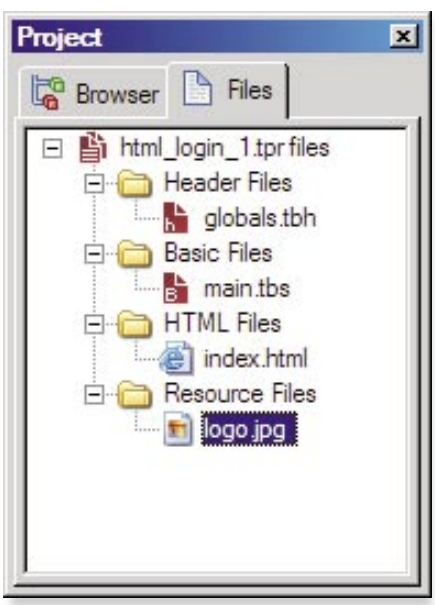
Ethernet w jednym module, część 3
Taiko: Ethernet w Basicu

W tej części zajmiemy się uruchomieniem przykładowego programu w module EM202 wykorzystującego wbudowany WEB Server. Na końcu zastanowimy się nad algorytmem programu sprzętowej bramki SMS (tajemniczy projekt z poprzedniej części), by w kolejnej części artykułu zająć się jego implementacją w języku Tibbo BASIC.

Pusty projekt

Uruchamiamy TIDE i z menu *File–>New Project* wybieramy: *1– Empty Project*. Nadajemy mu nazwę, po czym klikamy na przycisk *Browse* pola *Target Address*, co spowoduje uruchomienie Device Explorer'a z dostępnymi fizycznie modułami. Wybieramy jeden (bądź jedyny) z nich i klikamy na przycisk *Select*, a następnie *Close*. W oknie *New Project* klikamy OK, co spowoduje wyświetlenie obszaru roboczego projektu. Po lewej stronie w zakładce Files widzimy pliki dołączone do projektu. Widnieją w nim dwie pozycje: *global.tbh* – pusty plik przeznaczony do definiowania zmiennych globalnych oraz funkcji, *main.tbs* – główny plik programu. Klikamy dwukrotnie na ten plik w celu jego edycji.

Pierwszą rzeczą, jaką musimy wykonać jest konfiguracja systemu tuż po wystąpieniu sygnału zerowania (a więc też po podaniu zasilania). TiOS generuje wtedy zdarzenie nazywane *on_system_init*.



Rys. 9. Okno projektu ze wszystkimi niezbędnymi plikami

Klikamy zatem w oknie *Project* na zakładkę *Browser* i rozwijamy *Platform Events*. Lista, którą otrzymujemy stanowi dla nas bardzo istotną informację. Widzimy tu wszystkie możliwe zdarzenia, jakie mogą wystąpić w naszym systemie. W **tab. 1** znajduje się zestawienie wszystkich pozycji z listy.

Z uwagi na to, że na samym początku pracy sytemu musimy go odpowiednio skonfigurować, klikamy dwukrotnie na *event on_sys_init*, co spowoduje pojawienie się szablonu obsługi zdarzenia. Podstawową czynnością, jaką nasz system musi wykonać, aby zaistnieć w sieci jako WEB Server, jest konfiguracja protokołu IP. Na potrzeby aplikacji zakładamy, że urządzenie ma stały, niezmienny adres IP. Adres jest przechowywany we własności net.ip (własność obiektu net). Adres IP możemy zarówno odczytywać, jak i zapisywać (przez podanie *stringu*), natomiast jego domyślna wartość to adres loopback 127.0.0.1. W naszym kodzie dopisujemy zatem (adres IP powinien być tak dobrany, aby moduł był osiągalny w podsieci):

```
net.ip  =  "192.168.0.10"
```

