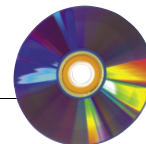
Poprzednie odcinki kursu na CD >>



Moduły GSM w praktyce (3)

Pierwsze kroki w środowisku OpenAT

Telefonia GSM w układach automatyki i telemetrii już od kilkunastu lat jest powszechnie wykorzystywana na szeroką skalę. Na łamach Elektroniki Praktycznej wielokrotnie publikowano projekty wykorzystujące do komunikacji telefon lub moduł GSM. We wszystkich spotykanych do tej pory rozwiązaniach rolę układu sterującego najczęściej pełnił niezależny mikrokontroler komunikujący się z telefonem/modułem GSM przez port szeregowy, wykorzystując odpowiednie komendy AT.

Po zapoznaniu się z wymaganiami stawianymi przez środowisko Open AT zaprezentujemy teraz bardzo prosty program, którego zadaniem jest miganie diodami LED znajdującymi się na płytce ewaluacyjnej. Aby nie trzeba było lutować dodatkowych diod do złącza modułu, postanowiono wykorzystać diody LED w założeniu służące do sygnalizacji stanu linii portu szeregowego UART1 modułu Q2686. W przypadku, gdy linie RXD i CTS portu szeregowego są skonfigurowane jako porty IO, możemy sterować nimi bezpośrednio za pomocą rozkazów sterujących liniami IO. Działanie tego programu jest bardzo proste i polega na naprzemiennym zaświecaniu i wyłączaniu diod RXD i CTS, które są dołączone odpowiednio do linii GPIO15 oraz GPIO16. Częstotliwość migania diod będzie równa 2 razy na sekundę. Kod programu przestawiono na **list. 2**.

Działanie programu rozpoczyna się od funkcji *adl_main*, w której za pomocą funkcji *adl_ioSubscribe* są inicjalizowane porty IO. Funkcja ta przyjmuje następujące parametry i posiada następujący prototyp:

```
s32 adl_ioSubscribe(u32 GpioNb, adl_ioConfig_t
*GpioConfig,
u8 PollingTimerType,u32 PollingTime,s32
GpioEventHandle);
```

`GpioNb` – parametr ten zawiera rozmiar tablicy konfiguracyjnej portów IO GpioConfig.

`GpioConfig` – tablica określająca konfigurację portów IO. Struktura ta jest zdefiniowana następująco:

```
typedef struct
{
 adl_ioLabel_u eLabel;
 u32 Pad;
 adl_io_Direction_e eDirection;
 adl_io_State_e eState;
} adl_ioConfig_t
```

Pole `eLabel` – Służy do ustalania linii portu IO, której dotyczy konfiguracja. Poszczególne linie IO są zdefiniowane za pomocą symboli od `ADL_IO_Q2686_GPIO_1` do `ADL_IO_Q2686_GPIO_44` odpowiadających odpowiednio liniom IO od 1 do 44.

Pole `eDirection` – Pozwala na ustalenie czy linia została skonfigurowana jako wejście (`ADL_IO_INPUT`), czy jako wyjście (`ADL_IO_OUTPUT`).

Pole `eState` – Ma znaczenie tylko w przypadku, gdy dana linia jest skonfigurowana jako wyjściowa, i pozwala określić jej stan początkowy. Jeżeli zmienna ta przyjmie wartość `ADL_IO_LOW`, wówczas linia zostanie ustawiona w stan niski, natomiast jeżeli zmienna przyjmie wartość `ADL_IO_HIGH`, wówczas linia ta będzie miała początkowo stan wysoki.

