Program obowiazkowy

1. **WSTEP**
2. **KONWENCJE**

    Odstępy
    Klamry
    Wcięcia
    Poziomy zagnieżdżenia
    Nazwy zmiennych/funkcji
    Derektywa "#define"
    Format stałych
    Typy zmiennych
    Zakres zmiennych
    Pętle
    Operatory
    Komentarze

3. **FUNKCJE**

    Łańcuchy znakowe – operacje proste

    CopyString(char cSource[], char cDestination[])
    eCompareString(char cStr1[], char cStr2[])
    AppendString (char cSourceStr[],char cDestinationStr[])
    ReplaceCharactersInString(char cString[],char OldChar,char NewChar)

    Łańcuchy znakowe – konwersje

    UIntToHexStr (unsigned int uiValue, char cStr[])
    eHexStringToUInt(char cStr[],unsigned int *puiValue)
    AppendUIntToString (unsigned int uiValue, char cDestinationStr[])

    Dekodowanie komunikatów

    ucFindTokensInString (char *String)
    eSringToKeyword (char cStr[],enum eKeywordType *peKeyword)
    DecodeTokens(void)
    DecodeMsg(char *String)

    Testy

1. Wstep

Program Obowiązkowy jest to zestaw funkcji do implementacji w języku C. Funkcje podzielone są na grupy.
Wykonanie *Programu Obowiązkowego* wymaga znajomości języka C w zakresie odpowiadającym Symfonii C++ część I do rozdziału 8.9.2.
Przed przystąpieniem do implementacji funkcji należy zapoznać się konwencjami programowania w języku C przedstawionymi w rozdziale „Konwencje”.
Funkcje należy implementować w środowisku kompilatora KEIL dla mikrokontrolerów ARM, którego wersje darmową można uzyskać ze strony firmy KEIL.

Funkcje należy dostarczać do poprawy prowadzącemu zajęcia w postaci *wydruku* składającego się z jednej lub więcej kartek spiętych zszywaczem.
Wydruk powinien zawierać jedną i tylko jedną grupę funkcji.
Wydruki mogą być dwustronne.
Funkcje należy umieszczać w formatce znajdującej się w pliku „FormatkaNaWydruki.doc” tak aby nie zmienić formatu tekstu (typ i rozmiar czcionki oraz odstępy miedzy liniami).
W nagłówku formatki powinno znajdować się nazwisko i imię oraz nazwa grupy funkcji
Na jednej stronie powinno znajdować się tyle funkcji ile to możliwe z zachowaniem odstępów między funkcjami Jeżeli to możliwe jedna funkcja powinna znajdować się tylko na jednej stronie.
Plik kartek należy zszywać w lewym górnym rogu.
Pliki kartek z naniesionymi uwagami prowadzącego należy zachować do końca kursu.

PRZED ODDANIEM FUNKCJI NALEŻY SPRAWDZIĆ CZY:

- SIĘ KOMPILUJĄ
- SĄ ZGODNE ZE SPECYFIKACJĄ
- SĄ ZGODNE Z KONWENCJAMI

2. Konwencje

Odstępy

W celu uwypuklenia struktury programu należy stosować odstępy pomiędzy poszczególnymi blokami kodu. W przypadku funkcji powinny to wyglądać w sposób następujący:

```
nazwafunkcji(argumenty){
   1 linia odstepu
   definicje zmiennych lokalnych
   1 linia odstepu
   reszta funkcji
}
```

Klamry

Należy zawsze stosowac klamry, nawet gdy zawiarac one maja jedna linijke kodu albo pusta instrukcje
Np.

```
if(UartError==1){
   copyStrToTxBuff("ERR");
}
```

lub

```
for ( ucStrCtr = 0 ; NULL == cString1[ucStrCtr] ; ucStrCtr ++ ) {}
```

Wcięcia

W celu uwypuklenia struktury programu należy stosować wcięcia (za pomocą spacji)

```
// ZLE!!!
if (UartTxBusy != 1)
{
if(sendOK==1)
{
copyStrToTxBuff("OK");
}
else if(UartError==1){
copyStrToTxBuff("ERR");
}
}

// DOBRZE
if(UartTxBusy != 1) {
   if(sendOK==1){
      copyStrToTxBuff("OK");
   }
   else if(UartError==1){
      copyStrToTxBuff("ERR");
   }
}
```

Poziomy zagnieżdżenia

Jednym z często popełnianych błędów w stylu kodowania powodującym „zaciemninie” kodu jest stosowanie zbyt dużego poziomu zagnieżdżenia. Dlatego poniżej zostaną zdefiniowane poziomy zagnieżdżenia.

