Na prawach rękopisu

INSTYTUT CYBERNETYKI TECHNICZNEJ
POLITECHNIKI WROCŁAWSKIEJ
Raport serii Sprawozdania nr 31/94

OS–9
– modułowy, wielozadaniowy system czasu rzeczywistego

Marek Wnuk

Słowa kluczowe: system operacyjny, wielozadaniowość, system modułowy, mikroprocesor,

Wrocław 1994

Spis treści

1 Wstęp **4**

2 Modułowa budowa OS–9 **5**
2.1 Moduły pamięciowe OS–9 . . . 5
2.2 Budowa modułu (na przykładzie modułu typu *program*) . . . 6
2.3 Moduły w plikach . . . 9
2.4 Grupy modułów . . . 10
2.5 Pamięć w OS–9 . . . 10
2.6 Pamięć kolorowana (*coloured memory*) . . . 10
2.7 Przydzielanie pamięci (*memory allocation*) . . . 11
2.8 Ochrona pamięci (*inter-task memory protection*) . . . 11

3 System wejścia–wyjścia w OS–9 **13**
3.1 Struktura wejścia–wyjścia w OS–9 . . . 13
3.2 Podsystem wejścia–wyjścia . . . 14
3.3 Moduły obsługi plików i moduły sterowników . . . 15
3.4 Deskryptory urządzeń . . . 15
3.5 Ścieżki i pliki . . . 17
3.6 Zezwolenia (atrybuty) i tryby . . . 18
3.7 Systemowe funkcje wejścia-wyjścia . . . 19
3.8 Deskryptor ścieżki . . . 20

4 Programy użytkowe OS–9 **23**
4.1 Podstawowe komendy OS–9 . . . 23
4.1.1 Shell . . . 25
4.1.2 Sh . . . 27
4.1.3 Edytor . . . 29
4.1.4 Grep . . . 29
4.1.5 Sed . . . 31
4.1.6 Gawk (GNU awk) . . . 32
4.2 Środowisko programowania w OS–9 . . . 33
4.2.1 Kompilator . . . 34
4.2.2 Make . . . 36
4.2.3 Debuggery . . . 37

5 Internet Support Package (ISP) **39**
5.1 Sockman . . . 39
5.2 Ifman . . . 40
5.3 Programy użytkowe pakietu ISP . . . 40
5.4 Gniazdka (*sockets*) . . . 41
5.5 Biblioteka **netdb.l** dla OS–9 . . . 42
5.6 Biblioteka **socklib.l** dla OS–9 . . . 42

6 Procesy w OS–9 **43**

7 Podział czasu w OS–9 **45**

8 Komunikacja między procesami w OS–9 **47**
8.1 Sygnały 47
8.2 Łącza 50
8.3 Zdarzenia (*events*) 52
8.4 Alarmy 56

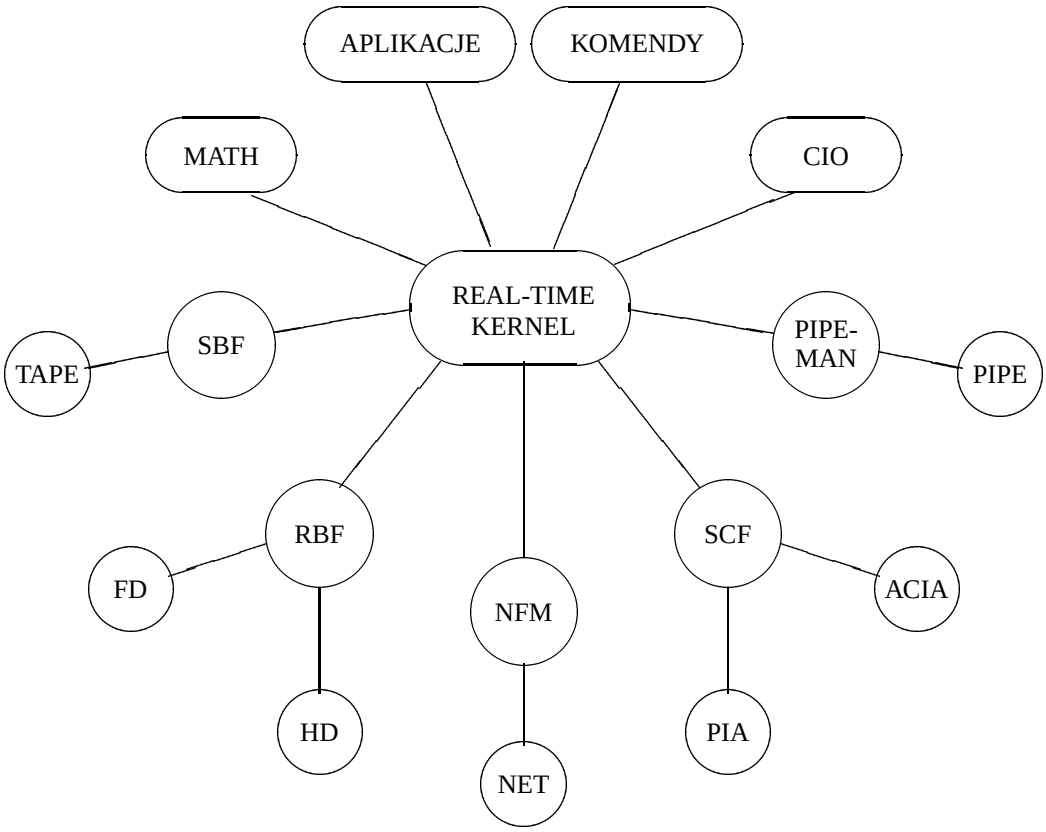
Spis rysunków

1 Struktura modułowego systemu czasu rzeczywistego OS–9 5
2 OS–9 Internet 39

1 Wstęp

OS–9 firmy Microware jest kompletnym, wielozadaniowym, modułowym systemem czasu rzeczywistego. Zawiera nie tylko jądro czasu rzeczywistego (*real-time kernel*) wraz z towarzyszącymi mu modułami, ale również moduły obsługi plików (*file managers*) i sterowniki urządzeń (*device drivers*) umożliwiające tworzenie elastycznego systemu wejścia - wyjścia. Udostępnia mechanizmy podziału czasu (*time sharing*) i wielozadaniowości (*multi-tasking*). Jest standardowo wyposażony w Unix-o podobny interfejs użytkownika (*shell*, hierarchiczny system plików, komendy ułatwiające administrowanie systemem). W wersji profesjonalnej (*Professional OS–9*) jest wyposażony w kompilatory języków wyższego rzędu, debuggery i narzędzia komunikacyjne umożliwiające współpracę z innymi systemami. Modułowa budowa OS–9 umożliwia przystosowanie rozmiarów i możliwości systemu docelowego do potrzeb konkretnej aplikacji, bez zamykania drogi ewentualnej dalszej rozbudowy. Je st dostępny w w wersji *Industrial OS–9* jako przeciwny biegun bardzo szerokiego spektru zastosowań (od prostych sterowników - w wersji zROMowanej do dużych systemów pracujących w sieciach). Warto wspomnieć, ze jedno z jego wcieleń zyskuje coraz większą popularność w CDI jako CD-RTOS.

Uwaga:
Niniejszy opis tego systemu jest oparty na dokumentacji fabrycznej, dostarczanej przez Microware i na źródłach dostarczonych wraz z systemem. Część plików definiujących struktury wewnętrzne systemu przytoczono w Dodatkach, jednak pełne zapoznanie się z budową OS–9 wymaga dostępu do oryginalnych plików systemowych.



Rysunek 1: Struktura modułowego systemu czasu rzeczywistego OS–9

2 Modułowa budowa OS–9

2.1 Moduły pamięciowe OS–9

Koncepcja modułów pamięciowych OS–9 stanowi sedno wielu zalet, które pozwalają na stosowanie tego systemu w szerokim zakresie aplikacji. Pozwala ona systemowi operacyjnemu panować nad programami, wspólnymi obszarami danych, składnikami systemu operacyjnego. Niezależnie od ich bezwzględnego położenia w pamięci, można się do nich odwoływać przez nazwę, podobnie jak do plików na dysku.
Moduł OS–9 może zawierać program, dane, część systemu operacyjnego. Zaczyna się nagłówkiem, a kończy cykliczną sumą kontrolną (CRC).
Nagłówek zawiera informacje o module, niezbędne do jego rozpoznania i prawidłowego wykorzystania w systemie. Pozwala on znaleźć moduł w pamięci przy starcie systemu, zainicjować odpowiednie obszary danych i sprawdzić jego poprawność, dzięki sumie kontrolnej i CRC. Zawiera też wskaźnik

do nazwy modułu.
Moduł jest dostępny przez nazwę, musi ona być unikalna wśród modułów załadowanych do pamięci, choć na dysku (w plikach) może się powtarzać.
Adresy wszystkich załadowanych aktualnie modułów są zebrane w kartotece modułów (*module directory*), która jest tablicą w pamięci obsługiwaną przez system operacyjny. Każdy jej element zawiera:

- adres modułu
- licznik biezących użytkowników (*link count*)
- identyfikator grupy
- sumę kontrolną nagłówka modułu

Moduł jest dostępny w systemie dzięki funkcji systemowej F$Link (*link to module*), której dostarcza się nazwę modułu. Jądro systemu (*kernel*) przegląda moduły w kartotece, adres modułu wskazuje na jego nagłówek, w którym z kolei znajduje się wskaźnik do nazwy. Jej zgodność z nazwą zadaną kończy poszukiwanie.
Moduły mogą znajdować się w pamięci stałej (ROM), lub być ładowane przy starcie systemu, lub w dowolnej chwili podczas jego pracy. Podczas ładowania jądro systemu rezerwuje pamięć zgodnie z rozmiarem pliku, wpisuje jego zawartość i przeszukuje zapisany obszar w celu wykrycia poprawnych modułów. Znalezione moduły są dołączane do kartoteki modułów.
Każde użycie modułu (F$Link) powoduje zwiększenie licznika (*link count*), a każde zakończenie używania (F$Unlk) - zmniejszenie. Moduły o własności *sticky* pozostają w kartotece również przy zerowej wartości licznika, co pozwala uniknąć wielokrotnego ładowania z pliku często używanych modułów. Przy braku miejsca w pamięci modułu o zerowym liczniku użyć są usuwane. Większość podstawowych programów użytkowych firmy Microware ma własność *sticky*. Moduły bez tej własności są usuwane przy zerowaniu licznika użyć.
Programy występują wyłącznie w formie modułów. Mogą być ładowane przez system w dowolny obszar pamięci. Są one napisane w sposób niezależny od lokalizacji (*position-independent code*). Obszary danych są odrębne od obszarów programu i mogą być umieszczane w dowolnym miejscu pamięci. Pozwala to na ROM-owalność i współużywalność modułów programowych.
System operacyjny OS–9 jest również podzielony na moduły, co ułatwia jego dopasowanie do konkretnych potrzeb wynikających z aplikacji. Moduły umieszczone w ROM i pliku ładowanym przy starcie systemu (*bootfile*) są automatycznie włączane do kartoteki modułów, nie ma więc potrzeby specjalnego budowania systemu, a jedynie trzeba dodawać lub wymieniać moduły.

2.2 Budowa modułu (na przykładzie modułu typu *program*)

Nagłówek modułu i CRC są automatycznie tworzone przez linker w trakcie kompilacji lub asemblacji programu. Składa się on z części wspólnej dla wszystkich modułów zakończonej sumą kontrolną i części specyficznej dla każdego typu modułu (zawartego w części wspólnej). Definicje modułów znajdują się w plikach **module.a** i **module.h**. Przytoczono je w Dodatku A.

Przykład: moduł programowy

```
------------------------------------------------------------------------
adres   wielkosc    nazwa w C   nazwa w asm
```

```
-----   --------    ---------   --------------  WSPOLNA CZESC NAGLOWKA
$0000   word (2)    _msync      M$ID            Znacznik modulu (\$4AFC)
$0002   word (2)    _msysrev    M$SysRev        Numer wersji systemu
$0004   long (4)    _msize      M$Size          Wielkosc modulu
$0008   long (4)    _mowner     M$Owner         Wlasciciel modulu
$000C   long (4)    _mname      M$Name          Wskaznik nazwy modulu
$0010   word (2)    _maccess    M$Accs          Zezwolenia na dostep
$0012   word (2)    _mtylan     M$Type,M$Lang   Typ i jezyk
$0014   word (2)    _mattrev    M$Attr,M$Revs   Atrybuty i wersja modulu
$0016   word (2)    _medit      M$Edit          Numer edycji modulu
$0018   long (4)    _musage     M$Usage         Wskaznik komentarzy
$001C   long (4)    _msymbol    M$Symbol        Wskaznik tablicy symboli
$0020   word (2)    _mident                     Kod identyfikacyjny modulu
$0022               _mspare                     Zarezerwowane
$002e   word (2)    _mparity    M$Parity        Suma kontrolna naglowka

-----   --------    ---------   --------------  ROZSZERZENIE NAGLOWKA (Prog)
$0030   long (4)    _mexec      M$Exec          Wskaznik startu programu
$0034   long (4)    _mexcpt     M$Excpt         Wskaznik obslugi TRAP
$0038   long (4)    _mdata      M$Mem           Wielkosc obszaru danych
$003C   long (4)    _mstack     M$Stack         Wielkosc stosu
$0040   long (4)    _midata     M$IData         Wskaznik inicjacji danych
$0044   long (4)    _midref     M$IRefs         Wskaznik inicjacji wskaznikow

-----   --------    ---------   --------------  WLASCIWY PROGRAM
$0048
...

-----   --------    ---------   --------------  CYKLICZNA SUMA KONTROLNA
        long (4)                                CRC modulu

--------------------------------------------------------------------------------
```

Znacznik modułu (*Sync Word*)
Wartość $4AFC (nielegalna jako instrukcja w M68xxx) jest znacznikiem początku modułu w pamięci. Ułatwia ona poszukiwanie modułów w pamięci przez jądro systemu przy starcie i ładowaniu z pliku. Po jej znalezieniu sprawdza się sumę kontrolną nagłówka, a trzeciej kolejności - CRC.
Numer wersji systemu (*System Revision ID*)
Oznacza wersję nagłówka (dotąd nie zmieniana), pozwala na zachowanie zgodności w dół przy nowych wersjach systemu i ewentualnych rozszerzeniach.
Wielkość modułu (*Module Size*)
Ilość bajtów wraz ze znacznikiem i CRC.
Właściciel modułu (*User and Group*)
Numer grupy (bardziej znaczące słowo) i użytkownika (mniej znaczące słowo), który stworzył moduł. Używane wraz z zezwoleniami na dostep i przywilejami wywoływania niektórych usług systemowych (tylko dla grupy 0).

Wskaźnik nazwy modułu (*Offset to Module Name*)
Adres (względem początku modułu) napisu zawierającego nazwę modułu (zakończonego znakiem *null*). Jest on używany do wyszukiwania modułu przy pomocy funkcji systemowych (*link to module*).
Zezwolenia na dostep (*Access Permissions*)

```
15  14  13  12  11  10   9   8   7   6   5   4   3   2   1   0

--  --  --  --  --  pe  pw  pr  --  ge  gw  gr  --  ce  cw  cr

gdzie: p - public,  g - group, c - creator,

       w - write   - pozwala pisac w przylaczonym module
       r - read    - pozwala ladowac, przylaczac i odlaczac modul
       e - execute - pozwala dodatkowo uruchamiac modul
```

Typ i język (*Type and Language*)
Typ (starszy bajt):

```
 1 - program (Prog)
 2 - podprogram (Subr)
 4 - dane (Data)
11 - obsluga TRAP (Trap)
12 - systemowy (Sys)
13 - manager plikow (Fman)
14 - sterownik (Driv)
15 - deskryptor (Desc)
```

Język (mlodszy bajt):

```
 1 - kod maszynowy
 2 - kod posreni Basic

(nie uzywane):

 3 - kod posredni Pascal
 4 - kod posredni C
 5 - kod posredni Cobol
 6 - kod posredni Fortran
```

Atrybuty i wersja modułu (*Attributes and Revision*)
Atrybuty (starszy bajt):

```
    7       6       5       4       3       2       1       0

sharable sticky supervisor  -       -       -       -       -
```

sharable oznacza możliwość równoczesnego używania modułu przez wiele procesów
sticky oznacza pozostawanie w pamięci i kartotece modułów mimo wyzerowania licznika użytkowników (*link count*)
supervisor oznacza moduł pracujacy z ustawionym systemowym trybem pracy procesora
Wersja modułu (młodszy bajt) pozwala na zastępowanie nowszymi wersjami starszych (np będących w ROM lub *bootfile*). Podczas ładowania modułów z pliku jądro sprawdza, czy ładowany moduł ma wyższy numer wersji od ewentualnie istniejącego w kartotece modułu o tej samej nazwie i w takim przypadku podstawia nowszy moduł w miejsce starszego.
Numer edycji modułu (*Edition Number*)
Służy do porządkowania wersji programow, nie jest używany przez jądro systemu.
Wskaźnik komentarzy (*Offset to usage comments string*)
Wskaźnik tablicy symboli (*Offset to symbol table*)
Kod identyfikacyjny modułu (*Ident code*)
Nie są dotąd używane.
Suma kontrolna nagłowka (*Header Parity*)
Zanegowane słowo będące wynikiem wykonania operacji EXOR na wszystkich kolejnych słowach nagłówka. Wykonanie EXOR na całym nagłówku wraz z tą sumą winno dać wynik $FFFF.
Wskaźnik startu programu (*Offset to Program Entry Point*)
Z adresu modułu i tego wskaźnika jądro wylicza rzeczywisty adres startu programu zawartego w module.
Wskaźnik obsługi TRAP (*Offset to Default Trap Entry Point*)
Względny adres (jak poprzednio) procedury obsługi instrukcji TRAP, dla których nie okreslono indywidualnej obsługi.
Wielkość obszaru danych (*Minimum Program Data Space*)
Łączna wielkość pamięci statycznej (zmiennych) programu.
Wielkość stosu (*Minimum Program Stack Space*)
Rezerwowana na stos ilość pamięci wynosi domyslnie 3kB, można ją zwiększyć odpowiednią opcją dla linkera.
Wskaźnik inicjacji danych (*Offset to Data Initialization Table*)
Względny adres tablicy (tworzonej przez linker) zawierającej wartości służące do zainicjowania zmiennych statycznych przy tworzeniu pamięci statycznej procesu.
Wskaźnik inicjacji wskaźników (*Offset to Data Pointers Initialization Table*)
W czasie tworzenia modułu przez linker nie jest znane jego położenie w pamięci. Aby móc zainicjować wskaźniki do funkcji lub do zmiennych statycznych, trzeba umieścić w module tablicę z odpowiednimi względnymi (w stosunku do początku modułu) adresami.
CRC modułu (*Module CRC*)
Cykliczna suma kontrolna jest 24-bitowa. Startowa wartość akumulatora wynosi $FFFFFF, a wynik dzielenia całego modułu przez wielomian charakterystyczny jest po zanegowaniu wpisywany jako długie słowo (najstarszy bajt = 0) na końcu modułu. Przy sprawdzaniu (wraz z CRC) wynik powinien wynosić $800FE3. Odpowiednia funkcja systemowa ma symbol F$CRC.

