uIP

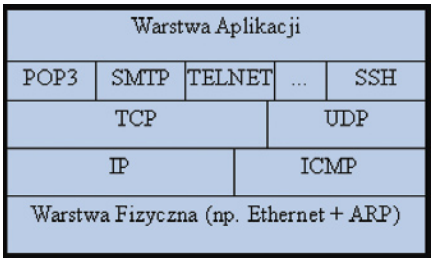
Sposób na TCP/IP w małych mikrokontrolerach

Trudno sobie dzisiaj wyobrazić życie bez Internetu. Do dyspozycji mamy coraz szybsze łącza DSL oraz coraz więcej, coraz bardziej wyrafinowanych usług. Wiele spraw możemy dziś załatwić za pomocą Internetu bez odchodzenia od komputera. Oprócz komputerów do sieci są dołączane różnorodne urządzenia elektroniczne – od drukarek, po telewizory – tak więc coraz częściej w praktyce konstruktora elektronika spotykać się będziemy z koniecznością dołączenia urządzenia do Internetu lub sieci lokalnej.

Najprostszym ze sposobów podłączenia własnego urządzenia do Internetu jest wykorzystanie gotowych modułów (np. Tibbo lub Digi). Moduły te posiadają wyjście w postaci portu RS232, które może być dołączone do portu szeregowego mikrokontrolera. Obsługa takiego modułu sprowadza się jedynie do umiejętności obsługi sprzętowego UART–u oraz wysyłania prostych komend tekstowych do modułu. Jedyną zaletą wykorzystania takiego modułu jest jego prostota, ponieważ konstruktor nie musi posiadać praktycznie żadnej wiedzy na temat protokołów sieciowych (TCP/IP). Rozwiązanie z wykorzystaniem gotowego modułu ethernetowego posiada szereg wad, które w zasadzie dyskwalifikują je w profesjonalnych rozwiązaniach. Do wad tych możemy zaliczyć: wysokie koszty modułu uniemożliwiające wykorzystanie go w aplikacjach niskobudżetowych, niewielka prędkość transmisji danych (która wynika z wąskiego gardła, jakim jest transmisja szeregowa z wykorzystaniem RS232), brak swobody programowania – użytkownik jest skazany na funkcje, które zostały zaimplementowane przez producenta modułu.

Jeżeli chcemy dołączyć urządzenie do sieci dużo lepszym rozwiązaniem jest wykorzystanie stosu TCP/IP zaimplementowanego w systemie mikrokontrolerowym. Wiele nowych mikrokontrolerów z rdzeniem ARM na przykład: STR912, LPC23xx posiada wbudowany kontroler Ethernetu, który do prawidłowej pracy wymaga tylko dołączenia taniego układu warstwy fizycznej PHY, co dodatkowo zmniejsza koszty takiego rozwiązania.

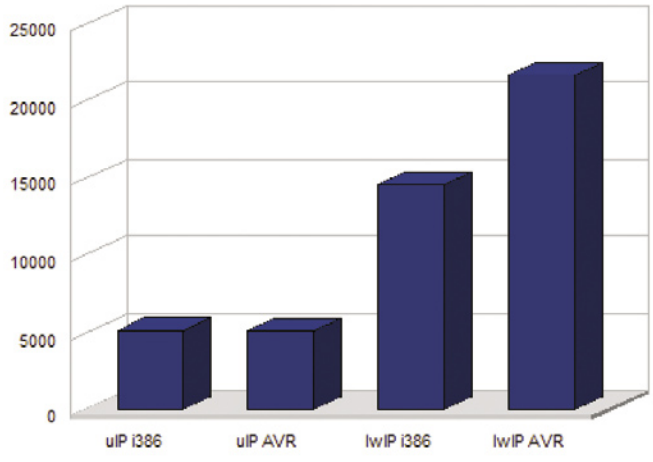
Podstawą Internetu jest protokół TCP/IP, który ma strukturę warstwową przedstawioną na **rys. 1**. Warstwa najniższa jest odpowiedzialna za fizyczny transport danych. Najczęściej w sieciach lokalnych jest wykorzystywany przewodowy protokół Ethernet, którego prędkość transmisji wynosi w zależności od wersji 10/100/1000 Mb/s. Każde urządzenie Ethernet ma unikatowy 6-bajtowy adres sprzętowy MAC, który jednoznacznie identyfikuje dane urządzenie. Pomiędzy węzłami Ethernet istnieje możliwość przesyłania pakietów pomiędzy urządzeniami znajdującymi się w obrębie jednej sieci fizycznej. Kolejną warstwą jest warstwa transportowa IP transportująca dane wewnątrz sieci. Każda ramka IP posiada 4–bajtowy IP adres źródłowy oraz 4–bajtowy IP adres docelowy określający miejsce przeznaczenia. Protokół ten nie zawiera żadnych mechanizmów kontrolnych gwarantujących dotarcie danych do adresata. W przypadku, gdy jako warstwa transportowa używamy Ethernetu, dochodzi dodatkowy protokół ARP, którego zadaniem jest tłumaczenie adresów sprzętowych Ethernet na adresy IP. Powyżej warstwy transportowej znajdują się kolejne warstwy protokołu TCP oraz UDP, których zadaniem jest organizowanie strumienia danych w kanały logiczne – porty. Protokół TCP jest protokołem połączeniowym gwarantującym, że strumień danych zawsze dotrze do adresata we właściwej kolejności, natomiast protokół UDP nie gwarantuje dotarcia danych do miejsca przeznaczenia, za to zapewnia niewielki narzut czasowy. Powyżej protokołów TCP oraz UDP znajduje się warstwa protokołów przypisanych do poszczegól-



