STM32
Migracja do Standard Peripheral Library wer. 3.1.0

We wcześniejszych numerach EP wielokrotnie przedstawiane były aplikacje pisane dla mikrokontrolerów STM32 z wykorzystaniem biblioteki API dostarczanej przez ST Microelectronics. Jednak w myśl słów: kto nie idzie naprzód, ten się cofa – firma ST przygotowała nową wersję biblioteki. W artykule przedstawiono skrótowo informacje dotyczące zarówno starszej wersji biblioteki (V2.0.3), oraz – szczegółowej – nowej wersji 3.1.0. Omówiono ponadto sposób migracji projektu ze starej wersji biblioteki do nowej.

Operowanie bezpośrednio na rejestrach jakiekolwiek 32–bitowego procesora lub mikrokontrolera nie należy do zadań łatwych. Mimo, iż samo napisanie (nawet stosunkowo zaawansowanych) aplikacji przy użyciu nazw rejestrów jest możliwe, to wprowadzanie zmian do istniejącego kodu po upływie na przykład kilku miesięcy, dodatkowo przez osobę, która nie jest autorem programu, jest w zasadzie niemożliwe do wykonania w sensownym czasie. Z powyższych względów, jeśli to możliwe, czyli np. czas wykonania kodu nie jest krytyczny, programiści wykorzystują funkcje o mniej, lub bardziej kojarzących się nazwach, do operacji często na zespołach wielu rejestrów.

Jeżeli mamy do czynienia z bardzo skomplikowanym projektem, wykorzystującym wiele peryferiów mikrokontrolera, to napisanie stosownych funkcji jest praco- i czasochłonne, ponadto wymaga bardzo dobrej znajomości architektury mikrokontrolera. Firma ST Microelectronics zauważyła ten problem i udostępnia kompletne biblioteki API, które pozwalają na pełną kontrolę nad mikrokontrolerami STM32. W pewnych przypadkach może oczywiście zajść potrzeba bezpośredniego odwołania się do rejestru, we wszystkich pozostałych funkcje API znacznie skracają czas potrzebny na napisanie i uruchomienie aplikacji.

STM32F10x firmware library V2.0.3

Biblioteka firmy STMicroelectronics dla mikrokontrolerów STM32 w wersji V2.0.3 (*FWLib*) została zastąpiona przez nowszą wersję i nie jest już dalej rozwijana. Mimo to, nadal można ze strony producenta pobrać tę wersję biblioteki, jednak tworząc nowy projekt należy już raczej skorzystać z nowszej wersji 3.1.0, która została omówiona w dalszej części artykułu. Poniżej przedstawione informacje mogą być przydatne podczas migracji do nowej wersji biblioteki.

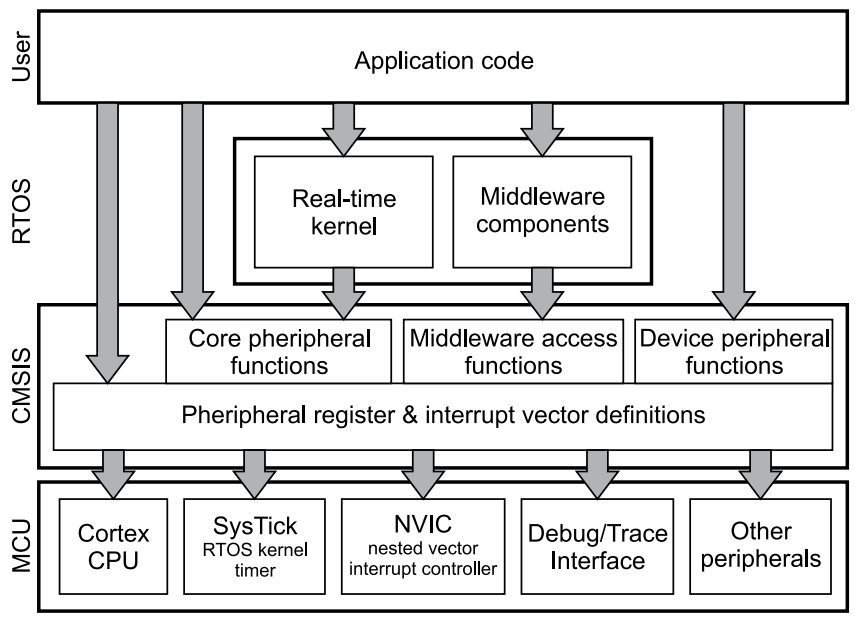
Biblioteka *FWLib* jest podzielona na wiele plików, każdy jest odpowiedzialny na obsługę jakiegoś elementu systemu mikroprocesorowego (układu peryferyjnego). Na przykład plik *stm32f10x_usart.c* zawiera wszystkie niezbędne funkcje do skonfigurowania portu USART oraz do nawiązania komunikacji. Plików bibliotecznych nie edytujemy, jedynie w nagłówkowym pliku *stm32f10x_conf.h*, używając komentarzy można włączać lub wyłączać obsługę poszczególnych urządzeń. Ważnym elementem biblioteki *FWLib* jest plik *stm32f10x_it.c*, w którym umieszczane są wszystkie funkcje obsługi przerwań.

