowany w latach 1970-tych jako „asembler wysokiego poziomu”
  - Pierwotnie dla komputerów PDP-11 i VAX-11
  - Dostęp do operacji niskopoziomowych
  - Proste odwzorowanie konstrukcji języka w kod maszynowy
  - Dostęp do większości operacji procesora z języka wysokiego poziomu

Dlaczego HLL?

- Łatwiejsze programowanie
- Szybsze debugowanie
- **Mniej pracy przy debugowaniu, bo wiele błędów może być wykrytych podczas kompilacji**
  - Ważny kieruek rozwoju języków programowania

C - standardy

- „K&R” (lata 1970.) – pierwotna wersja języka.
  - Brak prototypów funkcji, duża podatność na błędy
- „ANSI C” („C89”), właściwie ISO C90 - baza współczesnego języka C
- C99 – Stdint.h, typ _Bool, nowe atrybuty
- C11, C17/18 – niewielkie uzupełnienia
  - C17 jest poprawioną, ale nie rozszerzoną wersją C11
- C2x – w opracowaniu – udogodnienia zapisu, atrybuty

C i języki pochodne

- Większość współczesnych języków programowania bazuje na elementach składni C
  - C++, Objective C, C#, Java
- C++ NIE JEST rozszerzeniem C
  - To dwa odrębne języki, np.
    - int fun()
    - char napis[4] = ”abcd”;

Pliki źródłowe i nagłówkowe

- Pliki nagłówkowe powinny zawierać definicje preprocesora i deklaracje obiektów języka C nie będące definicjami*
  - Kompilacja samego pliku nagłówkowego nie powinna generować zawartości pamięci
  - Plik nagłówkowy może zawierać definicję funkcji rozwijanej (static inline), gdyż nie wymusza ona emisji kodu przez kompilator
- Pliki źródłowe mogą zawierać deklaracje i definicje funkcji i danych

Prototypy funkcji

- Definicje i deklaracje (prototypy) funkcji powinny zawierać specyfikacje typów argumentów (obowiązkowe w C2x).
- Jeżeli w module są funkcje publiczne, których prototypy znajdują się w pliku nagłówkowym, moduł ten powinien włączać plik nagłówkowy (diagnostyka błędów)
  - Niektóre kompilatory sygnalizują ostrzeżenie przy braku poprzedzenia definicji funkcji publicznej prototypem
- Definiowanie funkcji przed miejscem ich użycia ogranicza potrzebę ich deklarowania

Pliki nagłówkowe

- Nazwy definiowane przez preprocesor powinny być łatwo odróżnialne od nazw obiektów programu (np. wielkie litery)
- Zapobieganie wielokrotnemu włączeniu
  #ifndef MYHEADER_H_
  #define MYHEADER_H_
  … zawartość ...
  #endif

Pliki nagłówkowe

- Stałe preprocesora nie powinny zawierać rzutowania typów
  - np. #define X (uint32_t) 1
- Wszystkie wyrażenia stałe powinny być ujęte w nawiasy
  - #define Y ((X) + 1)
- Argumenty makrodefinicji w rozwinięciu powinny być ujęte w nawiasy
  - #define Z(a,b) ((a) * 2 + (b) * 3)

Typ int w C

- Typ umożliwiający reprezentację liczb całkowitych o zakresie nie mniejszym niż <-32768, 32767>
- Rozmiar zależny od implementacji (kompilatora), nie krótszy niż 16 bitów
  - W procesorach 8- i 16- bitowych – 16-bitowy
  - W procesorach 32- i 64-bitowych – 32-bitowy

C99

- Nowe słowa kluczowe:
  - inline, restrict, _Bool, _Complex, _Imaginary
- Typ _Bool
- Typy o określonych rozmiarach
- Jawne inicjowanie pól struktur/unii i elementów wektorów przez nazwę/indeks
- Komentarze w stylu C++ // do końca wiersza

_Bool

- Typ dla danych logicznych
- Dane przyjmują dwie wartości: 0 i 1
- Automatyczna konwersja przy podstawieniu danych innych typów, również zmiennopozycyjnych
  - Wartość 1 gdy dana po prawej stronie różna od 0
- Nagłówek stdbool.h zawiera typedef _Bool bool;

<stdint.h>

- Plik nagłówkowy zawierający definicje typów o jawnie określonych rozmiarach
  - uint8_t, uint16_t, uint32_t, ...
  - int8_t, int16_t, int32_t, …
- Również typy o jawnie określonym minimalnym rozmiarze, gwarantujące największą szybkość operacji
  - fast_uint8_t itd...

