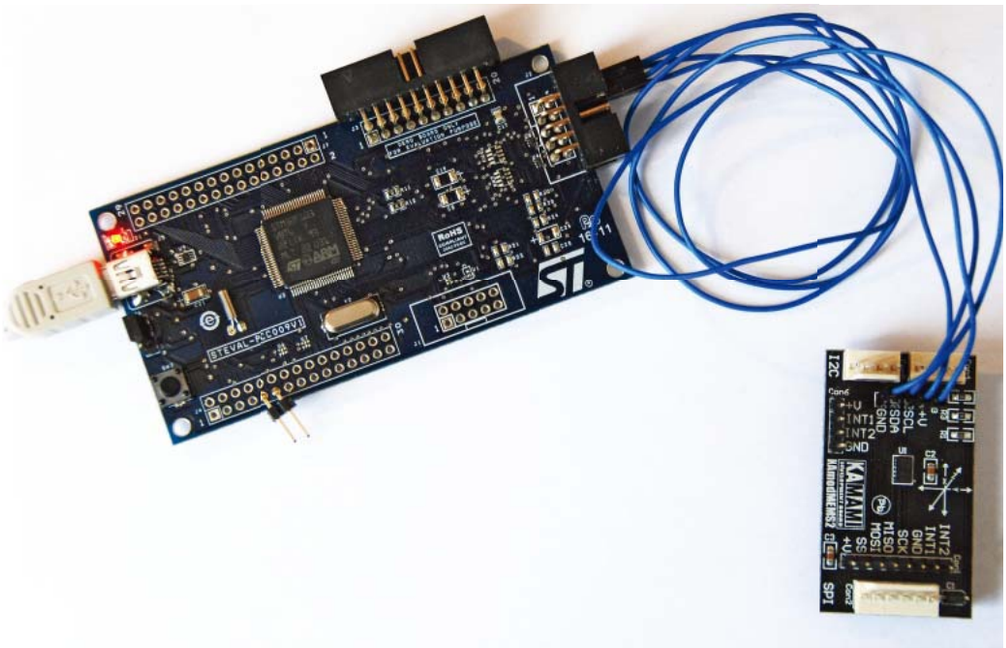
Zestaw STEVAL PCC009V1 i biblioteki Universal Dongle

W bogatej ofercie firmy ST Microelectronics można znaleźć dość ciekawy zestaw demonstracyjny o symbolu STEVAL PCC009V1. Jest on opisany jako „Universal USB to serial communication interface demonstration board based on the STM32”, czyli uniwersalny zestaw demonstracyjny komunikacji szeregowej dla USB oparty na STM32. W artykule pokazanych zostanie kilka funkcji tego zestawu oraz sposób jego wykorzystania do komunikacji pomiędzy komputerem PC a czujnikiem z interfejsami I²C i SPI. Ponadto, zostaną pokrótce przedstawione biblioteki SDK Universal Donlge pozwalające utworzyć własną aplikację z użyciem opisywanego zestawu.



Fotografia 1. Zestaw STEVAL PCC009V1 (po lewej) i moduł KAmodMEMS2 (po prawej)

Zestaw STEVAL PCC009V1

Jak wspomniano we wstępie, moduł opisany jest w ofercie STM jako *uniwersalny zestaw demonstracyjny komunikacji szeregowej dla USB oparty na STM32*. W praktyce można go jednak traktować dwojako. Albo jako zwykły zestaw uruchomieniowy z mikrokontrolerem STM32F103VB albo jako swego rodzaju „przejściówkę” z USB na interfejs szeregowy, ponieważ fabrycznie zestaw ma wgrany *firmware* spełniający taką właśnie funkcję. Na płytce PCC009V1 (**fotografia 1**), oprócz mikrokontrolera, przycisku *reset* i diody sygnalizującej załączenie zasilania, znajdziemy trzy złącza. Pierwsze z nich to złącze USB służące także do zasilania zestawu. Drugie złącze (oznaczone J3) to złącze interfejsu JTAG pozwalające na zaprogramowanie mikrokontrolera. Na trzecim złączu (J2) udostępniono linie interfejsów: I²C, SPI i SCI (*Serial Communication Interface*). Pod ostatnim oznaczeniem kryją się linie układu UART.

Tabela 1. Lista funkcji wyprowadzeń złącza J2

Pin	I²C	SPI	UART
1	SCL	GPIO 1	TX
2	SDA	GPIO 2	RX
3	GND		
4	V_{CON}		
5	GPIO 1	MISO	RTS
6	GPIO 2	SCK	CTS
7	GPIO 3	MOSI	GPIO 1
8	GPIO 4	NSS	GPIO 2
9	GPIO 5	GPIO 3	GPIO 3
10	GPIO 6	GPIO 4	GPIO 4

Zależnie od trybu pracy zestawu PCC009V1, na złączu J2 jest udostępniany jeden z ww. interfejsów, linia masy, zasilania oraz kilka linii GPIO ogólnego przeznaczenia. **Tabela 1** zawiera listę wyprowadzeń złącza J2 i opisy funkcji, które pełnią w poszczególnych trybach. Wystawiane na linii 4 napięcie V_{CON} zależy od ustawienia zworki J6 i może to być 3,3 V lub 5 V.