Następnie musimy zdecydować, ile socketów (gniazd) przygotujemy do obsługi klientów naszego serwera WWW. Liczba socketów obsługiwanych przez system operacyjny zależy od użytego modułu (mówi o tym własność sock.numofsock). TiOS zainstalowany na platformie EM202 może pracować maksymalnie z szesnastoma gniazdami jednocześnie. Dla każdego socketu z osobna musimy wykonać jego inicjalizację/konfigurację poprzez wykonanie trzech metod na obiekcie *sock:*

a) *syscall sock.rxbuffrq(numpages as byte) as byte* – (*rx buffer request*) – żądanie alokacji obszaru pamięci na bufor odbiorczy socketu. Rozmiar pamięci jest wyrażony w stronach, natomiast rozmiar strony to 256 bajtów. Faktyczny rozmiar bufora należy pomniejszyć o 16 bajtów użytych na wewnętrzne zmienne kontroli bufora. Metoda ta jedynie zgłasza do sytemu potrzebę alokacji pamięci, więc po jej wykonaniu bufor nie jest jeszcze gotowy. Właściwa alokacja wykona się po wywołaniu metody sys.buffalloc (zrobimy to po zdefiniowaniu parametrów dla wszystkich socketów). Metoda zwraca aktualną dostępną liczbę stron pamięci, która może zostać przeznaczona na bufor,

b) *syscall sock.txbuffrq(numpages as byte) as byte* – (*tx buffer request*) – metoda żądania alokacji bufora nadawczego socketu, analogiczna do poprzedniej,

c) *syscall sock.varbuffrq(numpages as byte) as byte* – (*http variables buffer request*) – metoda żądania alokacji bufora na stringi http, a więc jej wywołanie jest konieczne jedynie w przypadku przeznaczenia socketu do obsługi http. Podobnie jak dwie poprzednie metody, rozmiar alokowanej pamięci jest tu wyrażony w stronach.

Z uwagi na to, iż z założenia ruch TCP do i z modułu będzie tak naprawdę znikomy (w porównaniu z serwerami opartymi o platformę PC), to rozmiar wszystkich trzech buforów ustalimy na jedną stronę pamięci.