`PoolingTimerType` – Umożliwia określenie, z jakiego rodzaju timera

List. 2. Przykładowy program sterujący diodami LED modułu Q2686

```
#include „adl_global.h”
//-----------------------------------------------------------
//Rozmiar stosu dla programu
const u16 wm_apmCustomStackSize = 1024;

//-----------------------------------------------------------
//Konfiguracja diod led
#define LEDS_COUNT 2

const adl_ioConfig_t led_config[LEDS_COUNT] =
{
 {ADL_IO_Q2686_GPIO_15, 0, ADL_IO_OUTPUT, ADL_IO_LOW},
 {ADL_IO_Q2686_GPIO_16, 0, ADL_IO_OUTPUT, ADL_IO_LOW},
};

//Uchwyt do portu GPIO zawierajacy diody LED
s32 led_hwnd;

//-----------------------------------------------------------
//Zdarzenie od timera
void led_timer_handler( u8 id )
{
  static u8 l1,l2;
  /* Blinking led handler */
  if(l1)
  {
     adl_ioWriteSingle(led_hwnd,ADL_IO_Q2686_GPIO_15,ADL_IO_HIGH);
  }
  else
  {
     adl_ioWriteSingle(led_hwnd,ADL_IO_Q2686_GPIO_15,ADL_IO_LOW);
  }
  if(!l2)
  {
     adl_ioWriteSingle(led_hwnd,ADL_IO_Q2686_GPIO_16,ADL_IO_HIGH);
  }
  else
  {
     adl_ioWriteSingle(led_hwnd,ADL_IO_Q2686_GPIO_16,ADL_IO_LOW);
  }
  //Zaneguj zmienne
  l1= !l1;
  l2 = !l2;
}

//-----------------------------------------------------------
//Funkcja glowna konfigurujaca
void adl_main ( adl_InitType_e  InitType )
{
  led_hwnd = adl_ioSubscribe(LEDS_COUNT,led_config,0,0,0);
  /* Set 1s cyclic timer */
  adl_tmrSubscribe ( TRUE, 5, ADL_TMR_TYPE_100MS, led_timer_
handler );
}
//-----------------------------------------------------------
```

będą korzystały linie IO, jeżeli okresowe badanie stanu linii wejściowych jest potrzebne. Do dyspozycji mamy timer o standardowej rozdzielczości 100 ms (`ADL_TMR_TYPE_100MS`) oraz timer o wysokiej rozdzielczości 18,5 ms (`ADL_TMR_TYPE_TICK`). W przypadku, gdy nie badamy cyklicznie wejścia, wówczas zmienna ta powinna przyjąć wartość 0

`PoolingTime` – Jeżeli linia ta jest skonfigurowana w kierunku wejściowym, parametr ten umożliwia określenie czasu sprawdzania wejść w jednostkach timera. Zmienna ta pozwala więc określić, ile czasu upłynie pomiędzy kolejnym sprawdzeniem stanu wejść.

`GpioEventHandle` – Do zmiennej tej należy przekazać uchwyt zwrócony wcześniej przez funkcję *adl_ioEventSubscribe*. Zastosowanie tej funkcji wraz z *adl_ioEventSubscribe* umożliwia skonfigurowanie portów IO tak, aby zmiana stanu na określonych wejściach spowodowała wygenerowanie zdarzenia i wywołania odpowiedniej funkcji obsługi tego zdarzenia. Funkcja ta zwraca uchwyt funkcji, który jest wartością nieujemną. Uchwyt ten może być następnie wykorzystywany do wykonywania operacji na odpowiednio przypisanych portach IO. Jeżeli funkcja zwróci wartość ujemną, wówczas oznacza to, że wystąpił błąd.

W naszym przykładzie jako pierwszy argument funkcja przyjmuje wartość równą dwa, co oznacza, że będziemy przekazywać tablicę konfiguracyjną zawierającą dwa elementy. Do drugiego argumentu jest przekazywany adres do tablicy konfiguracyjnej `led_config`, która ustawia porty GPIO15 i GPIO16 jako wyjściowe oraz konfiguruje je w stanie niskim. Pozostałe parametry przyjmują wartość 0, ponieważ nie będziemy przypisywać i wykorzystywać zdarzenia informującego o zmianie stanu linii wejściowych. Ostatnią czynnością wykonywaną w funkcji inicjalizacyjnej (*adl_main*) jest przypisanie funkcji obsługi zdarzenia cyklicznego (*led_timer_handler*) tak, aby była ona wywoływana raz na 0,5 sekundy. Służy do tego celu funkcja *adl_tmrSubscribe*, która umożliwia skonfigurowanie timera i przypisanie do niego funkcji obsługi zdarzenia tak, aby była ona wywoływana w określonych odstępach czasu. Funkcja posiada następujący prototyp oraz przyjmuje następujące parametry:

```
adl_tmr_t *adl_tmrSubscribe(bool bCyclic, u32
TimerValue, u8 TimerType, adl_tmrHandler_t Timerhdl);
```

