OPEX – unikatowy system operacyjny dla mikrokontrolerów AVR

System operacyjny dla mikrokontrolera? Przecież to szaleństwo. Może jednak jest w tym szaleństwie jakaś metoda? System operacyjny zwalnia programistę z takich czynności, jak niskopoziomowa obsługa urządzeń peryferyjnych, ale nade wszystko dba o to, by wszystkie zadania aplikacji były realizowane prawidłowo i bez zakłóceń.

Podzielność uwagi, to cecha bardzo przydatna w życiu. Wiedzą o tym dobrze na przykład studenci, którym przyszło zdobywać wiedzę w akademickich pokoikach. Jakże często, gdy jeden współlokator pragnie przygotować się do ciężkiego kolokwium, drugi w tym samym czasie ma ochotę na słuchanie ciężkiego rock'a, nie mówiąc już o innych przyjemnościach. Kompromisowym rozwiązaniem może być np. umawianie się na okresową wyłączność dysponowania pokojem przez każdego z jego mieszkańców. Taka koncepcja jest na tyle uniwersalna, że bywa chętnie stosowana i w innych, zgoła odmiennych sytuacjach. Przyjrzyjmy się na przykład urządzeniom mikroprocesorowym. Konstrukcja wielu z nich powinna umożliwiać jednoczesną – z punktu widzenia użytkownika – realizację wielu zadań. Przydzielanie każdemu zadaniu odrębnego procesora, choć możliwe, rzadko kiedy znajduje ekonomiczne uzasadnienie. To tak, jak budowanie akademików tylko z 1–osobowymi pokojami. Problem trzeba było jednak jakoś rozwiązać, wymyślono więc systemy operacyjne, których specjalną odmianą są systemy operacyjne czasu rzeczywistego (*RTOS – Real Time Operation System*). Czy możliwe jest zatem stworzenie systemu RTOS dla małych mikrokontrolerów? No cóż. Odpowiedź będzie trochę wymijająca. I tak, i nie (z przewagą „nie"). Oczywiście wszystko zależy od tego, jakiej „wagi" będą zadania wykonywane w systemie oraz wielu innych czynników.

Czy OPEX = RTOS?

Próbę napisania systemu wielozadaniowego dla mikrokontrolerów AVR podjął Steve Childress. Efektem jego pracy jest pakiet funkcji przeznaczonych dla kompilatora AVRGCC, który można nazwać systemem operacyjnym. Zalecanym przez autora środowiskiem uruchomieniowym jest AtmanAvr. Software o nazwie OPEX (Operation Executive) na razie jest dostępny w wersji beta 1, należy więc z wyrozumiałością podchodzić do jego ewentualnych błędów i niedociągnięć. Wszystkie pliki zzipowane w jeden można pobrać ze strony *http://www.atmanecl.com/EnglishSite/opex.htm*. Jak zastrzega autor, OPEX nie jest wielozadaniowym systemem operacyjnym w rozumieniu wielozadaniowości z wywłaszczeniem. Przypomnijmy, że w takim przypadku o czasie dostępu do procesora przez dane zadanie decyduje system operacyjny. Czas ten jest ściśle określony, po jego upływie zadanie zostaje wyłączone, a sterowanie przekazane do następnego zadania. Dzięki takiej koncepcji uzyskuje się bezpieczną, quasi – jednoczesną pracę wielu aplikacji. Zawieszenie jednej z nich nie powoduje całkowitego zawieszenia systemu. Jest tak przynajmniej teoretycznie, bo jak wiemy, w praktyce czasami dochodzi jednak do takich sytuacji (przykładem systemu wielozadaniowego z wywłaszczeniem jest Windows 9x/NT/2000/XP). Inną koncepcją jest wielozadaniowość bez wywłaszczania (Windows 3x). Tu każda aplikacja (zadanie) ma określony czas na wykorzystanie procesora, po czym powinna przekazać sterowanie systemowi operacyjnemu. Jeśli tego nie zrobi (np. na skutek zawieszenia), zawieszeniu ulega sama aplikacja i cały system. OPEX jest bliższy tej drugiej koncepcji. Jest to system operacyjny przeznaczony dla urządzeń, w których występuje wiele zadań uruchamianych w stałych lub zmiennych interwałach czasowych, a także w ściśle wyznaczonych momentach (data, godzina).

