STRUKTURA PROJEKTÓW OPROGRAMOWANIA

ORAZ

STANDARD KODOWANA

DLA JĘZYKÓW C i C++

Autor: Łukasz Krzak
lkrzak@agh.edu.pl
Kraków 2007

1. Wstęp.

Niniejszy dokument ma za zadanie wyznaczenie jednolitego standardu zapisywania kodu oprogramowania powstającego w języku C lub C++. Mimo, iż dotyczy tylko dwóch języków programowania, proponowane standardy formatowania powinny przenosić się na wszystkie inne języki podobnego lub wyższego poziomu (na tyle na ile jest to możliwe).

Poniższe instrukcje zawierają wskazówki i zalecenia dotyczące struktury projektu, formatowania kodu, nazewnictwa, korzystania z narzędzi kontroli wersji itp. Materiały te są wynikiem prac nad kilkoma projektami, konsultacji z różnymi osobami zaangażowanymi w produkcję oprogramowania oraz poszukiwań w sieci i literaturze. Dotyczą głównie programowania systemów mikroprocesorowych, chociaż uwzględniono także tworzenie oprogramowania na platformie Windows (np. programy testujące).

2. Dobór narzędzi.

We wstępnym etapie planowania projektu ważny jest wybór odpowiednich narzędzi. Pozwolą one zaoszczędzić mnóstwo czasu i uniknąć częstych problemów. W trakcie pracy z oprogramowaniem w językach C i C++ kluczowymi narzędziami są:

- kompilator i linker (wraz z zestawem przynajmniej podstawowych bibliotek języka C), przystosowany do generowania kody wykonującego się na docelowym sprzęcie (np. danej rodzinie mikrokontrolerów). W przypadku języka C kompilator taki powinien mieć możliwość pracy w ścisłym standardzie ANSI C, chociaż dopuszczalne (choć nie zalecane) jest pisanie kodu zawierającego rozszerzenia specyficzne dla danego kompilatora.
- IDE (ang. *Integrated Design Environment*) czyli środowisko (najczęściej graficzne) zawierające edytor kodu. Edytor taki powinien mieć możliwość podświetlania składni C i C++, zarządzania wieloma folderami wchodzącymi w skład projektu oraz uruchamiania zewnętrznych narzędzi (np. kompilatora).
- Programator wraz oprogramowaniem, pozwalający na zaprogramowanie docelowego sprzętu
- Emulator (najczęściej JTAG), często pełniący funkcję programatora, ale pozwalający także na kontrolowane uruchamianie kodu w docelowym sprzęcie (krokowanie kodu, stawianie pułapek – jednym słowem debugging).
- Narzędzia kontroli wersji, pozwalające na inkrementalne zapisywanie zmian zachodzących w projekcie, tworzenie logu prac, bezkonfliktowe łączenie różnych gałęzi projektu oraz wyciąganie różnych wersji zapisanych w historii.
- Narzędzia wspierające dokumentację kodu. Mają za zadanie ułatwienie sporządzania dokumentacji oprogramowania.
- System operacyjny czasu rzeczywistego. Pozwala uprościć pisanie oprogramowania, obsługującego wiele zadań w ścisłych reżimach czasowych. Pozwala w łatwy sposób rozdzielać i sterować wykonywaniem zadań na pojedynczym procesorze. Zalecany w przypadku bardziej rozbudowanych projektów, wymagających przetwarzania informacji z wielu źródeł, sterowania wieloma wyjściami itp. Nie zalecany właściwie tylko w przypadku bardzo małych projektów.

W chwili pisania tego dokumentu następujące narzędzia zostały wytypowane, jako najbardziej korzystne:

Kompilator i linker: pakiet GCC. Obsługuje wiele architektur, jest darmowy, posiada dobre wsparcie ze strony środowiska użytkowników. Rozwijany na systemach unix/linux, ale często dostępny w postaci plików wykonywalnych na platformie Windows. Istnieją specjalne dystrybucje przeznaczone dla konkretnych architektur, takie jak WinAVR (dla rodziny Atmel AVR) WinARM (dla mikrokontrolerów z rdzeniem ARM) itp.