Użycie typów całkowitoliczbowych

- Tam, gdzie istotne jest jawne określenie rozmiaru danych, używamy typów z stdint.h
  - Czyli zawsze przy programowaniu µC
- Dane powinny być reprezentowane w najkrótszym typie obejmującym ich zakres
- Standard języka C wymaga wykonywania operacji arytmetycznych i logicznych w sposób równoważny wykonaniu ich na typie nie krótszym niż int/unsigned int
  - Nie ma potrzeby jawnego rzutowania krótszych typów na int po prawej stronie podstawienia

Bez znaku/ze znakiem

- Dane całkowitoliczbowe, które nie przyjmują wartości ujemnych, powinny być deklarowane w typach bez znaku (uintxx_t)
- Stałe używane w wyrażeniach z danymi bez znaku powinny być stałymi bez znaku – przyrostek u/U

Typ char

- To, czy jest to typ ze znakiem, zależy od implementacji
  - (obecnie zwykle bez znaku)
  - char, signed char i unsigned char – to w C trzy **różne** typy
    - Ale np. signed int to to samo, co int
- Typ char powinien być używany WYŁĄCZNIE do danych tekstowych – znaków i łańcuchów w kodzie ASCII (i innych 8-bitowych) oraz Unicode UTF-8

Długość danej, a wydajność

- Operacje na danych dłuższych niż 8/16 bitów w procesorach 8-/16-bitowych są kosztowne czasowo
- Operacje na danych 8- i 16-bitowych w językach wysokiego poziomu we współczesnych procesorach 32-bitowych mogą być nieco wolniejsze, niż operacje na danych 32-bitowych

Wydajność operacji we współczesnych µC

- Czas wykonania dodawania/odejmowania i operacji logicznych jest taki sam, jak czas przesłania danych
- Mnożenie:
  - Bez sprzętowego układu mnożącego – czas dodawania × liczba bitów mnożnika lub wolniej
  - Ze sprzętowym układem mnożącym – zwykle 1..5× wolniejsze od dodawania
- Dzielenie – zwykle min. 2× wolniejsze od mnożenia.

Długości danych

- Użycie 32-bitowych zmiennych lokalnych procedur w procesorach 32-bitowych nieco zwiększa możliwości optymalizacji kodu przez kompilator
- W praktyce współczesne kompilatory zwykle generują równie szybki kod, niezależnie od rozmiaru danych

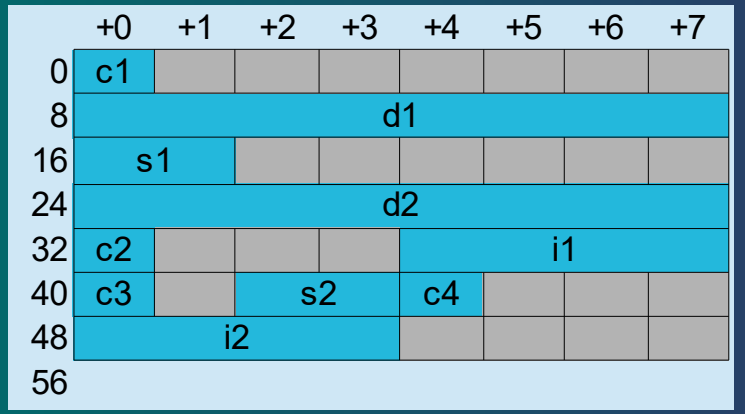
Wyrównanie naturalne

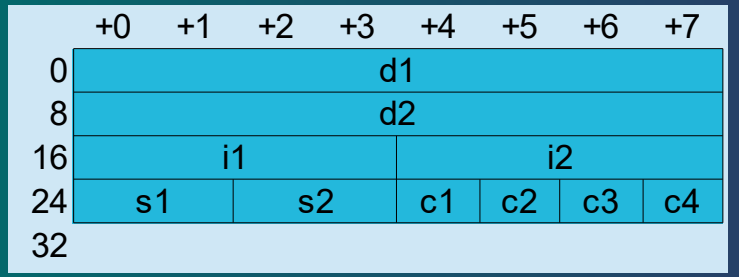
- W celu optymalizacji szybkości dostępu każda dana skalarna jest umieszczana przez kompilator w pamięci pod adresem podzielnym przez jej długość
- Kompilator nie może zmieniać kolejności pól struktury
  - Zajętość pamięci przez strukturę może zależeć od kolejności pól
- Operator sizeof zwraca odstęp pomiędzy początkami dwóch obiektów danego typu