CMSIS

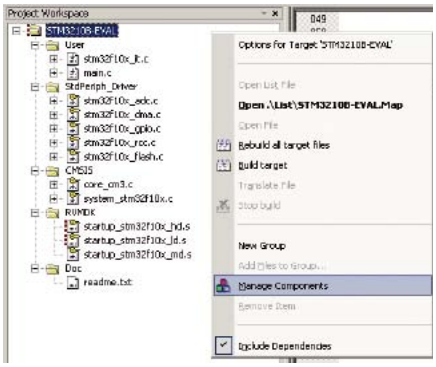
Biblioteka *STM32F10x Standard Peripherals Library V3.1.0* wykorzystuje standard CMSIS, a więc warto się nieco z nim zapoznać. Standard CMSIS (*Cortex Microcontroller Software Interface Standard*) jest to uniwersalny interfejs programowy, stworzony przez firmę ARM, który umożliwia komunikację z peryferiami i rdzeniem Cortex za pomocą ustandaryzowanych funkcji i definicji. CMSIS dostarcza mechanizmów do obsługi układów peryferyjnych, systemów operacyjnych czasu rzeczywistego oraz aplikacji wykorzystujących interfejsy komunikacyjne: Ethernet, UART oraz SPI.

Strukturę interfejsu CMSIS i jego miejsce w aplikacji przedstawiono na **rys. 1**. CMSIS ma docelowo obejmować cała rodzinę rdzeni z grupy Cortex-M, natomiast na chwilę obecną jest dostosowany do rdzeni Cortex -M0, oraz Cortex-M3. Ustandaryzowane interfejsu dostępu do układów peryferyjnych oraz samego rdzenia ma na celu ułatwienie przenoszenia aplikacji pomiędzy mikrokontrolerami różnych producentów, jak również uproszczenie procesu tworzenia aplikacji.

Standard CMSIS został podzielony na dwie podstawowe warstwy: *Core Peripheral Access Layer* oraz *Middleware Access Layer*. Pierwsza warstwa zawiera definicje nazw oraz umożliwia dostęp do rejestrów rdzenia oraz urządzeń peryferyjnych, natomiast druga udostępnia mechanizmy do współpracy z interfejsami komunikacyjnymi.



Rys. 1. Budowa interfejsu *CMSIS* i jego miejsce w aplikacji



Rys. 2. Struktura plików środowiska *µ*Vision dla projektu wykorzystującego przetwornik A/C

Dodatkowo, wymienione wyżej warstwy zostały rozszerzone przez producentów mikrokontrolerów, którzy współpracowali przy tworzeniu CMSIS, o dwie „warstwy": *Device Peripheral Access Layer* oraz *Access Functions for Peripherals*.