Rys. 1. Warstwy stosu uIP

nych aplikacji, takich jak POP3 – protokół odbioru poczty, SMTP – protokół transportowy poczty, HTTP – protokół transportu stron internetowych, oraz wiele innych protokołów. Powyżej tej warstwy znajduje się aplikacja docelowa korzystająca ze wspomnianych wyżej protokołów. Cały stos TCP/IP jest stosunkowo skomplikowany, a jego tradycyjna implementacja wymaga kilkuset kilobajtów pamięci kodu oraz danych. Klasyczny interfejs API stosu TCP/IP oparty jest o gniazda BSD, które zostały zaprojektowane z myślą o dużych komputerach. Interfejs BSD jest stosowany praktycznie we wszystkich systemach operacyjnych dla komputerów PC pracujących pod kontrolą systemu Linux lub Windows. Interfejs BSD wykorzystuje mechanizm blokowania w oczekiwaniu na dane oraz intensywnie wykorzystuje bufory pamięci, ponieważ na stosie TCP/IP spoczywa obowiązek retransmisji danych w przypadku niepowodzenia. Interfejs BSD jest niekwestionowanym standardem w przypadku stosów TCP/IP, w praktyce jednak nie nadaje się on do wykorzystania w mikrokontrolerach, ponieważ zbyt intensywnie korzysta z pamięci oraz wykorzystuje mechanizm blokowania w oczekiwaniu na dane, co wymaga użycia systemu operacyjnego. Specjalnie z myślą o małych urządzeniach powstała implementacja stosu uIP dostępna na licencji BSD pozbawiona wyżej wspomnianych wad. Stos uIP posiada cechy, dzięki którym nadaje się do implementacji w małych mikrokontrolerach 8–, 16–, 32–bitowych:

- niewielkie zapotrzebowanie na pamięć RAM i Flash,
- wsparcie dla protokołów ARP, SLIP, IP, UDP, ICMP (ping), TCP,
- dowolna liczba jednoczesnych połączeń, która jest konfigurowalna na etapie kompilacji,
- licencja umożliwiająca używanie uIP zarówno w projektach niekomercyjnych, jak i komercyjnych,
- bardzo dużo przykładowych aplikacji, umożliwiających zapoznanie się ze stosem,
- klient DNS oraz DHCP,
- zgodność z ANSI C umożliwiająca kompilację z wykorzystaniem dowolnego kompilatora,
- interfejs oparty o mechanizm zdarzeń umożliwiający implementację stosu bez wykorzystania systemu operacyjnego RTOS,
- zgodność ze standardem RFC.