Oprócz złącz J2 i J3, na płytce znajdują się pola dla złącz J1, J4, J7 i J8. Nie mają one jednak przylutowanych gniazd. Na złączu J1 wyprowadzono m.in. wejścia przetworników A/C. Na złączu J4 – linie portów GPIO B i GPIO E. Na złączu J7 – linie portów GPIO C i GPIO D, a na J8 – linie odpowiedzialne za obsługę kart SD. Pozwala to na wykorzystanie większości możliwości mikrokontrolera STM32F103.

W skład zestawu PCC009V1 – oprócz płytki demonstracyjnej – wchodzi też płyta CD, na której można znaleźć sterowniki i dokumentację do zestawu, program *Universal Dongle GUI* oraz biblioteki SDK *Universal Dongle*. Program ma własny interfejs graficzny i pozwala na komunikowanie się z układami wykorzystującymi interfejsy I²C, SPI lub SCI za pośrednictwem PCC009V1. Z kolei biblioteki pozwalają na zaimplementowanie tych funkcji we własnej aplikacji. Dostarczone na płycie CD sterowniki działają poprawnie w 32-bitowych wersjach systemu Windows. Dla wersji 64-bitowej może być konieczne uruchomienie maszyny wirtualnej. Po dołączeniu zestawu do komputera, komunikację z nim można nawiązać albo przez wspomniany program *Universal Dongle GUI*, albo za pośrednictwem funkcji z biblioteki SDK. Należy jednak w tym miejscu ostrzec, że zestawu podłączonego do komputera nie należy odłączać, zanim połączenie nie zostanie poprawnie zamknięte. W przeciwnym wypadku wyjęcie przewodu USB skutkuje zwykle całkowitym zawieszeniem się komputera (tzw. *bluescreen*).

Akcelerometr ST LIS35DE

W celu przetestowania i zademonstrowania działania zestawu PCC009V1 użyto modułu KAmodMEMS2 z oferty firmy KAMAMI. Jest to moduł oparty na układzie ST LIS35DE będącym 3-osiowym akcelerome-

trem MEMS o zakresie ±2 lub ±8g. Pomiary wykonywane są z rozdzielczością 8-bitową, z częstotliwością 100 lub 400 Hz. Ponadto, akcelerometr ma m.in. funkcje wykrywania pojedynczych i podwójnych kliknięć oraz wykrywania swobodnego spadku. Układ ten został wybrany ze względu na łatwą obsługę i jednoczesne wyposażenie w interfejsy I²C i SPI.

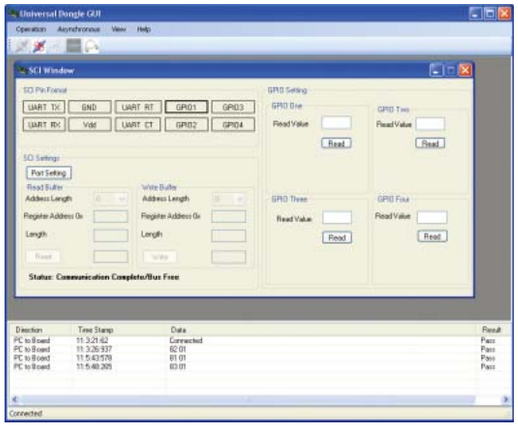
Konfiguracja układu oraz pobieranie z niego danych polegają na zapisie i odczycie odpowiednich rejestrów wewnętrznych. Z punktu widzenia obsługi podstawowej funkcji akcelerometru, czyli pomiaru przyśpieszeń, najważniejsze są rejestry:

- CTRL_REG1 (adres 0x20) – wybór częstotliwości i zakresu pomiaru, włączenie akcelerometru i pomiaru w poszczególnych osiach,
- CTRL_REG2 (0x21) – m.in. przywrócenie fabrycznych ustawień wszystkich rejestrów,
- OUT_X (0x29) – wynik pomiaru przyśpieszenia w osi X,
- OUT_Y (0x2B) – wynik pomiaru przyśpieszenia w osi Y,
- OUT_Z (0x2D) – wynik pomiaru przyśpieszenia w osi Z.

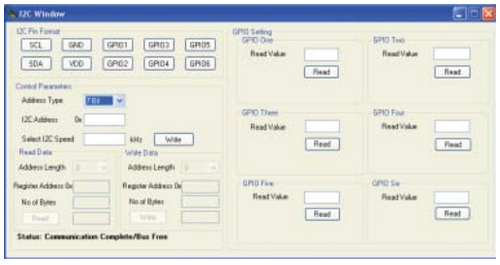
Ponadto, układ potrafi wykonywać automatyczną inkrementację adresów przy realizacji kolejnych odczytów, co pozwala m.in. na odczytanie wyników dla wszystkich osi bez konieczności osobnego przesyłania adresów kolejnych rejestrów. Należy jednak w tym wypadku pamiętać, że wyniki znajdują się w co drugim – poczynając od adresu 0x29 – rejestrze i dla pobrania wyników z trzech osi jest konieczne odczytanie pięciu kolejnych rejestrów, z których dwa są puste.