IDE: Eclipse. Darmowe środowisko napisane w javie. Wymaga zainstalowania wirtualnej maszyny JAVA. Obsługuje wiele języków programowania, łatwo integruje się z kompilatorem GCC. Dostępne są nawet specjalne dystrybucje dopasowane do konkretnej architektury (np. YAGARTO dla mikrokontrolerów z rdzeniem ARM). W przypadku oprogramowania pisanego dla systemu Windows wytypowano pakiet MS Visual Studio 2003, zawierające zarówno IDE jak i kompilator.

Narzędzie kontroli wersji: SVN (TortoiseSVN). Spełnia wszystkie wymagania i jest często wykorzystywana w małych i średnich projektach. W firmie znajduje się serwer SVN na którym

należy stworzyć dedykowane repozytorium realizowanego projektu oprogramowania. Więcej w rozdziale „Kontrola wersji z SVN”.

Narzędzia wspomagające dokumentację kodu: doxygen. Jest to program przetwarzający proste skrypty umieszczane bezpośrednio w komentarzach znajdujących się w plikach projektu. Ich użycie pozwalają automatycznie generować niskopoziomową dokumentację kodu.

Zaawansowane narzędzia wspomagające dokumentację projektów: Enterprise Architect. Jest to edytor języka UML, przeznaczony głównie do modelowania oprogramowania, ale świetnie spełniający tylko funkcję dokumentowania. Wymaga znajomości składni UML. Doskonale sprawdza się jako narzędzie do zbierania i zarządzania wymaganiami, jako wizualizacja struktury oprogramowania, procesów czy nawet pojedynczych funkcji. Posiada wsparcie dla wielu diagramów UML.

System operacyjny czasu rzeczywistego: FreeRTOS. Darmowy system sportowany na wiele różnych architektur. Posiada świetne wsparcie ze strony użytkowników i developerów. Pozwala konstruować nawet bardzo rozbudowane funkcjonalnie programy, zachowując przy tym przejrzystość i ścisłą kontrolę nad zależnościami czasowymi. Wymaga niewielkich zasobów systemowych (ok. 4kB pamięci programu i kilkaset bajtów pamięci operacyjnej). Istnieje możliwość zakupu profesjonalnej wersji ze wsparciem producenta.

Narzędzia programujące i emulujące nie zostały uwzględnione, ponieważ zależą silnie od docelowej platformy sprzętowej.

3. Struktura projektu.

Proponowana struktura projektu sprawdza się zarówno w przypadku systemów jedno- jaki i wielo-procesorowych. Generalnie projekt dzielony jest na pod-projekty, z których każdy generuje zestaw plików potrzebnych do zaprogramowania pojedynczej jednostki przetwarzającej (CPU, np. mikrokontrolera, procesora, FPGA itp.). Przykładowo, jeżeli projekt uwzględnia programowanie dwóch mikrokontrolerów, to projekty powinny być dla nich rozdzielone na dwa pod-projekty, chyba że ich podobieństwo jest na tyle duże, że można je uznać jako dwa szczególne przypadki jednego projektu. Dodatkowo, z każdego z pod-projektów wyróżnione powinny być ich części wspólne. Każdy pod-projekt oraz część wspólna ma własny folder w strukturze projektu.

