PRZEMYSŁOWY INSTYTUT AUTOMATYKI I POMIARÓW
MERA-PIAP
Al. Jerozolimskie 202 02-222 Warszawa Telefon 23-70-81

Ośrodek Automatyzacji Procesów Produkcji

Pracownia Oprogramowania Cyfrowych Systemów Sterowania

4440 BE 10

Główny wykonawca mgr inż. Bożena Dąbrowska

Wykonawcy: mgr inż. Wojciech Nikiel,
mgr inż. Stanisław Wółtański,
Rafał Waleriańczyk

Konsultant

Nr zlecenia 1119

Automatyczny system rozrządzania na stacji Lublin - Tatary.

Etap.3.
Mały System Operacyjny Czasu Rzeczywistego . Projekt i instrukcja obsługi.

Zleceniodawca

Pracę rozpoczęto dnia grudzień 87r. **zakończono dnia** 16.12.88r.

Kierownik Pracowni
Dąbrowska
mgr inż. B.Dąbrowska

Z-ca Dyrektora
d/s Automatyki
dr inż. T.Gałązka

Kierownik Ośrodka
mgr inż. A.Aderek

Praca zawiera:

stron 39
rysunków
fotografii
tabel
tablic
załączników

Nr rejestr. 6164

Rozdzielnik - ilość egz:

Egz. 1 BOINTE
Egz. 2 OAP-41
Egz. 3 CNTK
Egz. 4 WDOKP
Egz. 5 OAP-a/a
Egz. 6

Analiza deskryptorowa

System operacyjny, systemy komputerowe

TRANSPORT SZYNOWY, AUTOMATYZACJA, SYSTEM [illegible]

Analiza dokumentacyjna

Projekt i instrukcja użytkowania systemu operacyjnego

czasu rzeczywistego.

Tytuły poprzednich sprawozdań

UKD 681.518.5 System sterowania [illegible]

625.1 Kolejnictwo

PIAP-252/83-6000

Spis treści

3

1. Wprowadzenie

1.1 Przeznaczenie i charakterystyka systemu

Niniejszy opis stanowi instrukcję użytkową Małego 'Systemu Operacyjnego Czasu Rzeczywistego do Zastosowań Przemysłowych - SIRTOS. System ten, opracowany w Przemysłowym Instytucie Automatyki i Pomiarów w Warszawie, jest przeznaczony dla sterowników przemysłowych i mikrokomputerów 16-bitowych. Istnieje możliwość jego adaptacji dla mikrokomputerów 8 lub 32-bitowych. Opisywana wersja SIRTOS-A 2.0 pracuje z procesorem typu INTEL 8086 (8088, 80186, 80286). Stanowi ona jedynie jądro systemu R-T. Istnieje możliwość dołączenia obsługi pamięci zewnętrznej (np. dysków elastycznych standardu IBM PC) w formie zadania użytkowego. Zazwyczaj dla celów sterowania obiektem przemysłowym potrzebne są jedynie dodatkowe funkcje obsługi pakietów sprzęgu obiektowego (tzw. handlery). Możliwa jest również adaptacja systemu dla sterowników wieloprocesorowych.

System SIRTOS zawiera:

- koordynator zadań użytkowych
- monitor przerwań
- zestaw funkcji systemowych przeznaczonych do uruchamiania, synchronizacji i wymiany informacji między zadaniami oraz inicjacji wewnętrznych zasobów mikrokomputera
- opcjonalnie, uniwersalne zadania użytkowe obsługujące terminale (klawiatura, ekran monitora, drukarka) oraz testujące pamięć RAM
- procedury obsługi przerwania zegarowego czasu rzeczywistego wraz z datownikiem

Podstawowe jego charakterystyczne cechy to:

- obsługa zadań według ich priorytetów
- liczba zadań ograniczona wielkością dostępnej pamięci i szybkością procesora
- łatwe dołączanie zadań użytkowych
- kontrola przekroczenia stosu zadania oraz możliwość jego rozszerzania (powyżej 2 kB)
- mała zajętość pamięci ROM i RAM
- krótki czas obsługi własnej (przełączania zadań)
- dowolna liczba semaforów i skrzynek pocztowych
- realizacja systemu w języku C

Podobnie jak i w innych systemach czasu rzeczywistego funkcjonalny podział oprogramowania aplikacyjnego na zadania upraszcza projektowanie, zapewnia jego modularność, umożliwia równoległość wykonania. Realizacja priorytetowości obsługi gwarantuje efektywność oprogramowania. Szybkość reakcji systemu zwiększono dzięki kompilacji jądra z opcją "F" (optymalizacja czasu wykonania kosztem długości kodu programu) oraz unikaniu zbędnych wywołań koordynatora zadań. Parametry funkcji systemowych, są przekazywane przez stos zadania. System zawiera jedno zadanie systemowe (SYS),

dokonujące inicjacji całej konfiguracji aplikacyjnej. Implementacja została zrealizowana w języku AZTEC C (wersja 3.40) przy wykorzystaniu oprogramowania narzędziowego pracującego pod systemem PC DOS na komputerze IBM PC/AT. Instrukcja omawia realizację aplikacji w tzw. "małym modelu pamięci" (64 kB kodu, 64 kB danych), najbardziej efektywnym pod względem czasu wykonania i zwartości kodu wynikowego. W celu przygotowania oprogramowania aplikacyjnego, pracującego pod systemem SIRTOS, niezbędna jest dobra znajomość języka C - może być przydatna również znajomość asemblera mikroprocesora INTEL 8086.

1.2. Przygotowanie oprogramowania aplikacyjnego

Zadaniem twórcy systemu aplikacyjnego jest inicjacja odpowiednich obiektów danych systemowych zdefiniowanych w zbiorach:

- "gcc.h" (długość obszaru stosów, wielkość kwantu stosu, priorytety zadań)
- "gic.h" (parametry wewnętrznych układów we/wy)
- "vi.c" (wektor przerwań, rejestr CS)
- "uinit.c" (liczba zadań, tablica wejść, długości stosów)

Treść zadań użytkowych należy umieszczać we własnych zbiorach. Do pisania kodu źródłowego można używać edytora "Z", będącego elementem systemu AZTEC C lub innego uniwersalnego edytora, pracującego pod PC DOS-em (np. "kedit"). Funkcje obsługi przerwań należy standardowo umieścić w zbiorze "uints.c". Dodając nowe zbiory należy ich nazwy uwzględnić w zbiorze generatora MAKE ("makefile") o standardowej nazwie "nu". Nazwy głównych funkcji systemowych należy pisać dużymi literami. Testowanie i uruchamianie systemu użytkowego (po usunięciu błędów formalnych, sygnalizowanych przez kompilator i konsolidator) wykonuje się etapami. Szczegóły realizacji aplikacji są opisane w kolejnych punktach instrukcji.

2. Definicja, instalacja i inicjacja zadań użytkowych

Z punktu widzenia systemu SIRTOS, zadaniem użytkowym jest program, napisany w języku C , którego główną funkcją nie jest "main", lecz funkcja (typu "void"), o nazwie umieszczonej w tablicy wejść do zadań - "tentry", na pozycji wynikającej z nadanego temu zadaniu priorytetu. Numer zadania -równoważny przyznanemu priorytetowi - stanowi jego identyfikator. Priorytet zadania maleje wraz ze wzrostem numeru. Numer 0 jest zarezerwowany dla zadania systemowego SYS, wykonywanego jedynie w czasie inicjacji systemu oraz gdy żadne z zadań użytkowych nie jest gotowe do aktywacji. Funkcja "main" jest wejściową funkcją zadania SYS oraz równocześnie główną funkcją programu, którym - z punktu widzenia kompilatora - jest cały system aplikacyjny. Dołączenie zadania użytkowego do systemu polega na:

- nadaniu zadaniu priorytetu
- umieszczeniu nazwy wejściowej funkcji zadania w tablicy "tentry"
- przydzieleniu zadaniu liczby kwantów stosu w tablicy "lst"
- instalacji zadania w systemie przez wywołanie funkcji "instal"

Zaleca się nadawać priorytet zadaniu przez zdefiniowanie tzw. identyfikatora i przypisanie temu identyfikatorowi właściwego numeru np.:

```
#define SYS  0
#define TEH  1
#define PRT  3
#define BGT 12
```

Uwaga: Priorytet zadania maleje wraz ze wzrostem numeru (z wyjątkiem zadania SYS, które ma najniższy priorytet).

2.1. Tablica wejść

Tablica wejść do zadań definiuje ich adresy startowe. W systemie SIRTOS definicja ta polega na wpisaniu do właściwego elementu tablicy "tentry" (zgodnie z numerem - priorytetem) nazwy funkcji wejściowej zadania. Przykładowo, dla zadań TEH (numer 1), MON (numer 2), MMC (numer 3), BGT (numer 4), realizujących odpowiednio obsługę klawiatury terminala operatorskiego (funkcja wejściowa "kbord"), ekranu monitora operatorskiego (funkcja "tmon") dyrektyw komunikacji operator-system (funkcja "mmc") i testu pamięci RAM (funkcja "tbgt") tablica "tentry" ma postać (definicja wraz z inicjacją):

```
void ( *tentry [] ) () = {
                                main, kbord, tmon,
                                mmc,  tbgt,  NULLFP
} ;
```

NULLFP jest wskaźnikiem do funkcji pustej (ang. null function pointer) i rezerwuje w tablicy "tentry" miejsce na adres głównej funkcji zadania o numerze 5. Tablica "tentry" znajduje się standardowo w zbiorze "uinit.c" - dostarczanym klientowi w postaci źródłowej.

2.2. Przydział obszaru stosu

W zależności od modelu programu w systemie AZTEC C, zadaniom użytkowym zostanie przydzielona odpowiednia pula pamięci RAM na stosy. W tak zwanym małym modelu pamięci (ang. small code - small data) wystarczającym kwantem jest obszar 32 bajtow. Standardowo użytkownik ma do dyspozycji 64 takie kwanty pamięci (parametr SQ w zbiorze "gcc.h"). Zazwyczaj zadanie

6

potrzebuje 3 lub 4 kwanty (dla zadań wykorzystujących funkcje z wieksza liczbą parametrów). W przypadku zadań wykorzystujących funkcje wywoływane rekurencyjnie potrzebna liczba kwantów stosu zadania może być większa. Praktycznym sposobem oceny wielkości potrzebnego zadaniu stosu jest analiza wydruku zawartości pamięci przeznaczonej na stosy - po uruchomieniu wszystkich zadań systemu. Wydruk taki jest możliwy, gdy dysponujemy np. monitorem operatorskim zainstalowanym w pamięci ROM sterownika. Adres wierzchołka stosów (jedocześnie początek stosu zadania SYS) znajduje się w zmiennej "bstack" (pierwsza zmienna w grupie danych niezainicjowanych systemu), której adres można odczytać w zbiorze symboli "sirtos.sym", tworzonym przez konsolidator systemu AZTEC C (opcja -T). Standardowo w systemie aplikacyjnym, mieszczącym się w małym modelu pamięci, stos znajduje się za danymi zainicjowanymi a przed niezainicjowanymi i zajmuje łącznie 2 kB pamieci RAM. Dysponujac odpowiednio dużą pamiecią RAM można obszar stosów odpowiednio poszerzyć. Należy jednak pamiętać o właściwym zainicjowaniu wierzchołka stosu (nadaniu wartości zmiennej "bstack"). Zadanie systemowe SYS potrzebuje co najmniej trzy 32-dwu bajtowe kwanty pamięci RAM na swój stos. Przykładowo dla zadań użytkowych: TEH, MON, MMC, BGT (oraz zadania SYS) tablica "lst", przydzielająca zadaniom minimalne wielkości stosów ma postać:

```
char lst [] = { 3,    3,    3,
                4,    3,    0
};                                  /* łącznie 16 porcji z 64 */
```

W trakcie przełączania zadań system kontroluje przekroczenia puli pamięci przeznaczonej na stos aktywowanego zadania. W przypadku wykrycia zniszczenia stosu (przez zadanie o niższym numerze) nastąpi zawieszenie się systemu ("halt" systemu). Numer zadania, którego stos został uszkodzony znajduje się w zmiennej "act". W takim przypadku należy zwiększyć rozmiar stosu zadania o numerze niższym. Standardowo, w pamięci RAM dane są ułożone w następującej kolejności: wektor przerwań, dane zainicjowane (systemowe i użytkowe), stosy zadań (w kolejności malejącego numeru), dane niezainicjowane (systemowe i użytkowe).