Program Universal Dongle GUI

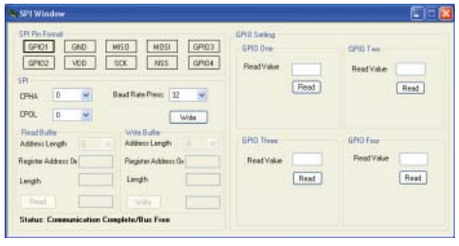
Jak wspomniano wcześniej, program *Universal Dongle GUI* pozwala na komunikowanie się z układami mającymi interfejsy I²C, SPI lub SCI za pośrednictwem PCC009V1. Po uruchomieniu programu należy wybrać z menu *Operation* polecenie *Connect*, co pozwala na nawiązanie połączenia z płytką PCC009V1. Następnie, ponownie z menu *Operation*, należy wybrać rodzaj interfejsu: *Synchronous* (dla SPI i I²C) lub *Asynchronous* (dla UART). W menu pojawi się wówczas dodatkowa opcja, zgodna z dokonanym wyborem, w której będzie można wybrać odpowiedni interfejs komunikacyjny. Po wybraniu interfejsu pojawi się okno z jego ustawieniami pozwalające na skonfigurowanie parametrów komunikacji z konkretnym urządzeniem zewnętrznym (np. czujnikiem), a następnie wysłanie i odebranie do/z niego danych. Należy oczywiście pamiętać, aby urządzenie dołączyć do złącza J2 zgodnie ze wskazaniami na ekranie. Trzeba zwrócić przy tym uwagę, że na ekranie pin numer 1 znajduje się po lewej, natomiast patrząc od przodu na złącze na płytce, jest on po prawej stronie. Dodatkowo warto zaznaczyć, że opisy w programie i w dokumentacji zestawu PCC009V1 są zgodne z rzeczywistymi funkcjami pinów złącza J2, natomiast ich opis w plikach pomocy do programu jest niestety błędny. Na **rysunkach 2**, **3** i **4** przedstawiono okna komunikacji dla poszczególnych obsługiwanych interfejsów. Przykładowy sposób wykorzystania tych okien zostanie omówiony na przykładzie akcelerometru LIS35DE.



Rysunek 2. Okno programu *Universal Dongle GUI* z oknem interfejsu SCI (UART)



Rysunek 3. Okno interfejsu I²C



Rysunek 4. Okno interfejsu SPI

Universal Dongle GUI: komunikacja z LIS35 – I²C

W wypadku użycia interfejsu I²C układ LIS35 ma adres 0x3A. Pierwszym krokiem, który trzeba wykonać w celu nawiązania komunikacji, jest skonfigurowanie parametrów transmisji. W oknie transmisji I²C (rys. 3) należy wybrać *Address type* – 7bit, w polu *I2C Address* wpisać 0x3A, a w polu *Select I2C speed* podać 100 kHz. Ustawienia wysyłane są do zestawu PCC009V1 po wciśnięciu przycisku *Write*. Kolejnym etapem jest skonfigurowanie akcelerometru. W tym celu w panelu *Write Data* należy wybrać *Address length* równy 1 (jest to długość adresów rejestrów wewnętrznych, w tym przypadku 1 bajt), a następnie w polu *Register Address* wpisać 0x21 oraz w polu *Number of bytes* wpisać 1 (wysłany zostanie jeden bajt danych). W ostatnim polu, obok przycisku *Write* należy wpisać w postaci dziesiętnej (a nie szesnastkowej) bajt danych do wysłania. W naszym przypadku niech będzie to 64, co spowoduje przywrócenie w układzie ustawień fabrycznych. Po jej wpisaniu należy kliknąć przycisk *Write*. W podobny sposób należy następnie wysłać do układu pod adres 0x20 wartość 71. Spowoduje ona włącznie układu, wybranie częstotliwości przetwarzania 100 kHz, wybranie zakresu ±2 g i włączenie pomiaru w trzech osiach akcelerometru. W przypadku, gdybyśmy chcieli do układu wysłać więcej niż 1 bajt, należy oczywiście wpisać ich liczbę w polu *Number of bytes*. Wówczas po wciśnięciu przycisku *Write* program poprosi o wskazanie pliku z kolejnymi wartościami zapisanego w formacie *.hex*.

Po skonfigurowaniu akcelerometru można rozpocząć odczyt danych. Np. w celu odczytania przyspieszenia w osi X należy w panelu *Read Data* wybrać *Address length* równy 1, w polu *Register Address* wpisać 0x29, a w polu *Number of bytes* wpisać 1. Po kliknięciu przycisku *Read* w polu obok niego pojawi się odczytana z akcelerometru wartość przyśpieszenia w osi X. Należy przy tym pamiętać, że jest to wartość „surowa", nieprzeliczona na jednostki g.

Universal Dongle GUI: komunikacja z LIS35 – SPI

Nawiązanie transmisji z układem LIS35 za pośrednictwem interfejsu SPI nastręczyło autorom nieco trudności. Ostatecznie okazało się, że w czasie komunikacji, mikrokontroler na płytce PCC009V1 zmienia stan linii nSS nie tylko na początku i końcu transmisji, ale także, na chwilę, po każdym wysłanym bajcie. Tymczasem akcelerometr traktuje każdą aktywację linii nSS jak początek kolejnej transmisji, więc w wypadku wysyłania danych z chwilową zmianą stanu nSS po każdym bajcie, drugi z bajtów akcelerometr zamiast traktować jak dane przynależące do przed chwilą przesłanego adresu, traktował jak początek nowej komendy (adres kolejnego rejestru). W efekcie, nigdy nie odczytywał danych konfiguracyjnych ani nie wysyłał danych ze swoich rejestrów. Program *Universal Dongle GUI* nie daje niestety możliwości zmiany tego zachowania. W związku z tym, jako rozwiązanie doraźne, odłączono od mo-

dułu KAmod przewód linii nSS i sterowano jej stanem ręcznie, poprzez zwieranie jej do masy na czas wysyłki kolejnych, 2-bajtowych komend i dołączanie do zasilania na czas wstrzymania transmisji. Okazało się to być skutecznym obejściem problemu i pozwoliło ostatecznie nawiązać transmisję z akcelerometrem za pośrednictwem SPI z poziomu programu *Universal Dongle GUI*.