Rozważmy przykład. Projekt zakłada istnienie dwóch urządzeń. Pierwsze urządzenie posiada dwa mikrokontrolery, z których jeden obsługuje interfejs użytkownika (np. klawiatura i wyświetlacz) a drugi realizuje jakiś algorytm sterowania. Drugie urządzenie jest sterowane i komunikuje się z mikrokontrolerem realizującym to sterowanie w urządzeniu pierwszym.
Załóżmy dodatkowo, że obsługa wyświetlacza wykorzystuje standardową bibliotekę o nazwie „GraphX", wewnątrz pierwszego urządzenia dwa mikrokontrolery komunikują się poprzez interfejs UART, zaś oba urządzenia, poprzez sieć CAN. Dodatkowo, możemy wymagać, aby pierwsze urządzenie komunikowało się z oprogramowaniem testowym na komputerze PC poprzez łącze USB. Struktura takiego projektu mogłaby wyglądać następująco:

projekt_sterowanie	-	główny folder projektu
common	-	folder grupujący biblioteki wspólne
GraphX	-	folder z plikami biblioteki GraphX
UART	-	folder z plikami mikrosterownika portu UART
CAN	-	folder z plikami mikrosterownika portu CAN
USB	-	folder z plikami sterownika portu USB
proj_device_1	-	folder grupujący pod-projekty dla urządzenia 1
proj_uc_1	-	folder z plikami programu dla mikrokontrolera 1 urządzenia 1
proj_uc_2	-	folder z plikami programu dla mikrokontrolera 2 urządzenia 1
proj_device_2	-	folder z plikami programu dla urządzenia 2
pc_test	-	folder z plikami aplikacji testującej, działającej na komputerze PC
doxygen	-	folder ze skryptami doxygen'a
doc	-	folder z dokumentacją oprogramowania
uml	-	folder z plikami języka uml (jeśli jest wykorzystywany)

Przy ustalonej strukturze folderów, każdy pod-projekt może korzystać z plików zawartych w innych folderach. Przykładowo, proj_uc_1 może korzystać z bibliotek wspólnych, aplikacja pc_test może korzystać z plików proj_device_2 itd.

Ta struktura prawdopodobnie nie uwzględnia niektórych przypadków, ale powinna się sprawdzić i być stosowana tam, gdzie jest to możliwe.

4. Nazewnictwo plików i folderów.

Unikamy stosowania dużych liter i spacji! Absolutnie niedopuszczalne jest stosowanie polskich liter w nazwach plików, jak również znaków specjalnych (-,+, kropka, przecinek, itp.). Nazwy plików mogą mieć długość większą od 8 znaków. Rozszerzenie pliku powinno jednoznacznie identyfikować jego zawartość (pliki *.c to pliki w języku C, *.cpp to pliki w języku C++ itp.). Nazwa pliku powinna sugerować, co się w nim znajduje, ale raczej staramy się nie rozbudowywać zbytnio nazw. Dobrze jest stosować skróty. Język nazw folderów jest dowolny, natomiast preferowany język nazw plików to angielski.

Jeżeli pewien fragment programu posiada nazwę własną (np. protokół transmisyjny, biblioteka funkcji itp.) to dobrze jest z niej skorzystać.

5. Struktura pod-projektów.

Struktura ta zależy niestety silnie od docelowego sprzętu, a co za tym idzie wybranych narzędzi kompilacji, debuggowania itp. Można jednak podać kilka zaleceń, ujednolicających wiele rozwiązań.

Struktura projektu powinna posiadać następujące pliki:

`config.h` – plik w którym grupowane są wszystkie definicje sterujące kompilacją całego pod-projektu, włączające/wyłączające pewną funkcjonalność. Mogą się tam znaleźć także definicje sterujące kompilacją bibliotek. Przykładowo mogą się tam znaleźć definicje rozmiarów buforów, opcje załączające do kompilacji kod przeznaczony dla debuggowania itp., włączające komunikację przez dany port itp.