Rys. 2. Porównanie wymaganej przez uIP pamięci dla różnych platform sprzętowych

Użycie minimalnej pojemności pamięci RAM osiągnięto dzięki wykorzystaniu globalnego bufora, który przechowuje jedynie jedną ramkę sieciową. Ponieważ stos jednocześnie przyjmuje i przetwarza tylko jedną ramkę, na kontrolerze Ethernet spoczywa obowiązek ich buforowania. Współczesne kontrolery sieciowe spełniają ten warunek, ponieważ posiadają od kilkunastu do kilkudziesięciu kilobajtów pamięci danych przeznaczonych na bufor. Aplikacja pracująca na „wierzchołku" uIP działa w oparciu o mechanizm zdarzeń i wywoływana jest w momencie wystąpienia określonych sytuacji, na przykład: odebranie pakietu danych, retransmisja pakietu, zakończenie połączenia itp. Wykorzystanie mechanizmu zdarzeń umożliwia działanie aplikacji bez konieczności używania systemu operacyjnego. Dzięki temu zyskujemy dodatkowo na wydajności, ponieważ nie musimy tracić czasu na przełączanie kontekstu procesora przez *scheduler*.

Zapotrzebowanie stosu uIP na pamięć RAM i Flash jest naprawdę bardzo małe. Stos uIP potrzebuje jedynie około 1500 bajtów pamięci RAM na bufor ramki sieciowej, oraz po kilkadziesiąt bajtów na każde aktywne połączenie sieciowe. W ekstremalnych przypadkach, gdy nie dysponujemy tak dużą ilością pamięci RAM, istnieje możliwość zmniejszenie bufora ramki np. do 512 bajtów. Stos będzie nadal działał, jednak ze względu na wydajność zalecane jest używanie bufora będącego rozmiarem maksymalnej ramki Ethernet, czyli około 1500 bajtów. Użycie pamięci Flash jest również niewielkie i w przypadku podstawowej funkcjonalności uIP zadowala się kilkunastoma kB pamięci Flash. Na **rys. 2** porównano rozmiary pamięci programu zajmowanej przez implementację stosu uIP na 8–bitowej architekturze AVR oraz na architekturze i386 w trybie 32–bitowym. Dla porównania, obok zamieszczono zajętość pamięci programu przez stos lwIP, który bazuje na klasycznej implementacji z buforami dynamicznymi oraz systemem operacyjnym (bez samego systemu oraz gniazd BSD).

Niewielka objętość stosu uIP jest dobrze widoczna w przypadku 8–bitowej architektury AVR, gdzie stos lwIP zajmuje ponad 4–krotnie więcej miejsca od uIP. Po dodaniu do stosu lwIP interfejsu BSD oraz małego systemu operacyjnego czasu rzeczywistego (*isix*) okazuje się, że całość na architekturze ARM (LPC23xx) zajmuje około 50 kB pamięci Flash i około 10…20 kB pamięci RAM. Co prawda takie ilości pamięci bez problemu są dostępne w nowszych mikrokontrolerach ARM, jednak używając uIP możemy mieć więcej miejsca dla własnego kodu lub zastosować mniejszy mikrokontroler obniżając koszty urządzenia. Kolejną zaletą uIP jest jego pełna przenośność na różne platformy, ponieważ został on napisany w ANSI C i nie wykorzystuje w zasadzie żadnych funkcji bibliotecznych (np. dynamicznego zarządzania pamięcią *malloc(), free()*).

Przenoszenie uIP na nową platformę sprzętową jest bardzo proste, należy tylko napisać lub zdobyć sterownik do urządzenia sieciowego, napisać prostą procedurę obsługi czasu wykorzystując na przykład dowolny układ czasowo–licznikowy mikrokontrolera oraz napisać prostą pętlę główną. Pętla główna obsługi stosu powinna cyklicznie robić

dwie rzeczy, mianowicie: wywołać funkcję obsługi urządzenia sieciowego sprawdzającą czy nowy pakiet przyszedł z sieci oraz sprawdzić czy upłynął określony odcinek czasu. Jeżeli zostanie odebrany pakiet z sieci, należy wywołać funkcję *uip_input()*, która dokonuje przetwarzania pakietu oraz wywołuje określoną aplikację na rzecz obsługi danego pakietu. W wyniku wywołania tej funkcji aplikacja lub aplikacje mogą wypełnić globalny bufor pakietu danymi do wysłania, więc należy sprawdzić zmienną globalną *uip_len* i jeżeli zmienna ta jest większa od 0, należy wywołać funkcję kontrolera Ethernet wysyłającą pakiet danych do sieci. Mechanizm czasowy używany jest do generowania potwierdzeń, retransmisji, informowania aplikacji, że znajduje się w stanie bezczynności itp.