W celu skonfigurowania transmisji w oknie transmisji SPI programu *Universal Dongle GUI* (rysunek 4) należy wybrać wartości *CPHA* i *CPOL*. Dotyczą one ustawienia sposobu generowania zegara taktującego na linii SCK. W przypadku akcelerometru LIS35 oczekuje on, że w stanie wstrzymania transmisji sygnał zegarowy będzie miał poziom wysoki (CPOL=1), a w czasie transmisji dane odczytywane będą na drugim, narastającym zboczu sygnału (CPHA=1). W polu *Baud rate* wybiera się prędkość transmisji, jednak wybór polega nie na podaniu jej wprost, a na określeniu dzielnika częstotliwości, który dzieli sygnał zegarowy taktujący podukład kontrolera SPI w mikrokontrolerze STM32F103. Ustawienia wysyłane są do zestawu PCC009V1 po wciśnięciu przycisku *Write*. Kolejnym etapem jest skonfigurowanie akcelerometru. Podobnie jak dla przypadku transmisji I²C opisanego wcześniej, należy najpierw wysłać wartość 64 pod adres 0x21, a następnie wartość 71 pod adres 0x20. Trzeba przy tym pamiętać o ręcznym ustawianiu niskiego stanu linii nSS na czas transmisji.

O ile w wypadku interfejsu I²C kierunek (zapis/odczyt) transmisji danych ustawiany jest najmłodszym bitem adresu urządzenia (co jest wykonywane automatycznie przez kontroler I²C w mikrokontrolerze STM32F103), o tyle w wypadku SPI kierunek należy określić ustawiając najstarszy bit adresu. W związku z tym, aby na przykład odczytać zawartość rejestru 0x29, należy jego adres podać jako 0xA9. Oprócz różnicy w sposobie podawania adresu, odczyt w oknie SPI programu *Universal Dongle GUI* jest wykonywany tak samo, jak w wypadku okna I²C.

Biblioteki SDK Universal Dongle

Gotowy interfejs GUI pozwala na nawiązanie komunikacji z testowanymi układami, jednak jego możliwości są ograniczone. Aby w pełni wykorzystać możliwości oferowane przez zestaw STEVAL PCC009V1, należy skorzystać z dostarczanych razem z nim na płycie CD bibliotek *SDK Universal Dongle*. Po instalacji *Universal Dongle GUI*, w podfolderze */Support*, można znaleźć 13 plików tej biblioteki. Głównymi z nich jest 6 plików .dll. Pliki I2C_SDK.DLL, SPI_SDK.DLL i SCI_SDK.DLL zawierają funkcje obsługi poszczególnych interfej-

Listing 1. Kod metody *buttonPolacz_Click( )*

```
private: System::Void buttonPolacz_Click(System::Object^  sender,
                                         System::EventArgs^  e) {
  int error=0;
  //wybór interfejsu I2C lub SPI
  if (radioButtonI2C->Checked) {interfejs=LIS35_Interface_I2C;}
  else
    if (radioButtonSPI->Checked) {interfejs=LIS35_Interface_SPI;}
    else {error=1; interfejs=0;}
  if (stanPolaczenia==false){ //Jeśli nie połączony - połącz
    if (error==0){
      if (interfejs==LIS35_Interface_I2C) {
        error=LIS35_I2C_Init(LIS35_I2C_ChipAddr);}
      if (interfejs==LIS35_Interface_SPI) {
        error=LIS35_SPI_Init();}
    }

    if (error==0){ //inicjalizacja i konfiguracja połączenia
      labelStanPol->Text=L"Połączono";
      stanPolaczenia=true;
      buttonPolacz->Text=L"Rozłącz";
      radioButtonI2C->Enabled=false;
      radioButtonSPI->Enabled=false;
      comboBoxZakres->Enabled=false;

      //podstawowa konfiguracja układu LIS35
      if (interfejs==LIS35_Interface_I2C) {
        error=LIS35_I2C_Config(comboBoxZakres->SelectedIndex);}
      if (interfejs==LIS35_Interface_SPI) {
        error=LIS35_SPI_Config(comboBoxZakres->SelectedIndex);}
      if (error==0) {
        //włączenie timera odczytu danych z układu
        timer1->Enabled=true;
      } else {
        labelStanPol->Text=L"Błąd konfiguracji układu LIS35";
      }

    } else {
      labelStanPol->Text=L"Błąd połączenia";
    }
  } else {                  //Rozłącz
    timer1->Enabled=false;
    if (interfejs==LIS35_Interface_I2C) {I2C_Disconnect();}
    if (interfejs==LIS35_Interface_SPI) {SPI_Disconnect();}
    stanPolaczenia=false;
    radioButtonI2C->Enabled=true;
    radioButtonSPI->Enabled=true;
    comboBoxZakres->Enabled=true;
    buttonPolacz->Text=L"Połącz";
    labelStanPol->Text=L"Nie podłączony";
  }
}
```