`includes.h` – plik w którym powinny znaleźć się wszystkie deklaracje #include załączające pliki z bibliotek języka C, kompilatora, środowiska itp. oraz inne pliki nagłówkowe, które mają być widoczne w całym projekcie. Dodatkowo powinny się tam znaleźć co najmniej 2 makrodefinicje:

```
#define ENTER_CRITICAL(TempSREG)
#define EXIT_CRITICAL(TempSREG)
```

których zadaniem jest odpowiednio wyłączanie i włączanie globalne przerwań. Przykładowe implementacje tych makr (dla procesora AVR) podano poniżej:

```
#define ENTER_CRITICAL(TempSREG)        do {                    \
                                            TempSREG = SREG;    \
                                            cli();              \
                                        } while (0)

#define EXIT_CRITICAL(TempSREG)         SREG = TempSREG
```

`globals.h / globals.c` – w tych plikach powinny znaleźć się deklaracje wszystkich zmiennych globalnych, np. w takiej postaci:

```
w globals.c:

   volatile uint32_t GlobalStatus;

w globals.h:

   extern uint32_t GlobalStatus;
```

6. Zalecenie dotyczące formatowania kodu.

6.1. Każdy plik powinien mieć w nagłówku ujednolicony komentarz. Zalecany format nagłówka:

```
/*
================================================================================
 Plik: errors.h
--------------------------------------------------------------------------------
 Autorzy:       Łukasz Krzak
 Projekt:       Nowe Radio
 Kompilator:    ARM-ELF-GCC 4.0.1 (WinARM)
 Hardware:      STR71x
 Doc:           doxygen 1.4.3
--------------------------------------------------------------------------------
 Zawiera:       funkcje obsługi błędów
--------------------------------------------------------------------------------
 Komentarze:

================================================================================
*/
```

6.2. Pliki powinny być podzielone na sekcje. Daną sekcję można pominąć, jednak ich kolejność musi zostać zachowana. Poniżej wymieniono sekcje dla plików implementacji (*.c i .cpp) oraz plików nagłówkowych (*.h).

Sekcje w plikach implementacji:

- Nagłówek pliku
- Sekcja załączania plików (deklaracje #include)
- Sekcja #define dla stałych (nazwa: CONSTANTS)
- Sekcja #define dla makr (nazwa: MACROS)
- Definicje lokalnych typów (nazwa: LOCAL TYPES)
- Definicje lokalnych zmiennych (nazwa: LOCAL VARIABLES)
- Definicje lokalnych tablic (nazwa: LOCAL TABLES)
- Definicje prototypów lokalnych funkcji (nazwa: LOCAL FUNCTION PROTOTYPES) – z zachowaniem kolejności w jakiej są implementowane.
- Globalne funkcje (nazwa: PUBLIC FUNCTIONS) – pogrupowane wg. funkcjonalności
- Lokalne funkcje (nazwa: LOCAL FUNCTIONS) – kolejność zgodna z prototypami

Sekcje w plikach nagłówkowych:

- Nagłówek pliku
- Sekcja #define dla stałych (nazwa: CONSTANTS)
- Sekcja #define dla makr (nazwa: MACROS)
- Definicje globalnych typów (nazwa: GLOBAL TYPES)
- Definicje eksportów (nazwa: EXPORTS)
- Prototypy funkcji publicznych (nazwa: PUBLIC FUNCTIONS)) – pogrupowane wg. funkcjonalności, tak jak w pliku implementacji

6.3. Każda sekcja powinna być poprzedzona blokiem komentarza, zawierającego nazwę sekcji:

```
// ------------------------------------------------------------------------
//  PUBLIC FUNCTIONS
// ------------------------------------------------------------------------
```

6.4. Wcięcia mają zawsze wielokrotność 4 spacji. Nie wolno używać znaku tabulacji (wg. ASCII 0x09).
6.5. Maksymalnie dozwolone są 132 znaki w jednej linii (zalecane 120).
6.6. Każda instrukcja (akcja) powinna być umieszczona w osobnej linii, nawet "pusta" instrukcja.
6.7. Każdy blok { } sformatowany na sposób K&R:

```
if (a > b) {
    a = b;
} else {
    x = y - z;
    z = -25;
}
```