Uwaga: Zaleca się przeznaczyć całą dostępną pamięć stosu zadaniom, zwłaszcza gdy oprogramowanie aplikacyjne wykorzystuje możliwość wielopoziomowej obsługi przerwań.

2.3. Instalacja zadań

Instalacja zadania polega na wpisaniu, przez funkcję "instal", w bloku kontrolnym zadania (TCB) parametrów:

- początkowy stan zadania ("śpiące"- ASLEEP)
- adres wejścia do zadania (zgodny z odpowiednim elementem tablicy "tentry")
- początek stosu zadania (wynika z liczby kwantów stosu zarezerwowanych przez zadania o wyższym priorytecie oraz zadanie SYS)

Sposób wywołania funkcji instalującej zadania w systemie wynika z postaci nagłówka jej definicji:

```
void instal ( numer_zadania )
     int numer_zadania ;
```

Zainicjowane w powyższy sposób zadanie może być następnie aktywowane za pomocą funkcji systemowych (START, STARTC, PERIOD). W trakcie pracy systemu ządanie może być wielokrotnie reaktywowane przez inne zadanie, przez siebie samo lub przez obsługę przerwań.

2.4. Blok kontrolny zadania

Tablica kontrolna zadań (TCB) składa się ze struktur o budowie:

```
struct tcblock {
     char state ;              /* stan zadania                    */
     void ( *entry ) () ;      /* adres wejścia do zadania        */
     unsigned int stack ;      /* początek stosu                  */
     unsigned int sp ;         /* wskaźnik stosu                  */
     char *asbc ;              /* adres semafora albo bufora      */
     int counter ;             /* licznik taktów 0.1 s            */
     unsigned int quanta ;     /* cykl restartu zadania w s       */
     unsigned int counts ;     /* licznik sekund w cyklu          */
     char intec ;              /* wskaźnik wejścia do sch:        */
} ;                            /* z zadania = 1                   */
                               /* układu przerwań = 0             */
```

System przechowuje w niej najważniejsze informacje o zadaniu. Modyfikują ją funkcje systemowe używane w zadaniach i obsłudze przerwań oraz bezpośrednio obsługa przerwania zegarowego i koordynator zadań.

3. Parametryzacja wewnętrznych układów we/wy sterownika przemysłowego

W celu ułatwienia użytkownikowi rozbudowy konfiguracji sterownika przemysłowego, oprogramowanie systemowe zawiera zestaw funkcji inicjujących podstawowe układy wejścia/wyjścia, takie jak: programowalne sterowniki przerwań - PIC (pracujące zazwyczaj w trybie master-slave), programowalne czasomierze odcinkowe - PIT, układy sprzęgów szeregowych - USART, sprzęgów równoległych (PIA), itp. Inicjacja tych wewnętrznych układów we/wy danego mikrokomputera polega na wywołaniu odpowiedniej funkcji systemowej z parametrem, będącym adresem tablicy lub adresem struktury, zawierającej niezbędne dane definiujące rodzaj pracy układu. Zazwyczaj użytkownik modyfikuje dane określające np.: sprzętowy adres sterownika, maskę przerwań, itp. Parametry dotyczące wewnętrznych układów we/wy znajdują się w zbiorze "gic.h". Po zdefiniowaniu wszystkich niezbędnych parametrów wewnętrznych układów we/wy sterownika

należy wywołania odpowiednich funkcji inicjujących te układy ("picinit", "timinit", "initV24", "piainit", itp.) z adresem tablicy (struktury) parametrów umieścić w funkcji "apinit", znajdującej się w zbiorze "uinit.c". Funkcja ta pracuje przy zamkniętym układzie przerwań. Zaleca się inicjację sterowników przerwań wykonać po inicjacji innych układów.

3.1. Programowalne sterowniki przerwań (PIC)

```
void picinit ( tablica_parametrów )
     char *tablica_parametrów ;
```

Funkcja "picinit" wykonuje inicjację jednego układu PIC typu 8259A. W tablicy parametrów "icw" powinny być umieszczone kolejno - dla każdego sterownika - następujące dane:

adres sterownika (adres portu we/wy)
słowo 1 inicjacji (przerwanie od narastającego zbocza, tryb master/slave, icw4; standardowo ICW1 = 0x11)
słowo 2 inicjacji (numer przerwania w wektorze; standardowo dla master-a ICW2 = 0x10, czyli 16-23)
słowo 3 inicjacji (pozycje slave-ów; przykładowo dla konfiguracji master i slave_1 oraz slave_2 ICW3 = 0x06 dla master-a, ICW3 = 0x01 dla slave_1, ICW3 = 0x02 dla slave_2)
słowo 4 inicjacji (SFNM - wielopoziomowy tryb pracy z z pełnym zagnieżdżaniem, tryb MCS-86/88; standardowo ICW4 = 0x11 dla master-a, ICW4 = 0x01 dla slave-ów)
słowo 1 operacji (rejęstr maskowania przerwań IRR; jedynka na najmniej znaczącym bicie oznacza zamaskowanie przerwania INT0, czyli o najwyższym priorytecie; standardowo OCW1 = 0x00 przy wszystkich przerwaniach aktywnych)

3.2. Programowalne czasomierze odcinkowe (PIT)

Do programowania układów czasomierzy odcinkowych typu 8253 służy funkcja "timinit".

```
void timinit ( tablica_parametrów )
     char *tablica_parametrów ;
```

Parametry funkcji są umieszczone w tablicy typu "char". Kolejne bajty tablicy zawierają: adres słowa komendy licznika, słowo sterujące, adres licznika, młodszy bajt podziału i starszy bajt podziału częstotliwości zegara. Funkcja, pod adres słowa komendy, wysyła słowo sterujące licznika, a następnie pod adres licznika wpisuje kolejno młodszy i starszy bajt podziału.

3.3. Sprzęgi szeregowe(USART)

Do programowania układów interfejsu szeregowego typu 8251A służy funkcja "initV24".

```
void initV24 ( tablica_parametrów )
     char *tablica_parametrów ;
```

Analogicznie jak dla poprzedniej funkcji parametry "initV24" umieszczone są w tablicy typu "char". Kolejne bajty tablicy zawierają : adres rejestru rozkazów, słowo rozkazu wewnętrznego zerowania rejestru IR, słowo trybu pracy, słowo rozkazu i adres słowa danych. Działanie funkcji jest następujące:

- trzykrotne wysłanie zerowego słowa do rejestru rozkazów USART-a
- wysłanie rozkazu z ustawionym bitem IR
- wpisanie słowa trybu pracy
- wpisanie słowa rozkazu
- dwukrotny odczyt rejestru danych

W celu ustawienia właściwej częstotliwości pracy układu USART-a, należy zaprogramować również współpracujący z nim licznik.

3.4. Sprzęgi równoległe (PIA)

Do programowania układów interfejsu równoległego typu 8255 służy funkcja "piainit".

```
void piainit ( tablica_parametrów )
     char *tablica_parametrów ;
```

Podobnie jak dla poprzedniej funkcji parametry "piainit" umieszczone są w tablicy typu "char". Kolejne bajty tablicy zawierają : adres rejestru rozkazów, słowo rozkazu definiującego tryb pracy układu i słowo rozkazu sterującego selektywnym ustawieniem bitów rejestru C. Funkcja wysyła kolejno oba słowa rozkazu pod adres rejestru rozkazów.

4. Dołączanie użytkowych funkcji obsługi przerwań

Podstawową czynnością użytkownika systemu SIRTOS, podczas konfigurowania oprogramowania aplikacyjnego, jest - obok dołączania zadania - dołączenie funkcji obsługi przerwania. Wyróżniono zewnętrzną i wewnętrzną funkcję obsługi przerwania. Nazwę zewnętrznej funkcji obsługi przerwania należy umieścić w wektorze przerwań tj. tablicy "vint", znajdującej się w zbiorze "vi.c". Wywołanie monitora przerwań następuje w funkcji zewnetrznej, natomiast funkcja wewnętrzna, wykonująca właściwą obsługę przerwania jest parametrem monitora przerwań.

4.1. Definiowanie wektora przerwań

Tablicę "vint", stanowiącą wektor przerwań systemu, należy zainicjować adresami zewnętrznych funkcji obsługi przerwań. Kolejność adresów wynika z przyporządkowania przerwaniom priorytetów przez sprzętowe podłączenie ich źródeł do poszczególnych wejść sterowników przerwań (PIC). W przypadku nieciągłości w numeracji przerwań podłączonych do różnych sterowników należy uwzględnić ten fakt przez wpisanie do wektora przerwań stałej NULLFP. Przykładowo wektor przerwań dla sterowników MASTER i SLAVE, do których podłączono następujące przerwania:

- przerwanie zegarowe 100 ms
  (na pozycję 0 sterownika MASTER)
- przerwanie RxRDY z układu USART
  (na pozycję 6 sterownika MASTER)
- przerwanie TxRDY z układu USART
  (na pozycję 7 sterownika MASTER)
- przerwanie zegarowe 1 s
  (na pozycję 0 sterownika SLAVE)

ma postać:

```
struct vinp vint [] = {      { 0, 0 },   {0, 0 }, { 0, 0 },

{ tim1, CS },  { NULLFP, CS }, { NULLFP, CS }, { NULLFP, CS },
{ NULLFP,CS }, { NULLFP, CS }, { rx, CS },     { tx, CS },

{ tim2, CS },  { NULLFP, CS }, { NULLFP, CS }, { NULLFP, CS },
{ NULLFP,CS }, { NULLFP, CS }, { NULLFP, CS }, { NULLFP, CS }
} ;
```

Dla powyższego przykładu, zakładając że rejestr segmentowy danych DS = 3, sterownikom MASTER i SLAVE należy przyporządkować odpowiednio przerwania o numerach 16-23 oraz 24-31.

Uwaga: Poszczególne elementy wektora przerwań należy, w małym modelu pamięci, uzupełnić stałą CS (wartość rejestru segmentowego kodu CS), gdyż element wektora jest zawsze tzw. długim adresem, niezależnie od modelu pamięci (pośredni call typu long). Początkowe trzy struktury (zainicjowane zerami) są niezbędne, ze względu na kompilator języka C.

4.2. Wywoływanie monitora przerwań

Nagłówek funkcji monitora przerwań ma postać:

```
void servints ( fint, parf )
     int ( *fint ) () ;
     char *parf ;
```

Monitor przerwań jest wywoływany w zewnętrznej funkcji obsługi przerwań; "fint" jest adresem wewnętrznej funkcji obsługi, a "parf" jest adresem przekazywanych jej parametrów.

W przypadku, gdy funkcja wewnętrzna nie używa parametrów, należy użyć adres pustego systemowego parametru - "dummy".

Ogólna postać zewnętrznej funkcji obsługi dowolnego przerwania jest następująca:

```
void exint ( )
     {
     savereg ( ) ;
     servints ( fint, apar ) ;
}
```

Funkcja "savereg" przechowuje aktywne rejestry na stosie zadania (SI, DI, DX, CX, BX, AX oraz opcjonalnie rejestr segmentowy ES). Funkcja "servints" jest monitorem przerwań systemu. Funkcja "fint" jest właściwą funkcją obsługi przerwania, a parametr "apar" adresem parametru (parametrów) tej funkcji.