Listing 2. Kod metody *timer1_Tick( )*

```
private: System::Void timer1_Tick(System::Object^  sender, System::EventArgs^  e) {
  static int  korekty[3]={0};  //korekty składowej stałej akcelerometru
  static int  czulosci[3]={0}; //czułości akcelerometru
  signed char bufor[5] ={0};   //bufor danych surowych
  float       buforK[5]={0};   //bufor danych skorygowanych,
                               //po przeliczeniu na przyspieszenia
  int         error=0;

  switch (comboBoxZakres->SelectedIndex){
    case 0:{ //korekty dla zakresu +-2g
             //korekta = -(wartość odczytana przy poł. poziomym)
             //czulosc=9.8067g / ((wartość przy poł. „+1g” +
             //                   - wartość przy poł. „-1g”) / 2) * 10000
             korekty[0] =-2;   korekty[1] =3;    korekty[2] =4;
             czulosci[0]=1816; czulosci[1]=1767; czulosci[2]=1767;
           } break;
    case 1:{ //korekty dla zakresu +-8g
             korekty[0] =-1;   korekty[1] =0;    korekty[2] =0;
             czulosci[0]=7005; czulosci[1]=6763; czulosci[2]=7544;
           } break;
  }

  if (interfejs==LIS35_Interface_I2C) {
    error=LIS35_I2C_ReadRegisters(LIS35_REG_OUTX,
                reinterpret_cast <unsigned char *>(bufor), 5);
  }
  if (interfejs==LIS35_Interface_SPI) {
    error=LIS35_SPI_ReadRegisters(LIS35_REG_OUTX,
                reinterpret_cast <unsigned char *>(bufor), 5);
  }
  if (error!=0) {labelStanPol->Text=L"Błąd transmsji! ("+ error.ToString()+")";}
  else {
    //x, y, z
    buforK[0]=((static_cast<float>(bufor[0])+korekty[0])*czulosci[0])/98067;
    buforK[2]=((static_cast<float>(bufor[2])+korekty[1])*czulosci[1])/98067;
    buforK[4]=((static_cast<float>(bufor[4])+korekty[2])*czulosci[2])/98067;

    labelXs->Text=bufor[0].ToString();
    labelYs->Text=bufor[2].ToString();
    labelZs->Text=bufor[4].ToString();

    labelXk->Text=buforK[0].ToString("0.00");
    labelYk->Text=buforK[2].ToString("0.00");
    labelZk->Text=buforK[4].ToString("0.00");
  }
}
```

Rysunek 5. Okno programu *LIS35UnivDongle Demo*

sów. Funkcje te są łatwo dostępne dla programistów, ponieważ do każdego z plików dołączone są odpowiednie pliki *.lib* oraz pliki nagłówkowe *.h*. Pozostałe 3 biblioteki DLL zawierają funkcje obsługi USB i warstwy sprzętowej zestawu uruchomieniowego i są wywoływane wewnętrznie przez funkcje z bibliotek SDK.

Zawartość plików *.dll* związanych z obsługą poszczególnych interfejsów jest niemal identyczna, tzn. zawierają one funkcje realizujące te same zadania, a różniące się tylko przedrostkami funkcji (np. *I2C_Connect*( ), *SPI_Connect*( ), *SCI_Connect*( )) i, w niektórych przypadkach, parametrami wywołania.

W celu nawiązania połączenia z wykorzystaniem wybranego interfejsu należy najpierw wywołać funkcję *XXX_Connect*( ) a następnie *XXX_InitXXXcommunication*( ), gdzie za *XXX* należy wpisać nazwę odpowiedniego interfejsu (np. *I2C*). Obie funkcje nie mają parametrów wywołania, a w przypadku niepowodzenia zwracają kod błędu. Kolejnym krokiem jest wywołanie funkcji *XXX_WriteXXXSettings*( ), która konfiguruje parametry połączenia z układem zewnętrznym, np. akcelerometrem LIS35DE. Funkcja przyjmuje trzy parametry, kolejno: tablicę z ustawieniami konfiguracyjnymi, długość danych do przesłania przez interfejs (w opisywanym przypadku 0) i liczbę bajtów, które mają pozostać niewysłane (również 0, co oznacza, że wysyłane są wszystkie dane). Zawartość wspomnianej tablicy stanowią pola typu *unsigned char* z ustawieniami konfiguracyjnymi. Znaczenie poszczególnych pól zależy od wybranego interfejsu. W przypadku interfejsu SPI są to kolejno (tablica 6-cio bajtowa): CPHA, CPOL, *Daisy Chain* (0 oznacza brak łańcuszka), liczba urządzeń w łańcuchu, starszy bajt wartości dzielnika częstotliwości sygnału taktującego kontroler SPI w mikrokontrolerze i młodszy bajt tegoż dzielnika. Dla interfejsu I²C są to kolejno (5 bajtów): tryb adresowania (0 oznacza tryb 7-bit.), dwa najstarsze bity adresu w 10-bit. trybie adresowania (w trybie 7-bit są one zerowe), adres urządzenia (lub młodsza część adresu 10-bit.), starszy bajt prędkości transmisji podanej w kHz, młodszy bajt prędkości transmisji. W przypadku interfejsu SCI (UART) są to kolejno (7 bajtów): 3 bajty przepływności (*baud rate*) poczynając od najbardziej znaczącego, liczba bitów danych, rodzaj parzystości (0=*Even*, 1=*Odd*, 3=brak, 4=*Mark*, 5=*Space*), liczba bitów stopu, kontrola przepływu. Do zakończenia połączenia służy funkcja *XXX_Disconnect*( ).