Tab. 1. Zestawienie wszystkich możliwych zdarzeń systemu w module EM202

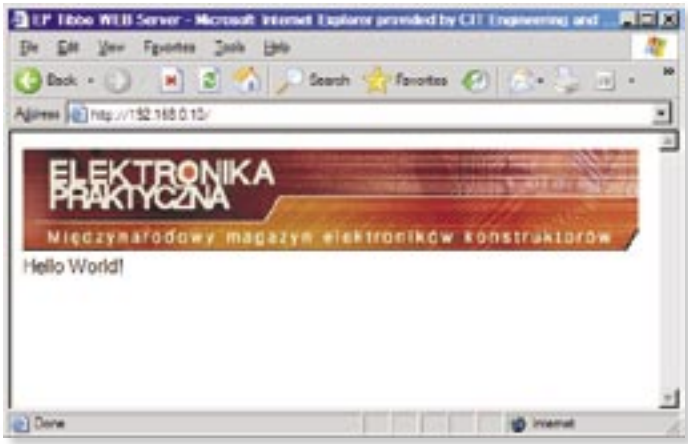
Nazwa	Opis
on_button_pressed	Jedną z linii wejściowych modułu EM202 jest linia nazwana MD (*Mode Selection Pin*). Służy ona do wprowadzania modułu w tryb konfiguracji poprzez UART. Moduł w wersji EV posiada na płytce przycisk (oraz rezystor podciągający) – „button" występujący w nazwie eventu, to właśnie mikrostyk podłączony do linii MD. Zatem linię tę możemy wykorzystywać analogicznie do linii przerwań zewnętrznych mikrokontrolerów. Zdarzenie jest generowane przy wciśnięciu tegoż przycisku (zbocze opadające).
on_button_released	Zwolnienie przycisku (zbocze narastające)
on_net_link_change	Zmiana stanu linku warstwy fizycznej. Innymi słowy podłączenie/odłączenie modułu od/do aktywnego segmentu sieci (innego urządzenia sieciowego).
on_net_overrun	Zdarzenie generowane w momencie przepełnienia bufora odbiorczego kontrolera NIC (*Network Interface Controller*). W przypadku modułu EM202 jest to układ Davicom DM9000EP. Jeśli nie obsłużymy tego zdarzenia, TiOS nie będzie generował kolejnych.
on_pat	Moduł posiada dwie linie wyjściowe przeznaczone do sterownia diodami LED. Diody te są opisane w TAIKO jako obiekt *pat* posiadający metodę *play* pozwalającą zapalać i gasić diody w różnej konfiguracji. Zdarzenie występuje po wykonaniu metody *play* na obiekcie *pat*.
on_ser_data_arrival	Zdarzenie generowane, gdy w buforze odbiorczym układu UART znajduje się przynajmniej jeden znak. Zdarzenie musi być obsłużone przy pomocy metody odczytu bufora odbiorczego *ser.getdata()*. Jeśli opuścimy funkcję obsługującą zdarzenie, podczas gdy w buforze ciągle będą dane, system natychmiast wygeneruje kolejne.
on_ser_data_sent	Analogiczne zdarzenie do poprzedniego, jednak mechanizm jego generacji nie jest tak oczywisty. Zdarzenie może zostać wygenerowane jednokrotnie jedynie po wywołaniu metody *ser.notifysent()* przyjmującej jako argument liczbę bajtów. Potwierdzenie nadania (*notification*) będzie wygenerowane w postaci zdarzenia *on_ser_data_sent*, gdy liczba bajtów oczekujących na transmisję w buforze nadawczym UART–u będzie mniejsza niż liczba bajtów przekazana do *ser.notifysent()*. Po jednym wywołaniu *ser.notifysent()* może nastąpić tylko jedno zdarzenie. Jeśli chcemy zapewnić sobie ciągłe powiadamianie o fakcie wysłania danych, musimy po każdym wystąpieniu zdarzenia wywołać *ser.notifysent()*.
on_ser_esc	Zdarzenie generowane po wykryciu w buforze odbiorczym znaku (np. terminującego) zdefiniowanego uprzednio poprzez własność *ser.escchar()*.
on_ser_overrun	Przepełnienie bufora odbiorczego UART. Zdarzenie raz wygenerowane nie wystąpi po raz drugi do momentu zwolnienia miejsca buforze.
on_sock_data_arrival	Zdarzenie występuje po pojawieniu się danych w buforze odbiorczym socketu (gniazda) zdefiniowanego do obsługi TCP lub UDP. Występuje tu jednak istotna różnica: W przypadku TCP, gdy podczas jednorazowej obsługi zdarzenia nie zdążymy przetworzyć (odczytać) całego strumienia danych, porcja nieodczytana zostaje zachowana, a system natychmiast generuje kolejne zdarzenie. Natomiast protokół UDP z racji swojej natury niedeterministyczny (nie dający nam gwarancji dostarczenia danych) traktowany jest tak przez system, a więc dane, których nie zdążymy przetworzyć w trakcie jednej obsługi zdarzenia są tracone.
on_sock_data_sent	Podobnie jak w przypadku układu UART: Zdarzenie generowane po tym, jak liczba znaków w buforze nadawczym socketu jest mniejsza, niż ta zdefiniowana przy pomocy metody sock.notifysent(). Metoda ta musi być wywołana za każdym razem, gdy chcemy, aby zdarzenie to zostało wygenerowane.
on_sock_event	Zdarzenie generowane po zmianie stanu socketu. Np. w przypadku protokołu TCP zmiana stanu może oznaczać ustanowienie nowego połączenia.
on_sock_inband	Dla każdego socketu system alokuje bufor CMD (*command*). Jest on odpowiedzialny za przechowywanie tzw. komend *inband*. Są to komendy konfigurujące moduł poprzez połączenie TCP lub UDP. Są one zaszyte w regularnym strumieniu danych i opatrzone warunkiem (sekwencją znaków) startu i stopu. Jeśli urządzenie napotka w datagramie protokołu warunek startu, następuje ekstrahowanie komendy i umieszczenie jej w buforze CMD. Jeśli to nastąpi, generowane jest omawiane zdarzenie.
on_sock_overrun	Zdarzenie informujące o przepełnieniu bufora odbiorczego. Teoretycznie zdarzenie może wystąpić tylko podczas pracy z protokołem UDP z racji braku potwierdzeń nadchodzących pakietów.
on_sys_init	Bardzo ważne zdarzenie. Występuje tuż po zerowaniu systemu. W praktyce fragment kodu odpowiedzialny za obsługę tego zdarzenia będzie tak naprawdę kodem zawierającym konfigurację systemu.
on_sys_timer	Zdarzenie generowane w sposób ciągły, co 0,5 s. Może posłużyć do wykonywania jakiegoś periodycznego procesu. Należy pamiętać, iż nie dysponujemy możliwością użycia jakiejś głównej pętli, która będzie się odbywała w nieskończoność! W celu zasymulowania takiej pętli (opóźnionej o 0,5 s) możemy wykorzystać to zdarzenie.