Cechą charakterystyczną systemów RTOS jest duże zapotrzebowanie na pamięć RAM. W klasycznych rozwiązaniach każde zadanie dysponuje własnym stosem, dość kłopotliwe staje się oszacowanie niezbędnej wielkości RAM–u oraz panowanie nad prawidłowym przełączaniem stosów. Niemała jest również wielkość potrzebnej pamięci programu i jak już wiemy duża moc obliczeniowa procesora. W systemie OPEX autor zastosował dość innowacyjne podejście. Występuje tu tylko jeden stos dla wszystkich zadań (nawet samopowtarzalnych). Specjalna funkcja tzw. *schedulera* czuwa nad uruchamianiem zadań według założonego porządku. Uwzględnia przy tym flagi ustawiane przez jedne zadania dla innych, odebranie danej z portu szeregowego mikrokontrolera (w szczególności znaku końca linii), upływu określonego interwału czasu od poprzedniego wywołania zadania, wykrycia określonej chwili (data, godzina). Funkcje schedulera pozwalają zarówno kreować zadania, jak i koń-

List. 1. Struktura służąca do zapisu danych o czasie astronomicznym (data/godzina)

```
typedef struct time_date_node //musi być upo-
rządkowana malejąco
   {
      unsigned char year; //0..99 aktualnego
wieku
      unsigned char month;
      unsigned char day;
      unsigned char hour;
      unsigned char minute;
      unsigned char second;
      unsigned char tick;
      unsigned char weekday; //0..6 pole to
nie jest brane pod uwagę przy porównaniach
   } DATE_TIME ;
//Przesunięcie względem GMT jest zapisywane
odrębnie
```

czyć ich działanie, występują różne rodzaje statusów operacji np.: *wait–for–flag*, *set–flag*. W systemie OPEX zaimplementowano obsługę zegara czasu astronomicznego. Niezbędny jest do tego oscylator o częstotliwości 32768 kHz. Tak jak zwykły zegarek mechaniczny, również zegar systemu OPEX odmierza stałe interwały czasu (ticks). Domyślny takt zegara jest równy 1/128 sekundy, ale może być łatwo zmieniony.

Uruchamianie aplikacji wielozadaniowych wymaga odpowiedniego wsparcia narzędziowego. Środowisko uruchomieniowe, jakim w przypadku mikrokontrolerów AVR może być np. AtmanAvr nie zawiera żadnych specjalnie do tego celu przewidzianych elementów. Jest to typowe IDE, przeznaczone do uruchamiania wszelkich systemów z mikrokontrolerem AVR. Aby choć trochę ułatwić życie użytkownikom, autor OPEX–u zaimplementował w systemie kilka funkcji wykorzystywanych do monitorowania i debugowania uruchamianej aplikacji. Dzięki nim można np. utworzyć listę wszystkich zadań w aplikacji, wyświetlić ich status, zajętość pamięci RAM, wykorzystanie stosu.

O stabilności systemu czasu rzeczywistego (mimo wszystko będę tak nazywał OPEX) w dużej mierze decyduje zarządzanie pamięcią RAM. Kluczową rolę odgrywa tu występująca w języku C funkcja *malloc()*. W systemie OPEX występuje własna jej implementacja. Funkcja ta dba o prawidłowe przydzielanie i zwalnianie pamięci dla poszczególnych zadań. Jest ona intensywnie wykorzystywana przez różne funkcje systemowe.

Jak już wiemy, w OPEX–owych aplikacjach poszczególne zadania są wywoływane okresowo przez schedulera. Bywają jednak chwile, w których żadne zadanie użytkownika nie jest aktywne. System operacyjny musi jednak pracować nawet wtedy. Wypełnianiem czasu procesora zajmuje się specjalne zadanie o nazwie „idle". Jest ono wywoływane z częstotliwością ok. 100000 razy na sekundę (w systemie z mikrokontrolerem AVR 6 MHz) i realizuje m.in. monitorowanie portów we/wy bez korzystania z systemu przerwań.