Po nawiązaniu i skonfigurowania połączenia można zacząć wysyłać i odbierać dane do/z rejestrów. Do zapisu danych służy funkcja *XXX_WriteRegisterAddress*(*BYTE* bCommand, BYTE* bRegisterData, UINT uiLengthOfData, UINT uiResidueData*) gdzie:

bCommand – 7-bajtowa tablica zawierająca komendę sterującą,

bRegisterData – wskaźnik do zmiennej zawierającej dane do zapisania do rejestru,

uiLengthOfData – rozmiar zmiennej *bRegisterData*,

uiResidueData – liczba bajtów, które mają zostać pominięte przy zapisie (zwykle 0).

Kolejne pola tablicy *bCommand* mają następujące znaczenie: długość adresu (w bajtach), adres (łącznie 4 bajty podawane od najbardziej do najmniej znaczącego) i rozmiar danych do zapisu (2 bajty poczynając od bardziej znaczącego). Wartość ostatniego z parametrów powinna zgadzać sie z wartością zmiennej *uiLengthOfData*, jednak testy wykazują, że jest ona prawdopodobnie ignorowana i o rozmiarze danych decyduje tylko zmienna *uiLengthOfData*.

Do odczytu danych służy funkcja *XXX_ReadRegisterAddress*(*BYTE* bCommand, BYTE* bRegisterData, UINT uiLengthOfData, UINT bResidueData*). Znaczenie poszczególnych parametrów jest takie samo jak w poprzedniej funkcji, z tym, że dane odczytane z układu zapisywane są na zmienną wskazywaną przez *bRegisterData*.

Jak wspomniano, część linii złącza J2 może pracować jako zwykłe linie GPIO. Do ich konfigurowania służy funkcja *XXX_WriteGPIOSetting*(*BYTE* bGPIOSettingData, UINT iLengthOfData, UINT uiResidueData*) gdzie:

bGPIOSettingDat – 4-bajtowa tablica zawierająca ustawienia wybranej linii portu,

uiLengthOfData – rozmiar danych (w tym przypadku 0),

uiResidueData – liczba bajtów, które mają zostać pominięte przy zapisie (0).

Znaczenie kolejnych bajtów tablicy *bGPIOSettingDat* jest następujące: numer linii GPIO, tryb pracy (0 – wejście, 1 – wyście), funkcja (0 – zwykłe wejście/wyjście, 1 – wejście sygnału przerwania lub wyjście PWM), aktywne zbocze sygnału przerwania (0 – opadające, 1 – narastające). Ustawienie wybranej linii GPIO w zadany stan realizuje funkcja *XXX_WriteXXXGPIO*(*BYTE bGPIONumber, BYTE* bGPIOData, UINT uiLengthOfData, UINT uiResidueData*) gdzie:

bGPIONumber – numer linii GPIO,

bGPIOData – stan linii (0 lub 1),

uiLengthOfData – rozmiar danych (w tym wypadku 1),

uiResidueData – liczba bajtów, które mają zostać pominięte przy zapisie (0).

Do odczytu stanu linii służy funkcja *XXX_ReadXXXGPIO*(*BYTE bGPIONumber, BYTE* bGPIOData, UINT uiLengthOfData, UINT uiResidueData*), której parametry wywołania są takie same jak dla funkcji *XXX_WriteXXXGPIO*( ).

Przykład wykorzystania bibliotek SDK *Universal Dongle*

W celu zaprezentowania praktycznego wykorzystania bibliotek SDK *Universal Dongle* napisano w środowisku *Microsoft Visual C++ 2010 Express* program *LIS35UnivDongle Demo*, którego okno pokazano na **rysunku 5**. Program pozwala na wybranie interfejsu połączenia z akcelerometrem oraz zakresu pomiaru. Po połączeniu rozpoczyna cykliczne wyświetlanie surowych danych pobranych z układu LIS35 oraz obliczonych na ich podstawie wartości przyśpieszeń.

Na potrzeby programu napisano także dwie biblioteki obsługi akcelerometru LIS35DE. Obie zawierają w zasadzie takie same funkcje, upraszczające korzystanie z bibliotek SDK. Są to:

LIS35_XXX_Init( ) – inicjalizacja interfejsu I²C lub SPI,

LIS35_XXX_Config( ) – podstawowa konfiguracja układu LIS35,

LIS35_XXX_WriteRegister( ) – zapis wskazanego rejestru układu LIS35,

LIS35_XXX_ReadRegister( ) – odczyt wskazanego rejestru układu LIS35,

LIS35_XXX_ReadRegisters( ) – odczyt kilku kolejnych rejestrów układu LIS35 z wykorzystaniem autoinkrementacji,

LIS35_XXX_Status() – odczyt stanu interfejsu.