2.3 Moduły w plikach

Moduły mogą być umieszczone w plikach pojedynczo, lub grupami. Często (ale nie koniecznie) nazwa pliku jest zgodna z nazwą modułu. Program load używa funkcji systemowej F$Load do zarezerwowania pamięci na ładowany plik, załadowania go, sprawdzenia (F$ValMod) modułów znalezio-

nych w zapisanym obszarze. Licznik użyć (*link count*) pierwszego modułu z pliku jest ustawiany w kartotece modułów na 1, pozostałych - na 0. Unlink (F$UnLink) zmniejsza o 1 licznik użyć modułu. Jeśli w ten sposób osiąga on 0, to moduł (o ile nie jest *sticky*) jest usuwany z kartoteki, a zajmowana przezeń pamięć jest zwalniana. Moduły *sticky* wymagają zmniejszenia link count do wartości -1.
Uruchamianie nowego procesu (F$Fork) powoduje poszukiwanie modułu o zadanej nazwie w kartotece modułów pamięciowych, a w przypadku jego braku - załadowanie pliku o tej nazwie z katalogu programów (*execution directory*) i uruchomienie pierwszego modułu z tego pliku (niezależnie od jego nazwy). Moduły nie mające zezwolenia na uruchamianie (*pe,ge,ue*) są odrzucane z błędem E_NEMOD - moduł nieuruchamialny.

2.4 Grupy modułów

Moduły ładowane z jednego pliku tworzą grupę modułów. Identyfikatorem grupy jest adres pierwszego z modułów. W kartotece modułów każdy moduł z grupy ma w odpowiednim polu wpisany ten identyfikator. Moduły tworzące grupę mogą być usunięte wyłącznie razem, gdy wszystkie osiągną zerową (lub -1) wartość *link count*. Moduły zawarte w ROM i bootfile tworzą również grupy.
Ochrona przed zwalnianiem modułów ogranicza się do uniemożliwienia odłączenia modułu we/wy (Fman, Desc, Driv), który jest używany (w tablicy urządzeń we/wy). Nowa wersja systemu (OS–9 v. 2.4.3 - 1992) chroni również moduły główne istniejących procesów.

2.5 Pamięć w OS–9

Jądro systemu OS–9 (*kernel*) dynamicznie przydziela pamięć i zwraca do puli wolnej pamięci w miarę potrzeby. Nie jest używany mechanizm pamięci wirtualnej, programy używają pośredniego adresowania względem bazowych rejestrów adresowych.
Jądro podczas uruchamiania procesu przydziela mu pamięć statyczną i obszar stosu według modułu głównego (zawierającego pierwotny program). Podczas działania, program może dynamicznie żądać przydzielania pamięci i zwalniać ją (do 32 obszarów łącznie). W miarę przydzielania pamięci dokonyawne jest (w miarę możliwości) scalanie sąsiadujących obszarów.
Komenda mfree -e pozwala obejrzeć mapę wolnych (w danej chwili) obszarów pamięci.

2.6 Pamięć kolorowana (*coloured memory*)

Microware wprowadza typy (”kolory”) pamięci i priorytety dostępu, by ułatwić wpływanie przez użytkownika systemu na decydowanie o kolejności i sposobie wykorzystywania zasobów pamięci RAM. Niektóre części pamięci komputera są szybsze (na płycie procesora), inne - wolniejsze (poprzez magistralę kasety). Ponadto, niektóre obszary mogą mieć specjalne własności (dwubramowa pamięć karty video, bufor komunikacyjny pakietu podrzędnego, pamięć z podtrzymaniem bateryjnym).
Przydzielenie każdemu specyficznemu obszarowi innego numeru (koloru) pozwala żądać przydzielenia pamięci z tego właśnie obszaru bez definiowania fizycznych adresów. Ma to istotny wpływ na uniezależnienie aplikacji od sprzętu.
Różnice w czasie dostępu do pamięci wspomniane wyżej można uwzględnić przez ustalenie odpowiednich priorytetów dla poszczególnych bloków (niezależnie od ich koloru).
Lista kolorowych obszarów pamięci jest umieszczona w module init, łatwe jest więc konfigurowanie pamięci przez użytkownika.
Dwie usługi systemowe:

_srqmem() (F$SRqMem) i _srqcmem() (F$SRqCMem)
pozwalają w programach żądać konkretnego, lub dowolnego typu (koloru) pamięci. Można również użyć koloru pamięci jako parametru przy funkcjach F$Load i F$DatMod.
Pamięć o zadanym kolorze jest przydzielana według priorytetów w danym typie (kolorze), a bez specyfikacji koloru - zgodnie z priorytetami wszystkich typów łącznie.

2.7 Przydzielanie pamięci (*memory allocation*)

Jądro systemu OS–9 korzysta z oddzielnych list wolnej pamięci dla każdego priorytetu każdego koloru. Każdy wolny obszar ma w nagłówku:

```
$0000   long (4)    adres nastepnego elementu
$0004   long (4)    adres poprzedniego elementu
$0008   long (4)    wielkosc obszaru (w bajtach)
```

Jądro obsługuje również tablicę obszarów znalezionych przy starcie systemu, opisująca adres początku i końca, numer koloru, priorytet i atrybuty każdego obszaru pamięci.
Elementy tablicy są połączone w dwukierunkową listę uporządkowaną według priorytetów:

```
$0000   long (4)    adres poczatku obszaru
$0004   long (4)    adres konca + 1
$0008   long (4)    adres nastepnego elementu listy
$000C   long (4)    adres poprzedniego elementu listy
$0010   long (4)    adres pierwszego wolnego podobszaru
$0014   long (4)    adres ostatniego wolnego podobszaru
$0018   long (4)    zarezerwowane
$001C   long (4)    adres obszaru po ew. translacji
$0020   long (4)    suma wolnych podobszarow
$0024   word (2)    atrybuty (z modulu init)
$0028   word (2)    numer koloru
$0028   word (2)    priorytet
```

By przydzielić pamięć, kernel przegląda listę aż do znalezienia obszaryu o dostatecznym rozmiarze, pobiera adres pierwszego i ostatniego podobszaru wolnego, a następnie przeszukuje listę wolnych obszarów (wspomnianą wcześniej). Po znalezieniu wystarczająco dużego podobszaru, jądro przydziela żądaną jego część (z zaokrąglęniem do jednostki alokacji - 16 bajtów). W przypadku braku odpowiedniego podobszaru następuje powrót do przeszukiwania listy obszarów w tablicy.
W przypadku żądania przydziału dowolnej pamięci poszukiwanie odbywa się identycznie, lecz niezależnie od koloru. Poszukiwanie w tym przypadku kończy się po osiągnięciu priorytetu 0. Taka pamięć może być przydzielona tylko żądaniem z zadanym kolorem.

2.8 Ochrona pamięci (*inter-task memory protection*)

OS–9 nie korzysta z translacji adresów (pamięć wirtualna). Sprzęt do zarządzania pamięcią (MMU) jest używany do zabezpieczania obszarów pamięci przed niepożądanym dostępem z innego procesu. Dla procesorów rodziny 68xxx posiadających MMU istnieje specjalny moduł ssm (*system security module*), który musi być w pamięci podczas startu systemu (w ROM lub *bootfile*).

Uruchamiany proces ma przydzieloną pamięć statyczną i stos, a także pamięć zajętą przez jego główny moduł (*primary module*). Każdy proces ma swą mapę pamięci, która jest modyfikowana przy przydzielaniu i zwalnianiu obszarów przez program. Obszar tworzonych, lub przyłączanych modułów daych jest dodawany do mapy pamięci procesu zgodnie z zezwoleniami na dostęp zawartymi w nagłówku modułu.
Obecność ssm może utrudniać bezpośredni dostęp z programu użytkownika do sprzętu (porty). Nie jest to wprawdzie praktyka zalecana (należy używać systemu we/wy), ale spotykana w małych systemach i konkretnych aplikacjach. Dla ominięcia tego problemu ssm dodaje do funkcji systemowych dwie dodatkowe:
F$Permit i F$Protect
pozwalające programom, których właścicielem (twórcą) jest *root* (*super user*), dodawać do mapy pamięci procesu specyficzne obszary.
Uwaga:
Dokładne opisy struktur omawianych wyżej można znaleźć w dokumentacji systemu ([10] oraz w plikach znajdujących się w /dd/DEFS (w szczególności **module.a** i **module.h** - zamieszczonych w dodatkach).

3 System wejścia–wyjścia w OS–9

W OS–9 różne urządzenia zewnętrzne są widoczne dla użytkownika w sposób ujednolicony. Do obsługi każdego urządzenia niezbędne są trzy moduły:

- deskryptor urządzenia (*Device Descriptor* - np. **term**, **d0**, **dd**)
- moduł obsługi plików (*File Manager* - np. **SCF**, **RBF**, **SBF**)
- sterownik urządzenia (*Device Driver* - np. **rbVMSC**)

Deskryptor urządzenia zawiera nazwy pozostałych modułów. System wykonuje operacje na fizycznym urządzeniu na podstawie wywołania tego urządzenia przez nazwę deskryptora.
Różne klasy urządzeń mają oczywiście różne zestawy operacji, które można na nich wykonać (np. `dir`, `format` są oczywiste dla dysku, niedostępne dla terminala). Mimo to unifikacja systemu wewy pozwala łatwo zastępować jedne urządzenia innymi bez modyfikacji programów. Każdy moduł obsługi plików ma następujące wejścia (realizuje następujące funkcje):

- create
- open
- makdir
- chgdir
- delete
- seek
- read
- write
- readln
- writeln
- getstat
- setstat
- close

3.1 Struktura wejścia–wyjścia w OS–9

System wejścia/wyjścia OS–9 jest skonstruowany hierarchicznie. Wszystkie wywołania systemowych funkcji we/wy trafiają do jądra systemu (moduł **kernel**), skąd są kierowane do odpowiedniego dla tej klasy urządzeń modułu obsługi plików (*file manager*). Wykonanie fizycznych operacji na konkretnym urządzeniu we/wy jest zlecane odpowiedniemu modułowi sterownika urządzenia (*device driver*).
Dopuszczalne jest istnienie wielu modułów obsługi plików i sterowników urządzeń. Każdy moduł obsługi plików przystosowany jest do pewnej klasy urządzeń. Przykłade m może być RBF (*Random Block File manager*) obsługujący urządzenia o blokowym dostępie swobodnym (dyski), lub SCF

(*Sequential Character File manager*) przeznaczony do obsługi urządzeń operujących strumieniem znaków (terminale, drukarki).
File manager wykonuje tylko operacje o charakterze logicznym, co pozwala zachować szeroki zakres jego przydatności. Fizyczne przesyłanie danych wykonywane jest na jego zlecenie przez sterownik urządzenia - moduł programowy napisany specjalnie dla konkretnego fizycznego urządzenia we/wy, jak UART, kontroler dysków, czy karta sieciowa.
Każde urządzenie jest opisane przez specjalny moduł danych - deskryptor urządzenia (*device descriptor*). Urządzenie jest w systemie znane pod nazwą deskryptora poprzedzoną przez "/". Przykład: moduł **term** opisuje urządzenie **/term**. Deskryptor zawiera nazwę modułu obsługi plików i modułu sterownika urządzenia, adres fizycznego urządzenia, zestaw parametrów i opcji definiujących konkretne własności urządzenia we/wy.
Taka konstrukcja systemu wejścia/wyjścia pozwala na użycie tego samego sterownika do wszystkich urządzeń korzystających z takiego samego sprzętu pzez zdefiniowanie osobnych deskryptorów z innymi adresami i wektorami obsługi przerwań. Można też na jednym fizycznym urządzeniu zdefiniować wiele urządzeń logicznych (deskryptory o różnych nazwach), różniących się parametrami, modułami obsługi plików, lub nawet modułami sterowników. Na przykład, dwa deskryptory dla tego samego portu szeregowego mogą pozwalać na współpracę z terminalem i drukarką, na pracę w trybie asynchronicznym i synchronicznym (w tym przypadku zazwyczaj różnią się modułem sterownika). Inny przykład to deskryptory dysków elastycznych o różnych formatach możliwych do obsłużenia w tym samym napędzie.

3.2 Podsystem wejścia–wyjścia

Podsystem we/wy składa się z modułu obsługi plików, sterownika urządzenia, deskryptora urządzenia i obszaru pamięci statycznej danego urządzenia. Wszystkie podsystemy są umieszczone w tablicy urządzeń (*device table*). Urządzenie we/wy może być miejscem składowania danych (jak dysk), lub interfejsem z otoczeniem (jak port szeregowy). Poza tym dostępne są urządzenia abstrakcyjne, nie mające fizykalnego charakteru, a istniejące w postaci reprezentacji w pamięci RAM. Takimi urządzeniami są łącza (*pipes*) i pseudodyski (*RAM disks*). Obsługa różnie zrealizowanych urządzeń przez jądro systemu (*kernel*) jest jednakowa.
System OS–9 dynamicznie inicjuje i zakańcza pracę podsystemów we/wy. Kernel dokonuje odpowiednich operacji w chwili otwierania ścieżki dostępu do pliku na danym urządzeniu wykonując funkcję I$Attach (wbudowana w jądro). Powoduje ona dołączenie (*link*) modułu deskryptora urządzenia (co zwiększa jego ilość użyć - *link count*). Jeśli w tablicy urządzeń nie ma deskryptora o tym samym adresie, to jądro tworzy podsystem we/wy:

1. pobiera nazwy modułu obsługi plików i sterownika z deskryptora;
2. przyłącza te moduły;
3. tworzy nowe miejsce w tablicy urządzeń;
4. ustawia ilość użyc urządzenia na 1;
5. rezerwuje miejsce w RAM na pamięć statyczną urządzenia;
6. uruchamia procedurę inicjacji (ze sterownika);
7. w przypadku powodzenia - wypełnia miejsce w tablicy urządzeń, przy błędzie - odłącza urządzenie;

Gdy w tablicy jest już taki deskryptor, wykonuje kroki 1 i 2 i zwiększa ilość użyć urządzenia.
I$Attach jest również wykonywana w trakcie komendy `iniz`, która pozwala zainicjować urządzenie jeszcze przed jego użyciem. Zapewnia to utrzymanie chwilowo zwolnionego urządzenia w tablicy (ważne zwłaszcza dla RAM-dysków).
Operacje odwrotną przeprowadza funkcja I$Detach (przy zamknięciu ścieżki):

1. wywołanie procedury deinicjacji (ze sterownika);
2. zwolnienie pamięci statycznej;
3. odłączenie deskryptora, sterownika i modułu obsługi plików;
4. zwolnienie miejsca w tablicy urządzeń;

Jeśli ilość użyć jest większa niż 1, wykonywany jest punkt 3 i zmniejszana ilość użyć urządzenia.
Komenda służąca do wykonania tej operacji interakcyjnie jest `deiniz`.
Przy funkcjach I$MakDir (tworzenie katalogu), I$ChgDir (zmiana katalogu bieżącego), I$Delete (kasowanie pliku) jądro systemu otwiera ścieżkę, wywołuje odpowiednią funkcję modułu obsługi plików i zamyka ścieżkę, co gwarantuje, że ścieżka jest otwarta, a urządzenie zainicjowane przy wywołaniu funkcji z modułu obsługi plików. I$ChgDir zwiększa również ilość użyć urządzenia w tablicy urządzeń, by zapobiec jego usunięciu nawet w przypadku zamknięcia wszystkich ścieżek. By usunąć urządzenie z tablicy trzeba więc wykonać `deiniz` tyle razy, ile łącznie wykonano `chd`, `chx` i `iniz` dla tego urządzenia. Komenda `devs` służy do przeglądania tablicy urządzeń.
Uwaga: funkcje inicjacji i deinicjacji urządzenia znajdujące się w sterowniku urządzenia są wywoływane bezpośrednio przez jądro (pozostałe - przez moduł obsługi plików).

3.3 Moduły obsługi plików i moduły sterowników

Podział funkcji pomiędzy moduł obsługi plików i sterownik jest umowny. Sterownik łącza (*pipe*) nie wykonuje żadnych funkcji, bo urządzenie to nie jest zależne od sprzetu (bufor FIFO w pamięci) i moduł obsługi plików może sam sobie poradzić zachowując w pełni uniwersalnosć.
Zazwyczaj moduł obsługi plików zawiera kod do operowania danymi na poziomie logicznym dla pewnego typu urządzeń. RBF - dla dysków z hierarchicznym systemem plików. Sterownik zawiera zaś kod do przeprowadzania fizycznych operacji na urządzeniu na zlecenie modułu obsługi plików. Ułatwia to pisanie sterowników dla nowego sprzętu, bo nie trzeba w pełni znać systemu plików by obsługiwać sprzęt. Dzięki temu przenoszenie OS–9 na nowy sprzęt jest zadaniem stosunkowo prostym.
Uwaga: furtką przewidzianą do rozszerzenia możliwości sterownika są funkcje *get status* i *set status*, pozwalające przekazywać zadania bezpośrednio do sterownika, bez udziału modułu obsługi plików.

3.4 Deskryptory urządzeń

Moduł deskryptora urządzenia jest małym modułem danych. Zawiera on część standardową i część opcjonalną, zależną od typu urządzenia (modułu obsługi plików). Część standardowa jest umieszczona po nagłówku modułu:

```
-----   --------    ---------   -------------- ROZSZERZENIE NAGLOWKA (Desc)
```

```
$0030   long (4)    _mport      M$Port          Adres portu
$0034   byte (1)    _mvector    M$Vector        Wektor obslugi przerwan
$0035   byte (1)    _mirqlvl    M$IRQLvl        Poziom przerwania
$0036   byte (1)    _mpriority  M$Prior         Priorytet przerwania
$0037   byte (1)    _mmode      M$Mode          Zakres mozliwosci urzadzenia
$0038   word (2)    _mfmgr      M$FMgr          Adres nazwy mod. obsl. plikow
$003a   word (2)    _mpdev      M$PDev          Adres nazwy sterownika
$003c   word (2)    _mdevcon    M$DevCon        Adres obszaru parametrow spec.

...

$0046   word (2)    _mopt       M$Opt           Wielkosc obszaru opcji

-----   --------    ---------   --------------  OBSZAR OPCJI

...

------------------------------------------------------------------------
```

Adres portu (*Port Address*)
Adres w przestrzeni pamięci, pod którym jest dostępny układ realizujący sterowanie urządzeniem. Jądro systemu sprawdza na jego podstawie unikalność deskryptora i ustawia V_PORT w pamięci statycznej urządzenia. Wykorzystanie przez sterownik nie jest obowiązkowe. Pozwala na użycie sterownika dla wielu układów tego samego typu.
Wektor obsługi przerwań (*Interrupt Vector*)
Numer wektora przerwania generowanego przez urządzenie (lub wektora bazowego, gdy generowanych jest wiele wektorów). Przy autowektorach musi wynosić 24+(M$IRQLvl).
Poziom przerwania (*Interrupt Level*)
Może być ustalony dla danego układu sprzętowo (od 1 do 7).
Priorytet przerwania (*Interrupt Polling Priority*)
Jeśli urządzenie ma unikalny wektor, to pole winno być zerowe, przy wykorzystaniu wspólnego wektora przez kilka urządzeń pole to decyduje o kolejności obsługi po równoczesnym zgłoszeniu przerwania.
Zakres możliwości urządzenia (*Device Mode Capabilities*)
Bajt flag sprawdzany przy otwieraniu ścieżki w zadanym trybie:

```
bit flaga
--- --------------------------------------------------
0 - read (odczyt),
1 - write (zapis),
2 - execute (wykonywanie programow),
5 - initial file size (predefiniowalna wielkosc pliku)
6 - non-sharable files (pliki nie wspoldzielone)
7 - directories (katalogi)
```

Adres nazwy modułu obsługi plików (*File Manager Name Offset*)