4.3. Okresowa blokada wybranych źródeł przerwań i całego układu

```
void mint ( adres_sterownika_przerwań, maska )
     char *adres_sterownika_przerwań ;
     char unsigned maska ;
```

Funkcja maskuje przerwania na sterowniku przerwań o podanym adresie, odpowiadające bitom ustawionym w masce na 1.

```
void unmint ( adres_sterownika_przerwań, maska )
     char *adres_sterownika_przerwań ;
     char unsigned maska ;
```

Funkcja odmaskowuje przerwania na sterowniku przerwań o podanym adresie odpowiadające bitom ustawionym w masce na 1.

```
void di ( )
```

Funkcja blokuje układ przerwań mikroprocesora.

```
void ei ( )
```

Funkcja odblokowuje układ przerwań mikroprocesora - użyta w zadaniu uruchamia koordynator zadań.

```
void dis ( )
```

Funkcja blokuje układ przerwań mikroprocesora.

```
void eis ( )
```

Funkcja odblokowuje układ przerwań mikroprocesora.

Funkcje "dis", "eis" mogą być używane w przypadku, gdy pomiędzy nimi nie są wołane żadne funkcje systemowe główne oraz funkcje inicjujące zasoby systemu. W przeciwnym wypadku należy użyć pary funkcji "di - ei". Jako para do "di" albo

"dis", może w zadaniu wystąpić funkcja systemowa główna, odblokowująca przerwania: PAUSE, STOP, WAIT, WAITOUT, SENDM lub WAITM albo START i KILL do samego siebie. Funkcje "dis" i "eis" działają szybciej niż para "di - ei", jednak te ostatnie są ogólniejsze.

4.4. Realizacja wielopoziomowej obsługi przerwań z pełnym zagnieżdżaniem

Standardowa inicjacja układów PIC, z tzw rozszerzonym podstawowym trybem pracy (ang. special fully nested mode - SFNM), umożliwia organizację wielopoziomowej obsługi przerwań, tj. przerwania o niższym priorytecie mogą być przerywane przez przerwania o wyższym priorytecie. Niezbędnym warunkiem takiej pracy jest odblokowanie układu przerwań w wewnętrzej funkcji obsługi (np. za pomocą funkcji "ei"). Należy jednak pamiętać, że przed wykonaniem powrotu z tej funkcji (do monitora przerwań) konieczne jest wówczas zablokowanie układu przerwań (np. funkcją "di"). Oczywiście niecelowe jest to w funkcji obsługującej przerwanie podłączone do zerowego wejścia sterownika MASTER (zazwyczaj zegar czasu rzeczywistego), gdyż i tak nie wystąpi przerwanie maskowalne o wyższym od niego priorytecie. W celu przekazania informacji z wewnętrznej funkcji obsługi przerwania do monitora przerwań, którego sterownika przerwanie dotyczy, należy w każdej z tych funkcji przypisać zmiennej "pic" numer właściwego PIC-a, tzn.: pic = MASTER (0) dla Master-a, pic = SLAVE_1 (1) dla Slave_1, itd. Nadanie wartości zmiennej "pic", w przypadku wielopoziomowej obsługi przerwań, musi mieć miejsce przed wykonaniem instrukcji "return", ale koniecznie po ponownym zablokowaniu układu przerwań.

5. Sterowanie, synchronizacja, oraz wymiana informacji między zadaniami

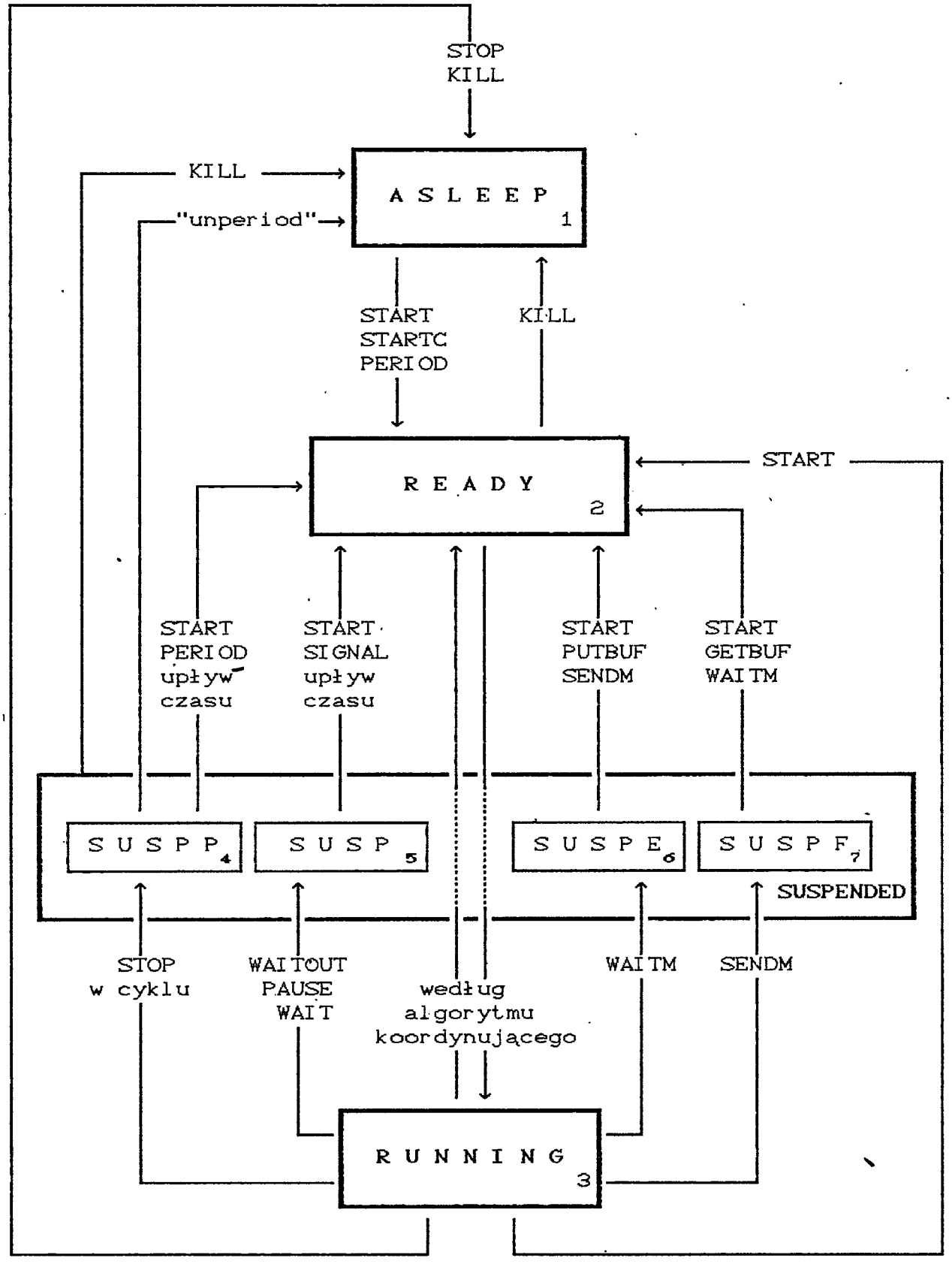
Zadanie użytkowe w systemie SIRTOS może znajdować się w jednym z czterech stanów: ASLEEP (1), READY (2), RUNNING (3) i SUSPENDED (4,5,6,7). W stanie ASLEEP znajdują się zadania nieaktywne. W stanie SUSPENDED znajdują się zadania oczekujące na spełnienie warunku:

SUSPP (4) - koniec okresu
SUSP (5) - otwarcie semafora lub upływ czasu
SUSPE (6) - adres informacji w buforze
SUSPF (7) - wolne miejsce w buforze

Wydzielenie czterech podstanów uprościło i przyśpieszyło działanie koordynatora zadań. W stanie READY znajdują się zadania gotowe do wykonania - tworzące priorytetową kolejkę. Spośród nich koordynator wybiera i uruchamia zadanie o najwyższym priorytecie (najniższym numerze). W stanie RUNNING (w systemie jednoprocesorowym) może znajdować się tylko jedno zadanie - aktualnie wykonywane. Jeżeli koordynator nie znajdzie zadania gotowego do uruchomienia, procesor zostaje zatrzymany w zadaniu systemowym (SYS) na instrukcji "hlt" - w oczekiwaniu na przerwanie zmieniające stan systemu. Zmianę stanu zadań powodują funkcje systemowe i upływ czasu. Funkcje systemowe są procedurami umożliwiającymi wykonywanie specjalnych czynności w systemie. W wyniku ich działania może ulec zmianie kolejka zadań aktywnych (READY). Przeważnie więc kończą się one wejściem do koordynatora zadań, który wyszukuje nowe zadanie o najwyższym priorytecie gotowe do uruchomienia. Funkcje te działają bezpośrednio na zasobach systemowych (np. bloku kontrolnym zadania - TCB). Ich wykonywanie nie może być z reguły przerywane i z tego powodu realizuje się je jako procedury pracujace przy zamkniętym układzie przerwań lub jako obsługę przerwań programowych. W systemie SIRTOS (wersja 2.2) przyjęto ten pierwszy sposób realizacji. Konsekwencją tego rozwiązania jest konieczność konsolidacji programu użytkowego razem z systemem operacyjnym. Wobec niewielkiej objętości jądra systemu (około 4 kB) oraz wykorzystaniu programu MAKE, nie stanowi to jednak żadnego problemu. Funkcje systèmowe pozostawiają układ przerwań w takim stanie w jakim znajdował się w momencie ich wołania. Jeżeli były one wołane przy zamkniętym układzie przerwań to ich działanie zostaje odłożone do momentu odblokowania układu przerwań. Wyjątkiem są funkcje, których użycie jest zabronione w obsłudze przerwań: PAUSE, STOP, WAIT, WAITOUT, SENDM i WAITM oraz dodatkowo START i KILL do samego siebie. Odblokowują one układ przerwań bezwarunkowo, gdyż ich działanie przy zablokowanym układzie przerwań nie ma sensu. Funkcje systemowe można podzielić na cztery grupy, omówione w kolejnych podrozdziałach. Stany zadań wraz z funkcjami systemowymi powodującymi ich zmianę przedstawia poniższy rysunek.

Uwaga: Użycie funkcji systemowych z parametrem nrt = 0 (SYS) jest zabronione.

14



Stany zadań w SIRTOS-ie. Przy strzałkach podano nazwy funkcji systemowych, powodujących zmianę stanu zadania.

5.1. Aktywacja zadań

```
void START ( numer_zadania )
     int numer_zadania ;
```

Funkcja START powoduje bezwarunkowe uruchomienie zadania o podanym numerze. Zadanie to zostaje wpisane do kolejki zadań gotowych do wykonania (READY) i rozpoczyna wykonywać się od początku (od adresu startu), zgodnie ze swoim priorytetem. Jeżeli zadanie było już wcześniej uruchomione, użycie funkcji START przerywa jego wykonanie. W takim przypadku zasoby systemowe używane przez to zadanie (np.: semafory, bufory cykliczne, itp.), pozostają w takim stanie w jakim znajdowały się w chwili przerwania zadania. Może prowadzić to do niewłaściwej pracy ponownie wystartowanego zadania lub innych zadań współpracujących z nim. Można uniknąć tego przez nadanie np. semaforom odpowiednich wartości początkowych. Każdy przypadek wymaga jednak odrębnej, szczegółowej analizy. Funkcja może być używana w obsłudze przerwań. Wywołanie przy zamkniętym układzie przerwań w zadaniu do samego siebie, odblokowuje układ przerwań.

```
int STARTC ( numer_zadania )
     int numer_zadania ;
```

Działanie tej funkcji systemowej jest analogiczne jak funkcji START, z tym że zadanie uruchamiane jest tylko wtedy, gdy jest ono w stanie uśpienia. Funkcja systemowa zrealizowana jako funkcja jezyka C typu "int", zwraca rezultat operacji - 1 (YES), gdy zadanie zostało wpisane do kolejki żadań gotowych (READY) albo 0 (NO) w przeciwnym przypadku. Funkcja może być używana w obsłudze przerwań.

```
void PERIOD ( numer_zadania, długość_cyklu )
     int numer_zadania ;
     unsigned int długość_cyklu ;
```

Parametrami funkcji systemowej są : numer uruchamianego zadania oraz długość cyklu (okres wznawiania) - podany w sekundach. Jeżeli zadanie jest śpiące (ASLEEP) wykonanie tej funkcji powoduje natychmiastowe wpisanie zadania do kolejki zadań gotowych do wykonania (READY) i cykliczne jego wznawianie z podanym okresem. Jeśli zadanie było już wcześniej aktywowane w jakikolwiek sposób i jest aktualnie wykonywane albo zawieszone, to po jego zakończeniu (wykonaniu przez nie funkcji STOP) zostanie natychmiast wpisane do kolejki zadań gotowych (READY). W przypadku, gdy okres wznawiania jest krótszy niż czas wykonania zadania ponowne jego uruchomienie nastąpi natychmiast po zakończeniu. Funkcja może być używana w obsłudze przerwań.