W celu obsługi zależności czasowych, co 0,5 sekundy powinna być cyklicznie wywoływana funkcja *uip_periodic()*. W wyniku wywołania tej funkcji aplikacja lub stos może również wygenerować ramkę danych do wysłania, więc po wywołaniu tej funkcji należy sprawdzić czy w buforze są dane i ewentualnie wywołać funkcję urządzenia sieciowego wysyłającą ramkę do sieci. W przypadku korzystania z Ethernetu, należy również napisać dodatkowy kawałek kodu wywołujący funkcję obsługi protokołu ARP.

uIP posiada dwa interfejsy dla tworzonych aplikacji: interfejs podstawowy (RAW API) oparty bezpośrednio o zdarzenia przychodzące z sieci oraz interfejs naśladujący gniazda BSD (*ProtoSocket*). W interfejsie podstawowym, każde zdarzenie z sieci powoduje wywołanie funkcji aplikacji z ustawionymi odpowiedni flagami zdarzeń. Flagi zdarzeń mogą być sprawdzane za pomocą odpowiednich makr sprawdzających wystąpienie zdarzenia. Aplikacja bazująca na interfejsie podstawowym powinna sprawdzić przyczynę, z jakiego powodu została wywołana, a następnie podjąć stosowną akcję, np. wysłać ponownie poprzedni pakiet, ponieważ został utracony. W ostatniej wersji uIP pojawił się nowy interfejs *ProtoSocket*, który w systemie pozbawionym systemu operacyjnego naśladuje interfejs gniazd BSD znanych z dużych systemów. Dzięki zestawowi makr nazwanych przez autora PROTOTHREAD, sprytnie wykorzystujących instrukcje *switch–case* zapamiętujących stan aplikacji w 32–bitowej zmiennej, udało się bez systemu operacyjnego oraz przełączania kontekstów procesora sprawić złudzenie pracy w środowisku wielowątkowym.

Przykładowa aplikacja: sterowanie diodami LED za pomocą protokołu telnet

Aby pokazać, że używanie stosu uIP nie musi być bardziej skomplikowane od wykorzystania różnych modułów dołączanych do portu szeregowego, pokażemy w jaki sposób „zmusić" stos uIP, aby za pomocą prostych poleceń wydawanych przez klienta telnetu sterował liniami mikrokontrolera, do których zostały dołączone diody LED. Jako platformę sprzętową wybrano zestaw ZL24ARM oraz ZL25ARM firmy Kamami wyposażony w mikrokontroler STR912, który ma wbudowany kontroler Ethernetu, wymagający tylko dołączenia układu PHY (w zestawie zastosowano PHY firmy STMicroelectronics STE100P).

Praktyczne próby wykazały, że uIP doskonale sobie radzi na dużo mniejszych układach, takich jak np. ATmega64 czy nawet poczciwy AT89C51RD2. Producent mikrokontrolera STR912 dostarcza wiele gotowych bibliotek, między innymi do obsługi wbudowanego interfejsu Ethernet, który po niewielkich przeróbkach można wykorzystać jako sterownik urządzenia sieciowego dla uIP. Kod stosu zawiera wiele przykładów pokazujących sposób jego używania min. przykładowy serwer WWW, serwer telnetu itp. Jako bazę naszego przykładu postanowiono wykorzystać kod przykładowego serwera telnet, który został uzupełniony o dodatkową komendę sterującą liniami GPIO. Użytkownik może łatwo rozbudowywać serwer telnetu wyposażając go w dodatkowe komendy według własnych potrzeb.

Kod pętli głównej stosu jest umieszczony w pliku *ethmain.c*, w którym znajduje się funkcja *main()*. W funkcji tej inicjalizowane są wszystkie układy peryferyjne mikrokontrolera niezbędne do prawidłowej pracy, takie jak pętla PLL, sygnały zegarowe, kontroler przerwań. Inicjalizowany jest także kontroler Ethernet zawarty we wnętrzu mikrokontrolera oraz układ czasowo–licznikowy, który jest odpowiedzialny za generowanie 0,5–sekundowych odcinków czasu, oraz. Aby nie komplikować zbytnio przykładu wybrano statyczną konfigurację adresów IP, która jest realizowana przez poniższy fragment kodu zawarty w funkcji *main()*:

```
uip_ipaddr(ipaddr, 192,168,16,222);
    uip_sethostaddr(ipaddr);
    uip_ipaddr(ipaddr, 192,168,16,1);
    uip_setdraddr(ipaddr);
    uip_ipaddr(ipaddr,
255,255,255,0);
    uip_setnetmask(ipaddr);

    telnetd_init();
```