Trzy wymienione powyżej parametry socketu dotyczyły sprzętowej warstwy modułu. Odwoływały się do systemu operacyjnego o przyznanie zasobów fizycznej pamięci. Kolejnym krokiem jest konfiguracja sposobu działania socketu. Ponownie ustalimy trzy parametry, posługując się tym razem własnościami obiektu *sock*:

a) *property sock.protocol as pl_sock_protocol* – protokół transportowy gniazda. Jako że http używa do transmisji protokołu TCP, musimy taki ustawić (domyślnym protokołem gniazda jest UDP),

b) *property sock.inconmode as pl_sock_inconmode* (*incoming connections mode*) – bardzo użyteczna własność, szczególnie dla systemów, w których duży nacisk kładzie się na zabezpieczenia dostępu. Umożliwia ona definiowanie akceptowalności połączeń przychodzących. Za jej pomocą możemy skonstruować prosty firewall chroniący zasoby systemu. Własność może przyjmować następujące wartości:
   - PL_SOCK_INCONMODE_NONE (0) – wartość domyślna. Socket nie będzie akceptował żadnych połączeń przychodzących,
   - PL_SOCK_INCONMODE_SPECIFIC_IPPORT – akceptowane będą połączenia jedynie z określonego adresu IP oraz portu, które są zdefiniowane za pomocą własności odpowiednio: *sock.targetip* i *sock.targetport*,
   - PL_SOCK_INCONMODE_SPECIFIC_IP_ANY_PORT – akceptowane będą jedynie połączenia z IP zdefiniowanego w *sock.targetip* i z dowolnego portu,

c) *property sock.httpportlist as string* – własność definiuje listę portów, na których będą akceptowane połączenia http. Jako wartość przyjmuje łańcuch znaków, w którym po przecinku wyliczone są kolejne porty. Wartość domyślna tej własności to pusty string, co jest jednocześnie informacją, że przychodzący strumień danych ma nie być interpretowany jako strumień http. Dla każdego z socketów możemy też zdefiniować wartość własności *sock.localportlist*, mówiącą o tym na jakich portach będzie odbywał się nasłuch dla przychodzących połączeń TCP, bądź UDP. Jeśli ten sam numer portu znaj-



Rys. 10. Okno przeglądarki WWW w pasku adresu IP modułu EM202

dzie się na obu listach (a więc tej definiującej „port http" i tej definiującej port TCP), wówczas przychodzący do niego strumień TCP będzie traktowany i interpretowany jako ciąg http. Własność ta niejako stanowi punkt uaktywniania funkcji WEB serwera modułu.

Zakładamy, zatem, że nasz serwer będzie w stanie obsługiwać naraz 8 połączeń. Kod inicjalizujący 8 socketów do pracy w trybie http został przedstawiony na **list. 1**.

„index.html"

Do przetestowania naszego prostego programu brakuje już tylko jednej rzeczy, a wiec prawidłowo sformatowanej strony html. Stwórzmy więc jej najprostszą odmianę, posługując się notatnikiem (**list. 2**) i zapiszmy w folderze projektu pod nazwą *index.html*, jeśli chcemy by była to strona domyślna.