Poziom zagnieżdżenia

1:

```
Main(x,y,y){
  A=1;
  B=2;
}
```

2:

```
Main(x,y,y){
  A=1;
  For(...){
     i=i+A;
  }
}
```

3:

```
Main(x,y,y){
  A=1;
  For(...){
     i=i+A;
     if(x==y){
        z=0;
     }
  }
}
```

Nazwy zmiennych/funkcji

Nazwa zmiennej powinna opisywać jej funkcję.
Nazwa funkcji powinna informować co dana funkcja robi.
Wyrazy w nazwach należy zaczynać od dużej litery.

```
unsigned int    i         // ZLE !!!

unsigned int    count     // TROCHE LEPIEJ – lepszy skrot niż jedna literka

unsigned int    counter    // ZNACZNIE LEPIEJ - lepszy caly wyraz niż skrot
                           // tylko ze niewiadomo co liczy counter

unsigned int    character_counter  // PRAWIE DOBRZE – ale zle sie czyta

unsigned int    CharacterCounter // DOBRZE - wyrazy mają się zaczynać
                                 //zaczyna od dużej litery
                                 //(bez podkreśleń)
```

Należy stosować **węgierską notację** (hungarian notation) (szczegóły -> internet)
Pozwala ona zorientować się jaki typ ma zmienna bez odwoływania się do jej definicji:

```
char OdebranyZnak          // ZLE !!!
char cOdebranyZnak         // DOBRZE
int StanPrzetwornika       // ZLE !!!
int iStanPrzetwornika      // DOBRZE
```

Jeżeli zmienna jest "unsigned" dodajemy prefix "u"

```
unsigned char     ucOdebranyZnak
```
unsigned int uiStanPrzetwornika

Węgierska notacje w nazwie tablic stosuje sie poprzez dodawanie prefiksu 'a'

```
unsigned char     aucRxBuffer[100]
```

Wegierska notacje w nazwie funkcji stosujemy tylko jeżeli funkcja zwraca jakąś wartość:

```
char cPodajOstatniZnak(void){
...
   return 'A';
}
```

Derektywa "#define"

Należy unikać bezpośredniego używania w kodzie programu stałych dosłownych zamiast tego należy stosować derektywe "#define". Nazwy derektyw należy pisać dużymi literami a wyrazy oddzielać znakiem podkreślenia
Wyjątkiem jest przypadek, w którym stała oznacza sama siebie, np.: inicjalizacja licznika, liczba zero, itp.
Jednak w wiekszosci przypadków stałe cos oznaczaja i wtedy należy uzyc derektywe "#define". Dobra praktyka jest też używanie zmiennych typu „enum” ale to zostanie opisane poźniej, na razie można zostac przy „define”.

```
// ZLE !!!

char   cRxBuffer[19]

main (){
.....
   ucBufferCounter ++;
   if (ucBufferCounter == 19)  bBufferOverflowError = 1;
....
}

// DOBRZE

#define BUFFER_SIZE    19

char   cRxBuffer[BUFFER_SIZE]

main (){
.....
   ucBufferCounter ++;
   if (ucBufferCounter == BUFFER_SIZE)  bBufferOverflowError = 1;
....
}
```

Format stałych

Stałe bitowe czyli stałe używane do operowania na konkretnych bitach (np. portu) powinny być zapisywane w formacie heksadecymalny

```
// Ustaw pierwszy i ostatni bit portu P1
P1 = 129;   //ZLE!!!
P1 = 0x81;  //DOBRZE
```

Stałe znakowe czyli stałe reprezentujące kod ASCII powinny być zapisywane w apstrofach

```
if (cZnak == 0x41) {} //ZLE !!!
if (cZnak == 'A') {}  //DOBRZE !!!
```

Typy zmiennych

Należy dopasowywać typ zmiennych do zadania. Rozmiar zmiennej powinien być jak najmniejszy ponieważ wielkość pamięci danych i kodu jest w mikrokontrolerach mocno ograniczona:

```
int  iSekundy  //ZLE !!!
char cSekundy  //DOBRZE
```

// zmienna typu int zajmuje 2 bajty może przyjmować wartości z zakresu od -32000 do 32000
// zmienna typu char zajmuje 1 bajt może przyjmować wartości z zakresu od -128 do 127
// Przyjęło się, że w minucie jest 60 sekund

// inkrementacja (++) zmiennej typu int zajmuje 4 instrukcje assemblera
// inkrementacja (++) zmiennej typu char zajmuje 1 instrukcje assemblera

Zakres zmiennych

Należy dopasowywać zakres zmiennych do zadania .
Zmienna powinna mieć zakres globalny tylko wtedy jeżeli służy do przesyłania wartości miedzy funkcjami, w przeciwnym razie zmienna powinna być deklarowana jako zmienna lokalna funkcji, w której jest wykorzystywana.