Wyrównanie i sizeof

```
struct st1 {
    char c1;
    double d1;
    short s1;
    double d2;
    char c2;
    int i1;
    char c3;
    short s2;
    char c4;
    int i2;
} e1;

struct st2 {
    double d1, d2;
    int i1, i2;
    short s1, s2;
    char c1, c2, c3, c4;
} e2;
```





Operator sizeof i rozmiar danych

- Operator sizeof zwraca odstęp pomiędzy początkami dwóch kolejnych danych danego typu w wektorze tych danych
  - Dla typów prostych i wektorów jest to użyteczny rozmiar danej w bajtach
- Liczba elementów wektora v[] dowolnego typu: sizeof(v) / sizeof(v[0])
- Długość stałego napisu: sizeof(napis) - 1

Atrybut const

- Oznacza stałą wartość danej, bez możliwości modyfikacji
- Dana musi być zainicjowana statycznie w definicji.
- Dana z atrybutem const nie musi być zrealizowana przez kompilator w pamięci – odwołanie może zostać zastąpione podstawieniem stałej natychmiastowej

const i wskaźniki

- char * p – zmienny wskaźnik na zmienną typu char
- const char * p – zmienny wskaźnik na stałą typu char
- char * const p – stały wskaźnik na zmienną typu char
- const char * const p – stały wskaźnik na stałą typu char

const i parametry procedur

- Atrybut const użyty przy specyfikacji parametrów procedury ułatwia optymalizację i wykrywanie błędów
  - void send_string(const char *s)
- Brak atrybutu const w takiej sytuacji powoduje generowanie przez kompilator nieuzasadnionych ostrzeżeń przy poprawnych wywołaniach funkcji ze stałymi argumentami z użyciem stałych

const i parametry procedur

- Atrybutu const należy używać zwłaszcza przy deklarowaniu parametrów jako wskaźników na łańcuchy i struktury, które nie są modyfikowane przez procedurę

restrict

- W deklaracji zmiennej wskaźnikowej oznacza, że żadna inna zmienna w bloku nie wskazuje tego samego obiektu
  - Obiekt nie jest modyfikowany wskutek dostępu poprzez inny wskaźnik
- Umożliwia optymalizację odwołań przez kompilator

Stałe i zmienne

- Wszystkie obiekty, którym nadajemy wartość stałą tylko jeden raz, powinny być stałymi, a nie zmiennymi
- Dotyczy to również struktur i wektorów
- Jednorazowe nadanie stałej wartości zmiennej jest nagminnym błędem w programach, również w wielu bibliotekach

Stałe i zmienne łańcuchowe w µC

- const char stalylancuch[] = "abc";
  - Stała łańcuchowa umieszczona w pamięci ROM
- const char *sptr = "abc";
  - zmienna wskaźnikowa umieszczona w sekcji DATA (pamięci RAM µC) zainicjowana adresem stałej łańcuchowej umieszczonej w sekcji CONST (w pamięci ROM µC)

volatile

- Atrybut volatile oznacza wyłącznie możliwość modyfikacji wartości zmiennej pomiędzy dwoma odwołaniami do niej w programie
- Zmienna może być modyfikowana przez sprzęt lub wywoływaną sprzętowo procedurę o wyższym priorytecie (np. procedurę obsługi przerwania)

volatile - skutki

- Każdy odczyt powoduje fizyczny odczyt danej
  - Również w zdaniu pustym `zmienna;`
- Każdy zapis jest wykonywany (również przy dwóch lub więcej kolejnych zapisach)
- Zmienna volatile (przynajmniej teoretycznie) nie musi być zaalokowana w pamięci

volatile

- Atrybut nie oznacza, że dostęp do danej będzie niepodzielny, np.
  - Podczas kopiowania struktury z atrybutem volatile modyfikacja kolejnego pola może nastąpić podczas odczytu poprzedniego
  - W procesorach 8-bitowych próba odczytu 16-bitowej zmiennej volatile podczas jej modyfikacji może zwrócić niepoprawną wartość
- Przed modyfikacją pomiędzy dostępami do części zmiennej chroni atrybut _Atomic