Wymagania sprzętowe

Niestety, zaimplementowanie systemu OPEX nie jest możliwe dla pierwszego, wziętego z półki mikrokontrolera AVR. Nawet uproszczone do maksimum funkcje wymagają w sumie niemałych zasobów. Minimalna wielość pamięci programu dla pełnej wersji systemu OPEX wynosi 16 MB, natomiast rozsądne minimum dla pamięci RAM, to 2 kB. Idealnym typem mikrokontrolera jest więc AVR ATmega32. W niektórych aplikacjach można sobie pozwolić na niepełne skompilowanie bibliotek. Mogą być wówczas wykorzystane nawet w układach z pamięcią programu mniejszą niż 8 kB. Takie chipy posiadają jednak na ogół nie więcej niż 1 kB pamięci RAM, co jest wartością absolutnie minimalną. W takich przypadkach mogą wystąpić problemy z rozmieszczeniem wszystkich zmiennych globalnych i statycznych, buforów portu szeregowego, itp.

Skondensowana charakterystyka systemu OPEX

Styl pisania programów działających w takim systemie, jak OPEX jest nieco odmienny niż dla standardowych aplikacji. Prawdopodobnie początkującemu programiście trudno będzie wyzbyć się pewnych manier, nabytych podczas wcześniejszej praktyki, a raczej braku nowych, charakterystycznych dla systemów RTOS. Tu wszystkie operacje wykonywane są przez zadania. Każde zadanie musi być zawczasu wykreowane, uruchomione i zamknięte po zrealizowaniu założonego celu. Istnieje przy tym wiele przypadków. Niektóre zadania mogą się uruchamiać samodzielnie po upływie określonego czasu lub w określonej chwili czasowej. Wywołanie zadania A może być przełożone podczas wykonywania zadania B. Zadanie A może oczekiwać na specjalny sygnał generowany przez zadanie B, C,... itd. Zadanie A może spowodować opuszczenie (*quit*) lub uśmiercenie (*kill*) zadania B. Szczególnym rodzajem zadania jest „*idle*". Jest ono uruchamiane okresowo w chwilach bezczynności procesora. Drugim, oprócz zadań, elementem systemu OPEX są zdarzenia. Zdarzenie, to fakt dający się zaobserwować przez funkcję obsługującą zadanie. Może to być np. ustawienie flagi (semafora), odebranie znaku z portu szeregowego, itp. Zdarzenie może więc powstać na przykład na skutek działania jakiegoś zadania lub procedury obsługi przerwania. Zadanie może wymagać do uruchomienia wystąpienia jednego lub wielu zdarzeń.

List. 2. Struktura służąca do zapisu informacji o zadaniu, tzw. blok sterujący zadaniem (TCB – Task Control Block)

```
typedef struct func_state_node
  {
      struct func_state_node *next; //wskaź-
nik do następnego rekordu
      DATE_TIME when;  // czas uruchomienia
zadania
      BYTE state;   // pole dla użytkownika
      void *pinfo;  // j.w.
      size_t *malloc_list;//wskaźnik do listy
alokacji
      char *name;   //nazwa zadania (łańcuch
znakowy)
      void (*funcp) (struct func_state_node
*me); //funkcja do wywołania
      BYTE *flags;   //wskaźnik do flag
      BYTE flagmask; //maska flag dla zadań
oczekujących na flagę
  } OPEX_TCB ;
```

Szczególną rolę w systemie OPEX pełni zegar czasu astronomicznego. Znaczna część zadań jest uruchamiana w oparciu o dane z tego zegara. Funkcje OPEX zapewniają prawidłowe ustawianie czasu i daty oraz obsługę zegara i kalendarza. Uwzględniane jest automatyczne przełączanie lat, korekta dni dla lat przestępnych, strefy czasowe (przesunięcie względem GMT) i czas letni. System dba o prawidłowe uporządkowanie zadań po każdej zmianie daty lub godziny.