Czytelnik w zależności od własnej konfiguracji sieci musi zmodyfikować ten fragment kodu ustawiając odpowiedni adres IP, maskę podsieci oraz adres bramy. Po zakończeniu konfiguracji sieci wywoływana jest funkcja inicjalizującą serwer telnetu wykorzystywany w naszej aplikacji. Po wstępnym skonfigurowaniu wszystkich parametrów program wchodzi do pętli głównej *while(1)*, która realizuje główny kod obsługi stosu uIP, według opisanych wcześniej mechanizmów. Aplikacja serwera *telnet* realizująca sterowanie portami GPIO zgodnie z konwencją katalogów stosu uIP znajduje się w katalogu *apps/telnetdlcd* i powstała poprzez przekopiowanie zawartości katalogu *apps/telnetd* zawierającego przykładową aplikację telnetu. Plik *telnetd.c* jest odpowiedzialny za implementację funkcjonalności serwera telnetu i obsługę protokołu, natomiast plik *shell.c* zawiera implementację przykładowych komend wykonujących różne czynności. Przykładowa aplikacja domyślnie zawiera jedynie dwie komendy:

– *help*, która wyświetla tekst pomocy,
– *exit*, która powoduje opuszczenie sesji *telnet*.

Pokażemy teraz w jaki sposób przykładową implementację serwera *telnetu* uzupełnić o dodatkową komendę umożliwiającą sterowanie stanem linii portu P7, do którego w zestawie ZL25ARM są dołączone diody LED D0…D7.

Do katalogu zawierającego kod naszego przykładowego serwera zostały dodane pliki *ledstr.{c,h}* zawierające funkcję *void ledInit(void)*, która inicjalizuje port P7 w kierunku wyjścia oraz *void ledSet(unsigned char mask)*, która w zależności od zawartości zmiennej *mask* ustawia wybrane linie portu P7 w stan 0 lub 1.

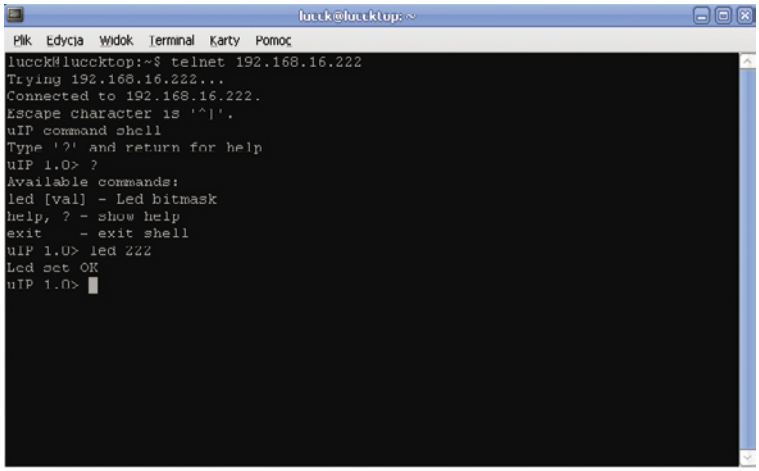
Aby można było sterować diodami LED należy na początku wywołać funkcję *ledInit()*, która konfiguruje linie portu w kierunku wyjścia. Wywołanie tej funkcji najlepiej jest wstawić do funkcji *void shell_init(void)*, która wywoływana jest w momencie inicjalizacji serwera *telnetd*. Aby dodać implementację kolejnej komendy należy uzupełnić globalną tablicę komend:

```
static struct ptentry parsetab[] =
{
  {„led”, led_set},
  {„help”, help},
  {„exit”, shell_quit},
  {„?”, help},
  /* Default action */
  {NULL, unknown}
};
```

Tablica ta składa się ze wskaźnika do łańcucha tekstowego zawierającego nazwę polecenia oraz wskaźnika do funkcji, która jest wywoływany w momencie, gdy użytkownik wpisze dane polecenie w oprogramowaniu klienckim. W naszym przypadku tablica została uzupełniona o komendę *led*, której wpisanie spowoduje wywołanie funkcji *led_set*, która została zdefiniowana następująco:

```
static void led_set(char *str)
{
 if(strlen(str) > 0)
 {
  int ledMask = atoi(&str[4]);
  if(ledMask || str[4]=='0')
  {
     ledSet(ledMask);
     shell_output(„Led set „,”OK”);
  }
  else
  {
     shell_output(„Error in set
LED”,””);
  }
 }
}
```