6.8. Sposób użycia spacji w zapisie wywoływania i deklaracji funkcji:

```
void func(int8_t arg1, int8_t arg2)
```

6.9. Wyrażenia z nawiasami zapisujemy bez spacji po nawiasie otwierającym i przed zamykającym:

```
x = (a + b) * c;
```

6.10. Nazwy stałych, makr, typów wyliczeniowych (enum) – duże litery, słowa oddzielone znakiem _ (underscore), np. MAX_BUFFER_SIZE, GET_LVALUE(x) itp.
6.11. Wszystkie makra w formie MACRO() nawet, gdy nie mają parametrów
6.12. Lokalne zmienne (np. w obrębie funkcji) – małe litery, słowa oddzielone znakiem _ (underscore), np. max_value, rx_data itp. Staramy używać się standardowych liter jako typowych zmiennych, np. i,j,k – jako liczniki pętli, p – jako wskaźniki itp.
6.13. Zmienne widoczne w obrębie stałego modułu – poprzedzone prefiksem identyfikującym moduł, kolejne słowa zaczynające się od dużych liter, np. DISP_BufSize, COMM_Channel itp.
6.14. Globalne zmienne w plikach „globals.*" - kolejne słowa zaczynające się od dużych liter, mogą być poprzedzone prefiksem modułu, z którym są związane, np. BufDesc, RFCOMM_Desc itp.
6.15. Lokalne funkcje - poprzedzone prefiksem identyfikującym moduł, kolejne słowa zaczynające się od dużych liter, zdefiniowane jako statyczne, np.:

```
static void COMM_PutChar()
```

6.16. Publiczne funkcje - poprzedzone prefiksem identyfikującym moduł, kolejne słowa zaczynające się od dużych liter, prototyp w pliku nagłówkowym np.:

```
void RFDLNK_SenfFrame();
```

6.17. Pliki nagłówkowe powinny być zabezpieczone przed powtórnym włączeniem. Dodatkowo każdy moduł powinien definiować swój numer wersji, tak jak w przykładzie poniżej (wersja 1.0) :

```
#ifndef COMM_H
#define COMM_H
#define COMM_VER_HI      1
#define COMM_VER_LO      0
…

#endif // COMM_H
```

6.18. Każdy plik powinien włączać wszystkie niezbędne definicje/deklaracje. Znaczy to, że każdy plik zawiera polecenie `#include` dla wszystkich innych plików, z których wykorzystywane są deklaracje lub zmienne.

6.19. Komentarz dla funkcji/zmiennej powinien być umieszczony przed jej deklaracją (funkcje/zmienne interfejsowe) lub definicją (lokalne). Wykorzystywany powinien być format doxygen'a

6.20. Nie wolno używać typów prostych bez wyspecyfikowanej długości, np. nie wolno używać `unsigned char`, zamiast tego powinno być `uint8_t`.

6.21. Zakazane są tzw. "magic numbers", czyli stałe liczbowe bezpośrednio w kodzie. Nawet jeżeli stała używana jest tylko raz to powinna być zdefiniowana jako `#define` lub `enum`. Istnieją wyjątki dotyczące najczęściej liczb 0 i 1 oraz maksymalnych zakresów typów, jak np. 0xff, 0xffff, 0xffffffff itp. Mogą one w uzasadnionych przypadkach być stosowane bezpośrednio w kodzie.

6.22. Wszystkie instrukcje `switch` powinny mieć etykietę `default`.

6.23. Złożone warunki i operacje logiczne, a także obliczenia powinny być zawsze zapisane z użyciem nawiasów, które określają kolejność działań (nawet jeżeli jest to nadmiarowe).

6.24. Jeżeli używamy wskaźnika do struktur lub unii to odwołanie do pól powinno być realizowane przez operator -> a nie przez *. np.:

```
CurrentControl->Text[i] = Value;
```

6.25. Arytmetyka na wskaźnikach powinna być ograniczona do minimum. A w przypadkach kiedy jest to nieuniknione (np. generowana jest o wiele dłuższa kod wynikowy) kod powinien być skomentowany, tak aby była jasna intencja autora.