Adres nazwy sterownika (*Device Driver Name Offset*)
Adres obszaru parametrów specyficznych dla urządzenia (*Device Configuration Offset*)
Adres ten wskazuje na tablicę zawierającą dodatkowe informacje o urządzeniu (0 - brak). Zawartość tablicy zależy od autora sterownika.
Uwaga: wszystkie adresy podaje się w modułach względem początku nagłówka.
Wielkość obszaru opcji (*Option Table Size*)
Obszar opcji zależy od autora modułu obsługi plików. Jest on kopiowany do obszaru opcji deskryptora ścieżki przy jego tworzeniu.
Dla RBF opcje są następujące:

```
-----   --------    --------------  OBSZAR OPCJI

$0048   byte (1)    DD_DTP          Typ urzadzenia (RBF - 1)
$0049   byte (1)    DD_DRV          Logiczny numer napedu
$004a   byte (1)    DD_STP          Predkosc krokowa glowic
$004b   byte (1)    DD_TYP          Typ dysku
$004c   byte (1)    DD_DNS          Gestosc zapisu
$004d   byte (1)
$004e   word (2)    DD_CYL          Ilosc cylindrow dla danych
$0050   byte (1)    DD_SID          Ilosc powierzchni (glowic)
$0051   byte (1)    DD_VFY          Flaga weryfikacji zapisu
$0052   word (2)    DD_SCT          Ilosc sektorow na sciezce
$0054   word (2)    DD_T0S          Ilosc sektorow na sciezce 0
$0056   word (2)    DD_SAS          Wielkosc segmentu
$0058   byte (1)    DD_ILV          Przeplot
$0059   byte (1)    DD_TFM          Tryb DMA
$005a   byte (1)    DD_TOffs        Numer pierwszego cylindra
$005b   byte (1)    DD_SOffs        Numer pierwszego sektora
$005c   word (2)    DD_SSize        Wielkosc bloku logicznego
$005e   word (2)    DD_Cntl         Slowo sterujace opcji
$0060   byte (1)    DD_Trys         Ilosc ponowien przy bledzie
$0061   byte (1)    DD_LUN          Numer napedu (SCSI LUN)
$0062   word (2)    DD_WPC          Prekompensacja
$0064   word (2)    DD_RWR          Ograniczenie pradu zapisu
$0066   word (2)    DD_Park         Cylinder parkowania
$0068   long (4)    DD_LSNOffs      Adres pierwszego sektora
$006c   word (2)    DD_TotCyl       Calkowita ilosc cylindrow
$006e   byte (1)    DD_CtrlrID      Numer kontrolera (SCSI)
$006f   byte (1)    DD_Rate         Predkosc
```

3.5 Ścieżki i pliki

Ścieżka (*path*) pozwala na dostęp na poziomie logicznym do danych, komend, statusu urządzenia z programu. Zamiast bezpośredniego komunikowania się z urządzeniem, program otwiera (*open*) ścieżkę dostępu przy pomocy odpowiedniego wywołania systemowego. System zwraca numer ścieżki,

przy pomocy ktorego odbywają się wszystkie dalsze operacje (odczyt, zapis, sterowanie ...). Po zakończeniu dostępu program wykonuje operację zamknięcia ścieżki (*close*).
Ścieżka jest reprezentowana przez strukturę w pamięci zwaną deskryptorem ścieżki (*path descriptor*), która jest tworzona przy otwieraniu i usuwana przy zamykaniu ścieżki. Przy zakończeniu procesu jądro samo zamyka wszystkie otwarte w nim ścieżki.
Plik jest strukturą danych w urządzeniu mającym pamięć. Pozwala to na przechowywanie wielu zbiorów danych na jednym urządzeniu i ich modyfikowanie. Moduły obsługi plików są tworzone tak, by jak najbardziej upodobnić (z punktu widzenia programu) pliki na różnych typach urządzeń do siebie.
Dysk jest urządzeniem pamięciowym o blokowym dostępie swobodnym - można czytać bloki w dowolnej kolejności. Tworzone są na nim katalogi (*directories*) będące plikami zawierającymi listę plików zawartych w katalogu. Pliki te mogą być również katalogami, co daje w efekcie drzewiastą strukturę (hierarchię) katalogów na dysku. Główny katalog stanowi korzeń tego drzewa (*root*).
Nazwa ścieżki (*pathlist*) jest napisem określającym urządzenie i/lub plik. Jest ona używana przy otwieraniu ścieżki. Nazwa ścieżki może zawierać wiele elementów oddzielonych separatorami ”/”:
/d0 - katalog główny dysku opisanego deskryptorem **d0**;
/h1/SUPWA/ex - plik **ex** w podkatalogu **SUPWA** katalogu głównego dysku **h1**;
Jeśli nazwa ścieżki nie zaczyna się od ”/”, to jest uważana za ścieżkę względną, począwszy od bieżącego katalogu danych, lub programów.
Funkcja systemowa F$PrsNam (*parse name*) dokonuje analizy nazwy (zarówno modułów jak i ścieżek). Jest ona niewrażliwa na duże i małe litery. Dopuszcza w nazwie : litery, podkreślenia ”_”, cyfry, kropki ”.”, znak dolara ”$”. Nazwa musi zawierać co najmniej jedną cyfrę, literę, lub podkreślenie.

3.6 Zezwolenia (atrybuty) i tryby

Zezwolenia (atrybuty) służą do ograniczania dostępu do urządzeń i plików. Przy otwieraniu ścieżki do urządzenia lub pliku, program podaje tryb, w jakim chce z niej korzystać. Jest to zestaw flag określający typ wykonywanych operacji. System operacyjny sprawdza, czy plik (urządzenie) ma zezwolenie na ten tryb dostępu i w razie braku zwraca błąd E$FNA (*file not accessible*). Zezwolenia mogą być pogrupowane na: użytkowników, grupowe i publiczne.
RBF sprawdza zezwolenia wzdłuż całej ścieżki dostępu (poczynając od urządzenia), także w przypadku ścieżek podanych w sposób względny.
Flagi trybu:

```
bit flaga
--- ---------------------------------------------------
0 - read (odczyt),
1 - write (zapis),
2 - execute (wykonywanie programu),
3 - nie uzywany
4 - nie uzywany
5 - initial file size (predefiniowana wielkosc pliku)
6 - non-sharable file (plik nie wspoldzielony)
7 - directory (katalog)
```

Zezwolenia dla plików RBF:

```
bit zezwolenie
--- --------------------------------------------------
0 - read (odczyt),
1 - write (zapis),
2 - execute (wykonywanie programu),
3 - public read
4 - public write
5 - public execute
6 - non-sharable file (plik nie wspoldzielony)
7 - directory (katalog)
```

Niestety, wspólne bity dla właściciela i grupy nie pozwalają grupować użytkowników (jak w unix-ie). Jest to historyczna pozostałość z OS–9/6809.

3.7 Systemowe funkcje wejścia-wyjścia

I$Attach: *Attach I/O Device*
Przyłącza urządzenie zapewniając jego inicjację, pamięć statyczną, umieszczenie w tablicy (jak opisano wcześniej). Nie ładuje modułów do pamięci (muszą być załadowane wcześniej - ROM, boot, load).
I$Detach: *Detach I/O Device*
Odłącza urządzenie (jak wcześniej).
I$Dup: *Duplicate Path*
Powiela ścieżkę, dając lokalny numer ścieżki dla wcześniej otwartej. Zwiększa sie liczniki użyć ścieżki (PD_COUNT, PD_CNT). Zwracany numer ścieżki jest najniższym z dostępnych.
I$Create: *Create New File*
Tworzy nowy plik i otwiera ścieżkę do niego. Jeśli moduł obsługi plików nie pozwala na ich tworzenie (np. SCF), to traktuje się to wywołanie jak I$Open.
I$Open: *Open Existing File*
Tworzony deskryptor ścieżki ma pola PD_COUNT i PD_CNT ustawione na 1, PD_MOD - zgodnie z podanym trybem. PD_USER ustawia się na numer grupy i użytkownika procesu. Po wywołaniu I$Attach do PD_DEV wpisuje się adres miejsca w tablicy urządzeń. Przy ścieżce względnej I$Attach wykonuje się dla urządzenia, którego adres w tablicy jest podany w polu bieżącego katalogu roboczego (danych lub programów - w zależności od atrybutu *execute*).
I$MakDir: *Make Directory File*
Podobnie, jak w I$Create, przy atrybytach *directory*. Po obsłużeniu przez moduł obsługi plików ścieżka jest zamykana (otwiera się ją tylko dla Fman).
I$ChgDir: *Change Default Directory*
Tymczasowo otwiera ścieżkę, wywołuje funkcję modułu obsługi plików i zamyka ścieżkę. Przy otwieraniu jądro dodaje atrybut *directory*. Adres urządzenia jest wpisywany do P$DIO - w deskryptorze procesu (jako bieżący katalog danych lub programów, w zależności od atrybutu *execute*). Jądro zwiększa licznik użyć urządzenia by zapobiec jego usunięciu z tablicy.
I$Delete: *Delete File*
Tymczasowo otwiera ścieżkę, wywołuje moduł obsługi plików i zamyka ścieżkę. Dla skasowania pliku trzeba mieć zezwolenie na zapis (wymaga tego Fman). RBF ponadto wymaga, by ewentualnie kasowany katalog był pusty, a dokładniej - pozwala usunąć atrybut *directory* tylko w tym przypadku.

I$Seek: *Change Current Position*
Przestawia bieżący wskaźnik w pliku (w otwartej ścieżce).
I$Read: *Read Data*
Czyta do bufora dane bez ich modyfikowania z otwartej ścieżki. Sprawdza (F$ChkMem) pozwolenie na zapis do bufora, jeśli nie jest w trybie nadzorcy.
I$Write: *Write Data*
Pisze z bufora dane bez ich modyfikowania do otwartej ścieżki. Sprawdza (F$ChkMem) pozwolenie na odczyt z bufora, jeśli nie jest w trybie nadzorcy.
I$ReadLn: *Read Line of ASCII Data*
Działa jak I$Read, lecz kończy czytanie na końcu linii (w OS–9 zazwyczaj $< CR >$=$13). Moduł obsługi plików może dostarczać prostych funkcji edycji linii (jak SCF).
I$WritLn: *Write Line of ASCII Data*
Działa jak I$Write, lecz zakłada, że moduł obsługi plików kończy na ¡CR¿ i dokonuje prostych konwersji (SCF - LF po CR, pauza po nowej stronie i rozwijanie tabulacji).
I$GetStt: *Get Path Status*
Pozwala na dostęp do takich własności urządzeń we/wy, które nie są osiągalne w inny sposób. Wymaga otwartej ścieżki. Przy wywołaniu podaje sie numer funkcji, który jest specyficzny dla sterownika urządzenia. Przyjęte jest, że sterownik zwraca w przypadku nieobsługiwanej funkcji kod E$UnkSvc - nieznana usługa. Wiele kodów jest zdefiniowanych przez Microware (plik /dd/DEFS/funcs.a), pozostałe są do dyspozycji twórców sterowników.
I$SetStt: *Set Path Status*
Jak I$GetStt, ale dla zadawania parametrów urządzeniom i realizowania specyficznych funkcji.
I$Close: *Close Path*
Zamyka otwartą ścieżkę. PD_COUNT i PD_CNT są zmniejszane, jeśli żaden inny proces z niej nie korzysta - jądro (*kernel*) wywołuje funkcję *close* modułu obsługi plików.
I$SGetSt: *Getstat using system path number*
Pozwala na dostęp do ścieżek nie będących własnością procesu. Wymaga podania systemowego (a nie lokalnego) numeru ścieżki. Można go znaleźć dzięki użyciu tablicy konwersji ścieżek zawartej w deskryptorze procesu będącego wlaścicielem ścieżki (F$GPrDsc). Przykład użycia - `procs`.

3.8 Deskryptor ścieżki

Deskryptor ścieżki jest strukturą w pamięci zajmującą 256 bajtów. Jest zerowany przy tworzeniu. Pierwsza połowa jest używana do przechowywania zmiennych potrzebnych przy obsłudze ścieżki. Jej początek jest wspólny dla wszystkich ścieżek:

```
adres   rozmiar     nazwa
-----   --------    --------------
$000    word (2)    PD_PD           Systemowy numer sciezki
$002    byte (1)    PD_MOD          Flagi trybu
$003    byte (1)    PD_CNT          Licznik uzyc (stary)
$004    long (4)    PD_DEV          Adres w tablicy urzadzen
$008    word (2)    PD_CPR          ID aktualnego procesu
$00a    long (4)    PD_RGS          Adres ramki stosu procesu
$00e    long (4)    PD_BUF          Adres bufora danych
$012    word (2)    PD_USER         Numer grupy ...
```

```
$014    word (2)                    ... i ID uzytkownika
$016    long (4)    PD_Paths        Nastepna sciezka na urzadzeniu
$01a    word (2)    PD_COUNT        Licznik uzyc sciezki (nowy)
$01c    word (2)    PD_LProc        ID ostatniego procesu
$01e    long (4)    PD_ErrNo        Numer ostatniego bledu
$022    long (4)    PD_SysGlob      Wskaznik pamieci lobalnej systemu
$026    word (2)
$000    word (2)
$02a    ...

-----   --------    --------------  OBSZAR OPCJI

$080    ...         PD_OPT

...

------------------------------------------------------------------------
```

Dalsza część zależy od modułu obsługi plików. Druga połowa ($80 - $FF) jest obszarem opcji, który jest wypełniany kopią obszaru opcji deskryptora urządzenia podczas otwierania ścieżki. Jego struktura zależy od modułu obsługi plików (*file manager*).
Dla RBF :

```
-----   --------    --------------
$080    byte (1)    PD_DTP          Typ urzadzenia (RBF - 1)
$081    byte (1)    PD_DRV          Logiczny numer napedu
$082    byte (1)    PD_STP          Predkosc krokowa glowic
$083    byte (1)    PD_TYP          Typ dysku
$084    byte (1)    PD_DNS          Gestosc zapisu
$085    byte (1)
$086    word (2)    PD_CYL          Ilosc cylindrow dla danych
$088    byte (1)    PD_SID          Ilosc powierzchni (glowic)
$089    byte (1)    PD_VFY          Flaga weryfikacji zapisu
$08a    word (2)    PD_SCT          Ilosc sektorow na sciezce
$08c    word (2)    PD_T0S          Ilosc sektorow na sciezce 0
$08e    word (2)    PD_SAS          Wielkosc segmentu
$090    byte (1)    PD_ILV          Przeplot
$091    byte (1)    PD_TFM          Tryb DMA
$092    byte (1)    PD_TOffs        Numer pierwszego cylindra
$093    byte (1)    PD_SOffs        Numer pierwszego sektora
$094    word (2)    PD_SSize        Wielkosc bloku logicznego
$096    word (2)    PD_Cntl         Slowo sterujace opcji
$098    byte (1)    PD_Trys         Ilosc ponowien przy bledzie
$099    byte (1)    PD_LUN          Numer napedu (SCSI LUN)
$09a    word (2)    PD_WPC          Prekompensacja
$09c    word (2)    PD_RWR          Ograniczenie pradu zapisu
```

```
$09e    word (2)    PD_Park         Cylinder parkowania
$0a0    long (4)    PD_LSNOffs      Adres pierwszego sektora
$0a4    word (2)    PD_TotCyl       Calkowita ilosc cylindrow
$0a6    byte (1)    PD_CtrlrID      Numer kontrolera (SCSI)
$0a7    byte (1)    PD_Rate         Predkosc
$0a8    long (4)    PD_ScsiOpt      Opcje SCSI
$0ac    long (4)    PD_MaxCnt       Najwiekszy ransfer
...

-----   --------    --------------  OBSZAR RBF

$0b5    byte (1)    PD_ATT          Atrybuty pliku
$0b6    long (4)    PD_FD           LSN deskryptora pliku
$0ba    long (4)    PD_DFD          LSN kartoteki macierzystej pliku
$0c0    long (4)    PD_DCP          Pozycja pliku w kartotece
$0c4    long (4)    PD_DVT          Adres miejsca w kartotece urzadzen
$0c8    long (4)    PD_SctSiz       Wielkosc sektora

...

$0e0    byte (1)    PD_NAME         Nazwa pliku - 32 znaki
```

4 Programy użytkowe OS–9

Programy użytkowe dostępne dla OS–9 pochodzą z różnych źródeł. Część z nich (komendy podstawowe, asembler, debuggery itp.) stanowią programy firmowe Microware ([14, 8, 7]). Istotnym uzupełnieniem jest oprogramowanie pochodzące od innych producentów ([12]) i z *public domain*. Warto również podkreślić, że firmy dostarczające sprzęt wyposażają go w niezbędne składniki programowe (*drivery*, *file managery*, przykłady aplikacji).

4.1 Podstawowe komendy OS–9

Microware dostarcza wraz z systemem w każdej z wersji zestaw podstawowych programów użytkowych pozwalających zarządzać systemem:

- attr - odczyt i zmiana atrybutów pliku
- backup - duplikowanie dysku
- binex - zamiana pliku na postać szesnastkową (S-rekordy)
- build - tworzenie krótkich plików tekstowych
- cmp - porównywanie plików
- code - wyświetlanie szesnastkowych kodów klawiszy
- copy - kopiowanie plików
- count - zliczanie znaków, słów i linii w pliku
- date - wyświetlanie daty i czasu
- dcheck - sprawdzanie poprawności katalogu/dysku
- deiniz - odłączenie urządzenia
- del - kasowanie plików
- deldir - kasowanie katalogów
- devs - wyświetlanie tablicy zainicjowanych urządzeń we/wy
- dir - wyświetlanie zawartości kartoteki
- dsave - kopiowanie poddrzewa katalogów
- dump - szesnastkowe wyświetlanie zawartości pliku
- echo - wysyłanie tekstu na ekran
- edt - edytor liniowy
- exbin - zamiana S-rekordów na postać binarną
- fixmod - odtworzenie sum kontrolnych i CRC modułu
- format - formatowanie dysków

- free - wyświetlanie wolnego miejsca na dysku
- grep - przeszukiwanie plików według wzorca
- help - wyświetlanie informacji o komendach
- ident - wyświetlanie informacji o modułach
- iniz - inicjowanie urządzeń we/wy
- link - przyłączanie modułu w pamięci
- list - wyświetlanie zawartości pliku
- load - ładowanie modułów z pliku do pamięci
- login - włączanie się do systemu (wielodostęp)
- makdir - tworzenie katalogu
- mdir - wyświetlanie kartoteki modułów
- merge - łączenie plików na wyjście standardowe
- mfree - wyświetlanie wolnego miejsca w pamięci
- pd - wyświetlanie bieżącej ścieżki danych
- pr - wyświetlanie pliku z formatowaniem
- procs - wyświetlanie aktualnych procesów
- qsort - szybkie sortowanie pliku w pamięci
- rename - zmienianie nazwy pliku
- save - składowanie modułów pamięciowych do plików
- shell - powłoka - interfejs użytkownika, język komend
- sh - powłoka - interfejs użytkownika, język komend
- sleep - zatrzymanie procesu na zadany czas, lub do przerwania
- tee - kopiowanie wejścia na kilka ścieżek wyjściowych
- tmode - wyświetlanie i ustawianie parametrów terminala
- touch - aktualizacja daty dostępu do pliku
- tr - zamiana znaków w pliku na inne (filtr)
- umacs - edytor ekranowy MicroEMACS 3.6 (mały)
- emacs - edytor ekranowy MicroEMACS 3.10 (rozbudowany)
- unlink - odłączanie modułów pamieciowych
- xmode - zmiana parametrów urządzenia znakowego

Każda z wymienionych komend podaje swój skrócony opis (składnia, opcje) po wywołaniu z opcją `-?`. Czasem więcej informacji można uzyskać po wywołaniu:

```
help <komenda>
```

4.1.1 Shell

Microware dostarcza wraz z systemem OS–9 program `shell` będący interpreterem komend użytkownika. Jest on w bardzo niewielkim stopniu podobny do unix-owego `sh`. W trybie interakcyjnym zgłasza się on znakiem ponaglenia (*prompt*) :

```
$
```

