Zarządzanie zadaniami w systemach embedded

Super Simple Tasker, część 2

Dla tych, którzy po raz pierwszy zetknęli się z systemami operacyjnymi typu embedded, jak na przykład µC/OS czy eCos, sposób ich działania może wydać się skomplikowany, sztuczny i mało intuicyjny. W miarę nabierania doświadczenia odczucie to wprawdzie słabnie, nadal pozostaje jednak pewna nieufność. W niniejszym artykule przedstawiamy prostsze rozwiązania alternatywne pozwalające zarządzać zadaniami w systemie embedded.

W pierwszym odcinku artykułu zostały wyjaśnione ogólne zasady zarządzania zadaniami w systemach *embedded*. Obecnie skoncentrujemy się na realizacji takiego systemu w oparciu o Super Simple Tasker.

Zalety SST

SST zużywa mniej zasobów mikrokontrolera w porównaniu z typowymi implementacjami systemów operacyjnych. Można to zaobserwować szczególnie w przypadku pamięci RAM używanej na potrzeby stosu. W systemach klasycznych każde z zadań ma własny stos i przypisany do niego obszar RAM–u. Nie można jednak łatwo przewidzieć, jak głębokiego stosu należy użyć dla każdego z zadań, więc jego wielkość jest przyjmowana z pewnym zapasem, a co za tym idzie – jego część będzie zawsze niewykorzystana. Jeżeli takie obszary przemnoży się przez liczbę zadań, wówczas może się okazać, że otrzymany wynik będzie stanowił całkiem istotną część RAM–u. Taki obszar mógłby być zagospodarowany w lepszy sposób (np. przez implementację szybszego algorytmu, mającego większe zapotrzebowanie na pamięć). Zamiast tego można by też użyć tańszego mikrokontrolera.

W SST zastosowano tylko jeden (ciągły) stos. Są na nim odkładane tylko te dane, które kompilator uzna za stosowne, można więc mówić o pełnej optymalizacji. Jako przykład można przytoczyć wersję SST opracowaną na mikrokontrolery typu ARM. Każdy kompilator C generujący kod dla tych procesorów stosuje się do ARM–owego standardu wywołania funkcji (*ARM procedure call standard* – APCS), opisującego m.in., które rejestry muszą być zawsze zapamiętane na stosie (przed rozpoczęciem kodu funkcji), a które mogą być nadpisane bez zapamiętywania. Podczas generowania kodu ISR przez kompilator wystarczy, że na początku kodu obsługi przerwania procesor zapamięta tylko te rejestry, które mogą być nadpisywane przez funkcje C – stanowią one tylko połowę rejestrów ARM–a. Pozostałe rejestry mogą być zapamiętane później, i to tylko wtedy, gdy będą potrzebne w danej funkcji (wywoływanej z ISR). Klasyczny OS musi w takim przypadku zapamiętać wszystkie rejestry. Co więcej, jeżeli w ISR będą wywoływane jakieś funkcje, ich uruchomienie także spowoduje kolejne zapisywanie rejestrów.

SST może być bardziej oszczędny również dlatego, że ze względu na swoją prostotę używa mniejszych struktur danych odpowiedzialnych za przechowywanie danych o zadaniach (*task control block* – TCB), a co za tym idzie, manipulacja tymi danymi wymaga mniej zasobów mikrokontrolera.

Sama prostota budowy SST także daje określone korzyści. Po pierwsze, kod napisany z użyciem takiego *schedulera* uruchamia się o wiele prościej. Dzieje się tak, gdyż w trakcie pracy systemu przez cały czas jest używany tylko jeden stos dostępny dla *debuggera* (dzięki temu nie trzeba używać specjalizowanych *debuggerów*). Łatwiej znajdować błędy również dlatego, że działanie całego systemu jest prostsze do zrozumienia, można więc skupić się tylko na aspektach istotnych dla konkretnej aplikacji.

Przykład użycia i implementacja SST

Warto zaznaczyć, że prezentowana, przykładowa (ale w pełni funkcjonalna i działająca) implementacja SST jest tylko jedną z wielu możliwych. W szczególności można łatwo wprowadzić dalsze uproszczenia, np. ograniczenie liczby zadań do 8 i przyjęcie tylko jednego zadania z danym priorytetem. W wyniku tego powstałby o wiele mniejszy kod wynikowy i dałoby się osiągnąć jeszcze mniejsze zużycie zasobów. Możliwa jest też modyfikacja odwrotna, tj. przystosowanie *schedulera* do obsługi większej liczby zadań (prezentowana implementacja listy zadań gotowych do wykonania staje się bardzo nieefektywna przy dużej liczbie zdań). Wszystko zależy od konkretnych potrzeb realizowanej aplikacji.