Przykłady

Rozpoczęcie pisania własnych programów działających w systemie OPEX będzie łatwiejsze po zapoznaniu się z przykładową aplikacją. Wyjaśniona jest w niej m.in. metoda obsługi interfejsu 1–wire (rejestrator temperatury DS1820), obsługa portów we/wy, eliminacja trzasków w przełącznikach mechanicznych, obsługa portu szeregowego, ustawianie kalendarza, korzystanie z funkcji monitora i debugera. Przykładowa aplikacja wykonuje 9 zadań użytkownika, realizujących wyżej wymienione operacje.

Zestawienie prototypów funkcji systemowych

W krótkim artykule nie ma miejsca na dokładne omówienie wszystkich funkcji systemu. Programistom posługującym się językiem C pewne wyobrażenie o możliwościach prezentowanego software'u i technice programowania może dać lista prototypów funkcji wraz z krótkimi objaśnieniami. Na list. 1 i 2 zostały przedstawione dwa najważniejsze typy strukturalne zmiennych systemowych.

Inicjalizacja systemu

void OPEX_init_dateTime(void); – inicjalizacja zegara czasu astronomicznego (data/godzina). Jeśli korzysta się ze specjalizowanych układów zewnętrznych, to odpowiednie funkcje obsługi muszą być umieszczone w pliku *AVRdependent.c*.

void OPEX_com_init(void); – inicjalizacja bufora portu szeregowego.

*void OPEX_sched_setIdleFunc(void(*funcp));* – funkcja bezczynności („*Idle*"). Jest ona wykonywana cyklicznie w chwilach bezczynności procesora. Wskaźnik będący argumentem tej funkcji może kierować wywołanie do kodu użytkownika, gdzie można np. zamieścić rozkaz uśpienia mikrokontrolera lub uwzględnić obsługę watchdoga.

void OPEX_sched_start(void); – start schedulera pilnującego kolejności uruchamiania zadań.

Obsługa zadań

*OPEX_TCB * OPEX_sched_new(void(funcp)(OPEX_TCB *),char *);* – kreowanie nowego zadania. Rozpoczęcie każdego nowego zadania jest możliwe dopiero po zakończeniu niniejszej funkcji. Wskaźnik będący pierwszym argumentem pozwala określić funkcję wywoływaną przy pierwszym uruchomieniu zadania. Drugi argument pozwala nadać zadaniu własną nazwę. Będzie ona później występowała w raportach monitora.

void OPEX_sched_quit(void); – zakończenie bieżącego zadania. Ta funkcja nie ma własnego wyjścia, wykonywany jest skok do schedulera.

*void OPEX_sched_kill(OPEX_TCB *);* – zakończenie zadania wskazywanego przez argument.

*void OPEX_sched_resched(OPEX_TCB *,days,hours,minutes,seconds, ticks);* – zmiana czasu uruchomienia zadania wskazywanego przez pierwszy argument funkcji. Zadanie to zostanie uruchomione po podanym upływie czasu.

*void OPEX_sched_schedAt(OPEX_TCB *,year,month,day,hour,minute, second,tick);* – zmiana czasu uruchomienia zadania wskazywanego przez pierwszy argument funkcji. Zadanie zostanie uruchomione o określonej porze (data/godzina).

*OPEX_TCB *OPEX_sched_get_named(char *);* – wyszukanie zadania o podanej nazwie. Zwracany jest adres TCB lub wartość NULL (jeśli nie zostało znalezione). Argument funkcji wskazuje na łańcuch znakowy, który musi odpowiadać zadaniu umieszczonemu w kolejce *schedQ* lub *flagwaitQ* (uwaga na zadania bez nazwy, którym zawsze będzie odpowiadała wartość NULL).