`Shell` czyta i interpretuje linię komendy wprowadzoną przez użytkownika i zakończoną znakiem $< CR >$ (*return*). Kolejne linie są pobierane ze standardowego wejścia aż do napotkania znaku końca pliku $< EOF >$, którym domyślnie jest $< ESC >$. Koniec pliku wejściowego powoduje zakończenie pracy `shell`. Znaki specjalne:

```
#   dodatkowy przydzial pamieci dla procesu
^   modyfikacja priorytetu procesu
>   skierowanie strumienia wyjsciowego (standard output)
<   skierowanie strumienia wejsciowego (standard input)
>>  skierowanie strumienia diagnostycznego (standard error)
;   separator kilku komend w linii
&   uruchomienie procesu drugoplanowego
!   lacze danych (pipe)
*   zastepuje dowolny ciag znakow w nazwie
?   zastepuje dowolny znak w nazwie
```

Przykłady:

```
$ dir a?*
```

wypisze pliki w bieżącej kartotece o co najmniej dwuliterowych nazwach rozpoczynających się od a lub A (w nazwach dla `shell` duże i małe litery nie są rozróżniane !)

```
$ list plik1 >plik2
```

przepisze plik 'plik1' do pliku 'plik2'
Każdy uzytkownik po włączeniu się do systemu otrzymuje własne środowisko (*environment*) określane przez `shell` za pośrednictwem zmiennych środowiskowych. Najważniejsze z nich:

- HOME - katalog własny użytkownika
- PORT - urządzenie komunikacyjne użytkownika
- SHELL - powłoka (interpreter komend)
- USER - nazwa użytkownika (*login name*)
- PATH - zestaw katalogów (oddzielonych ":"), które są przeszukiwane przez `shell` przy uruchamianiu programów
- PROMPT - ciąg znaków wypisywanych jako ponaglenie (np "$ ")
- TERM - typ terminala użytkownika (np vt100)

Zmienne te wyświetla się poleceniem `env`, a ustawia poleceniem `setenv`:

```
$ setenv TERM vt100
$ env

...

TERM = vt100
...
```

`Shell` uruchamiany przy włączaniu się uzytkownika (*login shell*) wykonuje komendy zawarte w pliku *.login* znajdującym sie w katalogu własnym ($HOME). Pozwala to przeprowadzić za każdym razem inicjację środowiska w szczególny dla każdego uzytkownika sposób. Analogicznie, przy kończeniu pracy przez *login shell* wykonywane są komendy z pliku *.logout*.
Wbudowane komendy `shell`:

- `ex` - uruchomienie procesu jako nakładki
- `chd` - zmiana katalogu danych
- `chx` - zmiana katalogu programów
- `kill` - porzucenie (zakończenie) procesu
- `logout` - zakończenie bieżącej powłoki (*shell*)
- `setenv` - ustawienie zmiennej środowiskowej
- `setpr` - ustawienie priorytetu
- `unsetenv` - usunięcie zmiennej środowiskowej
- `w` - oczekiwanie na koniec procesu
- `wait` - oczekiwanie na koniec wszystkich procesów potomnych

`Shell` wykonuje komendy zgodnie z następujacą procedurą:

1. Pobranie linii ze standardowego wejścia.
2. Przygotowanie komendy:
   - sprawdzenie składni
   - wyodrębnienie słowa kluczowego, parametrów i modyfikatorów
   - rozwinięcie znaków uniwersalnych (`*`,`?`)
3. Wykonanie komendy, jeśli jest ona wbudowana, lub poszukiwanie jej kolejno w:
   - kartotece modułów
   - katalogu programów
   - wszystkich katalogach ze zmiennej PATH

   W przypadku nieznalezienia komendy powrót z błędem *can't find command*
4. Załadowanie komendy do kartoteki modułów. Jeśli się nie powiedzie, to wykonanie pliku jako skryptu dla `shell`.

5. Uruchomienie programu:
   - bezpośrednio, jeśli jest to kod binarny
   - jako argumentu dla runb, gdy jest to kod pośredni dla BASIC
6. Przy braku powodzenia we wszystkich powyższych próbach - zwrócenie błędu: *can't execute command*

Komendy i skrypty muszą mieć atrybut ę"i/lub "pe".
Dokładniejszy opis `shell` można znalezć w dokumentacji OS–9 ([14]).

4.1.2 Sh

Powłoka `sh` powstała w ramach projektu TOP (*The OS–9 Project*) i jest dostępna publicznie. Jej własnosci są bardzo zbliżone do `sh` z systemu Unix, można się więc posłużyć dokumentacją tego systemu w postaci drukowanej, lub podrecznikiem systemowym (*man pages*) dostępnym za pośrednictwem komendy `man` w systemie Unix. Poza zachowaniem składni języka komend autorzy wprowadzili wiele udogodnień znanych z innych powłok (`csh, ksh, tcsh`), co ułatwia pracę interakcyjną z `sh` w systemie OS–9. Do tych udogodnień należą:

- pamięć wydawanych komend (*history*)
- możliwość edycji linii komendy przywołanej z historii
- podstawianie komend (*alias*)

`Sh` jest językiem programowania, który wykonuje komendy pobierane z terminala, lub pliku. Komenda może być prosta (jak w `shell`), lub złożona, z wykorzystaniem konstrukcji znanych z języków wyższego rzędu:

```
for <nazwa> [in <slowo> ...] do <lista> done

case <slowo> in [ <wzorzec> [| <wzorzec>] ... <lista>;; ] ... esac

if <lista> then <lista> [elif <lista> then <lista>] ... [else <lista>] fi

while <lista> do <lista> done
```

gdzie `<lista>` oznacza jedną, lub więcej komend połączonych sprzęgami "(*pipelines*).
`Sh` umożliwia kierowanie strumieni wejściowych i wyjściowych:

```
$ dir 1>plik1 2>plik2
```

spowoduje skierowanie standardowego wyjścia (strumień 1) na "plik1", a wyjścia diagnostycznego (*standard error* - strumień 2) na "plik2". Podwójne znaki >> oznaczają dopisywanie, zamiast zastępowania istniejących plików.
Rozwijanie nazw obejmuje (oprócz "*"i "?") wzorce wyliczane [...]:
`[acdz]*` oznacza wszystkie nazwy rozpoczynające się od a,c,d lub z
`[q-t]` oznacza litery q,r,s,t
Niektóre wbudowane komendy:

- `break` - wyjście z pętli `for` lub `while`
- `continue` - przejście do następnej iteracji
- `cd [arg]` - zmiana katalogu na `arg`, lub HOME
- `echo [arg]` - wypisanie `arg`
- `exec [arg]` - wykonanie nakładki
- `exit [n]` - zakończenie `sh` ze statusem = `n` (kod błędu)
- `pwd` - wypisanie bieżącego katalogu
- `read [nazwy]` - wczytanie linii i podstawienie wartości pod nazwy
- `return [n]` - wyjście z funkcji ze statusem = `n`
- `set [opt]` - ustawianie różnych opcji `sh`
- `shift [n]` - przenumerowanie parametrów pozycyjnych
- `trap [arg][n]` - podłożenie obsługi sygnału `n` przez komende `arg`
- `wait [n]` - oczekiwanie na koniec procesu `n` i zwrot jego statusu

Petla:

`for ... do ... done`

służy do wielokrotnego wykonania pewnej akcji dla różnych argumentów:

```
//h1//SUPWA//W> for i in *
> do
> list $i
> done
```

spowoduje kolejne wyświetlenie zawartości wszystkich plików w bieżącym katalogu. Warto zwrócić uwagę na nowy *prompt* ”> ”pojawiający się przy wpisywaniu kolejnych linii kontynuacyjnych pętli `for ... do`, które jest zakończone poleceniem `done`. $i jest typowym (jak w przypadku zmiennych) sposobem podstawienia bieżącej wartości zmiennej - tu roboczej - ”i”. ”*”jest rozwijana przez `sh` przed wykonaniem komendy w listę wszystkich plików w bieżącym katalogu, których nazwy pasują do wzorca (tu - wszystkie poza zaczynającymi się od ”.”). Jako zakres zmienności i można podać również dowolnie wyspecyfikowaną listę nazw.
Konstrukcja:

`case ... in ... esac`

jest przełącznikiem umożliwiającym wybór akcji w zależności od wartości parametru:

```
case $1 in
[1-3]) echo slabo;;
4) echo dobrze;;
[5-6]) echo swietnie;;
*) echo $1 to nie jest stopien
esac
```

Konstrukcja:

```
if ... than ... else ... fi
```

pozwala przełączać akcję w zależności od spełnienia warunku:

```
if test -f $1
then
echo $1 jest plikiem
elif
test -d $1
echo $1 jest katalogiem
else
echo nie rozumiem $1
fi
```

ten skrypt potrafi odróżnić pliki od katalogów dzięki zastosowaniu polecenia test. Polecenie to ma kilka opcji:

```
-f - czy argument jest plikiem
-d - czy argument jest katalogiem
-w - czy mamy zezwolenie na zapis
-r - czy mamy zezwolenie na odczyt
-s - czy plik ma niezerowy rozmiar
```

Opcje można łączyć operatorami and -a"i or -o":

```
test -f file -a -w file
```

określa, czy `file` jest plikiem, do którego możemy pisać.

4.1.3 Edytor

Edytor (MicroEMACS) jest wywoływany komendą `emacs nazwa_pliku`, po której edytowany plik wczytywany jest do pamięci i w pamięci obrabiany. W międzyczasie zawartość rzeczywistego pliku na dysku nie ulega zmianie. Przed wyjściem z edytora należy więc wydać komendę SAVE-FILE (zapis tekstu) aby wprowadzone zmiany zapisane zostały na dysk. Podczas długiej pracy nalezy to robić w regularnych odstępach czasu, gdyż w przypadku zawieszenia się systemu lub programu kilka godzin pracy może być stracone. Edytor poznaje czy wprowadzono zmiany w pliku które nie zostały zapisane i w przypadku ich stwierdzenia pyta czy zapisać plik.
Więcej komend można znaleźć w pliku **emacs.txt**, który zawiera opis edytora MicroEMACS 3.10. Plik **emacs.tut** zawiera krótki kurs posługiwania się tym edytorem.

4.1.4 Grep

`Grep` jest filtrem przegladającym plik w poszukiwaniu zadanego wzorca. Wzorzec zadaje się w postaci wyrażenia regularnego. Duże i małe litery nie są rozróżniane. Puste linie nie pasują do żadnego wzorca. Wyrażenie regularne powinno być ujęte w cudzysłowy w celu uniknięcia interpretacji jego cześci przez `sh`. Wyrażenia regularne budowane są z:

PODSTAWOWE KOMENDY EMACS

```
Ctrl-N        -Kursor w dol
Ctrl-P        -Kursor w gore
Ctrl-F        -Kursor w prawo
Ctrl-B        -Kursor w lewo
Ctrl-A        -Kursor na pocz. lini
Ctrl-E        -Kursor na koniec lini
ESC <         -Kursor na pocz tekstu
ESC >         -Kursor na koniec tekstu
ESC G         -Kursor do lini o danym numerze

ESC S         -Poszukiwanie wyrazu
Ctrl-R        -Poszukiwanie wyrazu wstecz
Ctrl-X S      -ISearch - poszukiwanie przyrostowe
Ctrl-X R      -ISearch do tylu

Ctrl-D        -Kasowanie znaku pod kursorem
Ctrl-K        -Kasowanie od kursora do konca lini
Ctrl-W        -Kasowanie obszaru kursor <--> znacznik i zapamietanie

ESC Space     -Ustawienie znacznika
ESC W         -Zapamietanie obszaru znacznik <--> kursor
Ctrl-Y        -Umieszczanie w miejscu kursora zapamietanego obszaru

ESC X         -Slowne podanie komendy

Ctrl-X O      -Zmiana aktywnego okna
Ctrl-X 0      -Zamykanie aktywnego okna
Ctrl-X 1      -Zamykanie pozostalych okien
Ctrl-X 2      -Podzial aktywnego okna na dwa
Ctrl-X Ctrl-S -Zapis tekstu
Ctrl-X Ctrl-C -Porzucenie edycji
Ctrl-X C      -Uruchomienie nowej powloki (shell)
ESC Z         -Koniec pracy z programem (zapis plikow)
```

```
x       zwykly znak (poza wymienionymi dalej) pasuje do takiego samego znaku
'\'     cytuje dowolny znak.  "\$" oznacza znak dolara.
'^'     na poczatku wyrazenia oznacza poczatek linii.
'$'     na koncu wyrazenia oznacza koniec linii.
'.'     oznacza dowolny znak oprocz konca linii.
':a'    dwukropek wydziela klasy znakow: ":a" - litery
':d'                                      ":d" - cyfry
':n'                                      ":n" - litery i cyfry
': '                                      ": " - spacje, tab, znaki sterujace
'*'     wyrazenie zakonczone gwiazdka oznacza dowolna (w tym zerowa)
        ilosc powtorzen tego wyrazenia: "fo*" oznacza "f", "fo"
        "foo", etc.
'+'     plus oznacza co najmniej jedno powtorzenie wyrazenia: "fo+"
        oznacza  "fo", etc.
'-'     minus po wyrazeniu oznacza jego opcjonalne wystepowanie we wzorcu.
'[]'    ciag znakow w klamrach pasuje do dowolnego znaku z tego ciagu.
        Jesli pierwszy znak w klamrach jest "^", to znaczenie warunku sie
        odwraca. Na przyklad: "[xyz]" pasuje do "xx" i źyx", zas "[^xyz]"
        pasuje do ąbc", ale nie do ąxb". Mozna okreslac przedzialy
        znakow przy pomocy -" Np. [a-z] oznacza wszystkie litery.
```

Złożenie wyrażeń regularnych jest wyrażeniem regularnym.

4.1.5 Sed

Jest to edytor ciągów znakowych używany jako filtr. Każda linia wejściowa jest czytana i wykonywane są na niej komendy edycji podane w programie. Wywołanie:

```
sed -e skrypt
```

lub:

```
sed -f plik
```

Pierwsza postać zawiera polecenia w linii wywołania, druga zawiera nazwę pliku z poleceniami. Polecenie ma ogólna postać:

```
adres1,adres2 zadanie argument
```

brak adresów oznacza każdą linię, pojedynczy adres jest równoważny dwu jednakowym adresom. Dwa adresy wyznaczają przedział (liczony w liniach), do którego odnosi się zadanie. Adres może być podany numerycznie (dziesiętnie), lub jako wzorzec (wyrażenie regularne) zawarty pomiędzy ”/.../”. ”+”i -ńie są dostępne przy adresowaniu linii.
Zadania:

```
s//.../.../ - podstawienie lancucha
d           - skasowanie linii
a\          - dopisanie tekstu na koncu linii
i\          - dopisanie tekstu na poczatku linii
c\          - zmiana linii
```

Każda linia tekstu po a, i, c (oprócz ostatniej) jest kończona ”.
Przykład:

```
echo $PATH|sed -e 's//:// //g'
```

gdzie g oznacza wykonanie podstawienia wszystkich, a nie tylko pierwszego z wystąpień łańcucha (tu ”:”).

4.1.6 Gawk (GNU awk)

Jest to programowalny filtr do tekstów. Pozwala stosować wyrażenia warunkowe, pętle, zmienne przy notacji podobnej do języka C.
Wywołanie:

```
    gawk program plik1 plik2 ...
```

wykonuje instrukcje dla gawk zawarte w pliku program, traktując jako wejsciowe dane zawartość plik1, plik2 itd. Jeśli nie ma plików, używa się wejścia standardowego. Wynik jest wysyłany na wyjście standardowe. Awk szuka w każdej linii wejściowej selektora, by w przypadku jego znalezienia podjać akcję:

```
BEGIN{ wyrazenia poczatkowe
}
{   selektor {akcja}

    ...

}
END{ wyrazenia koncowe
}
```

Par selektor-akcja może być kilka. Selektory są kolejno sprawdzane i akcje wykonywane, gdy odpowiednie selektory są prawdziwe. Przy braku selektora akcja jest wykonywana zawsze, brak akcji oznacza kopiowanie linii spełniających selektor na wyjście. Akcja może się składać z kilku wyrażeń oddzielonych ”;”. BEGIN i END są wykonywane raz, na początku i na końcu przetwarzania pliku. Każda wczytywana linia jest dzielona na pola oddzielone spacjami, lub znakiem wyspecyfikowanym jako FS=”:”(w tym przykładzie - dwukropek). $0 oznacza całą linie, $i - i-te pole. Ilość pol zawiera zmienną NF, ilość linii wejściowych - NR. Znak separacji linii może być zmieniony przez podstawienie do zmiennej RS. Nazwa pliku jest dostępna w zmiennej FILENAME.
Wyrażenia mogą dotyczyć napisów lub liczb, dostępne są operatory jak w C:

```
  +, -, *, //, %, ++, --, +=, -=, *=, //=, %=, <, >, <=, >=, ==, !=.
```

Zmienne mogą być skalarami, elementami tablic (x[i]), nazwami pól ($3).
Zmienne są inicjowane jako puste napisy.
`NF>5` jest prawdą, gdy jest więcej niż 5 pól w linii,
`$1<''s''` jest prawdą dla pola 1 zawierającego tekst poprzedzający leksykalnie “s”
jest operatorem dokładnej zgodności:
`$1 /abc|ABC/` jest prawdą dla linii, których pierwsze pole jest równe ”abc” lub ”ABC”
Akcja składa się z wyrażeń oddzielonych średnikami. dostępne są:

```
if (warunek) wyrazenie [ else wyrazenie ]
while (warunek) wyrazenie
for (wyrazenie;warunek;wyrazenie) wyrazenie
for (indeks tablicy) wyrazenie
break
continue
{[ wyrazenie ] ...)
zmienna=wyrazenie
print [ lista wyrazen ] [ >wyrazenie ]
printf format [, lista wyrazen ] [ >wyrazenie ]
next        \#pominiecie pozostalych selektorow w tej linii
exit        \#pominiecie reszty wejscia
```

Przykład:

```
if(\$2>\$1){
    x=\$1
    \$1=\$2
    \$2=x
}
```

kopiuje stdin na stdout zamieniając pola 2 i 1 gdy $2 jest większe od $1. Porównania są w miarę możliwości numeryczne, a w innych przypadkach - alfabetyczne.
Drukowanie:

```
print $2, $1
```

drukuje pole 2, nową linie i pole 1,

```
printf "%s, %s,\n", $1, $2
```

drukuje pole1 i pole 2 oddzielone przecinkiem.
Inne wbudowane funkcje to:

```
sqrt, log, exp, int
```

`length` - długość napisu
`substr(s,m,n)` -podciąg z s począwszy od m-tego znaku o długości nie mniejszej niż n
`index(s,t)` - pozycja pierwszego wystąpienia podciągu t w ciągu s.
Uwaga:
Dokumentacje tych i wielu innych komend przeniesionych do OS–9 z systemu Unix można znaleźć w dokumentacjach systemu Unix, lub książkach o tym systemie (...)

4.2 Środowisko programowania w OS–9

Tworzenie programu składa się z ciągu operacji powtarzanych aż do uzyskania zadawalającego wyniku:

- tworzenia (edycji) źródła,
- kompilacji/asemblacji do plików relokowalnych (*ROF - relocatable object file*),

- łączenia (*link*) ROF-ów w moduł programowy,
- testowania przy pomocy debuggera.

Narzędzia do tworzenia oprogramowania w OS–9 dostarczane przez Microware to:

- `cc` egzekutor kompilatora C
- `cpp` preprocesor C
- `c68` kompilator C dla 68000 i 68010
- `c68020` kompilator C dla 68020, 68030 i68040
- `o68` optymizator asemblera
- `r68` asembler dla 68000 i 68010
- `r68020` asembler dla 68020, 68030 i 68040
- `l68` linker
- `make` automatyczny zarządca kompilacji
- `debug` symboliczny debugger asemblera
- `srcdbg` symboliczny debugger C
- `sysdbg` debugger procesów w trybie systemowym

4.2.1 Kompilator

Kompilacja C składa się z trzech faz:

- wstępne przetwarzanie (`cpp`),
- kompilacja (`c68, c68020`),
- optymalizacja (`o68`).