Przykłady kodu prezentowane w artykule zostały napisane i uruchomione z użyciem kompilatora GNU GCC 4.1.1, ale nie zależą od jego wersji ani specyfiki. Do działania z innym kompilatorem wystarczy odpowiednio zadeklarować funkcje obsługi przerwań i użyć odpowiedniej składni wstawek asemblerowych.

System – w tym przypadku funkcja *main* – startuje z wyłączonymi przerwaniami, a jej zawartość to jedynie wywołanie funkcji (makra) SST_Run (**list. 3**). W trakcie wykonania SST_Run przeprowadzana jest inicjalizacja podsystemu SST i wywoływana jest funkcja użytkownika o nazwie SST_Start, która ma na celu zainicjowanie pracy reszty systemu (w tym zadań uruchamianych przez SST i przerwań). Na koniec następuje przełączenie do „pustego” (*idle*) zadania SST_OnIdle o najniższym możliwym priorytecie, które jest wykonywane w nieskończonej pętli (jedynej w całym kodzie). Zadanie to nie jest wyjątkiem i ma postać zwykłej funkcji bez nieskończonej pętli. Umieszcza się tu zwykle instrukcję przełączającą procesor w tryb o zmniejszonym poborze

List. 3. Inicjalizacja SST

```
#define SST_Run()    \
 SST_Init();         \
 for (;;)            \
 {                   \
   SST_OnIdle();     \
 }

void _SST_Init(void)
{
 SST_DECLARE_INT_USAGE;
 hp_task = SST_BAD_TASK_ID;                    1
 SST_Start();                                  2
 SST_INT_LOCK_AND_SAVE();                      3
 _SST_CurrPrio = SST_IDLE_TASK_
 PRIO;                                         4
 _SST_Schedule(SST_INT_STATE);                 5
 SST_INT_LOCK_RESTORE();                       6
}
```

mocy (komendy takie często zawieszają wykonywanie rozkazów, aż do nadejścia przerwania).

Odnośniki na list. 3 oznaczają:

1. Ustaw początek listy zadań oczekujących na wykonanie, na zadanie nieistniejące, co jest równoznaczne z wyczyszczeniem listy.
2. Wywołaj funkcję (inicjalizacyjną) użytkownika, która m.in. odblokowuje przerwania.
3. Zablokuj przerwania.
4. Ustaw priorytet aktualnie wykonywanego zadania na najmniejszy z możliwych.
5. Wywołaj *scheduler* dla zadań, które mogły w tym czasie znaleźć się na liście oczekujących na wykonanie.
6. Odblokuj przerwania.

W przykładzie przedstawionym na **list. 4** oprócz inicjalizacji specyficznej dla mikrokontrolera (UART, *timery*, I/O itp.) rejestrowane są w systemie 3 zadania. Pierwsze z nich (timer1_task_id) służy do „mrugania” LED–em. Drugie (btn_task_id) odpowiada za obsługę naciśnięcia przycisku i za wystartowanie lub zatrzymanie zadania timer1_task_id. Trzecie zadanie natomiast (interp_task_id) odczytuje dane z portu szeregowego i interpretuje je (w uproszczony sposób) jako komendy (modemu) AT. Odnośniki z list. 4 skomentowano poniżej:

1. Jeżeli wartość otrzymanego sygnału jest różna od zera...
2. To potraktuj jego parametr jako wartość flagi pozwalającej na sterowanie LED–em.
3. Wyślij do zadania timer1_task_id komunikat z wartością sygnału równą 1 i parametrem o wartości flagi t_enabled.

List. 4. Inicjalizacja SST z trzema zadaniami

```
void SST_Start(void)
{
 ioinit();
 timer1_task_id = SST_CreateTask(TIMER1_TASK_PRIO, timer1_task_func);
 btn_task_id = SST_CreateTask(KBD_TASK_PRIO, btn_task_func);
 interp_task_id = SST_CreateTask(INTERPETER_TASK_PRIO, interpreter_task_
func);
 clear_buf(&tx_buf);
 SST_INT_UNLOCK();
}

void timer1_task_func(SST_Event_T e)
{
 static is_enabled = TRUE;

 if (e.sig)                                    1
 {
   is_enabled = e.par;                         2
   LED_OFF(LED1);
 }
 else if (is_enabled)
 {
   LED_TOGGLE(LED1);
 }
 else
 {
 }
}

void btn_task_func(SST_Event_T e)
{
 static CPU_Base_T counter;
 static bool t_enabled = TRUE;

 if (IS_BTN_PRESS())
 {
   if (counter < DEBOUNCE_TIME)
   {
     counter++;
     if (DEBOUNCE_TIME == counter)
     {
       t_enabled = !t_enabled;
       SST_PostEvent(timer1_task_id, 1, t_enabled);   3
     }
   }
 }
 else
 {
 counter = 0;
 }
}
```

