Technologia GSM w elektronice (6)

Obsługa portu szeregowego, biblioteka WIP

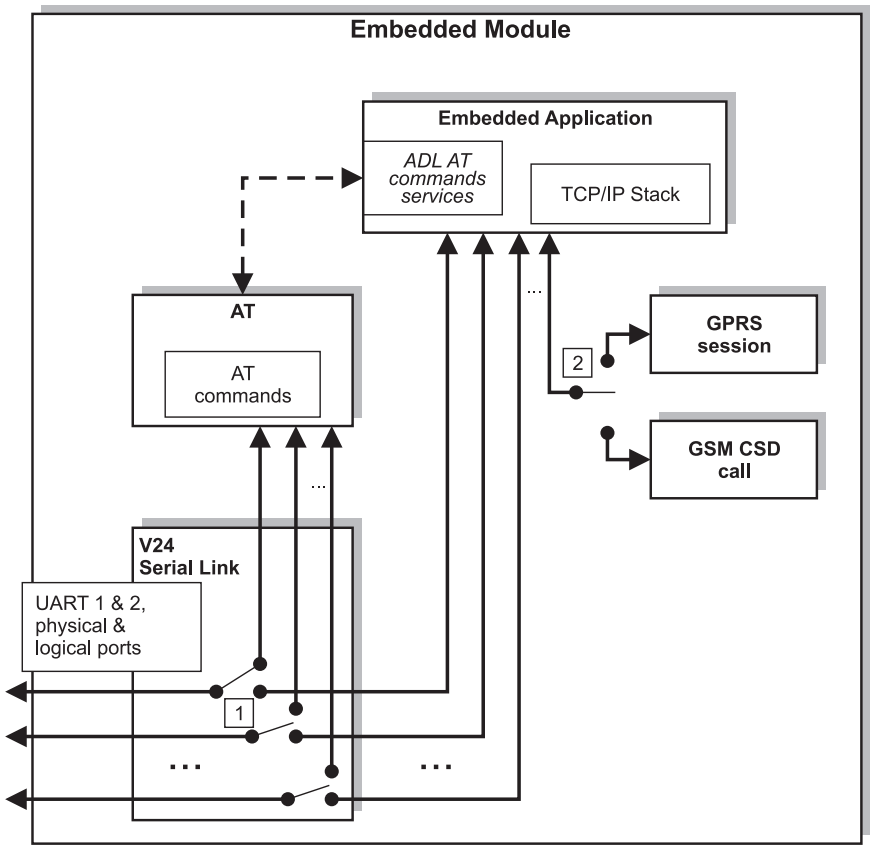
W poprzednim odcinku pokazaliśmy, jak z poziomu aplikacji wykorzystywać komendy AT oraz zapoznaliśmy się z różnymi metodami przechowywania danych w pamięci modemu. Kolejnym krokiem będzie obsługa portów szeregowych, a także wprowadzenie do biblioteki internetowej WIP (Wavecom IP).

Dodatkowe informacje:
Więcej informacji na temat produktów Sierra Wireless można znaleźć na stronach producenta: *www.sierrawireless.com* lub kontaktując się z firmą ACTE Sp. z o.o., która jest oficjalnym dystrybutorem opisywanych produktów oraz zapewnia pełne wsparcie techniczne.

Porty szeregowe (UART1, UART2, USB) w modułach AirPrime Q26 mają dwa tryby pracy: tryb komend oraz tryb danych. W poprzednich odcinkach kursu wysyłaliśmy z aplikacji dane przez port szeregowy, ale nie potrafiliśmy pobierać danych inaczej, niż przekazując je do aplikacji za pomocą utworzonych przez nas komend AT. Właśnie w ten sposób działa tryb komend (*command mode*). System przyjmuje i interpretuje tylko dane zaczynające się od przedrostka „AT”. Natomiast tryb danych (*data mode*) pozwala na pobieranie przez aplikację dowolnych danych, dzięki czemu modem może komunikować się z innymi urządzeniami, np. odbiornikami GPS wysyłającymi dane o pozycji w postaci tekstowych ramek NMEA.