5.2. Deaktywacja zadań

```
void unperiod ( numer_zadania )
     int numer_zadania ;
```

Funkcja pomocnicza "unperiod" kończy cykliczne pobudzanie zadania o numerze będącym jej argumentem. Jeżeli funkcja zostanie wywołana w trakcie wykonywania zadania, nie jest ono przerywane. Funkcja może być używana w obsłudze przerwań.

```
void PAUSE ( czas_oczekiwania )
     int czas_oczekiwania ;
```

Funkcja PAUSE wstrzymuje wykonywanie zadania (zmienia jego stan na SUSP) na czas_oczekiwania wyrażony w cyklach zegarowych (100 ms). Z takim też kwantem określona jest dokładność czasu oczekiwania. Konsekwencją kwantowania czasu jest to, że np. argument funkcji 1 powoduje zawieszenie zadania na czas od 0 do 100 ms, 2 - na czas od 100 do 200 ms, itd., czyli z niedomiarem mniejszym od cyklu zegara. Oczywiście po tym czasie zadanie zostaje wpisane jedynie do kolejki zadań gotowych (READY). W rzeczywistości czas oczekiwanima może być znacznie większy ze względu na wykonywanie się zadań o wyższych priorytetach, obsługę przerwań, oraz czas obsługi własnej systemu. Używanie funkcji w obsłudze przerwań jest zabronione. Wywołanie przy zamkniętym układzie przerwań w żadaniu - odblokowuje układ przerwań.

```
void STOP ( )
```

Funkcja systemowa STOP służy do zakończenia bieżącego zadania. Zmienia ona stan zwykłego zadania na ASLEEP, zaś zadania pobudzanego periodycznie na SUSPP. Ekstrakod ten musi wystąpić w każdym zadaniu co najmniej raz i kończyć wykonywanie zadania. Wyjątkiem są zadania działające w pętli bez zakończenia (ang. never ending), np. obsługi ekranu, klawiatury, itp. Używanie funkcji w obsłudze przerwań jest zabronione. Wywołanie przy zamkniętym układzie przerwań w zadaniu - odblokowuje układ przerwań.

```
void KILL ( numer_zadania )
     int numer_zadania ;
```

Funkcja systemowa KILL powoduje natychmiastowe i bezwarunkowe przerwanie zadania o numerze będącym jej argumentem (zmianę jego stanu na ASLEEP). Należy przy tym pamiętać, że przerwanie zadania może spowodować analogiczne zjawiska jak opisane przy funkcji START - związane z przerwaniem obsługi zasobów systemowych. Funkcja może być użyta w obsłudze przerwań. Wywołanie przy zamkniętym układzie przerwań w zadaniu do samego siebie, odblokowuje układ przerwań.

5.3. Synchronizacja zadań

Do synchronizacji zadań w systemie SIRTOS służą semafory binarne oraz funkcje systemowe operujące na nich. Semafory przyjmują wartość zera logicznego, gdy nadana jest im wartość 0 (OFF) albo jedynki logicznej - w pozostałych przypadkach (system jako jedynkę logiczną nadaje zawsze wartość 1 - ON). Semafory - z punktu widzenia kompilatora jezyka C - są globalnymi zmiennymi typu "char". Można więc nadawać im wartości początkowe na etapie kompilacji programu, tak jak innym zmiennym globalnym. Nadanie wartości semaforom możliwe jest również zwykłą instrukcją podstawienia albo za pomocą funkcji "presem". Ta ostatnia metoda jest nieznacznie wolniejsza, zwiększa za to czytelność programu.

```
void presem ( adres_semafora, wartość_inicjowana )
     char *adres_semafora ;
     char wartosc_inicjowana ;
```

Funkcja nadaje wartość_inicjowaną semaforowi binarnemu, do którego wskaźnik jest jej argumentem.

Możliwe jest również odczytanie wartości semafora. Aby operacja taka była poprawna należy zarówno ją jak i jej wykorzystanie wykonywać przy zamkniętym układzie przerwań.

```
void WAIT ( adres_semafora )
     char *adres_semafora ;
```

Funkcja systemowa WAIT wykonuje operację P na semaforze binarnym. Jeśli w chwili wywołania funkcji semafor, którego adres jest jej argumentem, jest "podniesiony" (przyjmuje wartość jedynki logicznej), to wynikiem działania funkcji jest "zamknięcie" semafora (realizowane przez zmianę jego wartości na zero logiczne - OFF), a wykonywanie zadania jest kontynuowane; w przeciwnym przypadku, zadanie zostaje zawieszone (SUSP) w oczekiwaniu na podniesienie semafora przez inne zadanie lub obsługę przerwania. Po podniesieniu semafora system automatycznie odwiesza zadanie wpisując je do kolejki zadań gotowych (READY). Używanie funkcji w obsłudze przerwań jest zabronione. Wywołanie przy zamkniętym układzie przerwań w zadaniu - odblokowuje układ przerwań.

```
int WAITOUT ( adres_semafora, ogranicznik_czasowy )
     char *adres_semafora ;
     int ogranicznik_czasowy ;
```

Funkcja systemowa WAITOUT jest rozszerzeniem funkcji WAIT. Przez odcinek czasu mniejszy od ogranicznika czasowego jej działanie jest identyczne z WAIT. Jeżeli czas zawieszenia zadania - określony liczbą cykli zegarowych (100 ms), przekroczy ograniczenie, to zadanie zostaje odwieszone i wpisane do kolejki zadań gotowych (READY). Funkcja zwraca kod przyczyny odwieszenia zadania - 1 (ON), gdy przyczyną odwieszenia było przekroczenie ograniczenia czasowego albo 0 (OFF), gdy odwieszenie zadania zostało spowodowane

podniesieniem semafora. Funkcja jest bardzo przydatna w przypadkach, gdy w wyniku działania programu oczekujemy przerwania, np.: przy obsłudze przetworników A/C, układów sprzęgu szeregowego (USART), itp. Użycie w takich przypadkach funkcji WAITOUT zamiast WAIT zapewnia możliwość wykrycia i obsługi sytuacji awaryjnej, polegającej na braku przerwania w przewidywanym czasie (spowodowanym np. awarią przetwornika). Oczywiście czas podany w ograniczniku czasowym musi być w takich przypadkach większy niż przewidywany czas oczekiwania na przerwanie. Używanie funkcji w obsłudze przerwań jest zabronione. Wywołanie przy zamkniętym układzie przerwań w zadaniu - odblokowuje układ przerwań.

```
void SIGNAL ( adres_semafora )
        char *adres_semafora ;
```

Funkcja ta stanowi parę dla funkcji systemowej WAIT (albo WAITOUT). Wykonuje ona operację V na semaforze binarnym. Podnosi semafor (nadaje mu wartość jedynki logicznej - ON), którego adres jest jej argumentem. Jednocześnie wpisuje ona do kolejki zadań gotowych (READY) wszystkie zadania oczekujące na tym semaforze. Funkcja może być wołana z obsługi przerwań.

5.4. Wymiana informacji pomiędzy zadaniami

Funkcje systemowe tej grupy służą do wymiany informacji pomiędzy zadaniami użytkowymi, wykorzystując w tym celu mechanizm tzw. skrzynki pocztowej (ang. mailbox). Skrzynki pocztowe są zrealizowane w systemie jako struktury typu "bch" i współpracują one z tablicą wskaźników do łańcuchów znaków, wykorzystaną w charakterze bufora cyklicznego. Definicja skrzynki powinna mieć postać:

```
char *nazwa_tablicy [ rozmiar_tablicy ] ;
struct bch nazwa_bufora =
        { EMPTY, rozmiar_tablicy, 0, 0, adres_tablicy } ;
```

Tablica może być lokalna. Liczba skrzynek pocztowych w systemie nie jest ograniczona. Stała EMPTY znajduje się w zbiorze stałych systemowych i przyjmuje wartość -1.

```
void SENDM ( adres_bufora, adres_informacji )
        struct bch *adres_bufora ;
        char *adres_informacji ;
```

Funkcja SENDM wpisuje adres informacji do skrzynki pocztowej, zorganizowanej jako bufor cykliczny. Adresy informacji w skrzynce pocztowej tworzą kolejkę - adres nowej informacji wpisywany jest na jej koniec (długość kolejki jest określana przy definiowaniu bufora). W przypadku braku miejsca w kolejce, zadanie zostaje zawieszone (SUSPF) do chwili zwolnienia miejsca na adres nowej informacji. Po zwolnieniu miejsca w skrzynce pocztowej system automatycznie wpisuje adres, na który poprzednio nie było miejsca, na koniec

kolejki adresów i odwiesza zadanie - wpisując je do kolejki zadań gotowych (READY). Używanie funkcji w obsłudze przerwań jest zabronione. Wywołanie przy zamkniętym układzie przerwań w zadaniu - odblokowuje układ przerwań.

```
void WAITM ( adres_bufora, adres_informacji )
     struct bch *adres_bufora ;
     char **adres_informacji ;
```

Odwrotne działanie do funkcji SENDM ma funkcja systemowa WAITM. Pobiera ona adres najstarszej informacji ze skrzynki pocztowej, będącej jej pierwszym argumentem i zwraca go pod adres, który jest jej drugim argumentem. W przypadku braku informacji - zadanie zostaje zawieszone (SUSPE) do chwili pojawienia się adresu informacji w skrzynce pocztowej. Z chwilą wpisania do skrzynki pocztowej adresu informacji system operacyjny automatycznie odwiesza zadanie - wpisując je do kolejki zadań gotowych (READY). Używanie funkcji w obsłudze przerwań jest zabronione. Wywołanie przy zamkniętym układzie przerwań w zadaniu - odblokowuje układ przerwań.

```
int PUTBUF ( adres_bufora, adres_informacji )
     struct bch *adres_bufora ;
     char *adres_informacji ;
```

```
int GETBUF ( adres_bufora, adres_informacji )
     struct bch *adres_bufora ;
     char **adres_informacji ;
```

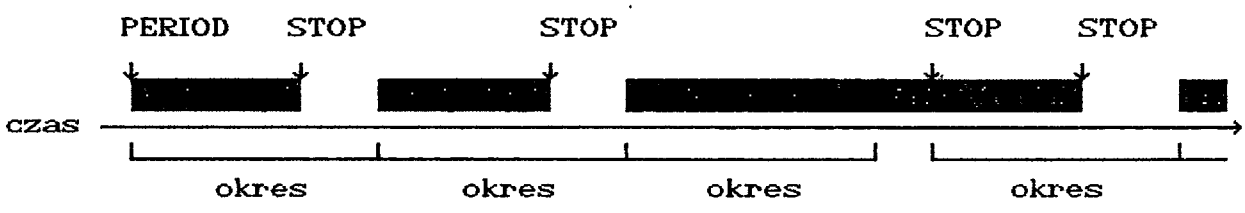
Analogiczne jak funkcje SENDM i WAITM działają odpowiednio funkcje systemowe PUTBUF i GETBUF. Jedyna różnica polega na tym, że w przypadku pełnego, bądź pustego bufora zadanie nie ulega automatycznie zawieszeniu. Funkcje te zwracają w takim przypadku jedynkę logiczną (ON), gdy operacja wpisania albo pobrania adresu ze skrzynki pocztowej nie powiodła się. W przeciwnym przypadku zwracane jest zero logiczne (OFF). Sposób obsługi obu tych przypadków należy do programu użytkowego. Funkcje PUTBUF i GETBUF mogą być wykorzystywane w procedurach obsługi przerwań.