Sekcje pamięci

- text/code – kod programu
- Static – dane statyczne
  - do sekcji tej należą dane deklarowane na poziomie zewnętrznym (poza funkcjami) oraz dane deklarowane wewnątrz funkcji z atrybutem static
  - składa się z podsekcji:
    - Const/rodata – stałych
    - Data – zmiennych o niezerowych wartościach początkowych
    - BSS – zmiennych zainicjowanych na 0
- Auto(stos) – argumenty i zmienne lokalne procedur
- Heap(sterta) – dane alokowane dynamicznie

Zajętość pamięci µC

- ROM:
  - kod programu,
  - Stałe,
  - wartości początkowe zmiennych niezerowych – obraz sekcji DATA
- RAM:
  - Zmienne statyczne (DATA i BSS)
  - Stos
  - sterta

Atrybut register

- Nie należy go używać
- Oznacza jedynie niemożność odwołania do obiektu przez adres.
- We współczesnych kompilatorach nie wpływa na optymalizację

Atrybut static

- Dwa całkowicie różne znaczenia, zależne od miejsca deklaracji obiektu
- Deklaracje zewnętrzne (funkcje, dane) – obiekt prywatny o zasięgu ograniczonym do modułu, w którym jest zadeklarowany
- Zmienne lokalne funkcji – obiekt alokowany statycznie, zachowuje wartość pomiędzy wywołaniami funkcji
  - obiekt taki nie jest alokowany na stosie

Statyczne dane lokalne

- Dane używane przez jedną funkcję, zachowujące wartość pomiędzy wywołaniami, np.
  - odliczanie czasu przy użyciu przerwania timera
  - zmienna przechowująca stan automatu realizowanego w przerwaniu

Stałe lokalne

- Zgodnie z zasadą ograniczania zasięgu stałe powinny być deklarowane wewnątrz funkcji…
- … ale wtedy mają one domyślnie klasę auto i są powoływane do życia przy każdym wejściu do funkcji
- → stałe wewnątrz funkcji należy deklarować z atrybutami static const

C11/C17 – słowa kluczowe

- Atrybut _Alignas – wymaganie wyrównania zgodnie z wymogiem dłuższego typu
- _Alignof – operator zwracający wyrównanie obiektu
- Atrybut _Atomic – gwarantuje niepodzielny dostęp do obiektu
- _Generic – operator wyboru akcji zależnie od typu obiektu – umożliwia polimorfizm w C

C11/C17 – słowa kluczowe (2)

- _Noreturn – atrybut funkcji zawierającej pętlę nieskończoną, w celu uniknięcia ostrzeżeń
- _Static_assert – ostrzeżenie kompilatora zależne od wartości wyrażenia stałego
- _Thread_local – atrybut prywatnych danych statycznych wątku

C2x (proponowany)

- nowe słowa kluczowe _Decimal32, _Decimal64, _Decimal128 – nowe typy dla dziesiętnych liczb zmiennopozycyjnych IEEE754-2008
- stałe binarne – prefiks 0b lub 0B
- w stałych całkowitych można oddzielać grupy cyfr apostrofem, np. 0b1010’0101, 12’345’678, 0xabcd’1234

Ciekawe zapisy w C

```
char c = "0123456789ABCDEF"[v & 0xf];

x = !!x;
```

Static – obiekty prywatne

- Każda funkcja lub dana zdefiniowana w pliku źródłowym, która nie jest używana przez inny plik źródłowy, powinna być zadeklarowana jako prywatna – static
- Funkcja main() nie może być prywatna, bo jest wywoływana przez moduł startowy