Flow Control Manager

Do obsługi portów w trybie danych służy mechanizm o nazwie *Flow Controll Manager* (FCM). Za pomocą serwisu FCM możemy zasubskrybować się nie tylko do portów szeregowych, ale również obsługiwać strumienie danych dla połączeń CSD oraz GPRS. Strukturę blokową FCM pokazano na **rysunku 1**.



Rysunek 1. Schemat blokowy Flow Control Manager

Tryb pracy *command mode* jest funkcjonalnością jedynie portów szeregowych, które mogą być przełączane pomiędzy *command* i *data*. Na rysunku 1 symbolizuje to przełącznik „1”. Podczas transmisji CSD i GPRS tryb komend nie może być używany, a przełącznik „2” na schemacie informuje, że oba strumienie danych nie mogą być obsługiwane jednocześnie. Domyślnie, gdy w modemie nie ma aplikacji Open AT lub aplikacja nie subskrybuje się do FCM, zachowanie się portów i strumieni CSD oraz GPRS jest kontrolowane przez firmware modemu i wygląda to następująco:

- dane CSD dla wywołanego połączenia GSM CSD są kierowane na port szeregowy, który zainicjował połączenie (ATD) lub dzięki któremu zostało ono odebrane (ATA).
- dane sesji GPRS są kierowane na port szeregowy, z użyciem którego została ona zainicjowana (komenda AT+CGDATA lub ATD).

Na **listingu 1** pokazano przykładowy program wykorzystujący mechanizm FCM do obsługi portu szeregowego. Przed uruchomieniem należy upewnić się czy UART2 jest włączony (włączanie – komenda AT+WMFM=0,1,2). Aplikacja subskrybuje się do portu UART2 za pomocą funkcji *adl_fcmSubscribe()*. Gdy subskrybcja powiedzie się, to automatycznie zostanie wywołana funkcja *Uart2_fun()* z obsługą zdarzenia *ADL_FCM_EVENT_FLOW_OPENNED*. To spowoduje przełączenie portu w tryb danych. Dodatkowo, utworzono komendę *AT+TEST*, która wysyła ustalony ciąg znaków na port UART2.

Listing 1. Program wykorzystujący mechanizm FCM do obsługi portu szeregowego

```
#include "adl_global.h"
const u16 wm_apmCustomStackSize = 1024;

u8 H_Uart2;

bool Uart2_fun (u8 event)
{
        switch (event){
                case ADL_FCM_EVENT_FLOW_OPENNED:
                        TRACE((1,"UART2 otwarty"));
                        adl_fcmSwitchV24State(H_Uart2, ADL_FCM_V24_STATE_DATA); //przelacz w tryb data
                        break;
                case ADL_FCM_EVENT_V24_DATA_MODE:
                        TRACE((1,"UART2 - data mode"));
                        break;
                case ADL_FCM_EVENT_V24_AT_MODE:
                        adl_atSendResponse(ADL_AT_PORT_TYPE (ADL_PORT_UART2, ADL_AT_RSP), "Uart2_AT mode\n\r");
                        adl_fcmUnsubscribe(H_Uart2);
                        break;
        }
        return TRUE;
}

bool Uart2_data (u16 DataLen, u8 * Data)
{
        ascii bufor[50];
        wm_memset(bufor,0,50);
        wm_memcpy(bufor,Data,DataLen);
        TRACE((1,"data length = %d",DataLen));
        bufor[DataLen]=0;
        TRACE((1,bufor));
        adl_atSendResponse(ADL_AT_PORT_TYPE(ADL_PORT_UART1,ADL_AT_RSP), "\n\r UART2 Data:");
        adl_atSendResponse(ADL_AT_PORT_TYPE(ADL_PORT_UART1,ADL_AT_RSP), bufor);
        return TRUE;
}

void Fun_test (adl_atCmdPreParser_t * paras)
{
        ascii * test_string="\r\nUART2 data mode test\r\n";
        if (paras->Type == ADL_CMD_TYPE_ACT) {
                adl_fcmSendData(H_Uart2, test_string, strlen(test_string));
                adl_atSendStdResponse(ADL_AT_RSP,ADL_STR_OK);
        }
}

void adl_main ( adl_InitType_e InitType )
{
        TRACE (( 1, "Embedded Application : Main" ));
        adl_atCmdSubscribe("AT+TEST", Fun_test, ADL_CMD_TYPE_ACT);
        H_Uart2 = adl_fcmSubscribe(ADL_PORT_UART2, Uart2_fun, Uart2_data);
}
```