Funkcje systemowe do wymiany adresów informacji działają według algorytmu FIFO. Możliwe jest ich mieszanie w parach, np. użycie PUTBUF-a z WAITM, itp. Przekazywana informacja może oczywiście mieć inny typ niż "char". Należy wówczas użyć operatora zamiany typów danych (cast), zarówno przy przesyłaniu adresu do bufora jak i przy jego pobieraniu.

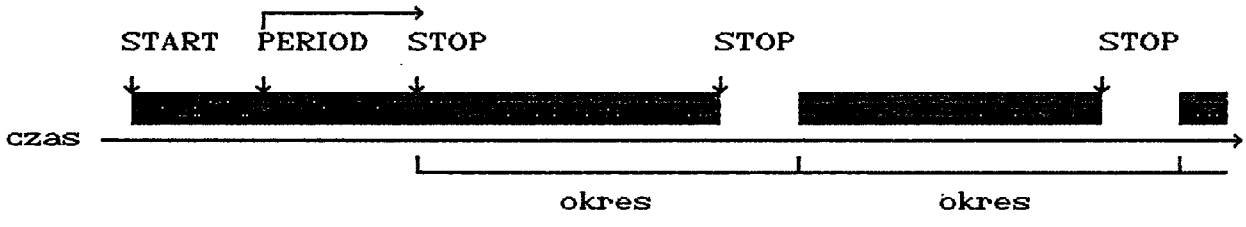
6. Uzależnienia czasowe i przykłady zastosowania funkcji systemowych

6.1. Działanie funkcji aktywacji i deaktywacji zadania

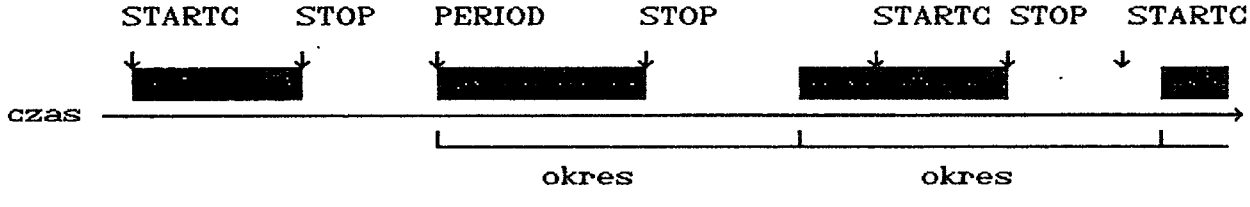
Niektóre przypadki uzależnień czasowych dla funkcji systemowych przedstawiają poniższe diagramy, na których zaczerniony prostokąt symbolizuje zadanie aktywne (nie będące w stanie ASLEEP).



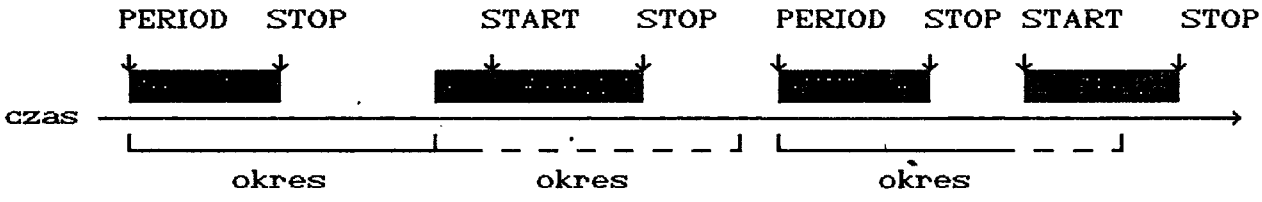
Przykład zachowania się systemu w przypadku, gdy okres pobudzania jest krótszy od czasu wykonania. Zadanie wznawiane jest natychmiast po zakończeniu (wykonaniu przez nie funkcji STOP).



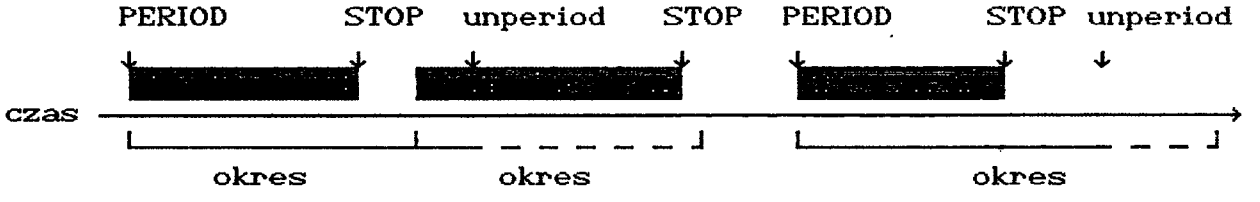
Działanie funkcji systemowej PERIOD wywołanej do zadania aktywnego (nie będącego w stanie ASLEEP). System rozpoczyna aktywację periodyczną zadania natychmiast po jego zakończeniu.



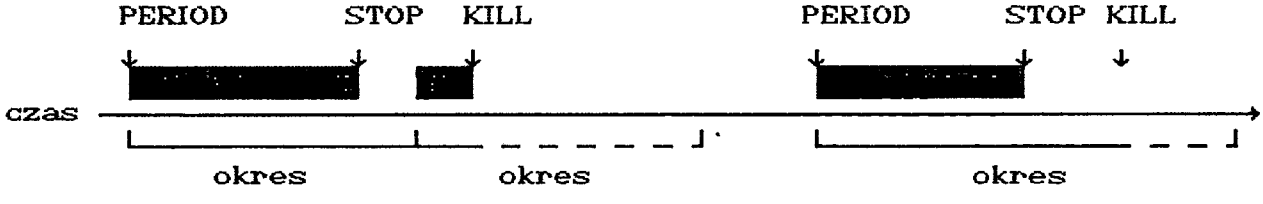
Przykłady działania funkcji systemowej STARTC. W drugim i trzecim przypadku nie powoduje ona żadnej akcji systemu, ponieważ została użyta do zadania aktywnego.



Przykład działania funkcji systemowej START w analogicznych przypadkach jak poprzednio - natychmiastowy start zadania od początku (od adresu wejścia).



Przykład zakończenia okresowego pobudzania zadania za pomocą funkcji systemowej "unperiod".



Przerwanie wykonywania zadania za pomocą funkcji systemowej KILL.

6.2. Wybrane przykłady wykorzystania funkcji systemowych

Sposób nadawania semaforom wartości początkowej oraz sposób korzystania z nich przedstawiają poniższe przykłady :

```
      .
      .
      .
char semafor1 = ON ;          /* nadanie wartości początkowej */
char semafor2, semafor3 ;     /* kompilator nadaje wartość 0  */
      .                       /* - ( OFF )                    */
      .
      .
presem ( &semafor2, ON ) ; /* nadanie wartości semaforom */
semafor3 = ON ;
      .
      .
      .
di ( ) ;                     /* zablokowanie układu przerwań */
if ( semafor1 ) {            /* sprawdzenie stanu semafora   */
            .
            .                /* semafor ustawiony - ON */
}
```

```
else {
            .
            .                    /* semafor zamknięty - OFF */
            .
}
ei ( ) ;                        /* odblokowanie układu przerwań */
      .
      .
      .
```

Za pomocą pary funkcji systemowych WAIT i SIGNAL można zorganizować synchronizację zadań np:

```
zadanie 1                                          zadanie 2

      .                                                  .
      .                                                  .
      .                                                  .
char sem = OFF ;                                   extern char sem ;
      .                                                  .
      .                                                  .
      .                                                  .
WAIT ( &sem ) ;      /* synchronizacja */          SIGNAL ( &sem ) ;
      .              /* na semaforze   */                .
      .                                                  .
      .                                                  .
```

oraz ochronę obszaru krytycznego :

```
zadanie 1                                          zadanie 2

      .                                                  .
      .                                                  .
      .                                                  .
char sem = ON ;                                    extern char sem ;
      .                                                  .
      .                                                  .
      .                                                  .
WAIT ( &sem ) ;                                    WAIT ( &sem ) ;
      .              /* obszar krytyczny  */             .
      .              /* - wykorzystanie   */             .
      .              /* wspólnych zasobów */             .
SIGNAL ( &sem ) ;                                  SIGNAL ( &sem ) ;
      .                                                  .
      .                                                  .
      .                                                  .
```

Wykorzystanie funkcji systemowej WAITOUT, z jednoczesną ochroną obszaru krytycznego - dostępu do przetwornika A/C.

```
zadanie
      .
      .
      .
WAIT ( &przetwornik ) ;      /* zajęcie obszaru krytycznego  */
presem ( &semafor, OFF ) ; /* wstępne opuszczenie semafora */
initac ( ... ) ;             /* procedura inicjująca pracę
                                   przetwornika A/C              */
if ( WAITOUT ( &semafor, MAXTIME ) {
      .                          /* obsługa błędnego działania   */
      .                          /* przetwornika        - brak   */
      .                          /* przerwania w czasie MAXTIME  */
}                                .
else {
      .                          /* obsługa prawidłowego    */
      .                          /* działania przetwornika */
      .
}
SIGNAL ( &przetwornik ) ;  /*zwolnienie obszaru krytycznego*/
      .
      .
      .
```

"zewnętrzna" funkcja obsługi przerwania z przetwornika A/C :

```
void opac ( )                  /* funkcja w wektorze przerwań */
      {
      savereg ( ) ;            /* zachowanie rejestrów na stosie*/
      servints ( ac, &semafor ) ;
}
```

"wewnętrzna" funkcja obsługi przerwania

```
int ac ( semafor )
      char *semafor ;
      {
      .
      .                          /* odczyt przetwornika          */
      .                          /* zdjęcie przerwania           */
      SIGNAL ( semafor ) ;  /* podniesienie semafora          */
      pic = 0 ;      /* ustawienie numeru sterownika przerwań */
      return ( ON ) ;          /* powrót z przerwania z wejściem*/
}                               /* do koordynatora zadań        */
```

Inny problem stwarza współpraca typu producent - konsument obsługi przerwań i zadania użytkowego. Ponieważ obsługa przerwania wykonuje się przy zablokowanym układzie przerwań i nie można użyć semaforów do synchronizacji z zadaniem użytkowym, ochrona rejonu krytycznego (w tym przypadku zmiennych obsługujących bufor) została zrealizowana poprzez zablokowanie układu przerwań w zadaniu. W przykładzie wykorzystano fakt, że funkcja systemowa WAIT odblokowuje układ przerwań.

"wewnętrzna" funkcja obsługi przerwania RxRdy

```
      .
      .
      .
char bufwe [ DBW ] ;
int pwm ;
int oz ;
char stbwe = EMPTY ;
      .
      .
      .
tmp = readb ( UDA0 ) ;          /* odczytanie znaku i zdjęcie
                                     przerwania */
if ( stbwe == FULL )
    return ( NO ) ;             /* znak nie został przyjęty
                                     - powrót do zadania     */
*( bufwe + pwm++ ) = tmp ;      /* wpisanie znaku do bufora */
if ( pwm == DBW )               /* czy koniec bufora ? */
    pwm = 0 ;
stbwe = pwm == oz ? FULL : ACCESS ; /* aktualizacja stanu
                                       bufora */
SIGNAL ( semafor ) ;
return ( YES ) ;    /* koniec komunikatu - powrót do        */
      .             /* przerwanego zadania z przeszukaniem */
      .             /* kolejki przez koordynator            */
      .
```

zadanie konsument

```
      .
      .
      .
extern char bufwe [ ], stbwe ;
extern int pwm, oz ;
      .
      .
      .
dis ( ) ;                       /* zablokowanie układu przerwań */
if ( stbwe == EMPTY ) {
    WAIT ( &keyboard ) ;        /* czekaj na przerwanie
                                     z klawiatury */
    dis ( ) ;                   /* zablokowanie układu przerwań */
}
mark = *( bufwe + oz++ ) ;  /* pobranie znaku z bufora
                                wejściowego */
if ( oz == DBW )            /* aktualizacja wskaźnika
                                ostatniego znaku */
    oz = 0 ;
stbwe = oz == pwm ? EMPTY : ACCESS ;      /* aktualizacja
                                             stanu bufora */
eis ( ) ;                   /* odblokowanie układu przerwań  */
      .
      .
      .
```

7. Generacja i instalacja kompletnego systemu

Przygotowanie oprogramowania aplikacyjnego pracującego pod systemem SIRTOS polega na napisaniu zadań użytkowych w języku C, z możliwością wykorzystania wstawek i funkcji asemblerowych, a następnie kompilacji i konsolidacji z systemem. W celu kompilacji i konsolidacji, a następnie konwersji na kod INTEL - HEX (wymagany przez programator pamięci PROM lub program transmisji do pamięci RAM sterownika) bardzo przydatny jest program MAKE, spełniający rolę generatora systemu aplikacyjnego. Zadaniem twórcy aplikacji jest poprawne sformułowanie zbioru "makefile", zawierającego strukturę zbiorów systemu aplikacyjnego, parametry kompilacji (np. opcje +F kompilatora) i parametry konsolidacji (np. wartość rejestru CS, nazwy bibliotek) oraz parametry konwersji na kod "hex" (np. pojemność układu pamięci PROM).