STM32F10x standard peripheral library V3.1.0

Biblioteka *STM32F10x Standard Peripherals Library v3.1.0.* (*StdPeriph_Lib*) została napisana zgodnie z formatem *Doxygen*, co znacznie upraszcza proces tworzenia dokumentacji oraz jej używanie. Cala dokumentacja omawianej biblioteki została umieszczona w pliku pomocy, a nie, jak było to dotychczas praktykowane przez STMicroelectronics, w pliku pdf. Nowy sposób dostarczania dokumentacji ułatwił przeglądanie jej zawartości, oraz wyszukiwanie informacji.

Wraz z archiwum biblioteki *StdPeriph_Lib* otrzymujemy szablony projektów do trzech najpopularniejszych kompilatorów (środowisk), a są to: IAR, Keil oraz darmowy kompilator GCC. Programista jest zatem zwolniony z obowiązku samodzielnego tworzenia projektu od podstaw. Struktura plików projektu, dla przykładu wykorzystującego przetwornik A/C,

```
[CMSIS]
  ├[Core]
  │  ├[CM3]
  │  │  ├[startup]
  │  │  │  ├[arm]
  │  │  │  │  ├startup_stm32f10x_cl.s
  │  │  │  │  ├startup_stm32f10x_hd.s
  │  │  │  │  ├startup_stm32f10x_ld.s
  │  │  │  │  └startup_stm32f10x_md.s
  │  │  │  ├[gcc]
  │  │  │  │  ├startup_stm32f10x_cl.s
  │  │  │  │  ├startup_stm32f10x_hd.s
  │  │  │  │  ├startup_stm32f10x_ld.s
  │  │  │  │  └startup_stm32f10x_md.s
  │  │  │  └[iar]
  │  │  │     ├startup_stm32f10x_cl.s
  │  │  │     ├startup_stm32f10x_hd.s
  │  │  │     ├startup_stm32f10x_ld.s
  │  │  │     └startup_stm32f10x_md.s
  │  │  ├core_cm3.c
  │  │  ├core_cm3.h
  │  │  ├stm32f10x.h
  │  │  ├system_stm32f10x.c
  │  │  └system_stm32f10x.h
  │  └[Documentation]
  │     └CMSIS_Core.htm
  └License.doc
```

Rys. 3. Struktura modułu *CMSIS*

zbudowanego w oparciu o szablon projektu dla środowiska *µ*Vision przedstawiono na **rys. 2**.

Jeśli szablon projektu zostanie skopiowany do innego katalogu, to należy poinformować kompilator, gdzie ma szukać stosownych plików oraz należy dodać wykorzystywane źródła do projektu. Dokonujemy tego (w środowisku *µ*Vision) poprzez prawe kliknięcie na nazwie projektu i wybór z menu kontekstowego opcji „*Manage Components*" – patrz rys. 2. Następnie odszukujemy na dysku właściwe pliki i je dodajemy.

Jak wyżej wspomniano, aby projekt poprawnie się kompilował, należy zaktualizować miejsca (ścieżeki), gdzie kompilator będzie szukał plików nagłówkowych **.h*. W tym celu wybieramy menu „*Project/Options for Target…*", po czym otworzy się okno, w którym na zakładce „*C/C++*" edytujemy ścieżkę poszukiwania plików nagłówkowych. W ten sposób przygotowany projekt, jeśli kod aplikacji jest pozbawiony błędów, powinien się bez błędów zbudować i załadować do pamięci mikrokontrolera.

Szybkie rozpoczęcie pracy z mikrokontrolerami STM32 znacznie ułatwiają dołączone do biblioteki *StdPeriph_Lib* przykładowe aplikacje, które pozwalają w krótkim czasie zapoznać się z możliwościami tych układów.

Struktura biblioteki *StdPeriph_Lib*

Pliki w bibliotece API zostały podzielone na dwa bloki (moduły). Są to katalogi *STM32F10x_StdPeriph_Driver* oraz *CMSIS*. Drzewo plików i katalogów modułu *CMSIS* biblioteki Standard Peripherals Library pokazano na **rys. 3**, natomiast drzewo modułu *STM32F10x_StdPeriph_Driver* na **rys. 4**.