*void OPEX_sched_on_flag(flag *,mask);* – oczekiwanie na zdarzenie. Zadanie zostanie zawieszone do chwili, aż wystąpi odpowiednie zdarzenie. Zdarzenie takie może być zapisane przez inne zadanie lub funkcję obsługującą przerwanie. Zdarzenia są zapisywane w bajcie pamięci RAM, który jest wskazywany przez pierwszy argument. W jednym bajcie można przekazać informację o 8 zdarzeniach. Maska pozwala ukryć zdarzenia, które powinny być zignorowane. Funkcja nie ma własnego zakończenia, wykonywany jest skok do schedulera. Zadania oczekujące na zdarzenie jest umieszczone w kolejce *flagwaitQ* typu LIFO (*last in – first out*).

*void OPEX_sched_change_flag(*flags,bitno,state);* – wyzwolenie zdarzenia, na które mogą oczekiwać zadania (jedno lub kilka). Funkcja ta przesuwa wszystkie zadania oczekujące do kolejki *schedQ* ze znacznikiem natychmiastowego wykonania.

Obsługa kalendarza i zegara czasu astronomicznego

*void OPEX_sched_now(OPEX_TCB *);* – zapisanie bieżącego czasu astronomicznego w TCB wskazywanym przez argument funkcji.

int OPEX_DayOfWeek(Year,Month,Day); – wyznaczenie dnia tygodnia dla podanej daty (niedziela=0).

*int OPEX_compare_dt(DATE_TIME *t1,DATE_TIME *t2);* – porównanie dwóch dat. W rezultacie obliczeń zwracane są wartości: –1 (t1<t2), 0 (t1=t2), +1 (t1<t2).

*void OPEX_format_time(char *buffer,DATE_TIME *t);* – formatowanie czasu astronomicznego do postaci łańcucha tekstowego.

*void OPEX_format_date(char *buffer,DATE_TIME *t);* – formatowanie daty do postaci łańcucha tekstowego. Za pomocą tej funkcji można również jednocześnie formatować datę i godzinę.

*void OPEX_date_add(DATE_TIME *,days,hours,minutes,seconds,ticks);* – dodanie przesunięcia do podanej w pierwszym argumencie daty/godziny.

*void OPEX_date_stuff(DATE_TIME *,year,month,day,hour,minute,second,tick);* – wpisanie daty i godziny do zmiennej wskazywanej przez pierwszy argument funkcji. Uwaga: funkcja nie sprawdza poprawności danych wejściowych.

void OPEX_sched_time_changed(void); – przeliczenie momentu wywołania wszystkich zadań do wykonania w przyszłości, zamiast wykonania natychmiastowego. Przeliczeniu nie podlegają jedynie zadania oczekujące na zdarzenie.

void OPEX_daylight_saving_adjust(); – uporządkowanie zapisów czasu z uwzględnieniem czasu letniego i stref czasowych

int OPEX_is_daylight_saving(void); – funkcja zwraca niezerową wartość, jeśli bieżąca godzina uwzględnia czas letni.

*void OPEX_daylight_saving_changes_at(flag,DATE_TIME *t);* – funkcja zwraca datę i godzinę, dla której zmieniono czas letni.

Obsługa wyjścia portu szeregowego

Int OPEX_putcNoBlock(BYTE); – wysłanie jednego bajtu (znak ASCII lub dana binarna). Jeśli port szeregowy jest zajęty wysyłaniem poprzedniej danej, to nowa dana jest buforowana. Bufor jest opróżniany przez obsługę przerwania nadajnika (odpowiednia funkcja jest zamieszczona w pliku *AVRdependent.c*). Dana przepada, jeśli bufor jest zapełniony, funkcja zwraca wówczas wartość niezerową. Udany zapis oznacza zwrócenie wartości zerowej.

void OPEX_putc(BYTE); – umieszczenie danej binarnej lub znaku ASCII w buforze nadajnika portu szeregowego. Opróżnienie bufora przebiega analogicznie, jak w poprzedniej funkcji. Jeśli bufor jest zapełniony zostaje cyklicznie wywoływana funkcja *OPEX_putcNoBlock()* aż do momentu, gdy zapis do bufora zostanie wykonany pomyślnie.

void OPEX_nl(void); – wysłanie znaku 0x0d ('\r'). Funkcja wykorzystuje *OPEX_putc()*.