6.26. W makrach funkcjonalnych (z parametrami) parametry powinny być zapisywane tylko w nawiasach. Np.

```
#define FUNC_MACRO(a) var =(a);
```

6.27. Makra składające się z wielu linii powinny być zawarte w instrukcji do { } while (0) Np.

```
#define DO_SOMETHING() \
do { \
    if (a > b) \
    { \
        a = b; \
    } \
} while (0);
```

6.27. Do sygnalizowania brakujących define-ów, stałych, makr i do sprawdzania poprawności ich wartości należy używać dyrektywy `#error`

6.28. Zaleca się nie używania instrukcji `loop` i `break` (poza switch). Jeżeli wydaje się to niezbędne to powinien być umieszczony komentarz z opisem powodu zastosowania takiego algorytmu.

6.29. Zaleca się w operatorze porównywania == zapis stałej (jeżeli występuje) po lewej stronie.

6.30. Wszystkie struktury i unie musza być zapisane z użyciem `typdef`. Nazwa typu powinna się zaczynać się na 'T'

6.31. W pliku c/cpp każda funkcja powinna zaczynać się od wyraźnego separatora (komentarza z nazwą funkcji) oraz kończyć komentarzem, również z nazwą funkcji

```
// ---------------------------------------------------------------------------
//  MOD_Func
// ---------------------------------------------------------------------------
void MOD_Func(int arg1, int arg2)
{
    ...
} /* MOD_Func */
```

7. Komentowanie z użyciem doxygen'a.

Doxygen definiuje różne metody komentowania poszczególnych elementów kodu programu. Poniżej przedstawiono główne zasady, których przestrzegania pozwoli wygenerować przejrzystą dokumentację. Nie zamykają one drogi do własnych rozwiązań, szczególnie jeśli są one uzupełnieniem poniższych zaleceń.

7.1. Całe oprogramowanie powinno być podzielone na grupy (najczęściej ze względu na funkcjonalność). Każdy plik powinien należeć do danej grupy, przy czym bardzo często już para plików: nagłówkowy (*.c) i implementacji (*.c lub *.cpp) tworzą osobną grupę, zamykając w sobie pewną funkcjonalność (np. obsługa portu szeregowego, bufora, protokołu transmisyjnego itp.). Grupa taka definiowana jest poleceniem `\defgroup` w pliku nagłówkowym, zaraz za definicjami zapobiegającymi powtórnemu załączeniu pliku. W opisie grupy powinny znaleźć się informacje o tym co dana grupa funkcjonalna robi, jak to robi, z czego korzysta, jak jej używać, na co uważać itp. Przykładowo:

```
#ifndef ERRORS_H
#define ERRORS_H

/** \defgroup errors Obsługa błędów
    \code
    #include "errors.h"
    \endcode

    Moduł zawiera funkcje i definicje odpowiedzialne za obsługę błędów,
zgłaszanych przez inne moduły oprogramowania, związane z działaniem
aplikacji. \n
...
*/
/*@{*/
```

... definicje ... makra ... prototypy funkcji ...

```
/*@}*/
```

7.2. W obrębie grupy występują definicje stałych, makr, typów, zmiennych czy wreszcie prototypów funkcji. Wszystkie elementy będące dostępne z poziomu innych plików (widziane globalnie), w szczególności wszystkie elementy znajdujące się w pliku nagłówkowym powinny posiadać swój własny komentarz w formacie doxygen'a.

7.3. Wszystkie definicje stałych, makr, typów, wyliczeń i zmiennych można komentować, używając klasycznych reguł, jak poniżej, jeżeli komentarz składa się z jednej linii:

```
/// Makro zwiększające o jeden wartość parametru
#define INC_MACRO(a)       (a)++
```

lub jeżeli komentarz zajmuje więcej:

```
/** Makro zwiększające o jeden wartość parametru. Może być wykorzystywane do
    działania na liczbach typu całkowitego oraz ... */
#define INC_MACRO(a)       (a)++
```