Dla każdej z rodzin mikrokontrolerów STM32 zostały przygotowane oddzielne pliki startowe. W **tab. 1** przedstawiono, który plik startowy należy wykorzystać z jaką rodziną STM32. Takich zestawów plików startowych otrzymujemy wraz z biblioteką API

List. 1. Fragment pliku *system_stm32f10x.c* – definicje częstotliwości zegara systemowego, wykorzystywane przez funkcję *SystemInit()*

```
/* #define SYSCLK_FREQ_HSE    HSE_
Value */
/* #define SYSCLK_FREQ_24MHz
24000000 */
/* #define SYSCLK_FREQ_36MHz
36000000 */
/* #define SYSCLK_FREQ_48MHz
48000000 */
/* #define SYSCLK_FREQ_56MHz
56000000 */
#define SYSCLK_FREQ_72MHz  72000000
```

Tab. 1. Pliki startowe dla poszczególnych rodzin mikrokontrolerów STM32

Plik	Rodzina STM32
startup_stm32f10x_cl.s	Connectivity line
startup_stm32f10x_hd.s	High Density
startup_stm32f10x_ld.s	Low Density
startup_stm32f10x_md.s	Medium Density

```
[STM32F10x_StdPeriph_Driver]
  ├[inc]
  │  ├misc.h
  │  ├stm32f10x_adc.h
  │  ├stm32f10x_bkp.h
  │  ├stm32f10x_can.h
  │  ├stm32f10x_crc.h
  │  ├stm32f10x_dac.h
  │  ├stm32f10x_dbgmcu.h
  │  ├stm32f10x_dma.h
  │  ├stm32f10x_exti.h
  │  ├stm32f10x_flash.h
  │  ├stm32f10x_fsmc.h
  │  ├stm32f10x_gpio.h
  │  ├stm32f10x_i2c.h
  │  ├stm32f10x_iwdg.h
  │  ├stm32f10x_pwr.h
  │  ├stm32f10x_rcc.h
  │  ├stm32f10x_rtc.h
  │  ├stm32f10x_sdio.h
  │  ├stm32f10x_spi.h
  │  ├stm32f10x_tim.h
  │  ├stm32f10x_usart.h
  │  └stm32f10x_wwdg.h
  └[src]
     ├misc.c
     ├stm32f10x_adc.c
     ├stm32f10x_bkp.c
     ├stm32f10x_can.c
     ├stm32f10x_crc.c
     ├stm32f10x_dac.c
     ├stm32f10x_dbgmcu.c
     ├stm32f10x_dma.c
     ├stm32f10x_exti.c
     ├stm32f10x_flash.c
     ├stm32f10x_fsmc.c
     ├stm32f10x_gpio.c
     ├stm32f10x_i2c.c
     ├stm32f10x_iwdg.c
     ├stm32f10x_pwr.c
     ├stm32f10x_rcc.c
     ├stm32f10x_rtc.c
     ├stm32f10x_sdio.c
     ├stm32f10x_spi.c
     ├stm32f10x_tim.c
     ├stm32f10x_usart.c
     └stm32f10x_wwdg.c
```

Rys. 4. Struktura modułu *STM32F10x_StdPeriph_Driver*

trzy – dla trzech najpopularniejszych kompilatorów: Keil, IAR oraz GCC.

Plik *system_stm32f10x.c* zawiera definicje i funkcje, które mogą być wykorzystane do konfiguracji sygnału zegarowego mikrokontrolera. Korzystając z funkcji zawartych w pliku należy w pierwszej kolejności wybrać, używając komentarzy, z jaką częstotliwością MCU ma pracować. Fragment, który należy w tym celu edytować został przedstawiony na **list. 1**. W kodzie aplikacji wystarczy teraz wywołać funkcję *SystemInit()*, a jej wykonanie spowoduje skonfigurowanie wszystkich sygnałów zegarowych (zegar systemowy, HCLK, PCLK2, PCLK1).