*void OPEX_puts(char *p);* – skopiowanie umieszczonego w pamięci RAM łańcucha ASCII zakończonego znakiem zero do bufora portu szeregowego (z wykorzystaniem *OPEX_putc()*). Domyślna wartość łańcucha, to „*something*".

*void OPEX_putline(char *p);* – j.w. z dodatkowym wywołaniem funkcji *OPEX_nl()*.

*void OPEX_puts_P(char *p);* – skopiowanie umieszczonego w

pamięci RAM łańcucha ASCII zakończonego znakiem zero do bufora wyjściowego portu szeregowego poprzez *OPEX_putc()*. Umieszczenie łańcucha w pamięci programu może być zrealizowane za pomocą polecenia makro AVRGCC – PSTR("*something*").

*void OPEX_putline_P(char *p);* – j.w. z dołączeniem znaku 0x0d ('\r').

int OPEX_txbuf_unused(void); – podanie wolnego miejsca w buforze wyjściowym nadajnika szeregowego.

int OPEX_txbuf_used(void); – podanie liczby bajtów pozostających do wysłania w buforze wyjściowym nadajnika szeregowego.

Obsługa wejścia portu szeregowego

*Int OPEX_com_peek(unsigned char *c,BYTE last);* – funkcja zwraca liczbę bajtów w buforze wejściowym portu szeregowego. Jeśli bufor nie jest pusty, to pierwszy argument służy do przekazania bajtu z bufora. Drugi argument określa, czy bajt pobrany z bufora jest pierwszym (0) lub ostatnim (nie 0) w buforze.

*int OPEX_getc(unsigned char *p);* – pobranie bajtu z bufora wejściowego portu szeregowego (jeśli jest do pobrania). Funkcja zwraca wartość 0, jeśli aktualnie nie ma żadnego bajtu w buforze wejściowym, w przeciwnym przypadku zwracana jest wartość niezerowa, a bajt z bufora jest zapisywany w zmiennej wskazywanej przez argument funkcji. Funkcja ta nigdy nie zapętla się w oczekiwaniu na daną.

*int OPEX_gets(char *p1,int maxchars,BYTE timeout_secs);* – pobranie łańcucha z bufora wejściowego portu szeregowego z timeout'em. Funkcja odczytuje ciąg ASCII zakończony znakiem 0x0d. Zwraca liczbę znaków znajdujących się w buforze wejściowym wskazywanym przez pierwszy argument. Maksymalna długość łańcucha jest ograniczona do wartości podanej w drugim argumencie i powinna gwarantować umieszczenie w buforze zerowego bajtu kończącego. Jeżeli trzeci argument ma wartość niezerową, to jego wartość jest równa timeout'owi wyrażonemu w sekundach. Przez ten czas funkcja odbiera dane i oczekuje na znak CR umieszczony bezpośrednio po zerowym bajcie kończącym, W wyniku przekroczenia timeout'u odebrany łańcuch może nie zawierać znaku CR. Funkcja ignoruje przychodzące znaki 0x0a ('\n').

Obsługa monitora i debugera

OPEX_set_monitoring(BYTE); – włączenie/wyłączenie funkcji monitorowania.

*void * OPEX_sched_monitor_redirect(void(*fp)(char));* – przekierowanie wyjścia tekstowego z funkcji monitoringu wymienionych niżej. Przesyłanie znaku będzie prowadzone przez podaną w argumencie funkcję, wartość *NULL* przywraca standardowe działanie wyjścia.

int OPEX_monitoring(void); – pobranie statusu monitoringu (włączony/wyłączony).

*void OPEX_task_report(char *p,OPEX_TCB *f);* – funkcja może być wykorzystywana przez zadanie do wysyłania raportu o przebiegu wykonywania zadania. Po sformatowaniu raportu może być on wysłany do zadania wskazanego w drugim argumencie. Funkcja nie sprawdza, czy takie zadanie istnieje, jeśli nie, mogą wystąpić problemy.

