

Technologia GSM w elektronice (9)

Open AT – Wielowątkowość, mechanizmy synchronizacji wątków



Po krótkiej przerwie kontynuujemy kurs dotyczący programowania modemów Sierra Wireless Air Prime, jednak teraz będziemy poruszać zagadnienia związane z zaawansowanymi technikami programowania. W tym artykule zajmiemy się zagadnieniem wielowątkowości i mechanizmów z tym powiązanych.

Wszystkie dotychczas prezentowane przez nas aplikacje składały się tylko z jednego wątku. Punktem wejściowym takiej aplikacji była zawsze funkcja `adl_main()`. Przy implementacji bardziej złożonych zagadnień programistycznych może istnieć potrzeba rozdzielenia zadań realizowanych przez aplikację na kilka wątków. System operacyjny Open AT pracujący w modułach i modemach Sierra Wireless Air Prime pozwala na uruchamianie aplikacji składających się z maksymalnie 64 wątków. Jej punktem wejściowym jest tablica inicjalizująca poszczególne wątki. Każda kolejna linia tablicy

odpowiada za opis jednego wątku. Koniec tablicy oznaczamy linią wypełnioną wartościami 0. Sposób tworzenia tablicy najlepiej będzie przedstawić na przykładzie – **listing 1**.

Jak widzimy na przedstawionym przykładzie tablica inicjalizująca jest typu `adl_InitTasks_t`. Deklaracja struktury wygląda następująco:

```
typedef struct
{
    void    (* EntryPoint) (void);
    u32    StackSize;
    const  ascii* Name;
    u8     Priority;
```

Dodatkowe materiały na CD i FTP:
<ftp://ep.com.pl>, user: 10142, pass: 5x7bu87r
 • poprzednie części kursu

```
} adl_InitTasks_t;
```

Znaczenie kolumn jest następujące:

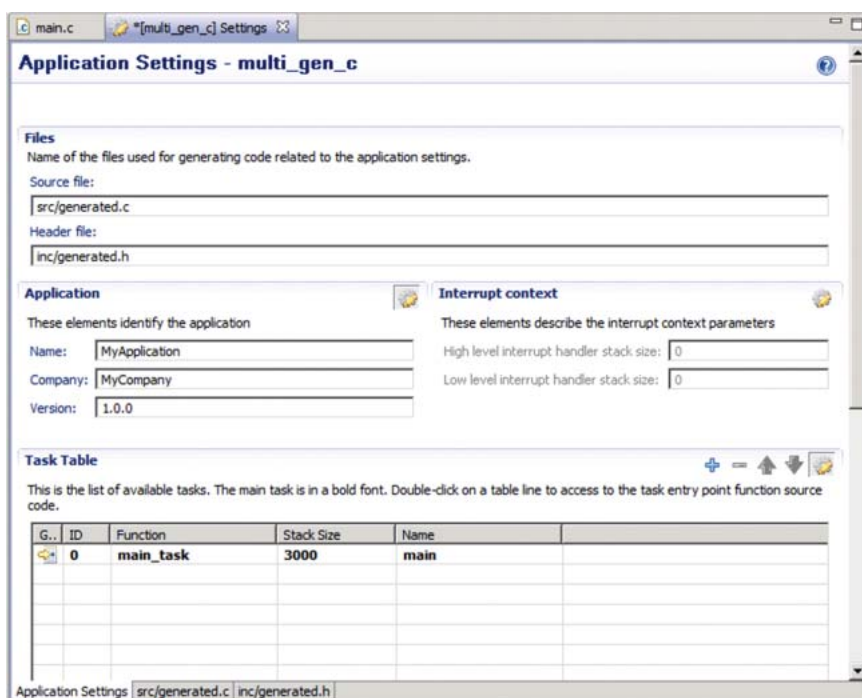
- Entry Point – funkcja wejściowa dla danego wątku, wykonywana każdorazowo po starcie według określonego priorytetu.
- Stack Size – rozmiar stosu dla danego wątku.
- Name – nazwa wewnętrzna wątku wykorzystywana do Trace i Error Service.
- Priority – priorytet danego wątku. Jest to liczba od 1 do wartości równej liczbie wątków. Im większa liczba, tym wyższy priorytet.