`bCyclic` – Parametr ten określa czy timer jest cykliczny (TRUE), czy jednorazowy (FALSE). Timer cykliczny działa do momentu jego zatrzymania za pomocą funkcji *adl_tmrUnsubscribe*, albo do momentu zatrzymania aplikacji, natomiast timer jednorazowy jest wywołany jednorazowo po upłynięciu określonego czasu.

`TimerValue` – Parametr ten określa, co ile cykli timera wywoływana będzie funkcja obsługi zdarzenia.

`TimerType` – umożliwia określenie z jakiego rodzaju timera będziemy korzystać. Do dyspozycji mamy timer o standardowej rozdzielczości 100 ms (`ADL_TMR_TYPE_100MS`) oraz timer o wysokiej rozdzielczości 18,5 ms (`ADL_TMR_TYPE_TICK`).

`Timerhdlr` – Wskaźnik do funkcji, która ma być uruchomiona w momencie wystąpienia zdarzenia. Funkcja obsługi zdarzenia od timera powinna mieć następujący nagłówek:

```
void timer_fun(u8 id );
```

Funkcja ta zwraca wartość wskaźnika do Timera lub wartość NULL w przypadku, gdy timera nie udało się uruchomić.

W przypadku naszego programu mrugającego diodami LED, funkcję wywołano w taki sposób, aby zdarzenie było wywoływane cyklicznie co 0,5 s z wykorzystaniem timera o niskiej rozdzielczości 100 ms. Po zakończeniu inicjalizacji funkcja kończy swoje działanie, a dalsza obsługa programu odbywa się w funkcji obsługi timera *led_timer_handler*. W funkcji tej zadeklarowano dwie zmienne statyczne `l1` i `l2`, które z każdym cyklem timera są zmieniane na przeciwne. W zależności od stanu tych zmiennych wywoływana jest funkcja *adl_ioWriteSingle*, która umożliwia zmianę stanu wybranej linii skonfigurowanej wcześniej za pomocą funkcji *adl_ioSubscribe*. Funkcja *adl_ioWriteSingle* jest zadeklarowana następująco oraz przyjmuje następujące parametry.

```
s32 adl_ioWriteSingle(s32 GpioHandle, u32 Gpio, u32
State);
```

`GpioHandle` – Parametr ten powinien zawierać uchwyt do portów GPIO, który został wcześniej przydzielony za pomocą funkcji *adl_ioSubscribe*.

`Gpio` – Numer portu IO, którego stan chcemy zmienić. Dozwolone są wartości od `ADL_IO_Q2686_GPIO_1` do `ADL_IO_Q2686_GPIO_44`. Musimy pamiętać, że port ten powinien należeć do skonfigurowanej puli, w przeciwnym razie funkcja zwróci błąd.

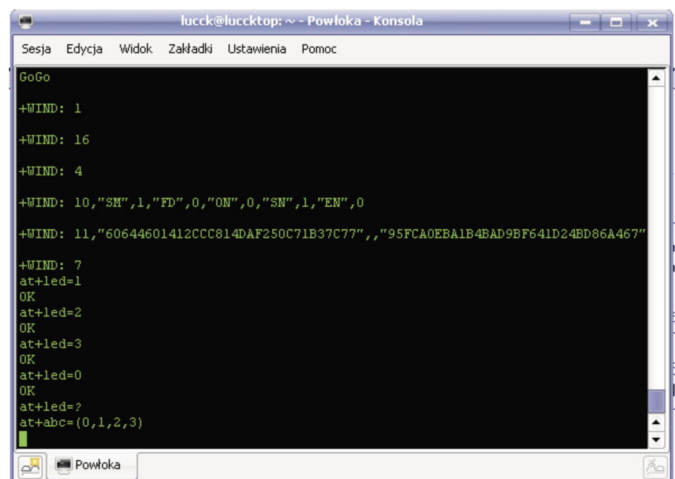
`State` – Stan, w jaki linia ma zostać ustawiona, dozwolone wartości to `ADL_IO_LOW` oraz `ADL_IO_HIGH`.