*void OPEX_sched_showQitem(OPEX_TCB *,char *,char);* – sformatowanie statusu wskazanego zadania do postaci tekstowej. Zadanie może być elementem kolejki posortowanej chronologicznie (*schedQ*) lub kolejki zadań oczekujących na zdarzenie (*flagwaitQ*). Drugi argument jest wskaźnikiem na bufor, w którym zostanie umieszczony raport. Jeśli trzeci argument jest niezerowy, to w sformatowanym raporcie powinny być dołączone flagi zdarzeń.

*void OPEX_sched_showQ(char *,BYTE);* – funkcja działa podobnie do poprzedniej, ale dotyczy kolejek dla zadań oczekujących na datę/godzinę i flagi.

*void OPEX_sched_show_mem(char *,unsigned int);* – formatowanie raportu o wykorzystaniu pamięci do postaci tekstowej. Raport zostanie umieszczony w buforze wskazanym przez argument.

Lepsza prawda zła, niż żadna

Autor OPEX–u wprawdzie zastrzega, że jego software nie jest prawdziwym system operacyjnym czasu rzeczywistego, ale choćby z samego faktu złożenia powyższej deklaracji wynika, że myśl taka „chodziła mu" widocznie po głowie. Obiektywnie patrząc, taka kwalifikacja byłaby jednak zbyt daleko idącym skojarzeniem. Można natomiast ze spokojniejszym sumieniem uznać OPEX, jako unikatowy system operacyjny przeznaczony dla mikrokontrolerów ATmega. Pozostaje jeszcze odpowiedzieć na pytanie, czy może on stanowić metodę pozwalającą rozwiązać problemy konstruktora – programisty, czy choćby ułatwi opracowywanie własnych aplikacji? Myślę, że – jeśli tak – to nie w każdym przypadku. Być może o zaletach stosowania OPEX–u można się przekonać dopiero po zrealizowaniu kilku własnych projektów. Po zapoznaniu się z filozofią tego systemu odnoszę wrażenie, że podobnym nakładem sił byłbym w stanie napisać program użytkowy własnymi metodami, w dodatku bardziej oszczędnie pod względem wykorzystania zasobów mikrokontrolera. Pamiętać przy tym trzeba, że OPEX jest systemem uniwersalnym i jako taki musi zawierać elementy, które nie koniecznie będą wykorzystywane w konkretnych sytuacjach. Czy warto sięgać po OPEX? No cóż, konstruktorzy dzielą się na takich, którzy lubią stabilność i pozostają wierni swoim wyuczonym narzędziom i metodom pracy oraz takich, którzy sięgają chętnie po każdą dostępną nowinkę. Czasami wygrywają jedni, czasami drudzy.

List. 3. Oryginalny przykład miniprogramu demonstracyjnego

```
//////////////////////////////////////////
//////////
// OPEXlib_Demo0.c - A very simple demo
#include „OPEX.h"
void serial_init(void); // these are in AVR-
dependent.c
void io_init(void);
void timer_init(void);
char StringBuf1[128]; // global scratch buffer
for string work

// THE ONE SCHEDULED TASK's FUNCTION
////////////////////
// say hello, then reschedule myself to run
again later
void task0(OPEX_TCB *me)
{
   OPEX_task_report(StringBuf1, me); // moni-
tor/debug report
   sprintf_P(StringBuf1, PSTR(„%s: pinfo =
%04d"), me->name, (int)me->pinfo++);
   OPEX_putline(StringBuf1);
   OPEX_sched_resched(me, 0, 0, 0, 1, 0); //
run me again later
}
/// MAIN ///
int main(void)
{
   io_init(); // see AVR_dependent for the-
se...
   timer_init();
   serial_init();
   OPEX_com_init(); // init OPEX serial I/O
   sei();

   OPEX_init_dateTime(); // set OPEX date and
time defaults
   OPEX_set_monitoring(1); // enable debug
monitoring

   for ( ;; )
   {
      OPEX_putline_P(PSTR(„Starting Scheduler")
); // print from flash memory
      OPEX_sched_new(&task0, „TASK0"); //
launch task, assume success doing so
      OPEX_sched_start(); // returns if all
tasks quit
   }
}
```

Jarosław Doliński, EP
jaroslaw.dolinski@ep.com.pl