Jeżeli istnieje konieczność zachowania wartości zmiennej lokalnej do następnego wywołania funkcji należy nadać jej (zmiennej) atrybut *static.*

```
//ZLE!!!

#define ROZMIAR_BUFORA    6

int            iBufor[ROZMIAR_BUFORA] = {2,3,4,2,1,3};
int            iNajmniejsza=0;
int            iNajwieksza=0;
unsigned char  ucIndeksBufora;

//-------------------------------------------------------------
unsigned char ZnjdzNajmniejszaLiczbeWBuforze(){

   iNajmniejsza=0;
   for(ucIndeksBufora=0;ucIndeksBufora<ROZMIAR_BUFORA;ucIndeksBufora){
      if(iNajmniejsza > iBufor[ucIndeksBufora]){
         iNajmniejsza=iBufor[ucIndeksBufora];
      }
   }
   return ucIndeksBufora;
}

//-------------------------------------------------------------
unsigned char ZnjdzNajwiekszaLiczbeWBuforze(){

   iNajwieksza=0;
   for(ucIndeksBufora=0;ucIndeksBufora<ROZMIAR_BUFORA;ucIndeksBufora){
      if(iNajwieksza < iBufor[ucIndeksBufora]){
         iNajwieksza=iBufor[ucIndeksBufora];
      }
   }
   return ucIndeksBufora;
}


//DOBRZE
#define ROZMIAR_BUFORA    6

int iBufor[ROZMIAR_BUFORA] = {2,3,4,2,1,3};

//-------------------------------------------------------------
unsigned char ZnjdzNajmniejszaLiczbeWBuforze(){

unsigned char  ucIndeksBufora;
int            iNajmniejsza=0;

   for(ucIndeksBufora=0;ucIndeksBufora<ROZMIAR_BUFORA;ucIndeksBufora){
      if(iNajmniejsza > iBufor[ucIndeksBufora]){
         iNajmniejsza=iBufor[ucIndeksBufora]
      }
   }
   return ucIndeksBufora;
}
//-------------------------------------------------------------
unsigned char ZnjdzNajwiekszaLiczbeWBuforze(){

unsigned char  ucIndeksBufora;
int            iNajwieksza=0;

    for (ucIndeksBufora=0;ucIndeksBufora<ROZMIAR_BUFORA;ucIndeksBufora){
        if(iNajwieksza < iBufor[ucIndeksBufora]){
            iNajwieksza=iBufor[ucIndeksBufora];
        }
   }
   return ucIndeksBufora;
}
```

Pętle

Do implementacji funkcji należy stosować TYLKO pętle „for” ponieważ w jednej linijce skupia ona inicjalizację, sprawdzenie warunku oraz uaktualnianie.
Jeśli to tylko możliwe należy stosować pełną wersję for-a

Dobrze

```
for(ucCharCounter=0; ucCharCounter <STRING_LENGTH; ucCharCounter ++){
    ..
    ..
}
```

Źle

```
for(ucCharCounter=0;;ucCharCounter ++){
    if (ucCharCounter <STRING_LENGTH) break;
    ..
    ..

}
```

Jeżeli „for” w pełnej formie niezbyt pasuje do przypadku można np. zrezygnować z warunku i dać go do środka pętli z wyjściem z pętli za pomocą *break-a* albo *returna*. Kryterium „rezygnacji” może być ilość operatorów w warunku, jeśli jest ich więcej niż 1 to lepiej dać warunek z wyjściem z pętli do środka.

Operatory (Nawiasy)

Nie należy zakładać priorytetów operatorów tylko używać nawiasów „wymuszające” kolejność operacji.

Podczas stosowania operatora równości „= =” stała powinna koniecznie znajdować się po lewej stronie. (Dlaczego ?)