Należy zwrócić uwagę na konieczność modyfikacji modułu inicjacji systemu (biblioteczny zbiór "srom.o" dla "małego modelu"), który konsolidator umieszcza na początku kodu wynikowego programu. Niezbędne są dwie modyfikacje:

- zamiana instrukcji "sti", przedwcześnie odblokowującej układ przerwań, na "nop" (FB na 90 pod adresem 5B)
- ustawienie wartości rejestru DS (standardowo na 0003 pod adresem 07), ze względu na niedziałanie opcji -d konsolidatora wersji 3.40

Powyższe modyfikacje można wykonać np. w programie transmisji kodu "hex" z IBM PC do pamięci RAM sterownika lub w monitorze operatorskim sterownika.

7.1. Postać zbioru "makefile"

Dla przedstawionego w poprzednich punktach przykładowego systemu aplikacyjnego (zadania: TEH, MON, PRT, MMC, i BGT) zbiór "makefile" o nazwie "nu" ma postać:

```
*=sirtos.
R=srom.o
S=srts.o ecods.o init.o ints.o
A=inita.o intsa.o proc.o
I=vi.o
U=uinit.o uints.o usersfs.o

T=tteh.o tmon.o tprt.o tmmc.o tbgt.o ram.o
M=-c 100 -d 3
K=8

$*hex: $*exe
        @hex86 -z -s$K $*.exe
        @echo SIRTOS OK!
        @echo ms makefile end

$*exe: $I $R $S $A $U $T
```

```
        @ln -T $M -o $@ $I $R $S $A $U $T -lc

$S:     cc +F $*
$U:     cc +F $*

srts.o:     gsc.h gse.h ees.h gcc.h
ecods.o:    gsc.h gse.h       gcc.h
init.o:     gsc.h gse.h ees.h gcc.h guc.h gue.h
ints.o:     gsc.h gse.h       gcc.h guc.h gue.h gic.h
vi.o:                         gcc.h                   gfe.h

uinit.o:    gsc.h gse.h       gcc.h guc.h gue.h gic.h gfe.h
uints.o:    gsc.h gse.h ees.h gcc.h guc.h gue.h       gfe.h
usersfs.o:                    gcc.h guc.h

tteh.o:     gsc.h       ees.h gcc.h guc.h gue.h gic.h
tmon.o:                 ees.h gcc.h guc.h       gic.h
tprt.o:                 ees.h gcc.h guc.h       gic.h
tmmc.o:                 ees.h gcc.h guc.h gue.h       gfe.h mmc.h
tbgt.o:                 ees.h gcc.h guc.h gue.h       gfe.h
ram.o:                  ees.h gcc.h guc.h gue.h
```

Uwaga: Wytłuszczonym drukiem wyróżniono niezmienne części zbioru "makefile".

Wywołanie programu "make" dla tego programu ma postać:

```
make -f nu
```

Należy zwrócić uwagę, że wynikiem procesu generacji będzie jeden zbiór o nazwie "sirtos.hex". W przypadku potrzeby podziału tego zbioru na mniejsze (np. 2 kilobajtowe) należy wykorzystać opcję "p" programu "hex86". Można w ten sposób utworzyć do 100 zbiorów "hex" (o nazwach "sirtos.h00", "sirtos.h01", "sirtos.h02", itd.).

7.2. Mapa pamięci systemu uruchomieniowego i docelowego

Po wygenerowaniu systemu aplikacyjnego oraz wstępnym uruchomieniu OFF-line (np. za pomocą debuggera AZTEC-a) fragmentów zadań użytkowych, należy dokonać próbnej instalacji całego systemu w docelowym sterowniku. Do czasu uzyskania wiarogodnej wersji zaleca się testowanie i stopniową rozbudowę oprogramowania wyłącznie w pamięci RAM. Poniższa tabela przedstawia różnicę wykorzystania pamięci sterownika, z mikroprocesorem INTEL 8086, na etapie uruchamiania i eksploatacji systemu.

Adres (hex)	Model pamięci w systemie uruchomieniowym	Model pamięci w systemie docelowym	Adres (hex)
00000 00*** RAM max 64 kB	wektory przerwań M86* i SIRTOS-a pole robocze M86 dane zainicjowane systemowe i użytkowe stosy zadań dane niezainicjowane systemowe i użytkowe	wektor przerwań SIRTOS-a dane zainicjowane systemowe i użytkowe stosy zadań dane niezainicjowane systemowe i użytkowe	00000 00*** RAM max 64 kB
RAM max 64 kB	jądro systemu SIRTOS kod zadań użytkowych kopia danych zainicjowanych	jądro systemu SIRTOS kod zadań użytkowych kopia danych zainicjowanych	ROM max 64 kB
ROM FFFF0 FFFFF	kod monitora M86 argument długiego skoku do M86	argument długiego skoku do SIRTOS-a	ROM FFFF0 FFFFF

* - monitor operatorski
*** - maksymalna długość wektora przerwań 1 kB (0 - 3FF)

7.3. Podstawowe parametry konfigurowania aplikacji

W trakcie tworzenia systemu aplikacyjnego, naturalną rzeczą jest jego przebudowa oraz rozbudowa. Powoduje to potrzebę dołączania dodatkowych zadań, zmiany w długości stosu zadania, przesuwanie początku kodu, itp. W celu umożliwienia użytkownikowi systemu SIRTOS dokonywania tego typu modyfikacji część parametrów jest zdefiniowana za pomocą dyrektyw "#define" preprocesora C. Parametry, charakteryzujące daną aplikację to:

- CS - wartość rejestru segmentowego kodu programu
- NRTMAX - maksymalny numer zadania w systemie
- SQ - wielkość kwantu pamięci stosu zadania
- K2 - łączna długość wszystkich stosów

8. Zalecana forma oprogramowania użytkowego

Przy pisaniu dużych programów w dowolnym jezyku poważnym problemem staje się rosnąca liczba odwołań i zmiennych globalnych. Bardzo ważny staje się wówczas tzw. styl programowania. Polega on na pisaniu programów w ściśle określonym formacie, strukturalnych oraz podzielonych na moduły. Za moduł uważać będziemy niezależnie kompilowany zbiór funkcji dostarczający pewnych usług innym fragmentom programu (też modułom). Wyróżnia się w nich wyraźnie funkcje i zmienne eksportowane (wynoszone poza moduł), oraz wewnętrzne (używane tylko w jednym module). Podstawowymi zaletami korzystania z modularyzacji są :

- ułatwiona dokumentacja programów
- zmniejszenie liczby powiązań
- umieszczenie w jednym miejscu fragmentów programu uzależnionych od sprzętu (przy zmianie sprzętu zachodzi wówczas konieczność wymiany tylko tego modułu)

Mechanizmami umożliwiajacymi budowę modularnej struktury dużych programów w jezyku C są :

- niezależna kompilacja zbiorów
- klasy pamięci "static" i "extern"
- definiowanie funkcji i zmiennych w plikach nagłówkowych (ang. header files)

8.1. Kod programu i zmienne

W module należy umieszczać funkcje dotyczące tematycznie jednego problemu. Stosować konsekwentnie klasę "static" dla funkcji i zmiennych wykorzystywanych tylko wewnątrz modułu, oraz umieszczać deklaracje zmiennych i funkcji wynoszone na zewnątrz (eksportowane) w zbiorach nagłówkowych. Dołącza je później preprocesor zleceniem "#include". Język C jedynie umożliwia modularyzację programu, ale jej nie wymusza (robią to niektóre z nowszych języków programowania). Nieumiejętny lub niekonsekwentny podział programu na moduły powoduje nadmierne rozrastanie się zbiorów nagłówkowych i niejasną strukturę programu, utrudniając w ten sposób jego uruchamianie, a nawet może prowadzić do błędów.

8.2. Dane użytkowe

W przypadku przygotowywania programu w wersji dostosowanej do pamieci ROM należy korzystać z faktu, że wszystkie zmienne globalne, którym nie została nadana wartość zostają wstępnie wyzerowane przez kompilator i nie należy nadawać im wstępnie wartości 0. Nadanie zmiennym globalnym wartości 0 powoduje umieszczenie ich pośród zmiennych zainicjowanych w pamięci ROM, a następnie przepisanie ich do pamięci RAM bezpośrednio przed uruchomieniem programu. Niepotrzebnie wzrasta w ten sposób zajętość pamięci ROM.

Nie zaleca się również stosowania zmiennych typu "char" w obliczeniach matematycznych i dla parametrów funkcji. Ponieważ obliczenia są przeprowadzane na zmiennych typu "int" - kompilator zmienia typ zmiennej. Również parametry funkcji przekazywane są przez stos zawsze jako "int". Zastosowanie zmiennych typu "char" ma sens wtedy, gdy rzeczywiście jest ich bardzo dużo i są przetwarzane jednakowo - zaoszczędzone miejsce na ich przechowywanie może wówczas być większe niż strata spowodowana zamianą typów (należy jednak liczyć się z pewnym zwolnieniem obliczeń), albo gdy są one traktowane jako wartości logiczne, bądź gdy nie są przetwarzane (np. teksty).

8.3. Ograniczenia wynikające z modelu pamięci

Opisywana wersja systemu operacyjnego SIRTOS pracuje w tzw. małym modelu pamięci - mały kod programu, małe dane. Kompilacja bez opcji, dotyczącej modelu pamięci, powoduje przyjęcie małego modelu. Dodatkowo, jest wymagane dołączenie modułu "srom.o" w trakcie konsolidacji programu. Mały model pamięci wprowadza ograniczenie długości kodu programu do 64 kB oraz ogranicza zajętość pamięci przez sumę wszystkich zmiennych globalnych, statycznych, stosu i sterty również do 64 kB. Dzięki temu wszystkie odwołania w programie - zarówno do funkcji jak i danych, są krótkie, bez modyfikacji rejestrów segmentowych. Rejestry te (DS, ES, SS i CS) są ustawiane przez kompilator tylko raz na początku programu. Ponadto rejestry ES i DS mają jednakową wartość. Kod programu jest dzięki temu zwarty i wykonuje się szybko. W małym modelu pamięci kompilator przydziela 2 kB na stos. W przypadku dużej liczby zadań użytkowych obszar ten może okazać się zbyt mały. Należy wówczas odpowiednio zainicjować zmienną "bstack" oraz przedefiniować parametr K2 (modyfikacja parametrów stosu w zbiorze bibliotecznym "stksiz.c dla wersji 3.40 jest nieskuteczna).

8.4. Zasady uruchamiania i testowania programu

W wielozadaniowych systemach czasu rzeczywistego dużym problemem jest uruchomienie wielu zadań użytkowych i procedur obsługujących przerwania. Każdy przypadek wymaga indywidualnego podejścia, można jednak sformuować pewne ogólne zasady uruchamiania i testowania programów wielozadaniowych.

Poszczególne zadania należy pisać jako oddzielne moduły. W razie potrzeby zadanie można również dzielić na mniejsze fragmenty. Należy korzystać z możliwości preprocesora, w szczególności umieszczać definicje funkcji i zmiennych w zbiorach nagłówkowych. Stosować konsekwentnie klasy pamięci "static" i "extern", jednym słowem - "dobry styl programowania".