Do modułu *STM32F10x_StdPeriph_Driver* należą pliki, które, wzorem starszej wersji biblioteki V2.0.3, zgodnie ze swoją nazwą zawierają funkcje API związane z poszczególnymi urządzeniami peryferyjnymi – patrz **rys. 4**. Tych plików bibliotecznych nie należy edytować.

Do każdego projektu są dołączone jeszcze dwa istotne pliki: *stm32f10x_conf.h* oraz *stm32f10x_it.c*. W pierwszym pliku nagłówkowym za pomocą komentarzy włączamy lub wyłączamy dołączanie do projektu plików nagłówkowych z modułu *STM32F10x_StdPeriph_Driver* – patrz **list. 2**.

Z punktu widzenia programisty jednym z najważniejszych plików jest *stm32f10x_it.c*, w którym umieszcza się wszystkie funkcje obsługi przerwań. Jego nieco zedytowany fragment wraz z pustymi funkcjami jest przestawiony na **list. 3**. Jak widać, jest to

List. 2. Obszar pliku *stm32f10x_conf.h*, w którym za pomocą komentarzy włącza się lub wyłącza dołączanie plików nagłówkowych

```
/* #include „stm32f10x_adc.h” */
/* #include „stm32f10x_bkp.h” */
/* #include „stm32f10x_can.h” */
/* #include „stm32f10x_crc.h” */
/* #include „stm32f10x_dac.h” */
/* #include „stm32f10x_dbgmcu.h” */
#include „stm32f10x_dma.h”
/* #include „stm32f10x_exti.h” */
/* #include „stm32f10x_flash.h” */
/* #include „stm32f10x_fsmc.h” */
#include „stm32f10x_gpio.h”
/* #include „stm32f10x_i2c.h” */
/* #include „stm32f10x_iwdg.h” */
/* #include „stm32f10x_pwr.h” */
#include „stm32f10x_rcc.h”
/* #include „stm32f10x_rtc.h” */
/* #include „stm32f10x_sdio.h” */
#include „stm32f10x_spi.h”
/* #include „stm32f10x_tim.h” */
/* #include „stm32f10x_usart.h” */
/* #include „stm32f10x_wwdg.h” */
/* #include „misc.h” */
```

List. 3. Fragment pliku *stm32f10x_it.c*

```
void UsageFault_Handler(void)
{
        while (1)
        {}
}

void SVC_Handler(void)
{
}

void DebugMon_Handler(void)
{
}

void PendSV_Handler(void)
{
}

void SysTick_Handler(void)
{
}
```

zestaw pustych funkcji, które jeżeli wystąpi odpowiednie przerwanie, są wywoływane. W stosunku do wersji tego pliku dostarczanej przez firmę STMicroelectronics zostały usunięte komentarze, które jak pokazała praktyka okazały się w tym miejscu zbędne. Nazwa funkcji obsługi danego przerwania jest już sama w sobie dość wymowna. W efekcie uzyskano bardziej zwięzły kod, nie tracąc tym samym na jego czytelności.

Zadaniem dewelopera jest umieszczenie w odpowiedniej funkcji obsługi przerwania kodu, jaki ma się wykonać po wystąpieniu określonego zdarzenia. Takie podejście sprawiło, że projekt jest przejrzysty, a obsługa wszystkich wyjątków znajduje się w jednym pliku i nie ma potrzeby, jak to zwykle bywa, samodzielnego definiowania funkcji wywoływanych w chwili, gdy system wykryje nadejście przerwania. Znacznie upraszcza i przyśpiesza to prace nad projektem, jednak nie należy zapominać, że aby tworzyć dobre i niezawodne aplikacje to trzeba bardzo dobrze rozumieć to, co się robi.