Nie należy stosować operatorów logiczny (negacja, iloczyn, suma) bezpośrednio do zmiennych bo w C niema zmiennych logicznych. ()

```
if (!a)        // ŹLE
if (0 == a)  // DOBRZE
if !(1 == a) // DOBRZE
```

Należy unikać skróconej notacji. Można stosować co najwyżej postinkrementację i postdekrementacjwa++ i a--.

```
a+=7        // ŹLE
a = a + 7   // DOBRZE
```

Komentarze

Należy unikać komentarzy w kodzie funkcji. Implementacja funkcji powinna być na tyle jasna aby nie wymagała komentarza. Komentarz powinien znajdować się w nagłówku funkcji i powinien informować o poprawnym sposobie użycia funkcji np. o dopuszczalnych wartościach argumentów wywołania funkcji.

3. Funkcje

Łańcuchy znakowe – operacje proste

Zakłada się że funkcje operują na tzw. „Null terminated” stringach. (patrz. Gręboszc).
Zakłada się że najdłuższy łańcuch znakowy będzie mieć 254 znaki razem ze znakiem NULL. W przypadku operacji na łańcuchach należy używać stałej NULL zamiast „0”.
(należy ją wcześniej zdefiniować NULL).
Nie należy tworzyć żadnych funkcji pomocniczych.
Funkcje należy implementować w kolejności podanej w specyfikacji.
Nie należy stosować operatora *sizeoff*
Kopiowanie

`CopyString(char pcSource[], char pcDestination[])`

Zadaniem funkcji jest skopiować łańcuch znakowy włącznie ze znakiem NULL z tablicy *source* do tablicy *destination*.
Maksymalny poziom zagnieżdżenia = 2.
Użyć „pełnego” fora tj. z inicjalizacją i inkrementacją licznika.

`eCompareString(char pcStr1[], char pcStr2[])`

Zadaniem funkcji jest porównywać łańcuchy znakowe zakończone znakiem NULL.
Jeżeli łańcuchy są sobie równe funkcja powinna zwracać „EQUEL” , w przeciwnym przypadku „NOTEQUAL”
Funkcja zwraca wartość typu „enum CompResult { DIFFERENT , EQUAL }”
W funkcji ni powinno znajdować się więcej niż 3 porównania (chociaż można „zejść” do 2)
W ystarczy jeden *for* i jeden *if.*
Wystarczy jedna zmienna lokalna pracujaca jako licznik petli.
Maksymalny poziom zagnieżdżenia = 3.

`AppendString (char pcSourceStr[],char pcDestinationStr[])`

Zadaniem funkcji jest dodać do łańcucha znakowego znajdującego się w `cDestinationStr` łańcuch znakowy znajdujący się w `cSourceStr`.
Pierwszy znak łańcucha `cSourceStr` powinien zostać nadpisany na NULL-u łańcucha `cDestinationStr` .
Należy wykorzystać jedna z wcześniejszych funkcji **.**
Wystarczy jeden for.
Nie używać bufora pomocniczego.

`ReplaceCharactersInString(char pcString[],char cOldChar,char cNewChar)`

Zadaniem funkcji jest zamienic w String-u wszystkie znaki OldChar na NewChar.

Łańcuchy znakowe – konwersje

`UIntToHexStr (unsigned int uiValue, char pcStr[])`

Zadaniem funkcji jest skonwertować liczbę typu „ui” na łańcuch tekstowy w formacie heksadecymalnym.
Łańcuch tekstowy powinien znaleźć się w tablicy „**`cStr”.`**
Łańcuch powinien kończyć się znakiem NULL.
Łańcuch musi zaczynać się od „0x” następnie mogą pojawiać się kody ascci z zakresów „0-9”, „A-F”
Przykłady:
uiValue = 0; cStr=”0x0000”+NULL
uiValue = 1; cStr=”0x0001”+NULL
uiValue = 65000; cStr=0xFDE8”+NULL
Przed implementacją należy zapoznać się z tablicą kodów *ascii* oraz z zapisem heksadecymalny.
Należy iterować po tetradach (niblach, tetradach) zmiennej *uiValue* od najmłodszej do najstarszej.
Używać operatorów bitowych w tym przesunięć. Maski bitowe powinny mieć odpowiedni format.
Nie stosować więcej niż jednej pętli w kodzie.
Nie modyfikować zmiennej wejściowej uiValue;
Nie stosować tablic ani cas-ów do przekodowywania liczby na znak.
Wystarcza dwie zmienne lokalne (wliczając licznik pętli).
Unikać powtórzeń kodu.