Listing 2. Program wykorzystujący bibliotekę WIP

```
#include "adl_global.h"
#include "wip.h"

// Mandatory variables
// wm_apmCustomStackSize
const u16 wm_apmCustomStackSize = 1024;
// Local variables
wip_bearer_t bearer_handle;
// Local functions
void evh_bearer( wip_bearer_t b, s8 event, void *ctx)//--
{
        ascii IpAddr[16];
        wip_in_addr_t appIpAddr;
        ascii string[100];
        s8 bearer_ans;
        switch(event)
        {
                case WIP_BEV_IP_CONNECTED:
                        adl_atSendResponse ( ADL_AT_RSP, "WIP: connected OK: jestem w evh_bearer\r\n");
//This API provides IP in network format (i.e u32 number)
                        wip_bearerGetOpts(b, WIP_BOPT_IP_ADDR, &appIpAddr, WIP_BOPT_END);
//This API converts the u32 IP address to a dotted notation IP address.
                        wip_inet_ntoa(appIpAddr , IpAddr, 16);
                        wm_sprintf(string,"WIP: IP %s\r\n",IpAddr);
                        adl_atSendResponse ( ADL_AT_RSP, string);
                        break;
                case WIP_BEV_IP_DISCONNECTED:
                        adl_atSendResponse ( ADL_AT_UNS, "WIP: disconnected\r\n");
                        break;
                case WIP_BEV_CONN_FAILED:
                        adl_atSendResponse ( ADL_AT_UNS, "WIP: WIP_BEV_CONN_FAILED\r\n");
                        break;
                case WIP_BEV_STOPPED:
                        bearer_ans = wip_bearerClose(bearer_handle);
                        TRACE((1,"Fun_close:wip_bearerClose = %d",bearer_ans));
                        break;
                default:
                        adl_atSendResponse ( ADL_AT_UNS, "WIP: other error\r\n");
                        break;
        }
}

void Fun_start(adl_atCmdPreParser_t * param) {

        s8 wynik;
```

Listing 2. c.d.

```
        ascii * GPRS_APN;
        ascii * GPRS_USER;
        ascii * GPRS_PASSWORD;
        TRACE((1,"W Fun_start"));
        if (param->Type == ADL_CMD_TYPE_PARA){
                GPRS_APN = ADL_GET_PARAM ( param, 0 );
                TRACE (( 3,GPRS_APN));
                GPRS_USER = ADL_GET_PARAM ( param, 1 );
                TRACE (( 3,GPRS_USER));
                GPRS_PASSWORD = ADL_GET_PARAM ( param, 2 );
                TRACE (( 3,GPRS_PASSWORD));
                wynik=wip_bearerOpen( &bearer_handle, "GPRS", evh_bearer, NULL);
                TRACE((1,"Fun_start:wip_bearerOpen = %d",wynik));
                wynik=wip_bearerSetOpts( bearer_handle,
                        WIP_BOPT_GPRS_APN, GPRS_APN,
                        WIP_BOPT_LOGIN, GPRS_USER,
                        WIP_BOPT_PASSWORD,  GPRS_PASSWORD, WIP_BOPT_END);
                TRACE((1,"Fun_start:wip_bearerSetOpts = %d",wynik));
                wynik=wip_bearerStart( bearer_handle);
                TRACE((1,"Fun_start:wip_bearerStart = %d",wynik));
                adl_atSendStdResponse(ADL_AT_RSP,ADL_STR_OK);
        }
}

void Fun_close(adl_atCmdPreParser_t * paras)
{
        s8 wynik;
        wynik = wip_bearerStop( bearer_handle);
        TRACE((1,"Fun_close:wip_bearerStop = %d",wynik));
        adl_atSendStdResponse(ADL_AT_RSP,ADL_STR_OK);
}

void adl_main ( adl_InitType_e InitType )
{
        TRACE (( 1, "Embedded Application : Main" ));
        wip_netInit();
        adl_atCmdSubscribe("AT+START", Fun_start, ADL_CMD_TYPE_PARA | 0x0031);
        adl_atCmdSubscribe("AT+CLOSE", Fun_close, ADL_CMD_TYPE_ACT);
}
```