List. 5. Funkcje pomocnicze do wysyłania danych przez port szeregowy

```
Static void printchar(uint8_t
byte)
{
 SST_Wait_Busy_While(IS_BUF_
FULL(tx_buf));                                 1
 put_buf(&tx_buf, byte);                       2
}

static void print(const char
*str)
{
 while (*str)
 {
   SST_Wait_Busy_While(IS_BUF_
FULL(tx_buf));
   put_buf(&tx_buf, *str);
   str++;
 }
}
```

Na **list. 5** przedstawiono funkcje pomocnicze służące do wysyłania danych przez port szeregowy z użyciem bufora tx_buf. Zaznaczone linie oznaczają:

1. Czekaj (nic nie rób lub uruchom zadania o wyższym priorytecie) dopóki bufor jest pełny.
2. Włóż bajt do bufora.

Zadanie interpretujące odebrane znaki jako komendy (modemu) typu AT przedstawiono na **list. 6.** Objaśnienia do niego są następujące:

1. Wyślij zwrotnie odebrany znak (*echo*).
2. Jeżeli jest to „Enter” (*cariage return* – CR)...
3. To wyślij znak nowej linii (*line feed* – LF).

Kod procedur obsługi przerwań (w tym przypadku dla LPC2114) *timera* (*timer 1 compare match 0*) i UART–a (UART 0, kolejka sprzętowa o długości 1 bajtu) przedstawiono na **list. 7**, a objaśnienia poniżej:

1. Wejście do ISR–a – zapamiętanie kontekstu przerwanego zdania, w przypadku ARM–a jest to również przełączenie się na zestaw rejestrów specyficznych dla trybu przerwań.
2. Wyzeruj flagę żądania przerwania dla *timera* 1.
3. Wykonaj kod *SST ISR entry* – pozwól na zagnieżdżanie przerwań i odblokuj je.
4. Przenieś zadanie obsługi przycisku na listę gotowych do wykonania – funkcja SST_ScheduleTask służy tutaj do przekazania (z użyciem SST_PostEvent) „pustego” komunikatu do zadania btn_task_id w celu jego uruchomienia.

MOTOROLA and the Stylized M Logo are registered in the US Patent & Trademark Office. © Motorola, Inc. 2006.

Modem G24 jest następcą modułu G20 całkowicie zgodnym programowo z popularnymi od lat modelami G18 i G20. Nowa generacja produktów Motoroli spełnia dyrektywę RoHS. W porównaniu do swoich poprzedników G24 został wyposażony w możliwość obsługi super szybkiej transmisji danych EDGE/GPRS oraz posiada wbudowane środowisko języka JAVA pozwalające na wyjątkowo proste i autonomiczne sterowanie modułem.

Terminal RB 24 zbudowany jest w oparciu o moduł Motoroli G24. Dostępny jest w trzech wersjach: USB, Bluetooth oraz RS232. Wersja USB to modem EDGE/GPRS/GSM dedykowany dla użytkownika, który poszukuje prostego i mobilnego dostępu do internetu. Realizuje wszystkie funkcje i dostępne usługi operatorów telefonii komórkowej. Dostęp do Internetu/APN (GPRS/EDGE), funkcje modemu GSM, SMS czy FAX to podstawowe funkcje produktu. Ważnym rozszerzeniem jest możliwość wykorzystania modemu jako telefonu komórkowego. Idealny w podróży - nie wymaga zasilacza zewnętrznego ani baterii. Dostarczany jest z kompletnym oprogramowaniem dla systemów Windows XP oraz Linux. Wersja Bluetooth rozszerza powyższe funkcje o możliwość bezprzewodowego podłączenia modemu do komputera w standardzie Bluetooth. RB24-RS to typowy terminal komunikujący się z PC w standardzie RS-232. Dzięki niewielkim rozmiarom, prostej instalacji z powodzeniem stosowany jest jako sterownik GSM/GPRS we wszelkich rozwiązaniach telemetrycznych.

ul. Szymanowskiego 13, 05-092 Łomianki
tel. +48 22 751 76 80, fax +48 22 751 76 81
m2m@elproma.com.pl