`eHexStringToUInt(char pcStr[],unsigned int *puiValue)`

Zadaniem funkcji jest zamienić łańcuch znakowy w formacie hexadecymalnym (duże litery) na wartość.
Adres łańcucha znajduje się w zmiennej *cStr*.
Zakłada się że jest to łańcuch typu *NULL terminated string*
Wartość przekazywana jest na zewnątrz funkcji poprzez wskaźnik *puiValue*
Łańcuch heksadecymalny akceptowany przez funkcje musi zaczynać się od „0x” .
Po „0x” musi znajdować się przynajmniej jeden znak różny od *NULL*.
Po „0x” nie może znajdować się więcej niż 4 znaki różne od *NULL*.
Zgodność z formatem powinna się zakończyć zwróceniem przez funkcje wartości *OK.* a niezgodność *ERROR*.
Funkcja zwraca wartość typu „enum Result { OK, ERROR }”
Maksymalnie jeden „for”.
W „forze” jedno odwołanie do tablicy „[]”. Nie używać wskaźników do iterowania po cStr.

`AppendUIntToString (unsigned int uiValue, char pcDestinationStr[])`

Zadaniem funkcji jest dodać liczbę w formacie heksadecymalnym (patrz `ucUIntToHexStr`) do łańcucha znakowego znajdującego się w `cDestinationStr.`
Pierwszy znak liczby powinien zostać nadpisany na NULL-u łańcucha `cDestinationStr`.
Nie należy tworzyć bufora pomocniczego – należy odpowiednio zastosować wskaźniki.

Dekodowanie komunikatów

Komunikat

Komunikat ma postać **łańcucha znakowego** zakończonego znakiem *NULL* i składa się z jednego lub więcej tokenów. **Tokeny** to sekwencje znaków oddzielone jednym lub wieloma **delimiterami**. Funkcję delimitera pełnić będzie znak spacji „\s” (20h, 32).
Przykład komunikatu pokazano poniżej:
`„Ola ma jeża”.`

Składa się on z trzech tokenów rozdzielonych dwoma delimiterami.

Dekodowanie

Dekodowanie komunikatu będzie polegało na policzeniu i zdekodowaniu poszczególnych tokenów. Zdekodowanie tokenu będzie polegać na określeniu jego **typu** i **wartości** a następnie zapamiętanie ich w odpowiedniej strukturze danych.

Typ tokenu

Będziemy rozróżniać trzy typy tokenów: KEYWORD, NUMBER, i STRING.
Token zostanie rozpoznany jako KEYWORD jeżeli będzie należał do listy słów kluczowych. W takim przypadku wartość tokenu stanowić będzie *enum*.
Przykład: Załóżmy, że zadeklarowano następującą listę słów kluczowych:

Keyword (enum)	odpowiadający jej łańcuch znakowy (char[])
LD	"load"
ST	"store"
RST	"reset"

Wtedy tokeny "store" i „reset" zostaną zdekodowane w sposób następujący

token	typ	wartość (enum)
„reset"	KEYWORD	RST
"store"	KEYWORD	ST

Token zostanie rozpoznany jako NUMBER jeżeli będzie spełniał format liczby zapisanej w kodzie heksadecymalnie. W takim przypadku wartość tokenu stanowić będzie wartość liczby.
Przykład:

token	typ	wartość (unsigned int)
„0x10"	NUMBER	16 (decymalnie)
„0x0A"	NUMBER	10 (decymalnie)

Token zostanie rozpoznany jako STRING jeżeli nie zostanie rozpoznany jako KEYWORD ani jako NUMBER. Wartość tokenu typu STRING stanowić będzie wskaźnik na ten token.
Przykład:

token	typ	wartość (char *)
„add"	STRING	Wskażnik na „add"
"subtract"	STRING	Wskażnik na "subtract"

Typ tokenu będzie przechowywany w zmiennej wyliczeniowej zdefiniowanej jak poniżej:

```
typedef enum TokenType {KEYWORD, NUMBER, STRING};
```

Wartość tokenu

Ponieważ typ tokenu może być różny (typ wyliczeniowy, liczba, łańcuch znakowy) dlatego jego wartość nie może być przechowywana w zmiennej jednego typu. Z tego względu do przechowywania wartości tokenu zostanie wykorzystana *unia zmiennych* zdefiniowana jak poniżej:

```
typedef enum KeywordCode { LD, ST, RST};

typedef union TokenValue
{
 enum KeywordCode    eKeyword; // jezeli KEYWORD
 unsigned int       uiNumber; // jezeli NUMBER
  char *             pcString; // jezeli STRING
};
```