Najwyższe priorytety (najniższe numery) powinny mieć zadania wykonywane często i szybko - najniższe, zadania wykonujące się długo i rzadko. Podobnie jest z przerwaniami. Procedury obsługi przerwań powinny decydować o przeszukiwaniu kolejki zadań tylko w przypadku, gdy zmieniają stan zadań.

Ponadto długie procedury obsługi przerwań -jeśli to możliwe- powinny odblokowywać układ przerwań, umożliwiając w ten sposób ich hierarchiczną obsługę.

W pierwszym etapie warto sprawdzić poprawność działania istotnych procedur użytkowych (bez wywoływania funkcji systemowych SIRTOS-a) pod debuggerem "db" systemu AZTEC - z możliwością pracy krokowej, stosowania pułapek (ang. breakpoints), śledzenia zmian wartości zmiennych, itp.; próby te wykonujemy na komputerze IBM PC pod PC DOS-em. Następnie, po przesłaniu całego systemu aplikacyjnego do sterownika, sprawdzamy poprawność oprogramowania pod systemem SIRTOS, z wykorzystaniem prostego monitora operatorskiego (np. M86), zawierającego zazwyczaj podstawowe operacje śledzenia poprawności programu (np.: uruchomienie z ustawioną pułapką, odczyt lub modyfikacja zawartości pamięci, przesuwanie bloku danych, itp.). Pamiętać należy jednak, że użycie go zakłóca zależności czasowe pomiędzy zadaniami. Na tym etapie przydatna jest znajomość budowy bloku kontrolnego zadania (TCB), zawartości stosów oraz wydruk tzw. tabeli symboli globalnych - zbioru "sirtos.sym" (opcja -T konsolidatora). System należy uruchamiać stopniowo, zaczynając od jednego zadania i jednego przerwania - najlepiej zegarowego. Stopniowo należy zwiększać liczbę zadań i obsługiwanych przerwań. Uruchamiając przerwania można posługiwać się funkcjami "mint" i "unmint" albo maskować je bezpośrednio zmieniając parametry funkcji "picinit" - inicjującej sterowniki przerwań. Zmiany i rozszerzenia należy wprowadzać etapami, po wyjaśnieniu i usunięciu wszystkich spostrzeżonych błędów. Należy unikać wprowadzania jednorazowo dużych zmian - jak również wielu zmian w różnych miejscach programu. Do dobrej praktyki programowania należy również przechowywanie co najmniej jednej kopii pisanego programu na dyskietkach i częsta jej aktualizacja.

9. Najczęściej spotykane błędy

W kolejnych podrozdziałach omówiono w skrócie niektóre błędy nie wykrywane przez kompilator i konsolidator systemu AZTEC-C oraz błędy popełniane w programowaniu pod systemem SIRTOS.

9.1. Błędy programowania w języku C

Niezgodność typów, polegająca na umieszczeniu w jednym module definicji "double x", a w drugim deklaracji "extern x" powoduje domyślnie przyjęcie "extern int x". Spowoduje to w konsekwencji trudny do wykrycia błąd, nie sygnalizowany ani w fazie kompilacji, ani konsolidacji programu. Umieszczenie deklaracji "extern" w zbiorze nagłówkowym, a następnie włączenie go zleceniem preprocesora "#include" do obu modułów pozwala uniknąć tego typu błędów.

Niezgodność parametrów formalnych i aktualnych (brak użycia konwersji typu - operator "cast"); np.: "long" dla stałych i wywołanie funkcji ze złym typem argumentu, podanie wartości zamiast wskaźnika. Błąd powoduje zwykle złe przekazanie parametrów i zaburzenie stosu.

Zły przydział pamięci, powodujący wzajemne zachodzenie na siebie obszarów kodu programu, danych lub stosu.

Przekroczenie dozwolonej wartości indeksu i wyjście poza tablicę (np. wywołanie funkcji systemowej z nieistniejącym - zbyt dużym - numerem zadania)

Użycie wskaźnika (ang. pointer) bez nadania mu wartości.

9.2. Błędy programowania pod systemem SIRTOS

Do złego stylu programowania w systemach czasu rzeczywistego należy pisanie zadań w sposób zależny od ich względnego priorytetu. Zmiana priorytetów powoduje wówczas trudny do wykrycia błąd w działającym poprzednio programie. Prawidłowe użycie semaforów i skrzynek pocztowych pozwala na uniknięcie tego błędu.

Nieumieszczenie funkcji systemowej STOP na końcu zadania użytkowego. Kompilator przed ostatnim nawiasem zamykającym, kończącym każdą funkcję, dopisuje instrukcję "return" (o ile nie została napisana jawnie). Wykonanie jej w funkcji głównej zadania nie ma żadnego sensu, powoduje natomiast nieprzewidziane zachowanie się programu (powyższa uwaga nie dotyczy zadań działających w pętli nieskończonej).

Błędy synchronizacji zadań, polegające na użyciu nie zainicjowanych semaforów.

Przydzielenie zadaniu zbyt małego obszaru stosu. Zadanie niszczy wówczas początek stosu następnego, w kolejności priorytetów, zadania. System wykrywa uszkodzenie stosu i zatrzymuje procesor instrukcją "hlt" (próba kontynuowania pracy powoduje nieprzewidziane zachowanie się programu).

Próba uruchomienia nie zainstalowanego zadania; powoduje odwołanie się do bloku kontrolnego, w którym zamiast adresów: stosu, wejścia do zadania, itd. znajdują się zera.

W przypadku korzystania z hierarchicznej obsługi przerwań, należy w procedurze obsługi odblokować układ przerwań procesora. Przed końcem procedury (wykonaniem instrukcji "return"), należy układ ponownie zablokować. Niezablokowanie układu przerwań spowoduje niemożność prawidłowego zakończenia obsługi (odtworzenia rejestrów, wysłania sygnału zakończenia obsługi przerwania do sterownika PIC, itp.).

Podobny błąd polega na nadaniu wartości zmiennej "pic" (lub nienadaniu żadnej wartości) w "wewnętrznej" funkcji obsługi przerwania, przed ponownym zablokowaniem układu przerwań - przy wielopoziomowej obsłudze.

Użycie głównej funkcji systemowej (np. SIGNAL), po zablokowaniu układu przerwań za pomocą funkcji "dis" - w zadaniu użytkowym, spowoduje niepożądane odblokowanie układu.

Analogiczy błąd, polegający na użyciu głównej funkcji systemowej w obsłudze przerwań po odblokowaniu układu za pomocą funkcji "eis" - spowoduje niepożądane zablokowanie układu. Dwa ostatnie błędy nie wystąpią, gdy użyjemy odpowiednio funkcji "di", "ei".

10. Podstawowe zadania użytkowe i związane z nimi funkcje obsługi przerwań

Do systemu operacyjnego SIRTOS dołączono uniwersalne zadania użytkowe: TEH, MON i PRT, obsługujące odpowiednio klawiaturę, ekran monitora i drukarkę. Zadania te współpracują z przerwaniami RxRDY, TxRDY i INTB, obsługiwanymi przez funkcje: "txrdy", "rxrdy" i "intb". Funkcje te znajdują się w zbiorze "uints.c". Do komunikacji pomiędzy zadaniami wykorzystano mechanizm buforów cyklicznych, obsługiwanych przez funkcje systemowe SENDM i WAITM oraz PUTBUF i GETBUF. System uzupełniono dodatkowo o obsługę przerwania zegara czasu rzeczywistego - funkcja "timer".

Od strony sprzętowej klawiatura i ekran monitora współpracuje z mikrokomputerem przez łącze szeregowe V24. Zarówno ze strony mikrokomputera jak i monitora, łącze to jest obsługiwane przez programowalny układ USART-a INTEL 8251A. Przyjęta szybkość transmisji wynosi 4800 bitów/s. Ze strony monitora jest ona ustawiana za pomocą połaczeń krosowych, natomiast ze strony procesora jest zadawana przez zaprogramowanie układu timera INTEL 8253 (realizuje to funkcja "timinit") i USART-a (.funkcja "initV24"). Układ USART-a jest zaprogramowany jednocześnie na odbiór i nadawanie. W czasie pracy generuje on dwa przerwania RxRDY - gotowość do odczytu znaku, oraz TxRDY - gotowość do transmisji. Łącze równoległe, zarówno od strony mikroprocesora jak i drukarki, obsługuje programowalny układ typu INTEL 8255. Brama B tego układu została zainicjowana do pracy w trybie 1 - jako wyjście (funkcja "piainit"). Układ ten generuje przerwanie INTB, zgłaszające swoją gotowość do transmisji.

10.1. Obsługa ekranu monitora

Do obsługi ekranu monitora służy zadanie MON. Steruje ono wysyłaniem komunikatów na ekran. Komunikatem jest ciąg bajtów, z których pierwszy stanowi semafor, a ostatni - kończący komunikat - jest zerowy (null byte). Pierwszy i ostatni bajt komunikatu nie są wysyłane na ekran. Komunikat musi zawierać sekwencje sterujące ekranem monitora. Zadania użytkowe albo procedury obsługi przerwań wysyłają adresy komunikatów do bufora ("skrzynki pocztowej") "buc_1" za pomocą funkcji systemowych SENDM albo PUTBUF. Zadanie MON czeka zawieszone na pojawienie się adresu komunikatu w buforze. Po pojawieniu się adresu komunikatu, zadanie zostaje uruchomione przez system operacyjny. Odmaskowuje ono przerwanie TxRDY i sprawdza gotowość USART-a do wysłania znaku (zakłada się, że ten port szeregowy nie jest wykorzystywany przez inne zadania); następnie drugi znak komunikatu wysyła na ekran monitora, po czym zawiesza się w oczekiwaniu na zakończenie komunikatu. Wysłanie komunikatu kolejno znak po znaku, aż do napotkania znacznika końca informacji (ang. null byte) , przejmuje obsługa przerwania TxRDY (funkcja "txrdy"). Po wysłaniu, komunikat zostaje oznaczony przez podniesienie semafora, który jest jego pierwszym bajtem. W razie potrzeby umożliwia to synchronizację zadań z wypisywaniem komunikatów na ekran. Jeżeli wypisywanie komunikatu nie zakończy się w określonym czasie (jest on zdefiniowany przez parametr MAXKOM), to system operacyjny uruchamia zadanie MON, które rozpoczyna obsługę stanu awaryjnego. Polega ona na wysłaniu odpowiedniego komunikatu o awarii, ponowieniu wysłania komunikatu podczas transmisji którego wystąpił błąd i podniesieniu jego semafora (w przypadku niepowodzenia ponownej transmisji). Analogicznie postępuje się w przypadku braku gotowości USART-a do transmisji. Po wysłaniu wszystkich znaków komunikatu, przez obsługę przerwania TxRDY, zadanie MON zostaje uruchomione ponownie przez funkcję "txrdy". Zadanie sprawdza zawartość bufora "buc_1". Jeżeli znajdują się tam adresy następnych komunikatów, zostają one wysłane na ekran w ten sam sposób. Adresy komunikatów są pobierane według kolejności przyjścia (kolejka typu FIFO). Jeżeli bufor "buc_1" jest pusty, to zostaje wysłany komunikat sterujący "akur", który powoduje powrót kursora na pozycję, na jakiej znajdował się przed rozpoczęciem wysyłania komunikatów. Następnie zadanie MON maskuje przerwanie TxRDY na poziomie układu przerwań procesora (układ USART-a nie jest przeprogramowywany) i zawiesza się w oczekiwaniu na nowy komunikat.