Listing 3. Przykład programu nawiązującego połączenie w trybie TCP Client

```
#include "adl_global.h"
#include "wip.h"
const u16 wm_apmCustomStackSize = 1024;
wip_bearer_t bearer_handle;
u8 H_Uart2;
wip_channel_t client, destination;

bool Uart2_fun (u8 event)
{
        switch (event){
                case ADL_FCM_EVENT_FLOW_OPENNED:
                        TRACE((1, "Uart2_Opened"));
                        adl_fcmSwitchV24State(H_Uart2, ADL_FCM_V24_STATE_DATA);
                break;
                case ADL_FCM_EVENT_V24_DATA_MODE:
                        TRACE((1, "Uart2_Datamode"));
                        break;
                case ADL_FCM_EVENT_V24_AT_MODE:
                        TRACE((1, "Uart2_AT_mode"));
                        break;
        }
        return TRUE;
}

bool Uart2_data (u16 DataLen, u8 * Data)
{
        wip_write( destination, Data, DataLen );
        return TRUE;
}

static void ClientHandler(wip_event_t *ev, void *ctx)
{
        ascii buffer [256];
        ascii* init_msg = "Halo z klienta WIP\n\r";
        int nread = 0;
        int nwrite =0 ;

        switch( ev->kind) {
        case WIP_CEV_OPEN:
                destination= ev->channel;
                break;
        case WIP_CEV_PEER_CLOSE:
                wip_close( ev->channel);
                break;
        case WIP_CEV_READ:
                while ( (nread = wip_read( ev->channel, buffer, sizeof( buffer))) >0 )
                        adl_fcmSendData(H_Uart2, (u8 *) buffer, nread);
                        break;
        case WIP_CEV_WRITE:
                nwrite=wip_write( ev->channel, init_msg, sizeof(init_msg)); //tekst powitalny
                break;
        }
}

static void finalizer( void *ctx ) {
        wip_bearerStop( bearer_handle); //socket zamkniety, zamknij teraz polaczenie
}

void evh_bearer( wip_bearer_t b, s8 event, void *ctx)//--
```

Listing 3. c.d.

```
{
        ascii IpAddr[16];
        wip_in_addr_t appIpAddr;
        ascii string[100];
        s8 bearer_ans;

        switch(event)
        {
                case  WIP_BEV_IP_CONNECTED:
                        adl_atSendResponse ( ADL_AT_RSP, "WIP: connected OK: jestem w evh_bearer\r\n");
                        wip_bearerGetOpts(b, WIP_BOPT_IP_ADDR, &appIpAddr, WIP_BOPT_END);
                        wip_inet_ntoa(appIpAddr , IpAddr, 16);
                        wm_sprintf(string,"WIP: IP %s\r\n",IpAddr);
                        adl_atSendResponse ( ADL_AT_RSP, string);
                        H_Uart2 = adl_fcmSubscribe(ADL_PORT_UART2, Uart2_fun, Uart2_data);
                        client = wip_TCPClientCreateOpts("172.29.0.51",1000, ClientHandler, NULL, WIP_COPT_FINALIZER,
finalizer,WIP_COPT_END);
                        break;
                case WIP_BEV_IP_DISCONNECTED:
                        adl_atSendResponse ( ADL_AT_UNS, "WIP: disconnected\r\n");
                        break;
                case WIP_BEV_CONN_FAILED:
                        adl_atSendResponse ( ADL_AT_UNS, "WIP: WIP_BEV_CONN_FAILED\r\n");
                        break;
                case WIP_BEV_STOPPED:
                        bearer_ans = wip_bearerClose(bearer_handle);
                        TRACE((1,"Fun_close:wip_bearerClose = %d",bearer_ans));
                        adl_atSendStdResponse(ADL_AT_RSP,ADL_STR_OK);
                        break;
                default:
                        adl_atSendResponse ( ADL_AT_UNS, "WIP: other error\r\n");
                        break;
        }
}

void Fun_start(adl_atCmdPreParser_t * param) {

        s8 wynik;
        ascii * GPRS_APN;
        ascii * GPRS_USER;
        ascii * GPRS_PASSWORD;

        TRACE((1,"W Fun_start"));
        if (param->Type == ADL_CMD_TYPE_PARA){
                GPRS_APN = ADL_GET_PARAM ( param, 0 );
                TRACE (( 3,GPRS_APN));
                GPRS_USER = ADL_GET_PARAM ( param, 1 );
                TRACE (( 3,GPRS_USER));
```