Funkcja zwraca wartość OK w przypadku prawidłowego ustawienia stanu portu, natomiast w przeciwnym przypadku zwracany jest kod błędu.

W naszym przypadku zmieniany jest na przeciwny stan linii GPIO15 oraz GPIO16, które są podłączone do odpowiednich diod LED znajdujących się na płytce prototypowej. Jak właśnie pokazaliśmy, sterowanie liniami IO w modułach WaveCom jest naprawdę bardzo proste. W dalszej części artykułu opiszemy, w jaki sposób możemy pisać własne komendy AT, rozszerzając w ten sposób funkcjonalność modułu.

Kolejna aplikacja – tworzenie własnych komend AT

Aplikacja przedstawiona powyżej to przykład bardzo często publikowanej wprawki typu „Hello World". Miała ona oswoić Czytelnika ze środowiskiem OpenAT oraz ideą pisania aplikacji wykorzystujących to środowisko. Ponieważ potrafimy już sterować portami IO modułu, napiszemy teraz program, który pozwoli sterować diodami LED za pomocą odpowiednich komend AT. Jak wiemy, komendy AT stanowią podstawowy mechanizm komunikacyjny pomiędzy modułem a urządzeniem zewnętrznym, na przykład komputerem PC, czy sterownikiem nadrzędnym. W przypadku zwykłych modułów GSM do dyspozycji mamy tylko zestaw komend przygotowanych przez producenta. Dla modułów WaveCom sytuacja jest bardziej komfortowa, ponieważ możemy pisać własne aplikacje, implementując przy tym własne komendy AT. Wykorzystując ten moduł możemy więc stworzyć w zasadzie dowolne komendy, jakie będą potrzebne dla urządzenia zewnętrznego. Wszystkie komendy AT rozpoczynają się od ciągu znaków AT. Po nim następuje znak +, a następnie występuje właściwa nazwa komendy oraz rodzaj operacji. Dozwolone operacje to: „=" – ustawienie określonej wartości, czyli wywołanie komendy wraz z określonym parametrem; „?" – pytanie o stan oraz „=?" pytanie o zakres dozwolonych wartości parametru. Na przykład AT+XYZ=10 spowoduje wykonanie komendy XYZ z parametrem równym 10, AT+XYZ? spowoduje zapytanie o aktualny stan tej komendy, natomiast rozkaz AT+XYZ=? spowoduje w terminalu zwrócenie dozwolonych wartości parametrów, jakie może przyjąć komenda. Napiszemy teraz program, który będzie implementował działanie komendy AT+LED. Jak wiemy, do dyspozycji mamy dwie diody LED, bit zerowy będzie więc odpowiadał za pierwszą diodę RXD, natomiast bit pierwszy będzie sterował diodą CTS. Wydając komendę AT+LED=1 włączymy diodę RXD, natomiast komenda



Rys. 16. Odpowiedzi programu na poszczególne opcje komendy AT+LED

Tab. 6. Opis flag dostępnych w opcjach komendy adl_atCmdSubscribe

Nazwa	Wartość	Opis
ADL_CMD_TYPE_PARA	0x0100	Komenda pozwala na wykonywanie rozkazów postaci AT+XYZ=a,b,c. To czy funkcja obsługi będzie wywołana, zależy od tego czy liczba parametrów jest prawidłowa
ADL_CMD_TYPE_TEST	0x0200	Komenda pozwala na wykonywanie operacji zapytania o dozwolone parametry (AT+XYZ=?)
ADL_CMD_TYPE_READ	0x0400	Komenda pozwala na wykonanie operacji zapytania o stan wewnętrzny komendy (AT+XYZ?)
ADL_CMD_TYPE_ACT	0x0800	Komenda pozwala na wykonanie rozkazów niezawierających parametrów (AT+XYZ)
ADL_CMD_TYPE_ROOT	0x1000	Ustawienie tej flagi powoduje, że cały łańcuch tekstowy wpisany na terminalu jest przekazywany do funkcji obsługi zdarzenia bez dodatkowej interpretacji prawidłowego formatu oraz innych znaczników