Parametry wywołania funkcji są identyczne dla obu wersji (SPI i I²C) z wyjątkiem funkcji *LIS35_I2C_Init*( ), która jako parametr przyjmuje adres układu. Dodatkowo, w bibliotece przeznaczonej do komunikacji z akcelerometrem za pośrednictwem SPI znajduje się funkcja *LIS35_SPI_SS_State*( ) pozwalająca sterować stanem linii nSS.

Oprócz bibliotek dedykowanych dla poszczególnych interfejsów przygotowano także plik nagłówkowy *LIS35_common.h*, w którym zawarto stałe opisujące m.in. adresy rejestrów układu LIS35DE oraz predefiniowane wartości kilku kluczowych bitów w wybranych rejestrach.

Aby móc we własnym programie skorzystać z bibliotek SDK *Universal Dongle* należy dodać pliki *I2C_SDK.LIB* i *.H* oraz *SPI_SDK.LIB* i *.H* do projektu *MSVisual C++*. Następnie, we właściwościach tworzonego projektu w polu *ConfigurationProperties > General > Common Language Runtime Support* konieczna jest zmiana ustawienia na *Common Language Runtime Support (/clr)*. Pamiętać należy o jej wprowadzeniu dla wszystkich rodzajów kompilacji (standardowo są to *Debug* i *Release*). Ponadto, ze względu na inne nazewnictwo typów zmiennych, w obu dodanych plikach nagłówkowych należy dopisać odpowiednie definicje typów, tj.:

```
#define BYTE unsigned char
#define UINT unsigned int
```

W czasie działania programu demonstracyjnego, po kliknięciu przez użytkownika

przycisku *Połącz*, jest wywołana metoda *buttonPolacz_Click*() (**listing 1**). Na jej początku następuje sprawdzenie, który z interfejsów został wybrany, a następnie realizowane jest połączenie z zestawem PCC009V1 i inicjalizacja wybranego interfejsu. Jeśli się ona powiedzie, przeprowadzana jest konfiguracja akcelerometru. Po jej pomyślnym zakończeniu aktywowany jest *timer*, który co około 1/10 sekundy wywołuje metodę *timer1_Tick*() (**listing 2**). Metoda ta odczytuje zawartość rejestrów akcelerometru zawierających dane o zmierzonych w poszczególnych osiach przyśpieszeniach, a następnie przelicza je na jednostki g z uwzględnieniem korekty składowej stałej oraz czułości. Wartości korekt należy wyznaczyć samemu dla konkretnego egzemplarza układu LIS35. Korekta składowej stałej polega na umieszczeniu danej osi w pozycji poziomej (aby nie działało na nią przyśpieszenie ziemskie) i odczytaniu wskazywanej wartości. Wówczas *korekta* = *-wartość odczytana*. W przypadku kalibracji czułości należy daną oś akcelerometru umieścić tak by działało na nią przyspieszenie ziemskie *w* najpierw w kierunku „+”, a następnie „-” po czym obliczyć czułość wg wzoru: *czulosc* = 98067 / ((w_{+1g} - w_{-1g}) / 2). Znając wartości korekt, skalibrowaną wartość przyspieszenia można obliczyć wg zależności: *przyspieszenie* = (*odczyt* + *korekta*) • *czułość* / 98067.

Jeśli połączenie z akcelerometrem jest aktywne, kliknięcie przez użytkownika przycisku *Rozłącz* (jest to ten sam przycisk, co *Połącz*) powoduje zatrzymanie *timera* i wywołanie funkcji *XXX_Disconnect*( ), która przerywa połączenie z modułem PCC009V1.

W celu realizacji opisanych zadań, metody *buttonPolacz_Click*( ) i *timer1_Tick*( ) wywołują funkcje z przygotowanych bibliotek obsługi układu LIS35. Na przykładzie biblioteki w wersji dla interfejsu SPI przedstawione zostaną trzy z nich. Funkcja *LIS35_SPI_Config*(*Range*) (**listing 3**) najpierw konfiguruje linię GPIO 1 do pracy jako linia wyjściowa, co jest potrzebne, aby zrealizować sterowanie linią nSS. Następnie wysyła do układu LIS35 polecenie wyzerowania ustawień konfiguracyjnych, po czym zapisuje nowe ustawienia (m.in. aktywuje pomiar w poszczególnych osiach układu oraz wybiera zakres pomiarowy w zależności od wartości zmiennej *Range*. Po zapisaniu ustawień funkcja odczytuje je z układu. Jeśli odczytana wartość jest taka sama jak zapisywana oznacza to, że komunikacja z układem działa poprawnie i przyjął on nowe ustawienia. Do zapisu wartości do rejestru wykorzystywana jest funkcja *LIS35_SPI_WriteRegister*(*RegisterAddress, RegisterData*) (**listing 4**), która jako parametry przyjmuje adres rejestru i wartość, którą należy do niego zapisać. Wewnętrznie funkcja steruje stanem linii nSS oraz wywołuje, w celu realizacji zapisu danych, funkcję *SPI_WriteRegisterAddress*( ) z biblioteki SDK *Universal Dongle*. Na **listingu 5** przedstawiono z kolei funkcję *LIS35_SPI_ReadRegisters*(*RegisterAddress, RegisterData, NoOfBytes*), która jako parametry przyjmuje adres pierwszego z rejestrów, które mają być odczytywane, wskaźnik do zmiennej (tablicy), w której mają być umieszczone odczytane wartości oraz liczbę bajtów do odczytu. Wewnętrzne funkcja steruje stanem linii nSS oraz wywołuje, w celu realizacji odczytu danych, funkcję *SPI_ReadRegisterAddress*( ) z biblioteki SDK *Universal Dongle*. Jednocześnie funkcja sama dba o odpowiednie uzupełnienie adresu o bity kierunku (odczyt) i autoinkrementacji adresów.