ELPROMA

List. 6. Zadanie interpretujące odebrane znaki jako komendy AT

```
typedef enum {
    IS_ENTER,
    IS_A,
    IS_T,
    IS_AT,
    IS_AT_EXT,
    IS_AT_EXT_C
} Interpreter_State_T;

void interpreter_task_func(SST_Event_T e)
{
    static Interpreter_State_T state = IS_A;
    static const char *cmd = NULL;

    printchar(e.par);                                   1
    if ('\r' == e.par)                                  2
    {
        printchar('\n');                                3
    }

    switch (state)
    {
        case IS_ENTER:
            if ('\r' == e.par)
            {
              if (cmd)
              {
                print(cmd);
                cmd = NULL;
              }
              state = IS_A;
            }
            else
            {
              cmd = NULL;
            }
            break;

        case IS_A:
            if (('a' == e.par) || ('A' == e.par))
            {
             state = IS_T;
            }
            else
            {
             state = (('\r' == e.par) ? IS_A : IS_ENTER);
            }
            break;

        case IS_T:
            if (('t' == e.par) || ('T' == e.par))
            {
             state = IS_AT;
            }
            else
            {
             state = (('\r' == e.par) ? IS_A : IS_ENTER);
            }
            break;

        case IS_AT:
            switch (e.par)
            {
             case 'z':
             case 'Z':
                // exec ATZ
                cmd = "Command ATZ\n\r";
                state = IS_ENTER;
                break;
.....

        default:
            state = IS_ENTER;
            break;
    }
}
```

List. 7. Funkcje obsługi przerwań timera I UART–a

```
void T1_MatchC0_ISR(void)                           1
{
 SST_DECLARE_ISR;
 T1_IR = 0x1;                                       2
 SST_ISR_ENTRY(SST_MIN_INT_
PRIO+1);                                            3
 SST_ScheduleTask(btn_task_id);                     4
 SST_ISR_EXIT();                                    5
}                                                   6

void UART0_ISR(void)
{
 SST_DECLARE_ISR;
 uint32_t byte;

 switch (UART0_IIR & 0x0E)
 {
 case 0x06: /* RLS */
 byte = UART0_LSR;
   LED_TOGGLE(LED7);
   break;
   case 0x04: /* RX */
   case 0x0C: /* CTI */
   byte = UART0_RBR;
   LED_TOGGLE(LED6);
   SST_ISR_ENTRY(SST_MIN_INT_
    PRIO+3);
   if (0==(UART0_LSR & (1 << 7)))
   {
   SST_PostEvent(interp_task_id,
    0, byte);
   }
   SST_ISR_EXIT();
   break;
   case 0x02: /* TX */
   LED_TOGGLE(LED5);
   if (!IS_BUF_EMPTY(tx_buf))
   {
    UART0_THR = get_buf(&tx_buf);
   }

   break;
 default:
 break;
 }

 VICVectAddr = 0xff;
}
```

List. 8. Definicje makr SST_ISR_ENTRY i SST_ISR_EXIT

```
#define SST_ISR_ENTRY(isrPrio)   \
 do {                            \
  _pin_ = _SST_CurrPrio;         \   1
  _SST_CurrPrio = (isrPrio);     \   2
  SST_Start_Nested_Ints();       \   3
  } while (0);
#define SST_ISR_EXIT()           \
 do {                            \
  SST_INT_LOCK_AND_SAVE();       \   4
  _SST_CurrPrio = _pin_;         \   5
  _SST_Schedule(SST_INT_STATE);\     6
  SST_Stop_Nested_Ints();        \   7
  } while (0)
```

5. Wykonaj kod *SST ISR exit* – zablokuj przerwania i wywołaj funkcję *schedulera*.
6. Wyjście z przerwania z odtworzeniem kontekstu przerwanego zadania.

SST_ISR_ENTRY i SST_ISR_EXIT są zdefiniowane jak na **list. 8**. Odnośniki na list. 8 oznaczają:

1. Zapamiętaj priorytet przerwanego zadania (nie trzeba w tym przypadku wprost wywoływać blokowania przerwań, ponieważ kod ISR jest standardowo wykonywany z zablokowanymi przerwaniami).
2. Ustaw priorytet aktualnie wykonywanego zadania na wartość odpowiadającą przerwaniu (wyższa niż wartość każdego normalnego zadania).
3. Wywołaj kod użytkownika pozwalający na zagnieżdżanie przerwań.
4. Zablokuj przerwania.
5. Przywróć priorytet aktualnie wykonywanego zadania do wartości odpowiadającej przerwanemu (przez przerwanie) zadaniu.
6. Wywołaj funkcję *schedulera* – jeżeli na liście zadań gotowych do wykonania są zadania o priorytecie większym niż _pin_, wówczas będą one wykonane.
7. Przywróć tryb pracy przerwań.

Niestety, nie udało się pomieścić w jednym odcinku artykułu wszystkich funkcji implementujących SST, temat będziemy więc kontynuować za miesiąc.

Artur Lipowski
LAL@pro.onet.pl