AT+LED=2 powoduje włączenie diody CTS, po zatwierdzeniu komendy moduł będzie odpowiadał poleceniem OK, natomiast w przypadku wpisania złej liczby moduł będzie odpowiadał ERROR, informując o błędnych parametrach komendy. Z pomocą komendy AT+LED=? możemy dowiedzieć się o dostępnych opcjach i dozwolonych wartościach. Na **rys. 16** przedstawiono odpowiedzi naszego programu na poszczególne opcje komendy AT+LED.

Jak zapewne domyślają się Czytelnicy, napisanie programu obsługującego komendy AT modułu będzie się sprowadzać do zarejestrowania funkcji obsługi zdarzenia, która będzie wywołana w momencie wystąpienia komendy AT+LED oraz funkcji bezpośredniej obsługi diod LED. Dodatkowo, API modułu zapewnia szereg funkcji ułatwiających tworzenie własnych komend AT, więc nie musimy się specjalnie męczyć z funkcjami przetwarzającymi łańcuchy tekstowe. Zarejestrowanie funkcji obsługi zdarzenia dla wybranej komendy AT umożliwia funkcja *adl_atCmdSubscribe*, która posiada następujący prototyp i przyjmuje następujące parametry:

```
s16 adl_atCmdSubscribe(ascii *Cmdstr, adl_atCmdHandler_t Cmdhdl, u16 Options);
```

`CmdStr` – Parametr ten powinien zawierać nazwę komendy, której funkcja ma dotyczyć. Musi być ona podana w pełnym formacie, czyli np. AT+LED.

`Cmdhdl` – Wskaźnik do funkcji obsługi zdarzenia, która będzie wywołana w momencie wpisania komendy w terminalu. Funkcja obsługi zdarzenia musi posiadać następujący prototyp:

```
void atcmd_callback(adl_atCmdPreParser_t *param);
```

`Options` – Opcje dodatkowe pozwalają określić, w jaki sposób komenda jest interpretowana i przekazywana do funkcji obsługi zdarzenia. Jest to kombinacja odpowiednich flag, gdzie znaczenie poszczególnych bitów jest następujące: bity 0...3, czyli 0x000a pozwalają określić minimalną liczbę parametrów, jaką przekazujemy do komendy; bity 4...7, czyli 0x00b0 pozwalają określić maksymalną liczbę parametrów przekazanych do komendy po przecinku (AT+XYZ=a,b,c). Pozostałe flagi opcji są opisane w **tab. 6**.

Funkcja zwraca wartość OK w przypadku powodzenia lub ERROR w przypadku, gdy nie udało się zarejestrować komendy. Od momentu prawidłowego zarejestrowania funkcji, każde prawidłowe wpisanie komendy terminalu spowoduje wywołanie funkcji obsługi zdarzenia komendy wraz z określonymi parametrem. Na podstawie parametru `param`, możemy odczytać parametry przekazane do komendy oraz określić rodzaj zdarzenia. Parametr ten jest zdefiniowany jako struktura zawierająca następujące pola:

```
typedef struct
{
 u16 Type;
 u8 NbPara;
 adl_at_Port_e Port;
 wm_lst_t ParaList;
 u16 StrLength;
 ascii StrData[1];
} adl_atCmdPreParser_t;
```