Znacznik <IMG> umieszczony w kodzie strony spowoduje wyświetlenie pliku *logo.jpg*, jeśli taki zostanie dołączony do projektu. Dodajemy obydwa pliki do projektu (**rys. 9**) i z menu *Debug* wybieramy *Run*. Po kompilacji, aplikacja zostanie załadowana do modułu. Aby sprawdzić jej działanie uruchamiamy przeglądarkę WWW i w miejsce adresu wpisujemy IP modułu (w powyższym przykładzie – 192.168.0.10). Efekt naszej pracy powinien być podobny do przedstawionego na **rys. 10** (z dokładnością do pliku *jpg*).

Bramka SMS

Do realizacji kompletnie sprzętowej i niezależnej bramki będziemy potrzebowali modułu EM202–EV (bądź EM202 z „osprzętem") i przemysłowego modułu GSM. Obydwa bloki będą się ze sobą komunikowały łączem szeregowym z prędkością 9600 b/s. Aplikacja zostanie oparta o szablon projektu DHCP + setup, co umożliwi automatyczne pozyskiwanie konfiguracji IP. Interfejs WWW urządzenia będzie pozwalał na całkowitą jego konfigurację (statyczne IP oraz PIN kod simkarty). Moduł GSM będzie sterowany komendami AT (wielokrotnie omawiane na łamach EP). Aplikacja sterująca zostanie podzielona na następujące moduły (nazwy modułów odpowiadają faktycznym nazwom plików .tbs):

1) *dhcp.tbs* – kod dostarczony razem z TIDE – zawiera niezbędne funkcje klienta DHCP.
2) *init.tbs* – zawierać będzie obsługę eventu on_sys_init. Odpowiedzialny będzie za inicjalizację warstwy sieciowej (obsługa DHCP) i transportowej/sesji – inicjalizacja socketów, jak również za inicjację portu szeregowego modułu EM202.
3) *serial.tbs* – obsługa portu szeregowego, a tak naprawdę tylko jego toru odbiorczego (event *on_serial_data_arrival*). Jeden z istotniejszych punktów aplikacji – kod zawarty w tym pliku będzie odpowiedzialny za interpretowanie komend (a raczej odpowiedzi) przychodzących z modułu. Do jego konstrukcji zostanie użyta instrukcja *case*, której element decyzyjny będzie ustalany w innych modułach aplikacji.
4) *wismo.tbs* – będzie zawierał funkcję obsługi modułu GSM (użyty przeze mnie moduł oparty jest o *chipset* Wismo). Kod będzie odpowiedzialny za prawidłowe formatowanie ciągów znaków, łącznie ze znakami nowej linii i powrotu karetki.

Interfejs WEB aplikacji zostanie rozbudowany do 3 plików html. Plik główny (*index.html*) będzie zawierał pola tekstowe do wprowadzania treści wiadomości sms oraz numeru telefonu oraz przyciski Konfiguracja oraz Wyślij. Plik *send.html* będzie wywoływany po kliknięciu przycisku Wyślij. Za pomocą metody GET http zostaną przekazane do niego treść wiadomości oraz numer telefonu, natomiast strona zwróci status doręczenia wiadomości. Plik *conf.html* będzie odpowiedzialny za konfigurację urządzenia.

Marcin Chruściel, EP
marcin.chrusciel@ep.com.pl

List. 1. Obsługa eventu on_sys_init

```
sub on_sys_init
'This event is always generated first. We use it to initialize
'our program.

dim s as byte

net.ip = "192.168.0.10" 'static IP address

for s = 0 to 8      'for all (8) sockets
  sock.num = s      'active socket

  sock.rxbuffrq(1)  'one mem page for rx buff
  sock.txbuffrq(1)  'one mem page for tx buff
  sock.varbuffrq(1) 'one mem page for http variables

  sock.protocol=PL_SOCK_PROTOCOL_TCP
  sock.inconmode=PL_SOCK_INCONMODE_ANY_IP_ANY_PORT
  sock.httpportlist="80"
 next s

 sys.buffalloc           'buffers mem allocation
end sub
```

List. 2. Zawartość pliku index.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD W3 HTML//EN">
<HTML>
 <HEAD>
  <TITLE>EP Tibbo WEB Server</TITLE>
 </HEAD>
 <BODY>
  <IMG SRC="logo.jpg">
  <BR><FONT FACE="Arial">Hello World!</FONT>
 </BODY>
</HTML>
```