List. 4. Fragment pliku *config.ini*, który opisuje zmiany, jakie zostaną wprowadzone do kodu po uruchomieniu aplikacji *MigrationScript*

```
u32<;,;>uint32_t
u16<;,;>uint16_t
u8<;,;>uint8_t
uc32<;,;>const uint32_t
uc16<;,;>const uint16_t
uc8<;,;>const uint8_t
volatile const<;,;>__I
volatile<;,;>__IO
NMIException<;,;>NMI_Handler
HardFaultException<;,;>HardFault_
Handler
MemManageException<;,;>MemManage_
Handler
BusFaultException<;,;>BusFault_
Handler
UsageFaultException<;,;>UsageFault_
Handler
SVCHandler<;,;>SVC_Handler
DebugMonitor<;,;>DebugMon_Handler
PendSVC<;,;>PendSV_Handler
```

Migracja ze starszej wersji biblioteki *STM32F10x firmware library*

Najistotniejsze zmiany w stosunku do poprzedniej wersji biblioteki API to kompatybilność nowej wersji z omówionym wyżej standardem CMSIS. Z punktu widzenia programisty, który wcześniej pracował z biblioteką *STM32F10x firmware library* ważne jest, że nowa wersja (z perspektywy wykorzystania funkcji API) w sumie nie wiele różni się od swojej poprzedniczki.

Biblioteka *STM32F10x Standard Peripherals Library* została tak napisana, aby uaktualnienie projektów wykorzystujących starszą wersję biblioteki (*STM32F10x firmware library*) nie nastręczało problemów. Ze strony internetowej firmy STMicroelectronics można pobrać archiwum o nazwie *an2953* zawierające aplikację *MigrationScript*, która po uruchomieniu automatycznie zmienia kod w plikach tak, aby był możliwie jak najbardziej kompatybilny z nową biblioteką. Niestety te fragmenty kodu, które wykorzystują funkcje całkowicie zmienione, lub usunięte w nowej bibliotece *STM32F10x Standard Peripherals Library* należy edytować ręcznie. Dotyczy to przede wszystkim systemowego timera SysTick.

Zmiany, jakie zostaną wprowadzone do kodu plików za pomocą aplikacji *MigrationScript*, są opisane w pliku *config.ini*, a jego fragment został przedstawiony na **list. 4**.

Z perspektywy programisty wykorzystującego funkcje API, najistotniejszymi zmianami są:

– inne definicje typów zmiennych, np. typ *u32* w nowej bibliotece jest reprezentowany przez *uint32_t*,
– nowe nazwy funkcji obsługi przerwań systemowych, np. *NMIException()* została zmieniona na *NMI_Handler()*,
– funkcje i makra asemblerowe związane z obsługą rdzenia, kontrolera przerwań NVIC i timera SysTick zostały zmienione i zamieszczone w pliku *misc.c*, *core_cm3.c* oraz *core_cm3.h*.

Ponadto zmienione zostały nazwy związane z przerwaniami i tak na przykład przerwanie EXTI0 nazywa się EXTI0_IRQn, w miejsce EXTI0_IRQChannel. Aktualizacji wymagają również pliki związane bezpośrednio z wykorzystywanym środowiskiem programistycznym, ustawienia projektu oraz przeorganizowanie struktury plików biblioteki.

List. 5. Konfiguracja timera SysTick z wykorzystaniem biblioteki *STM32F10x firmware library*

```
// SysTick bedzie taktowany z f =
// 72MHz/8 = 9MHz
SysTick_CLKSourceConfig(SysTick_
CLKSource_HCLK_Div8);

// Przerwanie ma byc co 1ms, f =
// 9MHz czyli liczy od 9000
SysTick_SetReload(9000);


// Odblokowanie przerwania od timera
// SysTick
SysTick_ITConfig(ENABLE);

// Wlaczenie timera
SysTick_CounterCmd(SysTick_Counter_
Enable);
```