Zmianę położenia kursora na ekranie realizuje się przez zmianę jego adresu w komunikacie "akur". Musi ona być dokonywana przy zamkniętym jego semaforze. Następnie należy wysłać na monitor jakikolwiek komunikat (w szczególności może to być komunikat "akur"), np.:

```
WAIT ( akur ) ;                  /* zamknięcie semafora       */
*( akur + 3 ) = wiersz ;         /* ustawienie numeru wiersza */
*( akur + 4 ) = kolumna ;        /* ustawienie numeru kolumny */
SIGNAL ( akur ) ;                /* otwarcie semafora         */
SENDM ( &buc_1, kom1 ) ;         /* wysłanie komunikatu kom1  */
                                 /* kursor na nowej pozycji   */
```

W przypadku, gdy wysyłanym komunikatem jest "akur" jawne otwarcie semafora jest zbędne. Zostanie on i tak otwarty po wysłaniu komunikatu na ekran.

10.2. Obsługa klawiatury

Obsługę klawiatury realizuje zadanie TEH i funkcja obsługi przerwania RxRDY. Po naciśnięciu klawisza jest generowane przerwanie RxRDY. Obsługa tego przerwania przez funkcję "rxrdy", polega na odczytaniu znaku z klawiatury, umieszczeniu go w buforze i uruchomieniu zadania TEH przez podniesienie semafora. Zadanie odczytuje przesłany znak i przeprowadza jego interpretację. "Zwykłe" znaki są gromadzone w buforze "buc_2" z jednoczesnym wysłaniem echa znaku na ekran monitora. Realizowane jest to poprzez wysłanie odpowiedniego komunikatu do zadania MON. Jednocześnie zadanie TEH aktualizuje adres kursora tak, aby wskazywał na następną pozycję za wysłanym echem. Następnie jest sprawdzana zawartość bufora przerwania i jeżeli jest on pusty, zadanie zawiesza się w oczekiwaniu na kolejny znak. Znaki rozpoznane jako "specjalne" są traktowane indywidualnie, np.: BS czyści bufor "buc_2" - w którym gromadzone są znaki, ESC - zostaje umieszczony zawsze na początku bufora i natychmiast wysłany. Znak CR kończy kompletowanie komunikatu i powoduje przesłanie go do zadania użytkowego.

Pracą zadania TEH sterują trzy zmienne - "dkom", "secret" i "single". Zmienna "dkom" ogranicza długość przyjmowanych komunikatów. Zmienna "secret" steruje wysyłaniem echa. Z definicji przyjmuje ona wartość 0 (OFF), co oznacza wysłanie echa. Zmiana tej wartości na 1 (ON) wstrzymuje wysyłanie echa na ekran monitora. Kursor przesuwa się jedynie po ekranie. Ustawienie zmiennej "single" na 1 (z definicji przyjmuje ona wartość 0 - OFF), powoduje, że znaki są wysyłane do zadania pojedynczo, natychmiast po odczytaniu (bez czekania na znak powrotu karetki - CR).

10.3. Obsługa drukarki

Drukarkę obsługuje zadanie PRT. Działa ono podobnie jak zadanie MON. /Komunikat jest zbudowany analogicznie jak dla zadania MON. Komunikat musi zawierać sekwencje sterujące drukarką (inne niż dla monitora). Zadania użytkowe albo procedury obsługi przerwań wysyłają adresy komunikatów do bufora "buc_3" za pomocą funkcji systemowych SENDM albo PUTBUF. Zadanie PRT czeka zawieszone na pojawienie się adresu

35

komunikatu w buforze. Po pojawieniu się adresu komunikatu zadanie zostaje uruchomione przez system operacyjny. Odmaskowuje ono przerwanie INTB i sprawdza gotowość interfejsu równoległego do wysłania znaku (zakłada się przy tym, że brama B tego portu, pracująca w trybie 1 jako wyjście, nie jest wykorzystywana przez inne zadania). Dalsze czynności są analogiczne jak w zadaniu MON (łącznie z obsługą sytuacji awaryjnych). Po wysłaniu wszystkich komunikatów zadanie PRT zawiesza się w oczekiwaniu na nowe.

10.4. Przerwanie zegarowe czasu rzeczywistego

Przerwanie zegarowe, o okresie 100 ms, jest otrzymywane w wyniku odpowiedniego zaprogramowania układu licznika typu INTEL 8253 (za pomocą funkcji "timinit"). Przerwanie to obsługuje funkcja "timer". Funkcjonalnie można wyodrębnić w niej część obsługującą system operacyjny oraz część użytkową, obejmującą zegar czasu astronomicznego i datownik. Część systemowa aktualizuje tablicę bloków kontrolnych zadań (TCB). Co 100 ms jest aktualizowany w niej licznik taktów - "counter", a co sekundę licznik sekund -"counts". Liczniki te umożliwiają pracę systemu operacyjnego w czasie rzeczywistym.

Część użytkowa funkcji "timer" obsługuje zegar i datownik systemowy. Zegar obsługują liczniki: cycle, cls, clm i clh - zliczające odpowiednio: cykle zegarowe 100 ms, sekundy, minuty i godziny. Aktualny czas umieszczany jest w tablicy "setcl" i co sekundę wysyłany na ekran monitora. W momencie startu systemu wszystkie liczniki są zerowe. W celu ustawienia czasu należy uaktualnić je jednocześnie z tablicą "setcl". Wszystkie te czynności należy wykonywać przy zablokowanym układzie przerwań.

Datownik systemowy obsługują liczniki: day_number, clday, clmonth i clyear - zliczające odpowiednio: dzień tygodnia, dzień miesiąca, miesiąc i rok. Komunikaty o dniu tygodnia zawiera tablica "day_name", natomiast datę - tablica "setdata". Komunikaty te są wysyłane na ekran jedynie po ich zmianie. Aktualizację datownika należy przeprowadzać analogicznie jak zegara. Datownik działa poprawnie do roku 2100.

11. Zestawienie funkcji systemowych i ich parametrów

Funkcje systemowe (główne) z parametrami		Działanie
void START (numer_zadania) int numer_zadania ;	x	bezwarunkowe uruchomienie zadania
int STARTC (numer_zadania) int numer_zadania ;	x	uruchomienie zadania ze stanu uśpienia
void PERIOD (numer_zadania, długość_cyklu) int numer_zadania ; unsigned int długość_cyklu ;	x	uruchomienie zadania wznawianego cyklicznie
void PAUSE (czas_oczekiwania) int czas_oczekiwania ;		wstrzymanie bieżącego zadania
void STOP ( )		zakończenie bieżącego zadania
void KILL (numer_zadania) int numer_zadania ;	x	bezwarunkowe zakończenie zadania
void WAIT (adres_semafora) char *adres_semafora ;		operacja P na semaforze binarnym
int WAITOUT(adres_semafora, ogranicznik_czasowy) char *adres_semafora ; int ogranicznik_czasowy ;		operacja P na semaforze binarnym z ogranicznikiem czasowym
void SIGNAL (adres_semafora) char *adres_semafora ;	x	operacja V na semaforze binarnym
void SENDM (adres_bufora, adres_informacji) struct bch *adres_bufora ; char *adres_informacji ;		wpisanie adresu informacji do skrzynki pocztowej -FIFO
void WAITM (adres_bufora, adres_informacji) struct bch *adres_bufora ; char **adres_informacji ;		odczyt adresu informacji ze skrzynki pocztowej -FIFO
int PUTBUF (adres_bufora, adres_informacji) struct bch *adres_bufora ; char *adres_informacji ;	x	wpisanie adresu informacji do skrzynki pocztowej -FIFO
int GETBUF (adres_bufora, adres_informacji) struct bch *adres_bufora ; char **adres_informacji ;	x	odczyt adresu informacji ze skrzynki pocztowej -FIFO

Funkcje systemowe pomocnicze z parametrami		Działanie
void picinit (tablica_parametrów) char *tablica_parametrów	x	inicjacja sterownika przerwań (8259A)
void timinit (tablica_parametrów) char *tablica_parametrów	x	inicjacja czasomierza (8253)
void piainit (tablica_parametrów) char *tablica_parametrów	x	inicjacja łącza równoległego (8255)
void initV24 (tablica_parametrów) char *tablica_parametrów	x	inicjacja USART-a (8251A)
void unperiod (numer_zadania) int numer_zadania ;	x	zakończenie cyklicznego pobudzania zadania
void presem(adres_semafora,wartość_inicjowana) char *adres_semafora ; char wartosc_inicjowana ;	x	nadanie wartości semaforowi binarnemu
void instal (numer_zadania) int numer_zadania ;	x	zainstalowanie zadania

Funkcje systemowe obsługi przerwań z parametrami		Działanie
void servints (funkcja_obsługi, parametr) int (*funkcja_obsługi) ( ) ; char *parametr ;	*+	ogólna obsługa przerwań - monitor przerwań
void mint(adres_sterownika_przerwań, maska) char *adres_sterownika_przerwań ; char unsigned maska ;	x	zamaskowanie przerwania (przerwań)
void unmint(adres_sterownika_przerwań, maska) char *adres_sterownika_przerwań ; char unsigned maska ;	x	odmaskowanie przerwania (przerwań)
void di ( )	x	zablokowanie układu przerwań
void ei ( )	x	odblokowanie układu przerwań
void dis ( )	x	zablokowanie układu przerwań
void eis ( )	x	odblokowanie układu przerwań
void savereg ( )	+	zachowanie rejestrów

x - Wykorzystanie możliwe w zadaniu i w monitorze przerwań
+ - Wykorzystanie tylko w obsłudze przerwań
* - Wywołanie przy zamkniętym układzie przerwań
  - Funkcje systemowe typu "int" podają wynik operacji:
    pozytywny - OFF (0), negatywny - ON (1)

Zestawienie parametrów funkcji systemowych

Parametr	zakres	jed.fiz.	typ zmiennej
numer_zadania	1 .. 2^{15}-1	bez wym.	int
wartość_inicjowana	logiczna	"	char
maska	8 bitów	"	unsigned char
czas_oczekiwania	0 .. 2^{15}-1	0.1 s	int
	> 54.5 min		
ogranicznik_czasowy	"	"	"
długość_cyklu	0 .. 2^{16}-1	s	unsigned int
	> 18 godz.		
funkcja_obsługi	w polu	bez wym.	int (*)()
adres_bufora	adresowym	"	struct bch*
adres_semafora	procesora	"	char*
adres_informacji	INTEL 8086	"	"
adres_sterownika_przerwań	(do 1 MB)	"	"
parametr	"	"	"
tablica_parametrów	"	"	"

12. Podstawowe parametry eksploatacyjne systemu SIRTOS

System SIRTOS, zrealizowany w jezyku AZTEC C i zainstalowany na przemysłowym sterowniku INTELDIGIT-PROWAY z jednostką centralną MM86, zbudowaną na mikroprocesorze INTEL 8086, charakteryzuje się następującymi parametrami:

- częstotliwość zegara systemowego: 4915.2 kHz,
- okres zegara czasu rzeczywistego: 100 ms,
- maksymalny czas przełączania zadania aktywnego: < 100 s/zadanie,
- średni czas przełączania zadania aktywnego: < 50 s/zadanie,
- zajętość pamięci przez jądro systemu: mniej niż 4 kB

13. Literatura

[1] Nikiel W, Wóltański St. - "SIRTOS - A Small Industrial Real Time Operating System for Microcomputers" IFIP/IFAC Working Conference on Hardware and Software for Real Time Process Control, Warszawa, Maj 1988.

[2] Aztec C86 for PC DOS, MS DOS and CP/M-86 version 3.40, Manx Software Systems, Inc., 1985.

[3] Kernighan B. W. Ritchie D.M. - " Język C ", WNT Warszawa 1987.

[4] Kubiczek M, Martin W. J. - " Język programowania 'C' przewodnik dla praktyków", ORG-SERVICE Gdańsk, 1988.

[5] Strzałkowski P., Wóltański St. - " Systemy operacyjne czasu rzeczywistego dla mikrokomputerów 16 - bitowych - charakterystyka porównawcza", Biuletyn PIAP nr 3/88.

[6] " DOS 3.20 Reference ", IBM Corp. and Microsoft, Inc., 1986.

[7] Dunaj J. -"PC DOS 3.10, 3.20, MS DOS 3.10", Biuletyn PIAP (w druku)

[8] INTEL Corp. - "The 8086 Family User's Manual", 1979.

[9] Deitel Harvey M. - "An introduction to operating systems", Addison-Wesley Publ. Comp., 1984.