Preprocesor tworzy roboczy plik z rozwiniętymi makrami (`#define`), wstawkami (`#include`) i uwzględnieniem kompilacji warunkowej (tt #ifdef...).
Kompilator tłumaczy program w C na język asemblera (opcja -a w `cc` pozwala zakończyć na tym etapie).
Optymizator przegląda źródłowy program w języku asemblera w poszukiwaniu ciągów instrukcji, które można poprawić i zmienia je.
Fazy te nie są zazwyczaj wywoływane ręcznie. Służy do tego program egzekutora cc, który jest wyposażony w liczne opcje:

```
-a          kompilacja do zrodla asemblera
-bg         tworzenie modulu sticky"
-bp         wyswietlanie komend poszczegolnych faz
-c          pozostawienie komentarzy w zrodle asemblera
-cs=<name>  wskazanie niestandardowego modulu startowego
-d<name>    zdefiniowanie nazwy dla preprocesora
```

```
-e=<n>      numer edycji modulu
-f=<path>   nazwa sciezki dla wyniku (wzgledem ęxec")
-fd=<path>  nazwa sciezki dla wyniku (wzgledem "data")
-g          dodanie informacji dla debuggera zrodlowego
-i          uzycie trapow we//wy (cio)
-j          zakaz tworzenia tablicy skokow
-k=<n>[w|l][cw|cl][f]
            procesor docelowy - 0=68000, 2=68020
            przesuniecia dla danych - w=word, l=long word
            przesuniecia dla kodu - cw=word, cl=long word
            koprocesor - f=68881 (tylko dla 68020)
-l=<name>   dodatkowa biblioteka
-lo=<opts>  opcje do przekazania dla linkera
-m=<n>[k]   dodatkowa pamiec stosu dla modulu wynikowego
-n=<name>   nazwa modulu wynikowego
-nl         pominiecie standardowych bibliotek
-o[=<n>]    poziom optymizacji (0, 1, 2)
-q          pominiecie komunikatow o postepach kompilacji
-r[=<dir>]  komilacja//asemblacja do poziomu plikow relokowalnych
-s          pominiecie kontroli stosu w programie
-t=<dir>    katalog dla plikow tymczasowych
-u<name>    pominiecie definicji nazwy
-v=<dir>    katalog dla plikow wlaczanych przez preprocesor
-w=<dir>    katalog dla plikow wlaczanych przez linker
-x          uzycie trapow dla funkcji matematycznych (math)
```

`Cc` i `make` używają pewnych konwencji w nazwach plików (rozszerzeń nazwy po ”.”). Część nazwy przed ostatnią kropką zwie się *root*. Tworzone pliki zachowują główną część nazwy (*”root”*) i otrzymują określone rozszerzenie:

```
".c"    zrodlo w C
".a"    zrodlo asemblerowe
".r"    plik relokowalny (ROF) - wynik asemblacji
      wynikowy modul programowy - wynik linkowania
".h"    pliki wlaczane do C
".d"    pliki wlaczane do asemblera
".l"    biblioteki ROF-ow
```

Biblioteki dołączane przez linker są pobierane z /dd/LIB (uwaga na -w”) i zależą od opcji cc:

```
      clibn.l, math.l, sys.l;
-x"    clib.l, sys.l;
-i"    cio.l, clibn.l, math.l, sys.l;
-ix"   cio.l, clib.l, sys.l;
```

Jako pierwszy moduł używany jest *cstart.r* (z /dd/LIB). Jest on odpowiedni tylko dla programów, dla modułów wynikowych innych typów (*trap, fman, ...*) trzeba go zastąpić kodem własnoręcznie napisanym w asemblerze.

Linker tworzy moduł wynikowy przez połączenie kilku modułów relokowalnych (ROF-ów). Moduły relokowalne mogą zawierać kod, definicje symboli, definicje pamięci statycznej. Mogą one być dostarczane jako pliki "*.r", lub biblioteki "*.l". Biblioteka zawiera kilka ROF-ów połączonych przez `merge`. Różnica polega na tym, że linker włącza do modułu wynikowego wszystkie ROF-y podane w linii wywołania (*.r), oraz tylko te ROF-y z bibliotek (*.l), które są niezbędne do zdefiniowania wszystkich nazw. Moduły ROF są włączane w kolejności wywołania w linii komendy l68. Moduły z bibliotek włącza się po wszystkich modułach "*.r". Podczas pracy linker buduje i aktualizuje tablicę nazw, które muszą być zdefiniowane. Przeglądanie bibliotek jest jednorazowe, moduły ROF nie definiujące potrzebnych nazw są pomijane. Jeśli ROF definiuje któraś z potrzebnych nazw, wszystkie jego nazwy publiczne (odwołania i definicje) są dołączane do tablicy.

4.2.2 Make

`Make` pozwala dokonywać automatycznie odpowiednich zmian w całej grupie plików według zadeklarowanych zależności pomiędzy nimi. Określa, czy któryś z plików wymaga uaktualnienia przez porównanie czasu jego ostatniej modyfikacji z takim samym czasem dla wszystkich plików, od których jest on uzależniony. Najczęściej jest wykorzystywany do kompilacji programów napisanych w językach wysokiego rzędu, ale moze też służyć do innych celów, gdy pliki wynikowe zależą od źródłowych, a procedury ich tworzenia są ustalone.
Opis tworzenia wyników dla `make` jest zawarty (domyślnie) w pliku o nazwie `makefile` w bieżącym katalogu danych. Można podać inną nazwę tego pliku w linii wywołania `make` (opcja -f). W *makefile* wystepują trzy typy zapisów:

- zależności - określające powiązania plików wynikowych ze źródłowymi:

  ```
  <plik\_wynikowy>:[[<plik>],<plik>]
  ```

  (rozpoczynające się w pierwszej kolumnie)

- komendy - podające sposób tworzenia pliku wynikowego (zaczynają się od znakow białych); dopuszczalna jest dowolna ilość dowolnych komend OS–9;

- komentarze - linie zaczynające się "*" lub "#" są ignorowane.

Linie dłuższe niż 256 znakow mogą być kontynuowane przy pomocy znaku "ś na końcu kontynuowanej linii.
Make czyta cały plik `makefile` i tworzy drzewo uzależnień na podstawie znalezionych zależności. W drugim przebiegu dodaje wbudowane zależności pomiędzy plikami relokowalnymi a zródłowymi, które nie zostały podane jawnie. W trzecim przebiegu sprawdzane i porównywane są daty plików według drzewa uzależnień. Jeśli data któregoś ze źródeł jest nowsza niż pliku wynikowego, to wykonywane są komendy podane po zależności (czas jest pamiętany w plikach z dokładnością do 1 minuty!).
`Make` używa mechanizmu makrodefinicji, co pozwala uzyskać większą elastyczność i wygodę użytkowania. Makra są definiowane w *makefile*, lub w linii wywołania w postaci:

```
<nazwa makra>=<rozwiniecie makra>
```

Pewne makra mają określone znaczenie:

```
ODIR    - katalog plikow bez rozszerzen
SDIR    - katalog plikow zrodlowych (.a, .c, ...)
RDIR    - katalog plikow relokowalnych (.r)
CFLAGS  - opcje dla kompilatora C
RFLAGS  - opcje dla asemblera
LFLAGS  - opcje dla linkera
CC      - nazwa kompilatora
RC      - nazwa asemblera
LC      - nazwa linkera
```

Makra są wywoływane przez poprzedzenie nazwy ujętej w nawiasy znakiem dolara. Za $(CC) podstawione zostanie domyślnie cc, o ile nie zdefiniujemy inaczej.

```
$@  jest zastepowane nazwa pliku, ktory ma byc wytworzony przez komende;
$*  jest zastepowane glowna czescia nazwy;
$?  jest zastepowane lista plikow zrodlowych nowszych, niz wynikowy.
```

Opcje make:

```
-b        zakaz uzycia wbudowanych regul
-bo       zakaz uzycia wbudowanych regul dla ROF
-d        tryb testowy (wypisywanie dat uzaleznien)
-dd       tryb testowy (wiecej informacji)
-f[=]<xxx> zadanie <xxx> jako makefile
-f-       pobieranie makefile z stdin
-i        ignorowanie bledow
-n        zakaz wykonywania komend (tylko echo)
-s        zakaz wypisywania komend (silent mode)
-t        odnowienie dat bez wykonywania komend
-u        bezwarunkowe wykonanie (niezaleznie od dat)
-x        uzycie kompilatora skrosnego
-z[=<path>] pobranie listy plikow do wykonania z stdin lub <path>
```

4.2.3 Debuggery

`Debug` jest programem symbolicznego debuggera na poziomie języka asemblera. Pozwala uruchamiać i poprawiać programy pracujące w trybie użytkownika przy pomocy specjalnych funkcji systemowych. Proces poddawany uruchamianiu i testowaniu istnieje w systemie jak w czasie normalnej pracy, ale jest uruchamiany dopiero po odpowiednim wywołaniu systemowym wykonanym przez proces rodzicielski (*debug*), co pozwala np. ustawić pułapki. `Debug` próbuje załadować tablicę symboli (**.stb*) o tej samej nazwie głównej, co uruchamiany program. Poszukuje kolejno w katalogu *exec*, a potem według zmiennej środowiskowej PATH. Przed przeszukaniem każdego katalogu sprawdzany jest podkatalog STB (o ile istnieje). Jeśli plik *.stb nie zostanie znaleziony, to niemożliwe jest korzystanie z nazw symbolicznych, ale *debug* pracuje poprawnie.
Debug pozwala na :

- pułapki programowe,
- podgląd/modyfikację pamięci,

- podgląd/modyfikację rejestrów procesora,
- deasemblację kodu z pamięci,
- uruchamianie procesów (*fork*),
- przyłączanie się (*linking*) do modułów,
- sterowanie uruchamianiem programu.

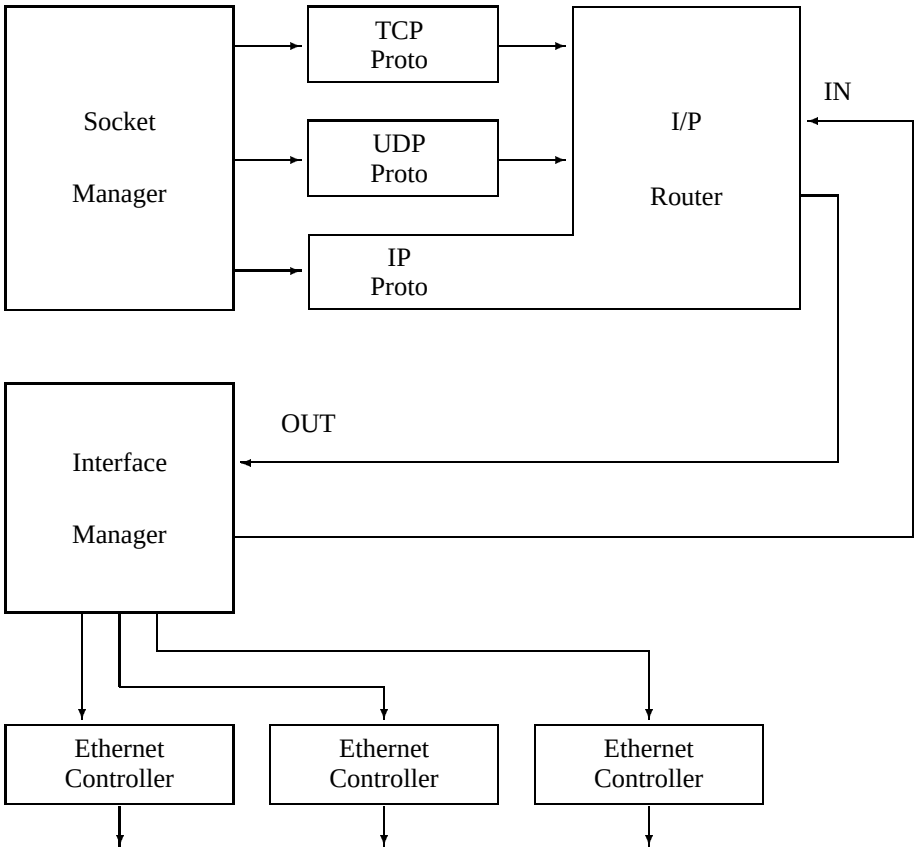
Ułatwienie stanowi operowanie na wyrażeniach przy wyliczaniu np. adresow:

```
dbg: d [.a0+6]+.d0 20
```

oznacza:
”wyświetl $20 bajtów od adresu otrzymanego przez dodanie zawąrtości komórki wskazanej przez a0 zwiekszonej o 6 do zawartości rejestru d0”.
Przyłączenie do modułu komendą ”l” powoduje umieszczenie w rejestrze relokacji (dostępnym jako ”.r7”) adresu początku modułu. Każdy z ośmiu rejestrów relokacji może być ustawiony jako adres bazowy komendą ”@”:

```
dbg: l term
dbg: @7
dbg: c 50
0x00000050+r7:18 .
dbg:
```

co ułatwia odwoływanie się do wnętrza modułu.
Debug pozwala uruchamiać program ze śledzeniem (*tracing*), co powoduje wolną pracę, ale pozwala ustawiać pułapki (*breakpoints*) w ROM, lub z pełną prędkością, ale pułapki są ustawiane tylko w RAM (jako nielegalne instrukcje, które za pośrednictwem trapu budzą debug).
`Srcdbg` działa podobnie, ale dodatkowo używa pliku **.dbg* tworzonego przez kompilator, który pozwala powiązać kod asemblerowy ze źródłem programu w C, a zatem wykonywać program krokowo według źródła w C, z wykorzystaniem nazw zmiennych i operacji na nich (obliczanie wyrażeń). Debugger źródłowy jest wyposażony w instrukcję użytkowania dostępną w czasie pracy dzięki komendzie `help`.



Rysunek 2: OS–9 Internet

5 Internet Support Package (ISP)

Pakiet ISP/OS–9 umożliwia komunikację pomiędzy systemami OS–9 i innymi systemami w sieci Internet z wykorzystaniem protokołu TCP/IP. Biblioteka funkcji języka C udostępnia interfejs programowy niemal identyczny z systemem gniazdek (*sockets*) znanym z Unix-a BSD. Poza licznymi programami o charakterze administracyjnym dostępne są usługi przesyłania plików (*ftp*) i zdalnego połączenia terminalowego (*telnet*).

5.1 Sockman

Moduł zarządzania plikami (*file manager*) **sockman** pozwala na łatwy dostęp z poziomu programu pisanego w języku C do protokołów sieciowych warstwy transportowej (**TCP/IP**). Umożliwia on przenoszenie sieciowych aplikacji z systemów typu Unix do OS–9 i odwrotnie. Protokoły adresowe (Internet itp.) zostały zaimplementowane w postaci modułów podprogramów, co pozwala na dynamiczne ładowanie jedynie aktualnie używanych, oraz ułatwia dalsze rozszerzanie systemu o ewentualne nowe protokoły.

5.2 Ifman

Moduł **ifman** (*Interface Manager*) uzupełnia gniazdka o niezależną od sprzętu warstwę umożliwiającą konfigurowanie interfejsów sieciowych i zarządzanie nimi. Protokoły wysokiego poziomu mogą dzięki niemu dopasowywać się do różnych interfejsów, co daje dużą elastyczność systemu.
Implementacja TCP/IP firmy Microware jest zgodna ze standardami amerykańskiego departamentu obrony (*United States DoD ArpaNet Standards*) i uwzględnia usprawnienia BSD4.3. Warstwa **IP** obsługuje zarówno statyczne trasowanie pakietów pomiędzy interfejsami lokalnymi, jak obsługę domy'slną (*default routing*) i za pośrednictwem bram (*gateway routing*). Ftp i telnet występują nie tylko jako programy uzytkowe (*client*), pozwalające korzystać z zasobów innych systemów, ale również jako serwery dostarczające analogicznych usług innym systemom.

5.3 Programy użytkowe pakietu ISP

Użytkownik pakietu ISP otrzymuje następujące programy użytkowe:

- `ftp` - *File Transfer* - program do przesyłania plików do/z innych systemów.
- `idbgen` - *Internet Database Generation* - program do budowania modułu danych **inetdb** na podstawie czterech plików konfiguracyjnych: **hosts**, **networks**, **protocols**, **services**.
- `idbdump` - *Internet Database Display* - program pozwalający przeglądać aktualną bazę (**inetdb**).
- `ifgen` - *IF Device Descriptor Generator* - program do tworzenia postaci źródłowych (asembler) deskryptorów wszystkich urządzeń opisanych w pliku **if_devices**.
- `ifstat` - *IF Device Status* - program do raportowania stanu urządzeń interfejsu sieciowego.
- `ispstart` - *Internet Startup* - program inicjujący pracę pakietu **ISP**.
- `lestat` - *Lance Device Status* - program generujący raport o stanie sterownika sieciowego typu **Lance**.
- `mbinstall` - *F_MBUF Installation* - program instalujący w systemie OS–9 funkcję (usługę systemową) **F_MBUF**.
- `telnet` - *Telnet User Interface* - program do komunikacji terminalowej pomiędzy systemami w sieci Internet. Umoliwia włączanie się (*log-in*) z innymi systemami.