Ograniczanie zasięgu danych

- Dana prywatna funkcji powinna być zadeklarowana wewnątrz najbardziej zagnieżdżonego bloku, w którym jest używana, bezpośrednio przed lub w miejscu jej pierwszego użycia
- Począwszy od C99 zmienna sterująca pętlą for może być zadeklarowana w for (), jak w C++
- W dotychczasowych standardach języka (< C2x) etykieta nie może poprzedzać deklaracji
  - Problem z ograniczeniem zasięgu w konstrukcji switch() - można wprowadzić blok pod case

Funkcje rozwijane (osadzone)

- Para atrybutów `static inline` jest sugestią dla kompilatora, że kod ciała funkcji powinien być wstawiony w miejsce jej wywołania
- Unika się w ten sposób czasochłonnego przekazywania sterowania do krótkich, prostych funkcji, zachowując kontrolę poprawności przez kompilator, niedostępną dla makrodefinicji procesora

Funkcje rozwijane (osadzone)

- Używamy praktycznie zawsze pary atrybutów `static inline`
- inline bez static wymusza na kompilatorze wygenerowanie zwyczajnego kodu funkcji, dostępnego dla innych modułów
  - Definicja taka nie powinna być umieszczana w pliku nagłówkowym, bo powoduje emisję kodu
  - Zbędna funkcja publiczna może być usunięta podczas konsolidacji

Procedura aktywna

- Procedura jest aktywna, jeśli jej wykonanie zostało rozpoczęte i nie zostało zakończone
- Oznacza to, że aktualnie jest wykonywana:
  - Ta procedura
  - Procedura wywołana przez nią (lub kolejna w hierarchii wywołań)
  - Procedura obsługi wyjątku wywołana podczas gdy nasza procedura była aktywna
  - Itd.

Wielobieżność (reentrancy)

- Możliwość równoczesnej wielokrotnej aktywacji procedury
  - np. z wątku głównego i z procedury obsługi przerwania
- Zwykle jednym z warunków wielobieżności jest niekorzystanie ze zmiennych statycznych
- Procedury operujące na peryferialach na ogół nie są wielobieżne!

Funkcje rozwijane, a makrodefinicje

- Nie należy nadużywać makrodefinicji do operacji na zmiennych
- Zastąpienie makrodefinicji funkcjami rozwijanymi poprawia diagnostykę błędów
- Wartość funkcji nie może być użyta do zainicjowania danej przez kompilator
  - W takim przypadku musimy użyć makrodefinicji

Atrybut __weak

- Nie należy do standardu języka – jest to rozszerzenie kompilatora
- Powoduje słabą definicję symbolu, np. funkcji, która może być przysłonięta przez inną, zwykłą definicję symbolu o tej samej nazwie na etapie konsolidacji
- Służy do definiowania pustych funkcji, gdy w niektórych wersjach projektu mamy inną wersję niepustą (np. obsługa wyjątków)

Stałe całkowitoliczbowe

- Domyślnie stała ma typ int (ze znakiem)
- W ARM jest to typ 32-bitowy – nie ma potrzeby stosowania przyrostka L
- Do stałych bez znaku (w tym masek bitowych) należy stosować przyrostek U
  - Kompilator nie będzie wtedy ostrzegał o przekroczeniu zakresu, gdy wartość wyrażenia stałego będzie >= 2^{31}

```
#define LEDR_MSK (1 << 31)     // warning
#define LEDG_MSK (1u << 31)    // no warning
```

Arytmetyka stałopozycyjna

- Operacje zmiennopozycyjne w procesorach bez sprzętowej jednostki zmiennopozycyjnej są wielokrotnie wolniejsze od stałopozycyjnych
- Dają również nieprecyzyjne wyniki
- Konwersja danej zmiennopozycyjnej do postaci znakowej (np. do wyświetlania) jest b. czasochłonna
- Rozwiązanie – reprezentacje stałopozycyjne

Arytmetyka stałopozycyjna

- Dane można reprezentować w podwielokrotnościach jednostek, np. w mV zamiast V, dziesiątych częściach stopni itd.
- Mnożenie przez liczbę niecałkowitą można zastąpić mnożeniem przez ułamek zwykły (mnożeniem i dzieleniem)
- Ułamek warto przekształcić (przybliżyć?) tak, by jego mianownik był potęgą dwójki.