List. 3. Kod programu obsługującego diody LED za pomocą komendy AT+LED

```
#include „adl_global.h"

//------------------------------------------------------------
//Rozmiar stosu dla aplikacji
const u16 wm_apmCustomStackSize = 1024;


//------------------------------------------------------------
//Konfiguracja diod led
#define LEDS_COUNT 2

const adl_ioConfig_t led_config[LEDS_COUNT] =
{
  ADL_IO_Q2686_GPIO_15, 0, ADL_IO_OUTPUT, ADL_IO_HIGH},
  ADL_IO_Q2686_GPIO_16, 1, ADL_IO_OUTPUT, ADL_IO_HIGH},
};

//Uchwyd do portu GPIO zawierajacy diody LED
s32 led_hwnd;

//------------------------------------------------------------
//Funkcja przechwytujaca wywolania komend AT
void atcmd_callback(adl_atCmdPreParser_t *param)
{
  if(param->Type == ADL_CMD_TYPE_TEST)
  {
   //Test parameter
    adl_atSendResponsePort(ADL_AT_RSP,param->Port,"\r\at+led=(0,1,2,3)\r\n");
  }
  else if(param->Type == ADL_CMD_TYPE_PARA)
  {
     //Set parameter
     int val = wm_atoi(ADL_GET_PARAM(param,0));
     if(val>3)
     {
       adl_atSendResponsePort(ADL_AT_RSP,param->Port,"\r\nERROR\r\n");
     }
     else
     {
       adl_atSendResponsePort(ADL_AT_RSP,param->Port,"\r\nOK\r\n");
       if(val & 1) adl_ioWriteSingle(led_hwnd,ADL_IO_Q2686_GPIO_15,ADL_IO_LOW);
       else adl_ioWriteSingle(led_hwnd,ADL_IO_Q2686_GPIO_15,ADL_IO_HIGH);
       if(val & 2) adl_ioWriteSingle(led_hwnd,ADL_IO_Q2686_GPIO_16,ADL_IO_LOW);
       else adl_ioWriteSingle(led_hwnd,ADL_IO_Q2686_GPIO_16,ADL_IO_HIGH);
     }
  }
}

//------------------------------------------------------------
//Funkcja glowna main
void adl_main ( adl_InitType_e  InitType )
{
  TRACE (( 1, „Embedded : Appli Init" ));
  adl_atCmdSubscribe(„at+led",atcmd_callback,ADL_CMD_TYPE_TEST|ADL_CMD_TYPE_PARA|0x11);
  led_hwnd = adl_ioSubscribe(LEDS_COUNT,led_config,0,0,0);
  adl_atSendResponse ( ADL_AT_UNS, „\r\nGoGo\r\n" );
}

//------------------------------------------------------------
```

`Type` – Określa rodzaj komendy, jaka ma być wykonana. Pozwala to ustalić czy jest to rozkaz z określonymi parametrami, zapytanie itp. Wartość ta jest określona przez wcześniej przypisane opcje komendy i może przyjmować wartości `ADL_CMD_TYPE_xxx`, które zostały opisane przy okazji omawiania argumentu `Options` funkcji *adl_atCmdSubscribe*.

`NbPara` – W przypadku, gdy `Type` zawiera wartość `ADL_CMD_TYPE_PARA`, pole to przekazuje liczbę parametrów, z jakimi została wywołana komenda, w przeciwnym przypadku pole to jest równe zero.

`Port` – Numer portu komunikacyjnego, z którego została wywołana komenda AT.

`ParaList` – Lista parametrów przekazanych jako argument do komendy AT. Poszczególne parametry mogą być pobrane z wykorzystaniem makrodefinicji *ADL_GET_PARAM(_p,_i)*, gdzie `p` jest parametrem przekazanym do funkcji obsługi zdarzenia, natomiast `i` jest numerem parametru. Makro to zwraca wskaźnik do łańcucha tekstowego zawierającego wskazany argument. Na przykład: jeżeli na terminalu wpisano komendę z jednym parametrem AT+XYZ="test", makro *ADL_GET_PARAM(param,0)* zwróci łańcuch „test" pozbawiony cudzysłowów.

`StrLength`, `StrData` – Parametry te zawierają odpowiednio łańcuch tekstowy oraz rozmiar tego łańcucha, zawierający sam łańcuch opisujący komendę, czyli np. jeżeli wywołamy komendę AT+XYZ=12, wówczas parametr ten będzie zawierał tekst AT+XYZ.