Tablica tokenów

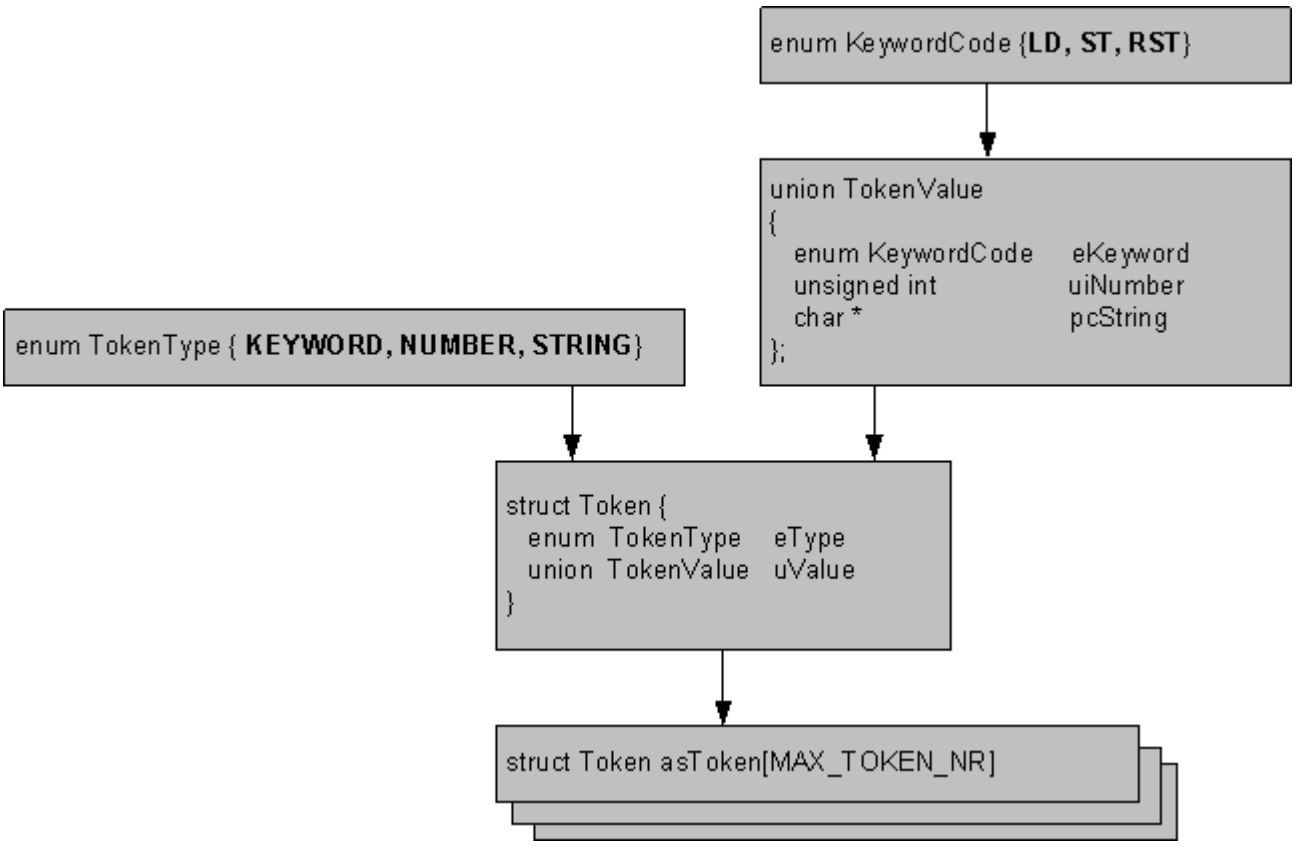
Typ i wartość tokenu będą przechowywane w jednej strukturze zdefiniowanej jak poniżej:

```
typedef enum TokenType { KEYWORD, NUMBER, STRING};

typedef struct Token
{
 enum  TokenType  eType;  // KEYWORD, NUMBER, STRING
 union TokenValue uValue; // enum,  unsigned int, char*
}
```

Ponieważ w pojedynczym komunikacie mamy najczęściej do czynienia więcej niż z jednym tokenem wiec wynik dekodowania będzie przechowywany tablicy tokenów zdefiniowanej jak poniżej:

```
#define MAX_TOKEN_NR 3 //maksymalna dopuszczalna ilość tokenów

struct Token asToken[MAX_TOKEN_NR]
```

Schemat blokowy odpowiadający tablicy `asToken` wraz z typami z których korzysta przedstawiono na rysunku:



Ilośc tokenów

Oprócz wypełnienia tablicy tokenów wynikiem dekodowania będzie również liczba odebranych tokenów zapamiętana w zmiennej:
unsigned char ucTokenNr;

Dekodowanie komunikatu będzie polegało na wypełnieniu tablicy sToken na podstawie odebranego łańcucha znakowego.