7.4. Poszczególne elementy kodu mogą być dodatkowo grupowane, np. definicje opisujące możliwe stany jakiegoś parametru. Należy wówczas skorzystać z polecenia `\name`.

```
/** \name Definicje identyfikatorów portu szeregowego
@{*/
/// identyfikator portu 0
#define UART0_ID                        1
/// identyfikator portu 1
#define UART1_ID                        2
/// identyfikator portu 2
#define UART2_ID                        3
/*@}*/
```

7.5. Komentarz dotyczący prototypów funkcji powinien zwierać informacje o tym co robi dana funkcja, w jaki sposób to robi, jak z niej korzystać itp. W wielu przypadkach może być również konieczne umieszczenie informacji o czasie wykonywania (dot. szczególnie systemów czasu rzeczywistego). Niezbędny jest również opis parametrów funkcji i rezultatu. Preferowany format podano poniżej.

```
/**
    Funkcja inicjalizuje moduł SCOM. Zeruje wszystkie zmienne i ustawienia.
    Powinna być wywoływana przed rozpoczęciem korzystania z modułu SCOM.
    \param pUSART wskaźnik do struktury opisu portu szeregowego.
    \return Rezultat modułu scom (zob. \ref TSCOM_Result). \n
            SCOM_RESULT_OK - inicjalizacja się udała, moduł jest gotowy do
                 pracy. \n
            SCOM_USART_TX_BUF_ERROR - zadany port szeregowy nie posiada
                 bufora nadawczego. \n
            SCOM_USART_RX_BUF_ERROR - zadany port szeregowy nie posiada
                 bufora odbiorczego. \n
*/
TSCOM_Result SCOM_Init(PUSART pUSART);
```

7.6. Komentarze znajdujące się w kodzie powinny jasno dokumentować intencje autora, zastosowane algorytmy, znaczenie zmiennych itp. W tym przypadku obowiązują klasyczne zasady komentowania kodu, w zasadzie niezależne od języka programowania.

7.7. Każdy plik powinien zawierać na samym końcu sekcję opisu pliku, wykorzystującą polecenie doxygen'a `\file`, w postaci:

```
/*! \file errors.c
    \brief Biblioteka funkcji obsługujących błędy w aplikacji.
*/
```

8. Kontrola wersji z SVN.

Narzędzie kontroli wersji pozwala na tworzenie dowolnych elektronicznych dokumentów poprzez budowanie historii ich zmian. Jednym z takich narzędzi jest system Subversion, zwany potocznie SVN. Aby łatwo i wygodnie korzystać z niego w systemie Windows, można skorzystać z klienta TortoiseSVN, który integruje się z powłoką, udostępniając nowe opcje kontekstowe dla plików i folderów.

Historia zmian plików, będących pod kontrolą wersji przechowywana jest na dysku w specjalnym folderze zwanym *repozytorium*. Do repozytorium mogą być dodawane dowolne pliki. W repozytorium stworzona może zostać także struktura folderów. Zaleca się stosowanie następujących zasad:

8.1. Każdy projekt ma osobne repozytorium.
8.2. W głównej gałęzi każdego repozytorium znajdują się tylko trzy foldery:
trunk
branches
tags
Wewnątrz nich mogą oczywiście znajdować się inne foldery.
8.3. W folderze trunk znajduje się aktualna wersja rozwojowa
8.4. W folderze branches znajdują się odgałęzienia projektów, rozwijane niezależnie od głównej wersji.
8.5. W folderze tags znajdują się obrazy wersji testowych, wersji demonstracyjnych, release'ów itp.

Przykładowa struktura folderów, wytłuszczone zostały foldery będące repozytoriami w sensie SVN. Wszystkie pliki i foldery wewnątrz nich są pod kontrolą wersji. Przykład nawiązuje do pkt. 3.

repositories
  software
    projekt_sterowanie
      trunk
        common
          GraphX
          UART
          CAN
          USB
        proj_device_1
          proj_uc_1
          proj_uc_2
        proj_device_2
        pc_test
        doxygen
        doc
        uml
      branches
      tags
    radiomodem_pr2007
      trunk
      branches
      tags