W najnowszej wersji Developer Studio proces tworzenia nowych wątków został zautomatyzowany. Po utworzeniu nowego projektu odszukujemy plik `generate.c` naszego projektu w oknie *Project Explorer*. Po kliknięciu na ten plik w oknie głównym pojawi nam się formatka pokazana na **rysunku 1**.

Oprócz informacji o wersji i autorze, które możemy zapisać w kodzie naszej aplikacji jest też możliwość tworzenia nowych wątków, a także zarządzanie nimi. Naciskając przycisk Add Task (symbol +) mamy możliwość wskazania jako funkcji wejściowej istniejącej już funkcji lub utworzenie nowej (**rysunek 2**).

Jeśli korzystamy ze wspomnianego kreatora, to tablica inicjalizująca jest umieszczona właśnie w pliku `generated.c`. Warto mieć to na uwadze analizując przykłady umieszczone w dalszej części artykułu.

Wszystkie wątki w aplikacji są tworzone statycznie i nie ma możliwości dynamicznego tworzenia wątków. Każdy wątek ma swój numer ID nadawany od 0 (pierwszy w kolejności wątek w tablicy wątków) do wartości równej ilości wątków pomniejszonej o 1. Identyfikator każdego wątku można sprawdzić za pomocą funkcji `adl_ctxGetID (void)`, a całkowitą ilość aktywnych wątków za pomocą `adl_ctxGetTa-`



Rysunek 1. Formatka tworzenia nowych wątków

Listing 1. Sposób tworzenia tablicy inicjalizującej wątki w OpenAT

```
#include "adl_global.h"

void MainTask ( void );
void SubTask1 ( void );

const adl_InitTasks_t adl_InitTasks [] = // Tablica inicjalizująca
{
  { MainTask, 3*1024, "MAIN", 2 }, // Main task
  { SubTask1, 3*1024, "SUB1", 1 }, // Sub task 1
  { NULL, 0, NULL, 0 }
};

void MainTask ( void )
{
  TRACE (( 1, "Multitasking - Main task" ));
  adl_atSendResponse ( ADL_AT_UNS, "\r\nHello from Main Task\r\n" );
}

void SubTask1 ( void )
{
  TRACE (( 1, "Multitasking - Sub task 1" ));
  adl_atSendResponse ( ADL_AT_UNS, "\r\n Hello from Sub Task1\r\n" );
}
```

Listing 2. Przykładowy program wykorzystujący mechanizm komunikatów

```
#include „adl_global.h”
#include "generated.h"

const adl_msgFilter_t MyFilter = {0, 0, ADL_MSG_ID_COMP_EQUAL, ADL_CTX_
ALL};
ascii * pozdrowienia = „tekst z main task”;

void main_task ( void )
{
  TRACE (( 1, "Task id %d", adl_ctxGetID()));
  adl_msgSend ( 1, 0, strlen(pozdrowienia)+1, (void*) pozdrowienia );
}

void MsgHandler_t2 ( u32 MsgIdentifier, adl_ctxID_e Source, u32 Length, void *
Data )
{
  ascii response [50]= {0};
  TRACE (( 1, "MsgHandler_t2, Task id %d", adl_ctxGetID()));
  wm_sprintf(response, "message from task: %d \r\nTresc:", Source);
  wm_strcat(response, Data);
  adl_atSendResponse ( ADL_AT_UNS, response);
}

void drugi_task ( void )
{
  TRACE (( 1, "Task id %d", adl_ctxGetID()));
  // Subscribe to Timer service
  s32 MyMsgHandle = adl_msgSubscribe ( &MyFilter, MsgHandler_t2 );
}
```