Arytmetyka zmiennopozycyjna, a stałopozycyjna

- Instrukcja:

```
uint32_t x, y;
x = ...;
y = x * 0.75;
```

  jest rozwijana jako:
  - konwersja x do typu double
  - mnożenie na typie double
  - konwersja wyniku do typu int
- Konwersje są zwykle bardziej czasochłonne od mnożenia

Arytmetyka zmiennopozycyjna, a stałopozycyjna

- Instrukcja:

```
uint32_t x, y;
x = ...;
y = x * 3 / 4;
```

  może być rozwinięta w dwa dodawania i dwa przesunięcia bitowe
- … ale wynik dzielenia jest obcięty w dół, a nie zaokrąglony do najbliższej wartości całkowitej
- W celu uzyskania zaokrąglenia przed dzieleniem dodajemy do dzielnej połowę dzielnika

```
y = (x * 3 + 2) / 4;
```

Optymalizacja operacji arytmetycznych

- Wyrażenia złożone warto przekształcić do postaci zawierającej najpierw dodawania i mnożenia, a na końcu jedno dzielenie
- Przy mnożeniu należy zwracać uwagę na możliwość przekroczenia zakresu
  - Warto przeprowadzić redukcję mnożników i dzielnika

Uśrednianie i filtrowanie

- Strumień danych często wymaga uśredniania w celu eliminacji szumów (np. odczyty ADC, pomiar periodycznych odcinków czasu)
- Uśrednianie pomiarów wymaga ich przechowywania
- Rozwiązanie: prosty filtr dolnoprzepustowy
  - Wymagane tylko jedno przesunięcie, jedno odejmowanie i jedno dodawanie na próbkę

Filtr dolnoprzepustowy

```
#define FILTER_SHIFT   3  // stala czasowa filtra

filter = filter – (filter >> FILTER_SHIFT) + sample;

// inicjowanie:
filter = firstsample << FILTER_SHIFT;
```

Przeliczanie pomiarów ADC

- Przefiltrowany odczyt ADC zamieniamy na wartość użyteczną, gdy chcemy jej użyć (np. wyświetlić)
- Warto sprowadzić tę operację do mnożenia przez ułamek, którego mianownik jest potęgą liczby 2
  - Przy odpowiednim doborze mianownika błąd wynikający ze zmiany mianownika jest mniejszy niż błąd przetwarzania ADC lub zaokrąglania wyniku

Naprzemienne nadawanie zmiennej dwóch wartości

- Zastosowanie:
  - Przełączanie buforów danych
  - Przełączanie pomiędzy dwiema wartościami (jaśniej/ciemniej)

```
uint8_t buf1[SIZE], buf2[SIZE];
uint8_t *bufptr = buf1;

...
bufptr ^= buf1 ^ buf2;
```

Sprawdzanie stanu przycisku

- Najprostszy przypadek: brak zakłóceń elektrycznych, tylko drgania styków
  - np. przycisk na płytce blisko uC
- Wystarczy testować stan w procedurze obsługi przerwania timera z okresem dłuższym od okresu drgań i reagować na zmianę, np.

```
static uint8_t kstate;

kstate = (kstate << 1 | KEY_PRESSED) & 3;
if (kstate == 1)
   OnKeyPressed();
```

Inicjowanie bloków peryferyjnych µC

- Zwykle moduły peryferyjne inicjujemy tylko jeden raz – przy starcie programu
- Wartości początkowe rejestrów sterujących są znane i można z nich korzystać
- Używamy głównie podstawień, a nie operacji logicznych.
  - Mniejsza zajętość pamięci, lepsza czytelność kodu

Struktura programu

- Pętla zdarzeń + przerwania
  - Problem z usypianiem – możliwość zgubienia zdarzenia lub opóźnienia reakcji w pętli zdarzeń
- RTOS
  - Łatwe programowanie w konwencji pętli zdarzeń
  - narzut czasu i pamięci
- Tylko przerwania, w tym generowane programowo
  - Najwyższa wydajność
  - Wymaga zmiany podejścia do programowania

Nieblokująca obsługa urządzeń

- W dobrze zorganizowanym oprogramowaniu nie występują pętle oczekiwania na upłynięcie czasu lub gotowość urządzenia
- Obsługa peryferiów przez przerwania, DMA, lub z użyciem timera (np. ADC)