Oprócz wymienionych programów dostarczane są serwery (*daemon servers*) i moduły obsługi połączeń (*connection handlers*):

- **ftpd** - *FTP Server Daemon*
- **ftpdc** - *FTP Server Connection Handler*
- **telnetd** - *Telnet Server Daemon*
- **telnetdc** - *Telnet Server Connection Handler*

5.4 Gniazdka (*sockets*)

Pakiet ISP dla systemu OS–9 jest oparty na gniazdkach (*sockets*) znanych z BSD Unix. Gniazdka stanowią zakończenia ścieżki komunikacyjnej pomiędzy systemami, lub w obrębie jednego systemu operacyjnego. Pozwala to jednemu z systemów wysyłać i odbierać informację z innych systemów w sieci. Ułatwiają one ponadto używanie protokołu TCP/IP z poziomu programu pisanego w języku C. Definicje ziązane z gniazdkami przytoczono w Dodatku C.
Każde gniazdko ma swoje rodzinę adresową (*address family*), protokół (*protocol*) i typ (*type*). W pakiecie OS–9/ISP dostępne są:

- trzy rodziny adresowe:
  - AF_INET - rodzina adresowa *ARPA Internet*
  - AF_UNIX - rodzina adresowa lokalna
  - AF_ETHER - rodzina adresowa *Ethernet*
- dwa protokoły wysokiego poziomu:
  - TCP - *Transmission Control Protocol*
  - UDP - *User Datagram Protocol*
- trzy typy gniazdek:
  - SOCK_STREAM - *Stream sokets* z protokołem TCP
  - SOCK_DGRAM - *Datagram sockets* z protokołem UDP
  - SOCK_RAW - *Raw sockets* z protokołem AF_ETHER

Gniazdka typu *stream* są dwukierunkowymi strumieniami danych podobnymi do łączy (*pipes*). Pracują w trybie połączeniowym. Gniazdko w tym trybie ma dokładnie określonego, ustalonego partnera (*peer*). Używają protokołu TCP, który zapewnia bezstratną i wolną od powtórzeń transmisję danych. W przypadku niemożności przesłania danych w określonym czasie (*time-out*) połączenie uważa się za zerwane i każda funkcja korzystająca z tej ścieżki zwraca błąd E_EOF.
Gniazdka typu *datagram* pozwalają na dwukierunkowe przesyłanie komunikatów (*messages*). Mogą pracować w trybie połączeniowym, lub bezpołączeniowym. Używają protokołu UDP, który nie gwarantuje zachowania kolejności, kompletności i niepowtartalności pakietów. Mimo tych wad w wielu przypadkach jest on przydatny ze względu na swą zwartość.
Gniazdka typu *raw* dają programiście dostęp do protokołów niskiego poziomu (w pakiecie OS–9/ISP obejmuje to tylko rodzinę AF_ETHER). Ich niewłaściwe uzycie wiąże się z możliwością spowodowania poważnych zaburzeń w pracy systemu i sieci, dlatego dostępne są tylko dla administratora systemu (grupa 0 - *super user*).
Utworzenie połączenia obejmuje kilka podstawowych etapów:

- postronie serwera:
  - utworzenie gniazdka funkcją `socket()`
  - nadanie mu nazwy funkcją `bind()`
  - ustawienie w tryb nasłuchu (`listen()`)
  - przyjęcie zgłoszenia klienta (`accept()`)

- po stronie klienta:
  - utworzenie gniazdka funkcją `socket()`
  - połączenie z nasłuchującym gniazdkiem (`connect()`)

Po utworzeniu połączenia pomiędzy gniazdkami po stronie klienta i serwera można przesyłać dane przy pomocy funkcji:

- `read()`
- `recv()`
- `recvfrom()`
- `write()`
- `send()`
- `sendto()`

5.5 Biblioteka netdb.l dla OS–9

W pakiecie ISP przewidziano funkcje dostępu do bazy danych Internet-u. Ze względu na romowalność systemu OS–9 baza danych jest zorganizowana w postaci modułu danych **inetdb** tworzonego z plików **hosts**, **protocols**, **networks** i **services** przy pomocy programu `idbgen`. Wszystkie funkcje korzystające z bazy **inetdb** muszą być przyłączone (*linked*) do tego modułu. Uzyskuje się to przez wywołanie jednej z funkji typu *set* (`sethostent()`, `setnetent()`, `setservent()` itp.), lub typu *get* (`gethostent`, `getnetent` itp.). Odłączenie od modułu **inetdb** następuje w wyniku wywołania jednej z funkcji typu *end* (`endhostent` itp.).
W bibliotece **netdb.l** znajdują się następujące funkcje:

- `endhostent()`, `endnetend()`, `endprotend()`, `endservent()`,
- `gethostent()`, `gethostbyaddr()`, `gethostbyname()`,
- `getnetend()`, `getnetbyaddr()`, `getnetbyname()`,
- `getprotend()`, `getprotbyname()`, `getprotbynumber()`,
- `getservent()`, `getservbyname()`, `getservbyport()`,
- `inet_addr()`, `inet_lnaof()`, `inet_makeaddr()`, `inetof()`, `inet_network()`, `inet_ntoa()`,
- `sethostent()`, `setnetend()`, `setprotend()`, `setservent()`,

5.6 Biblioteka socklib.l dla OS–9

W bibliotece **socklib.l** znajdują się następujące funkcje:
`_ss_sevent()`, `accept()`, `bind()`, `connect()`, `gethostname()`, `getpeername()`, `getsockname()`, `getsockopt()`, `listen()`, `recv()`, `recvfrom()`, `send()`, `sendto()`, `setsockopt()`, `shutdown()`, `socket()`.
Zapewniają one interfejs zgodny z koncepcją gniazdek systemu BSD4.3.

6 Procesy w OS–9

Proces składa się z programu, który został uruchomiony i dotąd się nie zakończył, oraz obszaru danych i struktury w pamięci używanej przez system do obsługi tego procesu. Struktura ta jest deskryptorem procesu (*process descriptor*). Ponieważ w OS–9 regułą jest używanie czystego kodu jeden program może mieć równocześnie wiele wcieleń - procesów używających tego samego programu, ale oddzielnych obszarów danych. Każdy z procesów jest obsługiwany przez kernel dzięki oddzielnemu deskryptorowi, który zawiera informacje o procesie.
Proces (zadanie) jest tworzony przez funkcję systemową F$Fork, która zwraca numer zwany identyfikatorem procesu (ID) jednoznacznie określający dany proces. Jest on używany przez inne funkcje systemowe do komunikowania się z procesem i zmieniania jego zachowania. Numer jest zwalniany po zakończeniu procesu i może być przydzielony innemu zadaniu. Nie jest jednak możliwe, by dwa procesy miały ten sam numer równocześnie. Procesu o numerze 0 nie ma, a numer jeden posiada proces systemowy.
Każdy proces ma priorytet, który jest mu przydzielany przy uruchamianiu (*fork*). Zazwyczaj jest on taki sam, jak procesu tworzącego nowy proces (procesu rodzicielskiego - *parent*). Może on być zmieniony funkcją F$Prior i stanowi podstawowy mechanizm określania przydziału czasu procesora dla procesu. ()
W typowych aplikacjach czasu rzeczywistego wiele procesów pracuje równoczesnie. Muszą one wymieniać między sobą dane, informacje sterujące i synchronizować się wzajemnie.
W danej chwili proces może być w jednym z kilku stanów:

- **Active** - żądający czasu procesora
- **Waiting** - czekający na zakończenie procesu potomnego
- **Sleeping** - czekający przez określony czas, lub na zewnętrzne zdarzenie, lub na sygnał od innego procesu
- **Waiting for event** - czekający na zdarzenie wewnętrzne (semafor)
- **Debugged** - czekający na zezwolenie kontynuacji od procesu rodzicielskiego (debuggera)
- **Dead** - czekający na odebranie statusu wyjściowego przez proces rodzicielski (po zakończeniu)

Czas procesora jest dzielony pomiędzy procesy aktywne w jednostkach zwanych *time slices*. Osiąga się to dzięki cyklicznym przerwaniom zegarowym zwanym *ticks*. Zazwyczaj w OS–9 segment składa się z dwóch tików po 10 ms każdy.
Wykonywanie kolejno wszystkich procesów w krótkich segmentach czasu sprawia wrażenie równoległej pracy kilku programów. Długość segmentu dobiera się kompromisowo - dostatecznie małą, by stwarzać wrażenie równoległości, dostatecznie dużą, by nie tracić zbyt wiele czasu na przełączanie procesów. Dzięki przydzielaniu czasu tylko procesom aktywnym unika się marnowania go. System podziału czasu w OS–9 zapewnia równomierne przydzielanie czasu wszystkim procesom (z uwzględnieniem priorytetów), o ile programista nie zażąda inaczej. Specjalne cechy systemu podziału czasu pozwalają wpływać na sposób przydziału czasu, co jest szczególne ważne w systemach czasu rzeczywistego.
Komenda `procs` pozwala obejrzeć procesy istniejące w systemie wraz z ich pewnymi cechami.
Proces jest uruchamiany przez funkcję F$Fork wywołaną z innego procesu, lub części systemu operacyjnego. Zakończenie procesu następuje w wyniku wykonania przez program funkcji F$Exit, lub

stwierdzenia błędu fatalnego (np. nieprzewidziany sygnał). Proces rodzicielski może przekazać wskaźnik obszaru parametrów, który będzie skopiowany do pamięci statycznej procesu potomnego. Może też zadać priorytet, zwiększyć przydział pamięci w stosunku do minimum zadeklarowanego w nagłówku modułu programu.
Proces może również przekształcić się w inny, wykonujący inny program. Służy do tego funkcja F$Chain. Jest ona podobna do F$Fork, lecz proces rodzicielski zostaje zakończony, a jego deskryptor jest użyty do uruchomienia nowego procesu, który ma ten sam identyfikator.
Kernel zarządza procesami dzięki deskryptorom procesów (Dodatek D). Są one zebrane w tablicy deskryptorów procesów, która zawiera adresy wszystkich deskryptorów. Identyfikator procesu (ID) jest indeksem w tej tablicy. Zerowe adresy w tablicy oznaczają brak procesu o tym ID. Tablica ta jest w miarę potrzeby powiększana (przepisywana do dwukrotnie większego obszaru).
Każdy proces ma początkowo rodzica - proces, który go uruchomił. Może on być osierocony przez zakończenie procesu rodzicielskiego przed zakończeniem potomka.
Proces, który kończy się nieosierocony, zwraca swój status wyjściowy procesowi rodzicielskiemu. Dzieje się to przez wykonanie w procesie rodzicielskim funkcji F$Wait. Zakończony (*dead*) proces może więc pozostawać w tablicy do czasu wykonania tej funkcji, lub zakończenia rodzica. Pozostawiany jest w tym przypadku tylko deskryptor, program i dane są zwalniane przez jądro OS–9. Jest to niezbędne dla zapewnienia poprawnej współpracy procesow w systemach wielozadaniowych, gdzie proces rodzicielski musi często znać status końcowy potomka. Z drugiej strony - trzeba sobie z tego zdawać sprawę, by nie być zaskoczonym pozorną ńieśmiertelnoscią"jakiegoś procesu.
Chcąc uniknąć oczekiwania na zakończenie procesu potomnego należy uruchomić proces pośredni, który z kolei uruchomi właściwy proces i zakończy się. W ten sposób właściwy proces pozostanie osierocony, bo nie ma mechanizmu ądopcji przez dziadków".
Procesy są wykonywane w trybie użytkownika (*user*) procesora 680x0. Elementy systemu pracują w trybie nadzorcy (*supervisor*). W trybie *user* niektóre operacje (np. maskowanie przerwań) są nielegalne. Procesy mogą być przerywane w dowolnym momencie przez przekazanie czasu procesora innym procesom. W trybie nadzorcy podział czasu jest zawieszony, co zapewnia nieprzerwaną pracę funkcji systemowych.
Można samodzielnie wprowadzić funkcję systemową (w sterowniku urządzenia, lub module dodatkowym jądra), która pozwala maskować i odmaskowywać przerwania itp. Poza tym w systemie podziału czasu istnieją mechanizmy pozwalające na modyfikowanie jego działania, lub wręcz wyłączanie.
Inną metodą jest uruchomienie procesu w trybie systemowym (*system state process*). Osiąga się to przez ustawienie flagi *supervisor state* w atrybutach modułu programowego (w nagłówku). Przy uruchamianiu takiego programu system wykonuje proces w trybie nadzorcy 680x0.
UWAGA: Należy pamiętać, że niektóre funkcje systemowe działają wtedy inaczej !

7 Podział czasu w OS–9

Wielozadaniowść jest bardzo ważną cechą systemów czasu rzeczywistego i systemów wielodostępnych. Zwykle te dwa zastosowania wymagają drastycznie różnych systemów operacyjnych, ale obsługa wielozadaniowości w OS–9 pozwala je pogodzić bez rezygnowania z wymagań. Algorytm kolejkowania procesów firmy Microware jest prosty, elegancki, szybki i łatwy w obsłudze. Podstawową metodą jest karuzela (*round robin*), ale dostępne opcje pozwalają zmianę jego zachowania, w krańcowym przypadku do algorytmu czysto priorytetowego (jak w prostych systemach czasu rzeczywistego).
Funkcje obsługi procesów mieszczą się w jądrze (*kernel*). Ich celem jest udostępnianie wielu procesom czasu procesora. Jądro obsługuje listę procesów żądających czasu procesora (*active queue*). Każdy z procesów z tej kolejki otrzymuje czas procesora, o ile nie działa któryś z mechanizmów wywłaszczających. Proces bieżący (*current process*) nie znajduje się w kolejce. Zmienna globalna D_Proc wskazuje na jego deskryptor. W kolejce czekają te procesy, które żądają czasu procesora, ale go aktualnie nie dostały. Aby otrzymac czas procesora, proces musi zostać umieszczony w kolejce *active*. Usunięcie z tej kolejki może się dokonać wyłącznie przez wykonanie w procesie funkcji systemowej takiej jak czekanie na zdarzenie, uśpienie (*sleep*), lub zakończenie procesu przez kernel (*kill*). Przy uruchamianiu (*fork*) proces jest umieszczany w kolejce procesów aktywnych. przy pomocy funkcji F$AProc.
Kolejkowanie polega na umieszczaniu procesu w kolejce procesów aktywnych w zależności od priorytetu. Proces, który ma być uruchomiony jako następny znajduje się na czele kolejki. Dzięki temu przełączanie procesów jest szybkie. Usuwa się pierwszy proces z kolejki i czyni procesem bieżącym (funkcja F$NProc). Funkcja ta jest wywoływana tylko wtedy, gdy proces bieżący wywołał funkcję, która zawiesza jego pracę (*sleep*, *die*), lub przy powrocie do trybu użytkownika, gdy deskryptor procesu jest oznaczony jako przeterminowany (*timed out*). W tym ostatnim przypadku następuje umieszczenie tego procesu w kolejce *active*, jak przy uruchamianiu i dopiero potem - pobranie pierwszego w kolejce jako aktualnego. Możliwe jest więc wielokrotne kolejne przydzielenie odcinka czasu temu samemu procesowi. Jesli kolejka jest pusta, *time out* jest ignorowany dla oszczędzenia czasu przełączania procesów.
Procedura czasowa wywoływana przy każdym tiku zegara zmniejsza zmienną D_Slice, która oznacza ilosć tików do końca odcinka czasu (*slice*). Po jej wyzerowaniu, jądro ustawia flagę *timed out* w polu P$State deskryptora procesu bieżącego. Przełączenie odbywa się w chwili, gdy kernel ma wrócić do trybu użytkownika i zauważa *time out*. Wtedy wykonuje się fukcję F$AProc, by wstawić bieżący proces do kolejki, a następnie F$NProc, by uruchomić następny. Dzięki temu funkcje systemowe (wykonywane w trybie nadzorcy - *supervisor*) są niepodzielne (o ile nie wykonają *sleep*).
F$NProc usuwa proces z początku kolejki i czyni go bieżącym (D_Proc). Ustatwia D_Slice na wartość pobraną z D_TSlice (tiki na odcinek). Jeśli nie ma żadnych procesów do wykonania, procesor wykonuje instrukcję *stop*, co zatrzymuje jego pracę aż do przerwania. Zazwyczaj D_TSlice wynosi 2, by uniknąć przydzielania zbyt małych odcinków czasu (system nie rozróżnia jednostek czasu poniżej *tick*). Zmienne globalne podane są w Dodatku B.
Przydział czasu jest zarządzany przez kombinację czterech reguł:

- karuzela (*round robin*). Czas procesora jest dzielony na odcinki, każdy z procesów aktywnych po kolei dostaje swój odcinek. Priorytety decydują o częstotliwości przydzielania odcinków.
- próg priorytetu (*minimum process priority*). Procesy o priorytecie niższym od progowego nie otrzymują odcinków czasu.
- wywłaszczanie priorytetowe. Proces o najwyższym priorytecie pozostaje bieżącym aż do własnej

rezygnacji (*sleep*), lub uruchomienia procesu o wyższym priorytecie. Dotyczy to tylko procesów o priorytecie powyżej progu. Poniżej działa dalej karuzela (o ile chwilowo nie ma procesów aktywnych o priorytecie powyżej progu).

- pojedynczy proces. Programista przejmuje zarządzanie procesami dzięki mechanizmowi wywłaszczania. Jeden z procesów jest wyznaczany jako bieżący i pozostaje nim tak długo, dopóki nie zostanie zastąpiony innym, lub mechanizm ten nie zostanie wyłączony (D_Sieze).

Kernel obsługuje zmienną D_ActAge - wiek kolejki procesów aktywnych. Ma ona poczatkową wartość $7FFF0000 i jest zmniejszana o 1 przy każdym wywołaniu F$AProc. Funkcja ta wylicza stałą kolejkujcą (*scheduling constant*) dla procesu umieszczanego w kolejce. W normalnym przypadku jest ona sumą D_ActAge (po zmniejszeniu) i priorytetu procesu. Deskryptor procesu jest usuwany z kolejki, w której się dotąd znajdował i umieszczany w kolejce *active* bezpośrednio przed procesem o niższych wartosciach kolejkujących (P$Sched). Jesli dodatkowo proces ma wyższy priorytet od bieżącego, to bieżący jest przeterminowywany (*timed out*). Powoduje to uczynienie aktywnym procesu z czoła kolejki natychmiast po zakończeniu pracy w trybie nadzorcy. Jest to bardzo ważne w systemach czasu rzeczywistego.
Algorytm rotacyjny może być zmodyfikowany przez ustalenie progu priorytetu. Jeśli proces o priorytecie niższym niż wartość zmiennej globalnej D_MinPty jest umieszczany w kolejce *active* (F$AProc), to jego stała kolejkowania jest zerowana. W ten sposób trafia on na koniec kolejki. F$NProc sprawdza priorytet procesu, który ma być uruchomiony i oznacza go jako *timed out* w przypadku nieprzekraczania D_MinPty. Dodatkowo uruchamiana jest funkcja F$AProc, by umieścić proces w kolejce *active*, co jak poprzednio wyzeruje P$Sched i ulokuje proces na końcu kolejki. Po ustawieniu D_MinPty wszystkie procesy z kolejki dokończą swe wywołania systemowe i zostaną umieszczone na końcu.
Ustawienie D_MinPty powoduje zablokowanie wykonywania procesów o priorytecie poniżej progu. Przy obniżeniu progu funkcja F$SetSys służąca do zmiany zmiennych globalnych przegląda wszystkie procesy o zerowych polach P$Sched i wywołuje F$AProc umożliwiając uruchomienie procesów, których priorytety znalazły się powyżej nowego progu.
Zmienna D_MaxAge stanowi inny próg. Jesli nie jest zerem, to procesy o priorytetach poniżej progu są obsługiwane karuzelowo, a pozostałe otrzymują stałą kolejkujacą $80000000+P$Prio. W ten sposób zawsze poprzedzają w kolejce inne procesy, a pomiędzy sobą są uporządkowane według priorytetow. Dopóki w kolejce są procesy o priorytecie powyżej progu, procesy mniej ważne nie otrzymują czasu ($7FFF0000+P$Prio nie może przekroczyć $7FFFFFFF). F$SetSys przy zmianie D_MaxAge wywołuje F$AProc, by zaktualizować natychmiast kolejkę procesów.
Mechanizm wyłączania kolejkowania polega na ustawieniu w zmiennej D_Sieze numeru (ID) procesu, który ma być bieżącym. Gdy proces o tym ID jest uaktywniany (F$AProc), otrzymuje stałą $FFFFFFFF, co ustawia go na czele kolejki. Gdy ma dodatkowo wyższy priorytet, niż proces bieżący, to powoduje jego przeterminowanie i sam staje się bieżącym (jak przy innych trybach kolejkowania) dzięki funkcji F$NProc. Po zawieszeniu, lub zakończeniu tego procesu funkcja F$NProc nie uruchomi innych (jak w przypadku pustej kolejki). Programista może więc dowolnie sterować kolejnością wykonywania programów.
W pierwszej kolejności działa modyfikacja D_Sieze, następnie D_MinPty, ostatnia - D_MaxAge.

8 Komunikacja między procesami w OS–9

8.1 Sygnały

Sygnały (*signals*) są przeznaczone do komunikacji pomiędzy niezależnymi, równoległymi procesami. Składają się one z identyfikatora procesu, dla którego są przeznaczone i z własnego numeru. Pewne sygnały są w systemie wstępnie zdefiniowane:

- 0 - kill - bezwzględne zakończenie procesu;
- 1 - wake-up - wznowienie procesu bedącego w stanie sleep;
- 2 - abort - przerwanie procesu ostatnio korzystającego z we/wy na terminal (Ctrl-E) - najczęściej exit(2);
- 3 - interrupt - (Ctrl-C) - najczęściej exit(3);
- 4 - hang-up - wysyłany przez SCF przy utracie łączności modemowej;
- 5-255 - zarezerwowane dla dalszych zastosowań systemowych;
- 256-65535 - dla użytkowników.