sksCount(). Wskazany wątek (grupa wątków) może być wstrzymany za pomocą funkcji *adl_ctxSuspend()* (grupa - *adl_ctxSuspendExt()*). Odwieszenie działania uzyskujemy za pomocą funkcji *adl_ctxResume()* (grupa - *adl_ctxResumeExt()*). Istnieje również funkcja - *adl_ctxSleep()*, która pozwala uśpić bieżący wątek na pewien czas pozwalając innym wątkom (o niższym priorytecie) na wykonanie swoich zadań. W dowolnej chwili możemy również sprawdzić stan wybranego wątku przy pomocy funkcji *adl_ctxGetState()*. Funkcja ta zwraca stan wątku, który może być opisany wartością typu *adl_ctxState_e*. Wygląda ona następująco:

```
typedef enum _adl_ctxState_e
{
  ADL_CTX_STATE_ACTIVE,
  ADL_CTX_STATE_WAIT_EVENT,
  ADL_CTX_STATE_WAIT_SEMAPHORE,
  ADL_CTX_STATE_WAIT_INNER_EVENT,
  ADL_CTX_STATE_SLEEPING,
  ADL_CTX_STATE_READY,
  ADL_CTX_STATE_PREEMPTED,
  ADL_CTX_STATE_SUSPENDED
} adl_ctxState_e;
```

Wątek może znajdować się w jednym z wymienionych niżej stanów:

- ADL_CTX_STATE_ACTIVE – wątek jest w danej chwili aktywny,
- ADL_CTX_STATE_WAIT_EVENT – wątek czeka na zdarzenie (np. SIM event, SMS lub inne); obecnie nie ma nic do przetworzenia,
- ADL_CTX_STATE_WAIT_SEMAPHORE – wątek czeka na zwolnienie semafora,
- ADL_CTX_STATE_WAIT_INNER_EVENT – wątek zamrożony, oczekujący na wygenerowanie odpowiedniego zdarzenia (event),
- ADL_CTX_STATE_SLEEPING – wątek uśpiony przy pomocy funkcji *adl_ctxSleep()*,
- ADL_CTX_STATE_READY – wątek ma zdarzenia do przetworzenia jednak obecnie wykonywany jest wątek o wyższym priorytecie,
- ADL_CTX_STATE_PREEMPTED – wątek podczas przetwarzania otrzymanego zdarzenia został wywłaszczony przez wątek o wyższym priorytecie,
- ADL_CTX_STATE_SUSPENDED – wykonywanie wątku zostało wstrzymane za pomocą funkcji *adl_ctxSuspend()*.

Jeśli nasza aplikacja składa się z kilku wątków możemy potrzebować sposobu na wymia-

nę danych między wątkami. Rozwiązaniem jest wykorzystanie mechanizmu komunikatów. Aby wątek mógł odbierać komunikaty musimy użyć funkcji subskrybującej, wskazując przy tym na funkcję odbierającą komunikaty. Przy subskrypcji wykorzystany jest filtr składający się z identyfikatora, maski oraz komparatora. W ten sposób można sprawić aby dany wątek odbierał tylko określone komunikaty, ignorując pozostałe. Taki system pozwala na stworzenie kategorii komunikatów, gdy w aplikacji przesyłanych jest wiele komunikatów i odbieranie tylko tych, które są przeznaczone dla danego wątku. Przykładowy program wykorzystujący mechanizm komunikatów przedstawiono na **listingu 2**.

W przykładzie z listingu 2, zarówno identyfikator jak i maska są równe 0, więc wątek odbiera wszystkie komunikaty. Jeśli weźmiemy pod uwagę następujący przykład:

Maska=0x0000F000

Identyfikator=0x00003000

Komparator=ADL_MSG_ID_COMP_EQUAL

Źródło=ADL_CTX_ALL

to wątek słuchający komunikatów selekcyonowanych przez tak zadeklarowany filtr odbierze tylko te, które będą miały wartość 3 na czwartym oktecie identyfikatora (0xXXXX3XXX). Źródłem pochodzenia w tym przypadku mogą być wszystkie wątki.

W przypadku programowania wielowątkowego pojawia się również zagadnienie kontroli dostępu przez wiele procesów do wspólnego zasobu. Przykładowo możemy wyobrazić sobie tablicę danych, do której dostęp ma kilka wątków. Może się zdarzyć, że wątek o niższym priorytecie zacznie zapisywać tam jakieś dane, a w połowie tej czynności zostanie wywłaszczony przez wątek o wyższym priorytecie, chcący tę tablicę odczytać. W takim przypadku najlepiej skorzystać z pomocy semafora. Przykładowe rozwiązanie pokazano na **listingu 3**.