Na podstawie parametrów przekazanych do funkcji obsługi zdarzenia możemy w bardzo łatwy sposób obsługiwać komendy AT bez konieczności wykonywania skomplikowanych zabiegów na łańcuchach tekstowych. Oprócz wykonania danego rozkazu, np. zapalenia diod LED, funkcja obsługi komendy powinna wysłać odpowiedź do terminala informującą o statusie wykonania komendy. Do tego celu możemy posłużyć się funkcją *adl_atSendResponsePort*, która jako pierwszy argument przyjmuje rodzaj operacji. W naszym przypadku powinniśmy tam przekazać wartość `ADL_AT_RSP` informującą o tym, że jest to odpowiedź na zapytanie AT. Drugi parametr powinien zawierać numer portu, do którego chcemy wysłać odpowiedź. W naszym przypadku numer portu możemy odczytać z parametru Port przekazanego jako argument do funkcji obsługi zdarzenia. Jako trzeci argument przekazujemy właściwy tekst, który ma zostać wysłany do portu. Na przykład, aby wysłać do terminala odpowiedź OK na jakąś komendę, należy do wspomnianej wyżej funkcji przekazać następujące parametry:

```
adl_atSendResponsePort(ADL_AT_
RSP,param->Port,"\r\nOK\r\n");
```

Na powyższym przykładzie widać, że tworzenie własnych komend AT z wykorzystaniem API modułu jest naprawdę bardzo proste i wymaga użycia niewielu funkcji. Na **list. 3** przedstawiono kod programu obsługi diody LED za pomocą komendy AT+LED.

Program rozpoczyna się od wykonania funkcji *adl_main*, w którym najpierw jest wywoływana funkcja rejestrująca funkcję obsługi zdarzenia obsługi komendy AT+LED, przekazana jako pierwszy argument. Jako drugi argument przekazujemy adres funkcji *at_cmd_callback*, która będzie wywoływana w momencie wystąpienia komendy AT. Trzeci argument zawiera znacznik opcji informujący, jakie opcje komenda będzie implementować. Przekazujemy tutaj flagi `ADL_CMD_TYPE_TEST` (implementacja zapytania o dozwolone wartości AT+LED=?), `ADL_CMD_TYPE_PARA` (implementacja wywołania komendy z przekazywaniem parametrów) oraz wartość `0x11` oznaczająca, że komenda musi zawierać tylko jeden parametr (minimalna liczba parametrów równa 1, maksymalna liczba parametrów równa 1). Kolejna czynność, jaką wykonuje funkcja główna to inicjalizacja portów IO, tak aby mogły one sterować diodami LED, o czym była mowa w poprzednim przykładzie. Na zakończenie funkcja wypisuje w terminalu ciąg znaków GoGo i kończy swoje działanie, a dalej realizację obsługi zdarzeń od komendy AT+LED realizuje zarejestrowana funkcja obsługi zdarzenia *at_cmd_callback*. W funkcji tej jest badany stan zmiennej `param->Type`. W przypadku, gdy przyjmie ona wartość `ADL_CMD_TYPE_TEST` oznacza to, że w terminalu wpisano komendę zapytania o dozwolone wartości (AT+LED=?). Jedyną odpowiedzią na to zdarzenie jest odesłanie do terminala łańcucha tekstowego at+led=(0,1,2,3). Jeżeli zmienna `param->Type` przyjmie wartość `ADL_CMD_TYPE_PARA`, wówczas w terminalu wpisano komendę zawierającą parametr, np. (AT+LED=2). Pobierana jest wtedy wartość parametru `ADL_GET_PARAM` i zamieniana liczba za pomocą funkcji *wm_atoi()*. Następnie jest sprawdzany dozwolony zakres wartości zmiennej (0...3), i jeżeli nie mieści się on w pożądanym zakresie, wysyłany jest do terminala komunikat ERROR, po czym funkcja kończy działanie. Jeżeli natomiast wartość parametru mieści się w dozwolonym zakresie, w zależności od stanu bitów 0...1 są załączane lub wyłączane diody RXD i CTS za pomocą funkcji obsługi portów IO omówione wcześniej.

Lucjan Bryndza SQ7FGB, EP
lucjan.bryndza@ep.com.pl

Dodatkowe materiały oraz poprzednio publikowane części kursu znajdują się na płycie EP-CD12/2008B

R E K L A M A