Przedstawione listingi pokazują sposób wykorzystania najważniejszych funkcji z bibliotek SDK oraz bibliotek obsługi układu LIS35 przygotowanych na potrzeby niniejszego artykułu.

Podsumowanie

Przedstawiony zestaw STEVAL PCC009V1 może być przydatny nie tylko jako zwykły zestaw demonstracyjny czy uruchomieniowy, ale także, dzięki bibliotekom SDK *Universal Dongle*, może służyć np. do testowania czy też szybkiego rozpoznania możliwości różnorodnych czujników i innych układów korzystających z interfejsów SPI, I²C bądź UART. Ponadto, wspomniane biblioteki mogą być również wykorzystane do komunikacji z kilkoma innymi modułami demonstracyjnymi STEVAL od ST Microelectronics.

Autorzy składają podziękowania firmie ST Microelectronics za wyrażenie zgody na opublikowania kodu programu wykorzystującego biblioteki SDK *Universal Dongle*

Marek Galewski
Politechnika Gdańska
marg@mech.pg.gda.pl
Piotr Reddig
piotrreddig@gmail.com

Listing 3. Kod funkcji *LIS35_SPI_Config( )*

```
int LIS35_SPI_Config(int Range)
{
  unsigned char RegVal, LIS35Settings;
  unsigned char bGPIOSettingData[3];
  bGPIOSettingData[0] = 1; //GPIO1
  bGPIOSettingData[1] = 1; //output
  bGPIOSettingData[2] = 0; //normal output
  bGPIOSettingData[3] = 0;
  //konfiguracja GPIO1 dla potrzeb linii nSS
  SPI_WriteGPIOSetting(bGPIOSettingData,0,0);
  //reset ustawień LIS35
  LIS35_SPI_WriteRegister(LIS35_REG_CR2, LIS35_REG_CR2_BOOT);
  //Zapis ustawień - aktywacja osi
  LIS35Settings = LIS35_REG_CR1_XEN | LIS35_REG_CR1_YEN | LIS35_REG_CR1_ZEN |
                  LIS35_REG_CR1_ACTIVE;
  //Wybór pełnego zakresu pomiaru
  if (Range==1) {LIS35Settings = LIS35Settings | LIS35_REG_CR1_FULL_SCALE;}
  LIS35_SPI_WriteRegister(LIS35_REG_CR1, LIS35Settings);
  //Odczyt konfiguracji, jeśli taka sama jak zapisano LIS35 działa poprawnie
  LIS35_SPI_ReadRegister(LIS35_REG_CR1, &RegVal);
  if (RegVal == LIS35Settings) return LIS35_OK;
  return LIS35_ERROR;
}
```

Listing 4. Kod funkcji *LIS35_SPI_WriteRegister( )*

```
int LIS35_SPI_WriteRegister(unsigned char RegisterAddress, unsigned char
RegisterData)
{
  unsigned char bCommand[7];
  bCommand[0] = 1; //długość adresu rejestru
  bCommand[1] = 0; //kolejne bajty adresu
  bCommand[2] = 0;
  bCommand[3] = 0;
  bCommand[4] = RegisterAddress;
  bCommand[5] = 0; //rozmiar danych
  bCommand[6] = 0x01;
  int bLengthOfData = 1;
  int error=0;

  LIS35_SPI_SS_State(LIS35_SPI_SS_ENABLE);
  error=SPI_WriteRegisterAddress(bCommand, &RegisterData, bLengthOfData, 0);
  LIS35_SPI_SS_State(LIS35_SPI_SS_DISABLE);
  return error;
}
```

Listing 5. Kod funkcji *LIS35_SPI_ReadRegisters( )*

```
int LIS35_SPI_ReadRegisters(unsigned char RegisterAddress, unsigned char*
RegisterData, unsigned char NoOfBytes)
{
  unsigned char bCommand[7];
  bCommand[0] = 1; //długość adresu rejestru
  bCommand[1] = 0; //kolejne bajty adresu
  bCommand[2] = 0;
  bCommand[3] = 0;
  bCommand[4] = RegisterAddress|LIS35_SPI_READ|LIS35_ADDR_INC_SPI;
  bCommand[5] = 0; //rozmiar danych
  bCommand[6] = NoOfBytes;
  int bLengthOfData = NoOfBytes;
  int error=0;

  LIS35_SPI_SS_State(LIS35_SPI_SS_ENABLE);
  error=SPI_ReadRegisterAddress(bCommand, RegisterData, bLengthOfData, 0);
  LIS35_SPI_SS_State(LIS35_SPI_SS_DISABLE);
  return error;
}
```