W przypadku gdy, *third_task()* zacznie wykonywać operacje na obiekcie *tablica*, to nie zostanie ona przerwana przez wątek o wyższym priorytecie – *second_task()* do momentu zwrócenia semafora przez *third_task()*. Do tego czasu działanie *second_task()* będzie zawieszona.

Kolejnym przydatnym mechanizmem, którego przykład użycia umieszczono na **listingu 4**, są *eventy* (zdarzenia). Poprzez wykorzystanie funkcji *adl_eventWait()* możemy nakazać wątkowi wstrzymanie swojego działania do momentu spełnienia określonych warunków, czyli zaistnienia pewnych zdarzeń (*events*). Pozwala to nam na synchronizację działania wątków. Jeśli chcemy skorzystać z tego mechanizmu, najpierw stworzymy *event* za pomocą funkcji *adl_eventCreate()* podając jako argument 32-bitową wartość, którą *event* ma przyjąć po zainicjalizowaniu. Funkcja czekająca na *event* - *adl_eventWait()* – jest wywoływana z maską, gdzie jedynie na odpowiednich pozycjach maski oznaczają, że na jedynek na tych pozycjach *eventu* czekamy. Oczywiście, można też zaznaczyć, że

czekamy do momentu, gdy jedynka pojawi się na którejkolwiek z oczekiwanych pozycji eventu (*ADL_EVENT_WAIT_ANY*). Jeśli natomiast wybrana opcja to *ADL_EVENT_WAIT_ALL* to wszystkie jedynki eventu muszą pokrywać się z jedynkami określonymi w masce funkcji *adl_eventWait()*. Do kasowania i ustawiania eventu służą odpowiednio funkcje *adl_eventClear()* i *adl_eventSet()*.

W przedstawionej na listingu 4 aplikacji komenda AT+START ustawia *event* pozwalając na działanie drugiego wątku. Komenda AT+STOP kasuje *event* powodując wstrzymanie funkcji *callback* dla timera z wątku drugiego.

Choć mechanizmy semaforów i *eventów* wydają się do siebie podobne, to warto jednak zwrócić uwagę na pewne różnice:

- semafora nie można wygenerować przed jego pobraniem, natomiast w przypadku *eventów* jest dozwolona dowolna kolejność użycia funkcji *adl_eventWait/adl_eventSet* oraz *adl_eventClear*
- jeśli więcej niż jeden wątek czeka na semafor, to w momencie jego zwrócenia tylko wątek o najwyższym priorytecie wśród czekających wznowia swoje działanie – reszta oczekuje; w przypadku *eventu*, jeśli kilka wątków czeka na niego, to w momencie wystąpienia wszystkie wątki wznowiają działanie.

Na koniec należy wspomnieć o pewnym ograniczeniu dotyczącym programowania wielowątkowego. Wszystkie funkcje zdarzeń z wyłączeniem timerów oraz *Message events* są wywoływane zawsze jako kontekst wątku o najwyższym priorytecie. Dla przykładu, nawet jeżeli w zadaniu (*task*) o ID 2 zasubskrybujemy się do karty SIM, to funkcja *SIMHandler* zostanie wywołana z kontekstem ID 0.

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.

Adrian Chrzanowski
Acte Sp. z o.o.