Proces, który odbierze sygnał inny niż wake-up zostanie zakonczony, o ile nie jest wyposażony w procedurę przechwytywania sygnałów (*kill* jest niemożliwy do przechwycenia, ale może być wysłany tylko przez właściciela procesu, lub super-usera (grupa 0). Następujące funkcje systemowe są przewidziane do obsługi sygnałów:

- F$Send - wysłanie sygnału;
- F$Icpt - zainstalowanie procedury przechwytywania sygnałów;
- F$Sleep - deaktywacja procesu do wyczerpania czasu, lub odebrania sygnału;
- F$SigMask - ustawianie maski sygnałów dla procesu.

Sygnał w OS–9 stanowi bardzo krótką wiadomość przesyłaną z jednego procesu do innego. Jest to mechanizm programowy, w odróżnieniu od przerwań, które są związane ze sprzętem. Podobieństwo polega jedynie na zdolności do asynchronicznego zmieniania przebiegu programu i możliwości maskowania.
Do wysyłania sygnałów służy funkcja F$Send. Podaje się jej numer (ID) procesu - adresata (są też sygnały wysyłane do wszystkich procesow - broadcasts). Proces wysyłający sygnał musi mieć taki sam numer grupy i użytkownika (ID), jak proces - adresat, lub być własnością administratora (grupa 0). Odebranie sygnału działa na proces dwojako:

- uaktywnia go (o ile dotąd nie był w kolejce *active*),
- przy najbliższym uruchomieniu w trybie użytkownika powoduje wykonanie procedury obsługi sygnałów (*signal handler function*).

Warto zauważyć, że w chwili wysyłania sygnału proces - adresat nie jest wykonywany w trybie użytkownika. Jeśli nawet proces wysyła sygnał do siebie samego, to w chwili wykonywania F$Send jest w trybie nadzorcy! W związku z tym, signal handler będzie uruchomiony najwcześniej, jak to możliwe.

Sygnał jest jedynym asynchronicznym sposobem komunikacji między procesami w OS–9. Ponadto, pozwala on oczekiwać na dostępność zasobów bez sprawdzania (*polling*), gdyż odebranie sygnału uaktywnia proces. Po zakończeniu procedury obsługi sygnałów w procesie, wykonywanie programu wraca do normy. Jeśli proces był w stanie *sleep* lub *wait*, to wykonuje się następne instrukcje po F$Sleep lub F$Wait.
Jeśli proces otrzymuje sygnał w trakcie wykonywania funkcji systemowej, to signal handler nie uruchamia się aż do powrotu do trybu użytkownika. Jeśli był uśpiony w funkcji systemowej (np. w sterowniku urządzenia oczekując na przerwanie), to jest uaktywniany. Trzeba więc sprawdzać, czy powrót z funkcji F$Wait oznacza jej wykonanie, czy odebranie sygnału.
Gdy proces ma podjąć pracę w trybie użytkownika (*user state*), kernel sprawdza, czy jakiś sygnał nie oczekuje na obsługę. O ile w procesie została zainstalowana funkcja obsługi sygnałów, to kernel buduje na stosie dodatkową ramkę i w ten sposób powoduje zmianę przebiegu programu tak, jakby program sam wywołał tę funkcję. Po jej zakończeniu następuje powrót do programu (jak z podprogramu).
Sygnały otrzymywane przez proces są kolejkowane (może ich być wiele pomiędzy kolejnymi odcinkami czasu pracy procesu). Pole P$Signal w deskryptorze procesu zawiera kod ostatnio otrzymanego sygnału, lub zero (jeśli nie ma sygnałów). Funkcja obsługi sygnałów powinna wracać przez F$RTE, co powoduje sprawdzenie obecności sygnałów i obsłużenie kolejno wszystkich oczekujących w kolejce.
UWAGA: sygnały S$Wake i S$Kill nie są kolejkowane!
Do instalowania pocedury obsługi sygnałów służy funkcja F$Icpt. Powinna ona być wywoływana na początku programu. Proces, który odbierze sygnał nie mając zainstalowanej procedury jego obsługi zostanie zakończony.
W bibliotece **C** jest funkcja `intercept()`, która służy do tego samego celu. Zapewnia ona dodatkowo powrót z funkcji obsługi sygnałów (która jest definiowana przez programistę jako zwykła funkcja w **C**) poprzez F$RTE. Jeśli jako jej argument, zamiast wskaźnika na funkcję obsługi sygnałów podamy NULL, to powoduje ona usunięcie signal handler-a.
Sygnały o kodach 2 do 31 sa źabójcze"(sterowniki urządzeń w stanie *sleep* przerywają pracę i zwracają błąd w przypadku uaktywnienia przez taki sygnał). Mogą one być zamaskowane przez ustawienie odpowiednich bitów w słowie P$SigMask deskryptora procesu. Nie są wtedy kolejkowane i nie uaktywniają procesu.
Sygnał o kodzie 0 (S$Kill) nie jest kolejkowany. Powoduje natomiast ustawienie flagi *condemned* w polu P$State deskryptora procesu. Przy próbie uruchomienia w trybie użytkownika powoduje to bezwarunkowe zakończenie procesu.
UWAGA: funkcja `kill()` w bibliotece **C** służy (podobnie jak w unix-ie) do wysyłania dowolnego sygnału.
Sygnał o kodzie 1 (S$Wake) służy do uaktywniania uśpionego sterownika urządzenia przez procedurę obsługi przerwania, nie powinien być używany w trybie użytkownika. Nie jest on kolejkowany i nie uruchamia procedury obsługi sygnałow, nie może więc też źabić"procesu, który nie potrafi go obsłużyć.
Funkcja F$SigMask służy do odraczania obsługi sygnałów. Zwiększa, zmniejsza, lub zeruje pole maski sygnałów (P$SigLvl) w deskryptorze procesu. Jeśli pole to nie jest zerowe, to sygnały są kolejkowane i powodują wprawdzie uaktywnienie procesu, ale nie uruchamiają procedury ich obsługi aż do wyzerowania tego pola. Parametr F$SigMask może być równy -1, 0, lub 1, co powoduje odpowiednio zmniejszenie (ale nie przy zerowej wartości), wyzerowanie, lub zwiększenie (maksymalnie do 255) P$SigLvl. Kernel zwiększa maskę sygnałów na czas wykonywania ich obsługi zmniejsza

```
/*********************************************************************

   Przyklad obslugi sygnalow

*********************************************************************/


int got10,          /* flaga sygnalu 10 */
    got20,          /* flaga sygnalu 20 */
    illegal;        /* sygnal nielegalny */

sighdlr(s)          /* funkcja obsugi sygnalow */
int s;              /* kod odebranego sygnalu */

{
    switch(s){      /* rejestracja odebranego sygnalu */

        case 10: got10=TRUE;break;
        case 20: got20=TRUE;break;
        default: illegal=s;break;
    }
}

main()
{
    while(TRUE){
        sigmask(1);                 /* zamaskowanie sygnalow */
        if(got10||got20||invalid){
            sigmask(0);             /* odmaskowanie sygnalow */
            if(got10) printf(żeceived signal 10\n");
            else if(got20) printf(żeceived signal 20\n");
            else exit(_errmsg(1,"invalid signal %d \n",illegal);
        }
        else tsleep(0);             /* uspienie na nieograniczony czas */
    }
}

/*********************************************************************/
```

przy F$RTE, co chroni przed rekurencją. Jeśli w funkcji obsługi P$SigLvl jest zwiększone, to po powrocie z niej sygnały pozostają zamaskowane.
Funkcje F$Sleep i F$Wait wywołane w trybie użytkownika zerują P$SigLvl. Typowa sekwencja synchronizacji procesówb wygląda następująco:

- zamaskowanie sygnałów (F$SigMask);
- sprawdzenie flagi ustawianej przez funkcję obsługującą sygnały;
- uśpienie procesu na nieokreślony czas.

W tym przykładzie sygnały są maskowane przed sprawdzeniem flag (faktu odebrania sygnału). W ten sposób ich odmaskowanie powodowane jest dopiero funkcją F$Sleep (w `tsleep()`). Flaga nie może się więc zmienić podczas jej sprawdzania (unikamy hazardu). Jeśli przy odmaskowywaniu przerwań

okaże się, że jakiś sygnał oczekuje na obsługę (*pending*), to F$Sleep kończy się i proces kontynuuje pracę, w przeciwnym przypadku jest umieszczany w kolejce procesów uśpionych (*sleeping*).

8.2 Łącza

Łącze (*pipe*) jest buforem typu FIFO (*first-in-first-out*). Jeden lub więcej procesów może pisać do tego bufora przy pomocy standardowych funkcji wyjścia, jeden lub więcej procesów może zeń czytać przy pomocy standardowych funkcji wejścia. Dane stanowią strumień bajtów, są czytane w kolejności, w jakiej nastąpił zapis. Każdy odczytany bajt jest usuwany z bufora (łącza), gdy wszystkie wpisane bajty są odczytane, bufor jest pusty. Dalszy odczyt jest możliwy dopiero po wpisaniu dalszych danych. Łącze ma ograniczoną pojemność. Gdy różnica pomiędzy ilościami wpisanych i odczytanych bajtów ją przekracza, dalszy wpis jest niemożliwy. Zazwyczaj mamy do czynienia z jednym procesem czytajacym z łącza, mimo wielu procesów piszacych doń.
Łącza w OS–9 są obsługiwane przez moduł obsługi plików **pipeman**. Jako bufory w pamięci nie są zależne od sprzętu i nie wymagają sterowników urządzeń. Ze względu na organizację systemu WE/WY używany jest pusty sterownik **null**, którego funkcje są puste. Deskryptorem urządzenia jest **pipe**. Łącza tworzy się zwykłymi funkcjami tworzenia ścieżki dostępu do urzadzenia **/pipe**. Mogą być one tworzone jako nazwane, lub nienazwane.
Łącze nienazwane jest tworzone przez:

```
path=create("//pipe",S_IREAD|S_IWRITE,S_IREAD|S_IWRITE);

path=open("//pipe",S_IREAD|S_IWRITE);
```

Łącze nazwane tworzy się podając drugi element nazwy ścieżki w funkcji `create()`:

```
path=create("//pipe//blabla",S_IREAD|S_IWRITE,S_IREAD|S_IWRITE);
```

podobnie jak w przypadku plików dyskowych (uwaga: nie ma podkartotek!). Utworzone łącze nazwane może być otwierane przy pomocy swej nazwy:

```
path=open("pipe//blabla",S_IREAD|S_IWRITE);
```

Domyślna wielkość bufora wynosi 90 bajtow, lecz można wyspecyfikować większą, podając opcję S_ISIZE:

```
path=create("//pipe//blabla",S_IREAD|S_IWRITE|S_ISIZE,S_IREAD|S_IWRITE,1000);
```

Utworzony bufor ma wielkość nie mniejszą od zadanego parametru, będącą wielokrotnością 16 (tu: 1008).
Wszystkie ścieżki otwarte z tą samą nazwą działają na wspólnym buforze, więc różne procesy mogą mieć dostęp do tego samego łącza. Inaczej jest przy łączach nienazwanych. Jedynym sposobem dostepu do tego samego łącza z kilku procesów jest dziedziczenie ścieżki do niego na skutek posiadania wspólnego przodka, który utworzył tę ścieżkę. Nie da się otworzyć wielu ścieżek do tego samego łącza nienazwanego, a tylko wiele kopii tej samej ścieżki.
Łącza mogą służyć zarowno do przesyłania danych pomiędzy procesami, jak i do synchronizowania procesów. Proces czytający z łącza jest zawieszany po jego opróżnieniu (chyba że nie jest otwarta żadna ścieżka w trybie zapisu, kiedy to zwracany jest błąd ”koniec pliku”), aż do pojawienia się nowych danych w buforze. Proces piszący do łącza jest zawieszany po jego zapelnieniu (o ile są

```
/*****************************************************************

   Przyklad sprzegania procesow przez lacza

*****************************************************************/

copy_in=dup(0);     /* kopia stdin */
copu_out=dup(1);    /* kopia stdout */
close(1);           /* zamkniecie stdout */

path=create("//pipe",S_IREAD|S_IWRITE,S_IREAD|S_IWRITE);
        /* lacze bedzie mialo numer sciezki 1 (pierwsza wolna) */

pid_1=os9exec(os9fork,"prg1",args1,environ,0,0);
        /* utworzenie pierwszego procesu (dziedziczy trzy sciezki) */

close(0);           /* zamkniecie stdin */
dup(1);             /* kopia sciezki do lacza z numerem 0 (stdin) */
close(1);           /* zamkniecie lacza (stdout) */
dup(copy_out);      /* odtworzenie oryginalnego stdout z kopii */

pid_2=os9exec(os9fork,"prg2",args2,environ,0,0);
        /* utworzenie drugiego procesu (dziedziczy trzy sciezki) */

close(0);           /* zamkniecie lacza (stdin) */
dup(copy_out);      /* odtworzenie stdin z kopii */
close(copy_in);     /* zamkniecie kopii stdin */
close(copy_out);    /* zamkniecie kopii stdout */

/*****************************************************************/
```

otwarte ścieżki w trybie odczytu, bowiem w przeciwnym przypadku zwracany jest błąd E_WRITE, by zapobiec blokadzie).
Łącza nazwane mogą istnieć mimo braku otwartych ścieżek, o ile zawierają dane. Proces piszący do zapełnionego łącza nazwanego jest zawieszany w każdym przypadku, w nadziei pojawienia się nowych otwartych ścieżek. Łącze nazwane jest automatycznie usuwane gdy jest puste i nie ma otwartych ścieżek. Można je także usunąć komendą:

```
> del //pipe//blabla
```

Można rownież odczytać katalog (jednopoziomowy!) urządzenia **/pipe**:

```
> dir //pipe
```

Analogicznie jak w urządzeniach typu SCF (terminale), łącza pozwalają pytać o status i wysyłać sygnały w chwili pojawienia się danych (funkcje SS_Ready i SS_SSig typu *set status*).
Shell używa łącz nienazwanych do tworzenia potoków:

```
> dir -ud | grep -v "//$" | del -z
```

```
Element tablicy zdarzen

--- --------  --------------

$00 word (2)  Event ID      - unikalny numer zdarzenia w systemie;
$02 byte (12) Event Name    - max. 11-znakowa nazwa zdarzenia;
$0e long (4)  Event Value   - 4-bajtowa wartosc zdarzenia;
$12 word (2)  Wait Incr.    - przyrost wartosci przy budzeniu;
$14 word (2)  Signal Incr.  - automatyczny przyrost wartosci;
$16 word (2)  Link Count    - ilosc uzyc zdarzenia;
$18 long (4)  Next Event    - wskaznik do nastepnego zdarzenia;
$1c long (4)  Prev. Event   - wskaznik do poprzedniego zdarzenia;
```

8.3 Zdarzenia (*events*)

Przy pomocy zdarzenia (*event*) można w OS–9 synchronizować procesy i przekazywać niewielkie ilości danych (wartość zdarzenia). W odróżnieniu od sygnałów, zdarzenia nie mogą zmieniać przebiegu programu (jak przerwania i sygnały), nie mogą uaktywniać procesów nie oczekujących na nie. Mają jednak liczne zalety wynikające z elastyczności (mogą przyjmować wiele wartości, są publiczne, są nieulotne). Definicje związane ze zdarzeniami znajdują się w Dodatku E.
Zdarzenie jest tworzone przez funkcję systemową, której dostarcza się nazwę i wartość początkową. Jądro umieszcza zdarzenie w tablicy po uprzednim sprawdzeniu, że nie ma innego o tej samej nazwie (do 12 znaków). Zdarzeniu jest przyporzadkowywany numer (ID) będący długim słowem, którego bardziej znaczącą połową jest numer kolejny z D_EvID (po zwiększeniu), a mniej znaczącą - indeks zdarzenia w tablicy. Tablica zdarzeń jest dynamicznie rozszerzalna, nie ma ograniczenia na ilość zdarzeń. Funkcja zwraca ID, lub błąd (gdy zdarzenie o podanej nazwie już istnieje).
Element tablicy zdarzeń zawiera licznik dowiązań (użyć), inicjowany na 1, wielkość przyrostu automatycznego (*signal increment*), i wielkość przyrostu przy budzeniu (*wakeup increment*). Wartość zdarzenia jest długą liczbą ze znakiem (*long*), a przyrosty są krótkimi liczbami ze znakiem (*short*). przyrost automatyczny (*signal increment*) jest dodawany do wartości zdarzenia przy wywołaniu funkcji *signal event* (nie ma ona nic wspólnego z sygnałami !). Drugi z przyrostów jest dodawany zawsze przy budzeniu procesu ze stanu oczekiwania na zdarzenie (na skutek zmiany wartości zdarzenia).
Po utworzeniu zdarzenia inne procesy mogą się doń przyłączyć (*link to event*) i uzyskać ID po podaniu nazwy (zwiększa to licznik użyć zdarzenia). Po zakończeniu korzystania ze zdarzenia proces może je zwolnić (*unlink from event*). Zdarzenie, którego licznik użyć (*link count*) spada do zera może zostać skasowane funkcją *delete event*. Miejsce w tablicy zostaje wtedy zwolnione. Przy kasowaniu zdarzenia wszystkie oczekujące nań procesy są budzone z błędem E$EvntID - niewłaściwy identyfikator zdarzenia.
UWAGA: Żadne zdarzenia nie są automatycznie usuwane !
Proces znający ID zdarzenia może:

- odczytać jego wartość
- zmienić tę wartość
- czekać na wpadnięcie tej wartości do określonego przedziału

Przy każdej zmianie wartości zdarzenia jądro sprawdza, czy nowa wartość pasuje do przedziałów zadeklarowanych przez procesy oczekujące na to zdarzenie. W tym przypadku jądro budzi proces zwracając wartość zdarzenia, a następnie modyfikuje tę wartość o *wakeup increment*.