Listing 3. c.d.

```
                GPRS_PASSWORD = ADL_GET_PARAM ( param, 2 );
                TRACE (( 3,GPRS_PASSWORD));
        wynik=wip_bearerOpen( &bearer_handle, "GPRS", evh_bearer, NULL);
                TRACE((1,"Fun_start:wip_bearerOpen = %d",wynik));
                wynik=wip_bearerSetOpts( bearer_handle,
                        WIP_BOPT_GPRS_APN, GPRS_APN,
                        WIP_BOPT_LOGIN,    GPRS_USER,
                        WIP_BOPT_PASSWORD,  GPRS_PASSWORD, WIP_BOPT_END);
                TRACE((1,"Fun_start:wip_bearerSetOpts = %d",wynik));
                wynik=wip_bearerStart( bearer_handle);
                TRACE((1,"Fun_start:wip_bearerStart = %d",wynik));
        adl_atSendStdResponse(ADL_AT_RSP,ADL_STR_OK);
        }
}

void Fun_close(adl_atCmdPreParser_t * paras)
{
        s8 wynik;
        wynik = wip_bearerStop( bearer_handle);
        wynik = wip_close(client);
        TRACE((1,"Fun_close:wip_close = %d",wynik));
        adl_fcmUnsubscribe ( H_Uart2 );
}

void adl_main ( adl_InitType_e InitType )
{
        TRACE (( 1, "Embedded Application : Main" ));
        wip_netInit();
        adl_atCmdSubscribe("AT+START", Fun_start, ADL_CMD_TYPE_PARA | 0x0031);
        adl_atCmdSubscribe("AT+CLOSE", Fun_close, ADL_CMD_TYPE_ACT);
}
```

Aby przetestować działanie aplikacji, należy podłączyć się do UART2 poprzez np. Hyper Terminal. Każdy znak wysłany na port UART2 zostanie odebrany przez aplikację, a następnie zostanie wysłane powiadomienie na UART1.

Internetowa biblioteka WIP

Jak wspomniałem wcześniej, mechanizmu FCM można również użyć do obsługi strumieni CSD oraz GPRS. Jeśli w przypadku CSD użycie mechanizmu FCM wydaje się zasadne, to dla GPRS byłoby o wiele bardziej skomplikowane i wiązałoby się na przykład z koniecznością samodzielnego formowania ramek TCP/IP. W takiej sytuacji z pomocą przychodzi dostępna w środowisku biblioteka internetowa WIP. Pozwala ona obsługę protokołów DHCP, NAT, ICMP, TCP, UDP, FTP, HTTP, POP3, SMTP, SNMP oraz na wysyłanie MMS-ów. Biblioteka jest dostępna w IDE w postaci pluginu o nazwie WIP i może być dołączona do projektu na etapie jego tworzenia (za pomocą kreatora) lub w dowolnym momencie, gdy zdecydujemy, że jest ona potrzebna (*Project –> Properties –> Open AT Application –> Plugin*). Opis biblioteki, w którym znajdziemy m.in. opis funkcji API oferowanych przez WIP, znajdziemy w dokumentacji *M2M Studio* (*Help –> Help Contents –> Plug-ins Documentation –> WIP Open AT Plug-in Package*).

Na **listingu 2** zamieszczono przykładowy program wykorzystujący bibliotekę WIP. Jej zadaniem jest nawiązanie połączenia GPRS przy użyciu biblioteki WIP.

Aplikacja tworzy dwie komendy AT – jedną do nawiązywania sesji GPRS (AT+START), a drugą do jej zamykania (AT+CLOSE). Komenda AT+START jako argumenty przyjmuje nazwę APN oraz opcjonalne login i hasło. Każdorazowo po nawiązaniu połączenia z APN wyświetlany jest adres IP przydzielony karcie SIM.