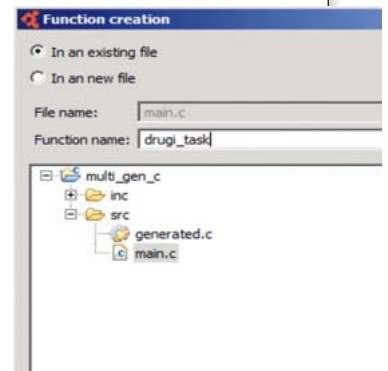
Listing 3. Przykład kontroli dostępu do wspólnego zasobu

```
ascii tablica [100] = {0};
s32 MySemHandle;

void main_task ( void )
{
    //gdzieś w programie
    MySemHandle = adl_semSubscribe ( 1 );
}

void third_task ( void )
{
    //gdzieś w programie
    adl_semConsume ( MySemHandle );
    zapisz ( tablica );
    adl_semProduce ( MySemHandle );
}

void second_task ( void )
{
    //gdzieś w programie
    adl_semConsume ( MySemHandle );
    odczytaj ( tablica );
    adl_semProduce ( MySemHandle );
}
```



Rysunek 2. Tworzenie funkcji wejściowej dla wątku

Listing 4. Przykładowy program wykorzystujący mechanizm zdarzeń (*events*).

```
#include "adl_global.h"
#include "generated.h"

static u32 MyEvent = 0;

void Fun_start(adl_atCmdPreParser_t * paras) {
    TRACE((1, "Fun_start"));
    adl_eventSet(MyEvent, 1); //ustaw msb
    adl_atSendStdResponse(ADL_AT_RSP, ADL_STR_OK);
}

void Fun_stop(adl_atCmdPreParser_t * paras) {
    TRACE((1, "Fun_stop"));
    adl_eventClear(MyEvent, 1, NULL); //skasuj msb
    adl_atSendStdResponse(ADL_AT_RSP, ADL_STR_OK);
}

void TimerHandler ( u8 ID )
{
    TRACE((1, "TimerHandler"));
    adl_eventWait(MyEvent, 1, NULL, ADL_EVENT_WAIT_ANY, ADL_EVENT_NO_TIMEOUT); //only msb wait
    adl_atSendResponse ( ADL_AT_UNSP, "\r\n Hello from SubTask1 TimerHandler \r\n" );
}

void main_task ( void )
{
    TRACE (( 1, "Task id %d", adl_ctxGetID()));

    adl_atSendResponse ( ADL_AT_UNSP, "\r\nHello from MainTask\r\n" );
    adl_atCmdSubscribe ("AT+START", Fun_start, ADL_CMD_TYPE_ACT);
    adl_atCmdSubscribe ("AT+STOP", Fun_stop, ADL_CMD_TYPE_ACT);
    MyEvent = adl_eventCreate(0);
}

void drugi_task ( void )
{
    // TODO Insert your task initialization code here
    TRACE (( 1, "Task id %d", adl_ctxGetID()));
    // Subscribe to Timer service
    adl_atSendResponse ( ADL_AT_UNSP, "\r\n Hello from task 2\r\n" );
    adl_tmrsSubscribe ( TRUE, 20, ADL_TMR_TYPE_100MS, TimerHandler );
}
```

R E K L A M A
www.conrad.pl

Zamówienia telefoniczne 801 005 133* lub (12) 376 00 22

Produkty w super cenach!

Cyfrowa stacja lutownicza Toolcraft ST50-D 50W

W zestawie grot w kształcie ołówka 0,2 x 25 mm

3 zdefiniowane temperatury grotu

Moc 50 W

Cena regularna 343,99

189,00

Taniej o 45%

<p>Stacja lutownicza Weller WTCP 51</p>  <p>Weller</p> <p>Nr prod. 81807 824,99 599,00</p>	<p>Szczytki do rozlutowywania SMD</p>  <p>Nr prod. 81875 120,99 106,89</p>	<p>Zestaw do lutowania Toolcraft z akcesoriami w walizce</p>  <p>TOOLCRAFT</p> <p>Nr prod. 58827 89,00 59,90</p>
<p>Multymetr cyfrowy Extech EX310</p>  <p>EXTECH</p> <p>Nr prod. 62232 346,33 84,68</p>	<p>Multymetr cyfrowy Voltcraft VC250</p>  <p>VOLTCRAFT</p> <p>Nr prod. 62401 152,42 109,45</p>	<p>Zestaw precyzyjnych narzędzi do prac elektronicznych i mechanicznych, 30 szt.</p>  <p>BASETech</p> <p>Nr prod. 61482 106,99 79,99</p>