List. 6. Konfiguracja timera SysTick z wykorzystaniem biblioteki *STM32F10x Standard Peripherals Library*

```
// SysTick bedzie taktowany
// z f = 72MHz/8 = 9MHz
SysTick_CLKSourceConfig(SysTick_
CLKSource_HCLK_Div8);

// Przerwanie ma byc co 1ms,f =
// 9MHz / 1000, czyli liczy od 9000
if (SysTick_Config(SysTick_Frequency
/ 1000))
{
        // W razie bledu petla
        // nieskonczona
        while (1);
}
```

Przykładowy fragment kodu, napisany z wykorzystaniem starszej biblioteki *STM32F10x firmware library* przedstawiono na **list. 5**, natomiast dla kontrastu na **list. 6** przedstawiono program napisany przy użyciu nowej biblioteki *Standard Peripherals Library*. Zadanie obydwu programów jest identyczne i polega na skonfigurowaniu do pracy systemowy timer SysTick.

Konfiguracja urządzeń peryferyjnych

Dla każdego urządzenia, czy jest to GPIO, kontroler przerwań, czy jakikolwiek inny element systemu, są stworzone odrębne typy danych. W przypadku portów wejścia/wyjścia nazywają się one: *GPIO_TypeDef*, oraz wykorzystywany do inicjalizacji typ *GPIO_InitTypeDef*. Dla programisty największe znaczenie ma typ inicjujący, ponieważ to właśnie zmienną tego typu jawnie tworzymy w pisanym kodzie.

Typ *GPIO_TypeDef* zapewnia dostęp do poszczególnych rejestrów mikrokontrolera i jest wykorzystywany przede wszystkim

List. 7. Wykorzystanie API – konfiguracja portu GPIOB

```
// Wyprowadzenia PB8, PB9 jako
// wyjscia push - pull
GPIO_InitStructure.GPIO_Pin = GPIO_
Pin_8 | GPIO_Pin_9;
GPIO_InitStructure.GPIO_Mode = GPIO_
Mode_Out_PP;
GPIO_InitStructure.GPIO_Speed =
GPIO_Speed_50MHz;

GPIO_Init(GPIOB, &GPIO_
InitStructure);
```

przez funkcje API, natomiast zmienna typu *GPIO_InitTypeDef* musi istnieć w każdej aplikacji wykorzystującej porty wejścia/wyjścia, ponieważ jest wykorzystywana do inicjalizowania i konfigurowania portów. Na **list. 7** jest przedstawiony kluczowy fragment kodu odpowiedzialny za konfigurację portu GPIOB.

Utworzona na początku zmienna *GPIO_InitStructure* jest, strukturą. Inicjowanie pinów, lub w szczególnym przypadku całego portu odbywa się w ten sposób, że wypełnia się poszczególne pola struktury, a następnie przekazuje się adres tak przygotowanej zmiennej do funkcji inicjującej.

W przedstawianej sytuacji w naszym kręgu zainteresowań leżą trzy pola struktury *GPIO_InitStructure*. W pierwszej kolejności ustalamy, które z pinów będą konfigurowane, następnie wybieramy żądany tryb pracy – w tym przypadku będzie to wyjście typu push-pull. Następnie ustalamy maksymalna prędkość, z jaką będą mogły pracować piny. Tak przygotowaną zmienną należy przekazać przez podanie jej adresu w argumencie do funkcji inicjującej *GPIO_Init()*. Drugim argumentem, jaki należy funkcji przekazać jest nazwa portu, do jakiego mają być zastosowane wybrane ustawienia.

Podsumowanie

Wykorzystywanie gotowych bibliotek jest bardzo przyjemne, ale nie zwalnia to programisty z obowiązku rozumienia, co właściwie (i w jaki sposób) poszczególne funkcje robią. Jest to bardzo ważne i zawsze należy o tym pamiętać. Z tego powodu, jeśli konstruktor chce w pełni panować nad wszystkimi elementami systemu, to i tak musi dokładnie zapoznać się z dokumentacją techniczną mikrokontrolera, a tym samym poznać jego architekturę.

Krzysztof Paprocki
paprocki.krzysztof@gmail.com