Funkcja jako argument *str* otrzymuje wskaźnik do łańcucha tekstowego, który został wprowadzony przez użytkownika. Poprzez dodatkowe sparsowanie tego łańcucha możemy zaimplementować opcjonalne parametry dla komendy. W naszym przypadku komenda jest bardzo prosta i ma postać *led x*, gdzie *x* jest liczbą z zakresu 0...255 odzwierciedlającą stan bitów bajtu, który będzie wystawiony na liniach portu P7. Działanie tej funkcji sprowadza się do jedynie do przekształcenia łańcucha tekstowego przekazanego przez użytkownika jako argument komendy LED na liczbę z zakresu 0...255, która następnie jest przekazywana do poprzednio omówionej funkcji *ledSet()*, której zadaniem jest wystawienie odpowiednich stanów logicznych na wyjściu portu P7.



Rys. 3. Widok okna terminala z aktywnym połączeniem telnet

Oprócz ustawienia stanu wybranych linii portu P7, do użytkownika wysyłany jest za pomocą funkcji *shell_output()*, komunikat tekstowy informujący o statusie wykonania komendy. W funkcji wykonywane jest również sprawdzenie czy użytkownik wpisał prawidłowy argument komendy, który powinien być liczbą z zakresu 0...255. W przypadku, gdy liczba jest nieprawidłowa, bity portu P7 nie są ustawiane, a do użytkownika wysyłana jest informacja o błędzie. W miarę potrzeb możemy również uzupełnić funkcję *help()* o dodatkowy opis zawierający informację na temat używania komendy *LED*.

Aby uruchomić poniższy przykład, należy w pliku *ethmain.c* wpisać prawidłowe ustawienia sieci, skompilować przykład za pomocą polecenia *make*, zaprogramować mikrokontroler STR912 zawartością pliku *ethuip.hex*, dołączyć kabel sieciowy do gniazda sieciowego w zestawie ZL25ARM. Aby połączyć się z serwerem telnetu, należy na komputerze PC otworzyć wiersz polecenia i wydać komendę: *telnet adres_ip_plytki*, gdzie jako *adres_ip_plytki* należy wpisać wcześniej skonfigurowany adres. Po wpisaniu tego polecenia na ekranie powinniśmy ujrzeć aktywne połączenie sieciowe z zestawem ZL25ARM (**rys. 3**).

Teraz możemy na przykład wpisać „?”, aby otrzymać informację o dostępnych komendach. Po wpisaniu komendy *led wartosc*, gdzie wartość jest liczbą z zakresu 0...255, diody D0...D7 zestawu ZL25ARM powinny zaświecić się według stanu bitów przekazanych jako argument komendy *led*. Opuszczenie sesji *telnet* następuje w momencie wydania komendy *exit*.

Zakończenie

Zaprezentowany artykuł miał na celu pokazanie, że wykorzystywanie stosu uIP (oraz innych podobnych rozwiązań) we własnych konstrukcjach nie jest dużo trudniejsze od stosowania gotowych modułów ethernetowych. Ponadto pozwala uzyskać znaczące korzyści w stosunku do rozwiązań z dodatkowym modułem, głównie w dziedzinie wydajności oraz ceny. Niewątpliwą zaletą uIP są jego bardzo skromne wymagania sprzętowe, pozwalające na uruchomienie nawet na małych 8–bitowych mikrokontrolerach oraz licencja pozwalająca na jego wykorzystywanie zarówno w projektach komercyjnych, jak i niekomercyjnych. Wraz z kodem stosu uIP dostarczanych jest wiele przykładów, na przykład serwer WWW, klient poczty, wysyłanie poczty, serwer *telnetu*. Przykłady te mogą być bardzo prosto adoptowane do własnych potrzeb, bez szczegółowej znajomości protokołów sieciowych (co pokazaliśmy w artykule), a w przypadku chęci napisania bardziej zaawansowanej aplikacji pozwalają na samodzielną naukę funkcji API stosu uIP. Z uwagi na ograniczone łamy niniejszego artykułu, nie zostały pokazane wszystkie możliwości stosu uIP, na przykład wykorzystanie DHCP, DNS oraz nie zostały opisane funkcje API czy konstrukcja stosu. Artykuł ten ma zachęcić czytelników do samodzielnego poznawania oraz eksperymentowania ze stosem uIP.

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