Przykład: Wynik rozkodowania komunikatu: „load 0x20 immediately” .

token	Type	Value
“load”	KEYWORD	LD
„0x20”	NUMBER	32 (decymalnie)
„immediately”	STRING	wskaznik na „immediately”

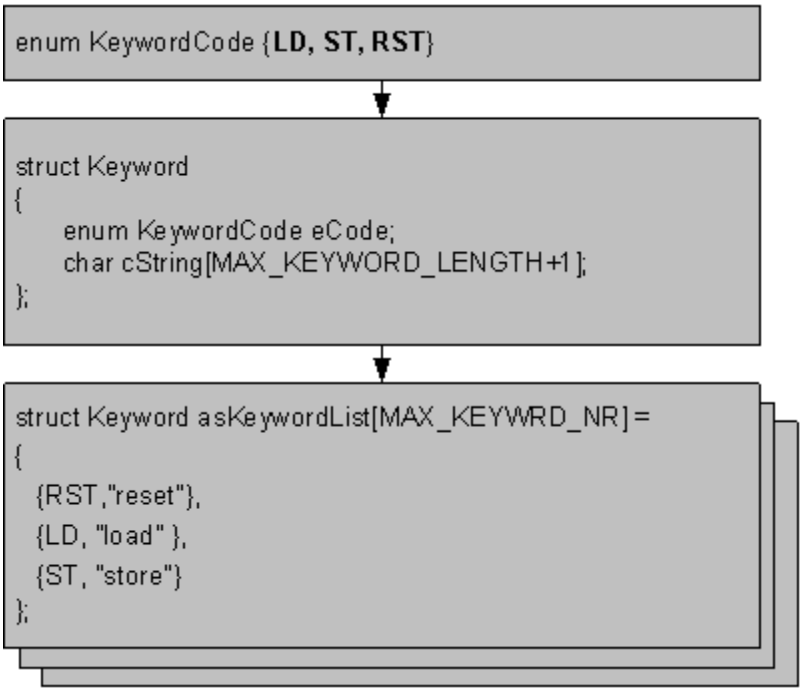
Lista słów kluczowych

Do sprawdzenia czy token jest typu KEYWORD musi istnieć lista łańcuchów znakowych rozpoznawanych jako słowo kluczowe. Elementem listy będzie struktura składająca się z dwóch elementów – kodu słowa kluczowego oraz związanego z nim łańcucha znakowego.

```
#define MAX_KEYWORD_STRING_LTH  10 // mksymalna dlugosc komendy

typedef enum KeywordCode { LD, ST, RST};

typedef struct Keyword
{
 enum KeywordCode eCode;
 char cString[MAX_KEYWORD_STRING_LTH + 1];
};
```

Deklaracja listy poprawnych słów kluczowych będzie wyglądać następująco:

```
#define MAX_KEYWORD_NR 3

struct Keyword asKeywordList[MAX_KEYWORD_NR]= ↵
{
 {RST,"reset"},
 {LD, "load" },
 {ST, "store"}
};
```



Zmienne

Przed implementacja funkcji należy zadeklarować następujące zmienne globalne:

```
asKeywordList[]    // uzywana przez bStringToCommand
asToken[]          // wypelniana przez DecodeMsg na podstawie
                   // cUartRxBuffer i asCommandList
ucTokenNr          // liczba tokenów w zdekodowanym komunikacie
```

Do dekodowania potrzebne będą następujące funkcje:

`ucFindTokensInString (char *pcString)`

Zadania

Wypełnić pola `uValue` tablicy a`sToken` wskaźnikami na początki tokenów znajdujących się w łańcuchu znakowym *String*.
Funkcja powinna zwracać ilość znalezionych tokenów.
Uwaga: Funkcja powinna indeksować nie więcej tokenów niż rozmiar tablicy a`sToken`.

Implementacja

Implementacja funkcji powinna mieć postać automatu.