Zauważmy że w funkcji *adl_main()*, oprócz funkcji tworzących komendy AT, znajduje się również funkcja *wip_netInit()*, która jest niezbędna do zainicjowania pracy biblioteki WIP. Jest to standardowy sposób inicjalizacji biblioteki z parametrami domyślnymi. Gdy chcemy zmienić niektóre z parametrów standardowych (np. maksymalną liczbę otwartych socketów), to wykorzystujemy funkcję *wip_netInitOpts()*. Obowiązkowa jest również dyrektywa *#include "wip.h"* dołączająca nagłówek biblioteki WIP do aplikacji.

Wewnątrz funkcji *Fun_start()* umieszczono sekwencję zestawiającą połączenie GPRS. Zdarzenia z tym związane są przekazywane funkcji wskazanej poprzez *wip_bearerOpen()*, czyli w naszym przypadku jest to *evh_bearer()*. Gdy aplikacja otrzyma zdarzenie *WIP_BEV_IP_CONNECTED* to oznacza, że połączenie zostało nawiązane. To właśnie w sekcji pod tym zdarzeniem można umieścić funkcję, która na przykład otworzy socket TCP lub rozpocznie otwieranie połączenia FTP.

Na **listingu 3** umieszczono aplikację, która jest modyfikacją poprzedniej. Jej zadaniem jest nawiązanie połączenia GPRS, a następnie połączenia TCP Client do zadanego adresu IP. Równolegle otwarty zostaje UART2 w trybie transmisji danych (*data mode*). Komunikacji przez UART2 jest dwukierunkowa.

W porównaniu programu z programem z listingu 2, po otrzymaniu zdarzenia *IP_BEV_IP_CONNECTED* nawiązywane jest połączenie *TCP Client* za pomocą funkcji *wip_TCPClientCreateOpts()*. Jako argumenty podawane są: adres IP hosta, numer portu, a także funkcja *ClientHandler()*, która zostanie wywołana, gdy wystąpi jakieś zdarzenie związane z tym połączeniem. Dodatkowo jako opcja podana jest funkcja finalizer(). Funkcja ta zostanie wywołana przez system operacyjny w momencie, gdy po użyciu funkcji *wip_close()* zostaną zwolnione wszystkie wcześniej przydzielone temu połączeniu zasoby. Zatem po wydaniu komendy *AT+CLOSE* najpierw zostanie zamknięty socket TCP. Dopiero po jego całkowitym zamknięciu funkcja *finalizer()* zacznie zamykać połączenie GPRS.

Przyjrzyjmy się jeszcze zdarzeniom, które są obsługiwane przez funkcję *ClientHandler()*. Zdarzenie *WIP_CEV_OPEN* zachodzi zaraz po tym, jak zostanie nawiązane połączenie TCP z hostem. Prawie równocześnie zachodzi zdarzenie *WIP_CEV_WRITE* oznaczające, że bufor nadawczy jest gotowy do przyjmowania danych. To zdarzenie zajdzie też w sytuacji, gdy wysyłamy dużą ilość danych szybciej, niż łącze GPRS jest w stanie je wysłać. Wtedy bufor zapełni się i funkcja zapisu *wip_write()* jako ilość wysłanych danych zwróci 0. Oznacza to, że bufor jest pełny i z dalszym wysyłaniem musimy się wstrzymać do kolejnego wystąpienia *WIP_CEV_WRITE*.

Zdarzenie WIP_CEV_READ zajdzie w momencie, gdy w buforze odbiorczym znajdą się dane do odczytu. To zdarzenie będzie się pojawiało za każdym razem, gdy modem odbierze nowe dane poprzez GPRS. Zdarzenie *WIP_CEV_PEER_CLOSE* oznacza, że nasze polaczenie zostało zamknięte przez drugą stronę.

W celu zapoznania się jak efektywnie obsługiwać powyższe zdarzenia w przypadku, gdy przesyłamy duże ilości danych, proponuję zapoznać się z aplikacją **tcp_client** dostępną jako aplikacja przykładowa w środowisku M2M Studio.

Aplikacja serwera TCP różniłaby się od zaprezentowanej jedynie komendą otwierającą socket. Zamiast *wip_TCPClientCreateOpts()*, należałoby użyć funkcji *wip_TCPServerCreate()*. Utworzenie takiej aplikacji polecam jako temat samodzielnego treningu. Oczywiście można się wspomóc aplikacją **tcp_serwer**, którą producent dołączył do IDE jako przykład programu użytkowego.

Adrian Chrzanowski
Acte Sp. z o.o.