- Automat powinien mieć dwa stany, TOKEN i DELIMITER, a jego wejściami powinny być kolejne znaki analizowanego łańcucha
- Funkcja powinna składać się tylko z definicji/inicjalizacji zmiennych i jednej pętli `for`.
- Pole warunku pętli for powinno być puste.
- W pętli powinny znajdować się tylko odwołanie do tablicy znaków i instrukcja `switch`.
- `Switch` powinien być sterowany stanem automatu.
- Poszczególne klauzule `case` powinny składać się tylko z jednej instrukcji `if-else-else...` oraz jednej instrukcji `break..`
- W warunkach `if`-else powinno znajdować tylko po jednym porównaniu.
- `If`-ów nie należy zagnieżdżać.
- Do tablicy znaków można się odwołać tylko dwa razy (max. 2 x [], zastosować zmienną pomocniczą)
- W `case`-ach można tylko ustawiać stan automatu.
- Jeżeli to możliwe unikać wskaźników

Testy

Ponadto należy sprawdzić czy funkcja jest odporna na:

- pusty łańcuch, tj. Łańcuch składający się z samych delimiterów
- delimiter przed pierwszym tokenem
- więcej niż jeden delimiter miedzy dwoma tokenami

`eSringToKeyword (char pcStr[],enum KeywordCode *peKeywordCode)`

Zadaniem funkcji jest zamienić łańcuch znakowy na kod słowa kluczowego na podstawie listy słów kluczowych.
W przypadku powodzenia funkcja powinna zwracać wartości OK.
W przypadku niepowodzenia funkcja powinna zwracać wartości ERROR.

`DecodeTokens(void)`

Zadaniem funkcji jest zdekodować wszystkie tokeny tj. dla każdego tokenu ustalić jego typ i wartość i wpisać je do tablicy a`sToken`.
Należy skorzystać ze wskaźników początków tokenów znajdujacych się w a`sToken[0..ucTokenNr].uValue.pcString` (patrz funkcja `ucFindTokensInString`).
1 x []
1 x for

DecodeMsg(char *pcString)

Na podstawie `String-a` i `asCommandList` wypełnia tablice `sToken` i ustawia zmienna `ucTokenNr`.
W tym celu:

- indeksuje początki tokenów
- zamienia wszystkie *delmitery* na *nulle*
- dekoduje poszczególne tokeny.

```
DecodeMsg
    FindTokensInString
    ReplaceCharactersInString
    DecodeTokens
      eStringToKeyword
      eHexStringToUInt
        eCompareString
```

Wcięcia reprezentują zagnieżdżenie funkcji.
Kolejność funkcji jest zgodna z kolejnością ich wywołań.

Testy

Każdej funkcji (oprócz funkcji „main”) powinna odpowiadać funkcja testująca.
Nazwa funkcji testującej powinna zaczynać się od „TestOf_” a kończyć pełną nazwą funkcji testowanej.
Format funkcji testujących powinien być zgodny z poniższym przykładem.

```
 void TestOf_eHexStringToUInt(void){
  // deklaracje zmiennych pomocniczych

  printf("bHexStringToUInt\n\n ");

  printf ("Test 1 - ");
  // krótki opis, jaki jest cel testu nr 1 dac w komentarzu
  // jakies przygotowania do testu1
  if (xx==yy) printf("OK\n") else printf("Error\n");

  printf ("Test 2 - ");
  // krótki opis jaki jest cel testu nr 1
  // jakies przygotowania do testu2
  if (xx==yy) printf("OK\n") else printf("Error\n");

  …
 }
```

To, co można i należy w nim modyfikować to:
Nazwy funkcji testowanej,
Warunki w „if-ach”,
Deklaracje zmiennych pomocniczych,
Krótki opis celu testu,
Przygotowania do testu.
Reszta powinna pozostać niezmieniona.
Uruchamianie testów powinno następować w funkcji `main` w sposób pokazany poniżej

```
 void main(void){

  printf("TESTY FUNKCJI DO OPERACJI NA STRINGACH \n\n\n ");

  TestOf_eHexStringToUInt();
  TestOf_xXXX();
  TestOf_yYYY();
  ….

  printf("TESTY FUNKCJI DO DEKODOWANIA KOMUNIKATÓW\n\n\n ");

  TestOf_aAAA();
  TestOf_bBBB();
 ….
 }
```

Uwaga: Jeżeli istnieje taka możliwość należy w testach używać funkcji do porównywania łańcuchów (oczywiście nie dotyczy samej funkcji do porównywania łańcuchów).