Wartość zdarzenia może zostać zmieniona przez:

- ustawienie bezwzględnej wartości (set absolute)
- dodanie (ze znakiem) wartości (set relative)
- dodanie przyrostu automatycznego (signal increment)
- chwilowe ustawienie bezwzględnej wartości (pulse event)

W tym ostatnim przypadku jądro nadaje nową wartość zdarzeniu tylko w celu obudzenia czekających procesow (o ile są), po czym przywraca pierwotną wartość.
Proces może czekać na zdarzenie na dwa sposoby - bezwzględny i względny. W pierwszym przypadku będzie obudzony, gdy wartość zdarzenia znajdzie się pomiędzy minimalnym i maksymalnym ograniczeniem zadanym w wywołaniu. Drugi wariant dotyczy zakresów podawanych względem aktualnej wartości zdarzenia. Jeśli funkcja czekania na zdarzenie jest wywołana w procesie w chwili, gdy warunek jest spełniony, to jest ona natychmiast kończona (przyrost *wakeup* jest dodawany !).
Jądro systemu zawiera jedną funkcję F$Event pozwalającą operować zdarzeniami. Odpowiednia akcja jest powodowana dzięki kodom operacji:

```
Ev$Link     - uzycie zdarzenia (wedlug nazwy);
Ev$UnLnk    - zwolnienie zdarzenia;
Ev$Creat    - utworzenie zdarzenia;
Ev$Delet    - usuniecie istniejacego zdarzenia;
Ev$Wait     - oczekiwanie na zdarzenie;
Ev$WaitR    - oczekiwanie na wzgledna wartosc zdarzenia;
Ev$Read     - odczytanie wartosci licznika bez oczekiwania;
Ev$Info     - odczytanie parametrow zdarzenia;
Ev$Pulse    - sygnalizacja wystapienia zdarzenia;
Ev$Signl    - sygnalizacja wystapienia zdarzenia;
Ev$Set      - ustawienie licznika i sygnalizacja zdarzenia;
Ev$SetR     - wzgledne ustawienie licznika i sygnalizacja;
```

Dokładniejszy opis powyższych funkcji można znale'zć w [10] (OS–9 System Calls). W języku **C** dostępne są funkcje realizujące analogiczne zadania (`_ev_link()`, `_ev_unlink()`, `_ev_creat()`, `_ev_delete()`, `_ev_wait()`, `_ev_waitr()`, `_ev_read()`, `_ev_info()`, `_ev_pulse()`, `_ev_signal()`, `_ev_set()`, `_ev_setr()`). Ich opisy można znaleźć w [8].
Operacja Ev$Wait powoduje zawieszenie procesu w przypadku, gdy wartość zdarzenia (licznika) nie mieści się w podanym przedziale. Proces jest umieszczany w kolejce i oczekuje na zmianę tej sytuacji. Gdy wartość zdarzenia mieści się w przedziale, jest ona zmieniana o Wait_increment i proces jest kontynuowany.
Operacja Ev$Signal powoduje zmianę wartości zdarzenia o Signal_increment i uaktywnia (budzi) te z procesów oczekujących na to zdarzenie, dla których aktualna (zmieniona) wartość zdarzenia jest odpowiednia (mieści się w przedziale).
Najprostszą formą synchronizacji procesów moze byc *pulsing*. Jeden proces tworzy zdarzenie, drugi się przyłącza. Wartość początkowa zdarzenia - 0, przyrost przy budzeniu - 0. Proces oczekujacy ustawia w wywołaniu Ev$Wait przedział od 1 do 1, a proces synchronizujący wywołuje Ev$Pulse z

wartością 1. Powoduje to zbudzenie procesu oczekującego na zdarzenie bez zmieniania wartości zdarzenia (pozostaje ona zerowa). Może się tu jednak zdarzyć, że proces zacznie oczekiwać na zdarzenie już po jego wystąpieniu, co spowoduje blokadę (zdarzenia nie są buforowane).
Bezpieczna forma synchronizacji procesów wymaga wzajemnych potwierdzeń (*handshake*). Zdarzenie jest tworzone jak poprzednio, proces oczekujący ustawia taki sam przedział, ale proces synchronizujący ustawia wartość zdarzenia na 1 (Ev$Set) i czeka na jego wyzerowanie (Ev$Wait). Proces synchronizowany (po wykonaniu odpowiednich operacji - na przykład przeczytaniu danych umieszczonych w pamięci przez proces synchronizujący) ustawia wartość zdarzenia na 0 i budzi w ten sposób proces synchronizujący.
Łatwo to uogólnić na przypadek obsługi bufora cyklicznego w module danych. Proces wpisujący dane do bufora powinien czekać w przypadku, gdy bufor jest pełny, proces odczytujący - gdy bufor jest pusty. Zdarzenie stanowi licznik znaków w buforze. Proces wpisujący ustawia przedział dla Ev$Wait od 0 do N-1 (gdzie N jest długością bufora), a proces czytający - od 1 do N. Proces wpisujący zwiększa zdarzenie o jeden przy każdym wpisie (Ev$SetR z argumentem 1), a proces czytający zmniejsza o 1 przy każdym odczycie (argument -1).
Zdarzenia można również użyć do synchronizacji przesyłania danych do wielu procesów. W tym celu proces nadający tworzy zdarzenie z przyrostem automatycznym równym -1 i przyrostem przy budzeniu równym 1. Po wpisaniu danych do modułu ustawia wartość zdarzenia na ustaloną dużą liczbę M. Procesy czekające na dane mają ustawiony dolny prog Ev$Wait na M, są więc budzone. Każde obudzenie procesu zwiększa wartość zdarzenia o 1, więc wartość zdarzenia wzrośnie w wyniku wykonania tej operacji o ilość obudzonych procesow. Każdy z procesow czytających wywołuje funkcję Ev$Signal, ktora zmniejsza wartość zdarzenia o 1. Proces nadający może teraz zmniejszyć wartość zdarzenia o M i czekać na jego powrót do wartości 0, co nastąpi po przeczytaniu wpisanych danych przez wszystkie procesy czytające. To ostatnie obudzenie zwiększy wprawdzie wartość zdarzenia do 1, ale przy progu M znacznie większym nie spowoduje to obudzenia procesów czytających. Procesy czytające muszą być gotowe do odczytu danych (być w funkcji Ev$Wait) w chwili ustawiania zdarzenia na M przez proces piszący, by nie stracić danych.
Użycie zdarzenia jako semafora regulującego dostęp do niepodzielnego zasobu polega na zmuszeniu procesu zgłaszającego żadanie do poczekania na zwolnienie zasobu. Zdarzenie tworzy się z wartością 0, przyrostem przy budzeniu 1 i przyrostem sygnałowym (automatycznym) -1. Proces żądający zasobu czeka na zerową wartość zdarzenia. Przyrost przy budzeniu powoduje ustawienie wartości 1, blokując dostęp dla innych, podobnych procesów. Po zakończeniu korzystania z zasobu proces zmniejsza wartość zdarzenia (Ev$Signal), co powoduje obudzenie kolejnego z oczekujących procesów.

```
/*****************************************************************/

    _ev_wait(event_id,0,0);             /* czekanie na zasob */
    us/eresource();                     /* u/zycie zasobu */
    _ev_signal(event_id,0);             /* zwolnienie zasobu */

/*****************************************************************/
```

Przykład użycia zdarzenia do przydziału zasobów (N drukarek).
Tworzymy zdarzenie o początkowej wartości N (wszystkie drukarki dostępne). Wait_increment ustalamy na -1 (zajęcie drukarki przez jakiś proces), a Signal_increment na 1 (zwolnienie drukarki).
Proces chcący korzystać z drukarki musi czekać (Ev$Wait) na wartość zdarzenia w przedziale [1,N].

```
/*********************************************************************/

Inicjacja zdarzenia:
--------------------

   q event_id =_ev_creat(0,1,-1,źd");       /* value = 0
                                            wait_inc = 1
                                            signal_inc = -1
                                            name = źd" */

Proces piszacy:
---------------

    new_data();                         /* wpis danych */
    _ev_set(event_id,M,EV_ALLPROCS);    /* budzenie proc. czytajacych */
    _ev_setr(event_id,-1*M,0);
    _ev_wait(event_id,0,0);             /* czekanie na zero */


Proces czytajacy:
-----------------

    _ev_wait(event_id,M,M+2000);        /* czekanie na wartosc M */
    tak/edata();                        /* czytanie danych */
    _ev_signal(event_id,0);             /* zmniejszenie zdarzenia o 1 */

/*********************************************************************/
```

Dzięki temu, póki są jakieś wolne drukarki, procesy są kontynuowane, a wartość zdarzenia odzwierciedla ilość wolnych drukarek. Po zakończeniu drukowania proces zwalnia drukarkę i sygnalizuje to (Ev$Signal). Ilość wolnych drukarek (wartość zdarzenia) jest zwiększana. Jeśli jakiś proces był zatrzymany na Ev$Wait z powodu braku drukarek (wartość zdarzenia poza [1,N]), to po zwróceniu drukarki przez inny proces (*Signal*) zostanie uaktywniony (zbudzony) i on z kolei będzie mógł zająć drukarkę (wartość zdarzenia zostanie zmieniona na skutek dokończenia funkcji Ev$Wait tego procesu).

8.4 Alarmy

Alarmy pozwalają zsynchronizować proces z czasem rzeczywistym odmierzanym przez zegar systemowy (moduł **clock**) w przerwaniach. Alarm instaluje się przy pomocy funkcji F$Alarm. Powoduje ona wysłanie (w określonej chwili) sygnału o określonym kodzie do procesu wywołującego. Są dwa typy alarmów: jednorazowe i cykliczne (okresowe). Alarm jednorazowy powoduje wysłanie sygnału po upływie określonej ilości tików (alarm względny), lub przy określonym stanie zegara (data, czas) - alarm bezwzględny. Alarm cykliczny powtarza wysyłanie sygnału z żądanym okresem aż do skasowania alarmu, lub zakonczenia procesu.
Alarm jednokrotny pozwala zorganizować time-out przy oczekiwaniu na asynchroniczne wydarzenie (np. przysłanie danych do portu). Alarm cykliczny pozwala wykonywać sekwencje instrukcji w odstępach niezależnych od warunków otoczenia (np. zajętość i szybkość procesora).
Można ustawić wiele alarmów równocześnie. F$Alarm zwraca unikalny numer alarmu (ID), który może służyć np. do jego usunięcia (funkcja F$Alarm). Usunięcie alarmu z ID=0 powoduje usunięcie wszystkich alarmów należących do danego procesu (np. przy zakończeniu procesu).
Informacje o alarmach są przechowywane w postaci łańcucha bloków opisu zagnieżdżonego w zmiennych globalnych systemu. Łańcuch ten jest uporządkowany według czasu uaktywnienia alarmu (pierwszy element będzie wykonany najwcześniej). Alarmy są wstawiane w odpowiednie miejsce łańcucha podczas tworzenia, a alarmy cykliczne są ponownie wstawiane po każdym wykonaniu (ze zmodyfikowanym czasem). Data i czas dla alarmów bezwzględnych jest przechowywana w formacie julianskim (dni od 2 stycznia roku -4712, sekundy od północy), więc alarmy te mogą być ustawiane z dokładnością do sekundy. Dla alarmów wzglednych i cyklicznych czas jest ustawiany w tikach i zapamiętywany w bloku opisu alarmu jako suma argumentu i bieżącej wartości zmiennej globalnej D_Ticks, zawierającej ilość tików od startu systemu. Alarmy te mogą więc być ustawiane z dokładnością do tiku (na ogół 10ms). Dla wygody programisty można podać czas w 1/256 sekundy, ustawiając najwyższy (31) bit argumentu. Jądro systemu (*kernel*) przekształca taki argument na tiki, co pozwala uniezależnić program od długości tiku.
Alarmy nie są wykonywane bezpośrednio przez procedurę obsługi przerwania zegarowego. W celu wysłania alarmu budzony jest proces systemowy (o numerze 1, niewidoczny w `procs`). Ma on najwyższy z możliwych priorytetow (65535), więc zostanie uruchomiony przed wszystkimi procesami w trybie użytkownika. Takie rozwiązanie powoduje, że procedura obsługi przerwania zegarowego nie trwa zbyt długo i inne funkcje systemowe mogą być wykonywane, a równocześnie wysłany do procesu sygnał dociera (z punktu widzenia użytkownika) równie szybko.
Proces systemowy po obudzeniu sprawdza każdy alarm w łańcychu, porównuje ustawiony w nim czas z D_Ticks (bieżąca wartość). W przypadku osiągnięcia, lub przekroczenia ustawionego czasu, wysyła ustawiony sygnał i usuwa blok z łańcucha (przy alarmie cyklicznym - dodaje okres alarmu do D_Ticks i ponownie wstawia blok do łańcucha). Po dotarciu do bloku opisu alarmu, którego czas jest większy od D_Ticks modyfikuje D_Elapse, o ile najbliższy alarm ma wystąpić wcześniej.

Zmienna D_Elapse służy do odliczania czasu uśpienia procesu systemowego, jest zmniejszana przez procedurę obsługi przerwania zegarowego (przy wartości 0 proces systemowy jest budzony). W drugiej kolejności przeglądane są alarmy bezwzględne i ich czas jest porównywany z D_Second, a data z D_Julian. W przypadku upłynięcia czasu obsługa jest analogiczna. Inną funkcją procesu systemowego jest budzenie procesów uśpionych na określony czas (F$Sleep). W tym celu przeglądana jest kolejka procesów uśpionych (*sleeping*) i w analogiczny sposob procesy, których czas minął są uaktywniane (F$AProc).
Uwaga:
Zmiana czasu i daty systemowej (F$STime) wpływa wyłącznie na alarmy bezwzględne, ponieważ D_Ticks nie ulega zmianie.
Do obsługi alarmów wykorzystywana jest jedna funkcja F$Alarm z kodami operacji (**funcs.a**), którym w bibliotece **C** odpowiadają oddzielne funkcje:

-	--------	----------	----------
0	A$Delete	alm_delete	skasowanie alarmu o zadanym ID (lub wszystkich, dla ID=0)
1	A$Set	alm_set	ustawienie alarmu wzglednego
2	A$Cycle	alm_cycle	ustawienie alarmu cyklicznego
3	A$AtDate	alm_atdate	ustawienie alarmu bezwzglednego w chwili zadanej w formacie gregorianskim (YYYYMMDD,00HHMMSS)
4	A$AtJul	alm_atjul	ustawienie alarmu bezwzglednego w chwili zadanej w formacie julianskim (dni od 2 stycznia -4712 roku, sekundy od polnocy)

Bibliografia

[1] S. R. Bourne, *The UNIX System*, ISBN 0-201-13791-7, Addison-Wesley Publishing Company, London, 1983.

[2] Paul S. Dayan, *The OS–9 Guru*, ISBN 0 9519228 0 7, Galactic Industrial Limited, Durham, 1992.

[3] Peter Dibble, *OS–9 Insights*, ISBN 0 918035 0105, Microware systems Corporation, Des Moines, 1988.

[4] Marc J. Rochkind, *Programowanie w systemie Unix dla zaawansowanych*, ISBN 83-204-1596-9, Wydawnictwa Naukowo-Techniczne, Warszawa, 1993.

[5] Abraham Silberschatz, James L. Peterson, Peter B. Galvin, *Podstawy systemów operacyjnych*, ISBN 83-204-1625-6, Wydawnictwa Naukowo-Techniczne, Warszawa, 1993.

[6] Peter P. Silvester, *System operacyjny Unix*, ISBN 83-204-1086-x, Wydawnictwa Naukowo-Techniczne, Warszawa, 1990.

[7] *OS–9 Assembler/Linker User's Manual*, ALD-68NA-68-MO, Microware Systems Corporation, Des Moines, 1989

[8] *OS–9 C Compiler User's Manual*, CCC-68-NA-68-MO, Microware Systems Corporation, Des Moines, 1989

[9] *OS–9 Technical I/O Manual*, OIO68NA68MO, Microware Systems Corporation, Des Moines, 1990

[10] *OS–9 Technical Manual*, OST68NA68MO, Microware systems Corporation, Des Moines, 1989.

[11] *The OS–9 Catalog*, OS968NA68SL, Microware systems Corporation, Des Moines, 1990.

[12] *The OS–9 Hardware & Software Sourcebook*, SHD68NA68SL, Microware systems Corporation, Des Moines, 1989.

[13] *Using OS–9 Internet*, INT-68-NA-68-MO, Microware systems Corporation, Des Moines, 1992.

[14] *Using Prfessional OS–9*, OSU68NA68MO, Microware systems Corporation, Des Moines, 1989.

Dodatek A

Definicje modułów pamięciowych (pliki: **/dd/DEFS/module.a**) i **/dd/DEFS/module.h**).

module.a

```
 opt -l
 nam Module Header Formats
*******************************
* Edition History
*  #   Date    Changes Made                                                by
* -- -------- ------------------------------------------------------------ ---
* 00 04-11-83 Converted to 68000 from OS-9 Level II Edition 7.             rfd
* 00 05-24-83 Removed DAT routines with conditional assembly.              rfd
* 00 06-21-83 Changed from "ds" to "do" (define offset).                   rfd
* 01 02-09-84 Converted for linkage use.                                   lac
* 02 03-21-84 Revised Trap definitions, and device type codes.             rfd
* 03 06-13-84 Changed module format.  Old modules incompatable.            rfd
* 04 10-12-84 Modified device descriptor format slightly.                  rfd
* 05 11-01-84 Changed names of "init" offsets and expanded.                rfd
* 06 11-01-84 Added type//language values (from oskdefs).                   rfd
* 07 01-11-85 Added M$Events to configuration module.                      rfd
* 08 06-27-85 Added M$Mode byte in device descriptors.                     rfd
*               ---- OS-9//68k V1.2 released ----
* 09 08-23-85 Added M$BlkSiz and M$MinBlk to "init" module.                rfd
* 10 01-31-86 Removed same.                                                rfd
* 11 07-02-86 Added M$SParam string in config module.                      rfd
* 12 09-04-86 Added MD$MChk header checksum.                               rfd
* 13 09-25-86 Added M$Compat for smoothing upgrade problems.               rfd
*               ---- OS-9//68k V2.0 released ----
* 14 01-30-87 Defined module access permission bits.                       rfd
*               ---- OS-9//68k V2.1 released ----
* 15 07-17-87 Added device type definitions for SBF, CDFM.                 rfd
*               ---- OS-9//68k V2.2 released ----
* 16 03-10-88 Added new device type definitions.                           rfd
* 17 04-28-88 Added M$MemList to init module, and DT_NRF.                  rfd
* 18 06-15-88 Added DT_VFM.                                                rfd
* 19 07-18-88 Added M$IPID.                                                rfd
* 20 09-07-88 Changed DT_VFM to DT_GFM.                                    rfd
*               ---- CD-RTOS V0.97 released ----
*               ---- CD-RTOS V0.99 released ----
* 21 11-02-88 Removed M$MaxMem (obsolete), added M$IRQStk.                 rfd
*               ---- OS-9//68k V2.2 edition #50 released ----
* 22 02-20-89 Added M$ColdTrys.                                            rfd
*               ---- OS-9//68k V2.3 released ----
* 23 03-09-90 Added M$Compat2.                                             wwb
*               ---- CD-RTOS V1.1 released ----
*               ---- OS-9//68k V2.4 released ----
* 24 01-29-91 Updated comments for M$CPUTyp.                               wwb
*               ---- OS-9//68k V2.4 68040 released ----
*               ---- OS-9//68k V2.4.3 released ----
```

```
*
edition set 24

 psect module,0,0,edition,0,0

*******************************
* Module Definitions
M$Rev:  equ 1 Module format revision (for D_Cigar)

*     Universal Module Offsets
 org 0
M$ID:       do.w 1 ID code
M$SysRev:   do.w 1 system revision check value
M$Size:     do.l 1 module size
M$Owner:    do.l 1 owner ID
M$Name:     do.l 1 name offset
M$Accs:     do.w 1 access permissions
M$Type:     do.b 1 type
M$Lang:     do.b 1 language
M$Attr:     do.b 1 attributes
M$Revs:     do.b 1 revision level
M$Edit:     do.w 1 edition number
M$Usage:    do.l 1 comment string offset
M$Symbol:   do.l 1 symbol table offset
            do.b 14 reserved
M$Parity:   do.w 1 header parity
M$IDSize:   equ . module ID size

**     Type-Dependent Module Offsets
* System, Program, Trap Handler, File Manager, Device Driver
M$Exec:     do.l 1 execution entry offset
M$Excpt:    do.l 1 exception entry offset

* Program, Trap Handler, Device Driver
M$Mem:      do.l 1 data area requirement

* Program, Trap Handler
M$Stack:    do.l 1 stack area requirement
M$IData:    do.l 1 initialized data ptr
M$IRefs:    do.l 1 initialized reference lists ptr

* User Trap Handler
M$Init:     do.l 1 initialization execution offset
M$Term:     do.l 1 termination execution offset

*   Device Descriptor Module
```

```
            org M$IDSize
M$Port:     do.l 1 port address
M$Vector:   do.b 1 hardware vector number
M$IRQLvl:   do.b 1 Interrupt hardware priority level
M$Prior:    do.b 1 interrupt (polling table) priority
M$Mode:     do.b 1 device mode capabilities
M$FMgr:     do.w 1 file manager name offset
M$PDev:     do.w 1 device driver name offset
M$DevCon:   do.w 1 offset of device dependent constants
            do.w 4 reserved
M$Opt:      do.w 1 device default option count
M$DTyp:     do.b 1 device type

*   Configuration Module Entry Offsets
 org M$IDSize
            do.l 1 reserved
M$PollSz:   do.w 1 number of entries in interrupt polling table
M$DevCnt:   do.w 1 number of entries in device table
M$Procs:    do.w 1 initial process table size (64)
M$Paths:    do.w 1 initial path table size (64)
M$SParam:   do.w 1 initial startup module parameter string
M$Sysgo:    do.w 1 initial startup module name offset
M$SysDev:   do.w 1 system device name
M$Consol:   do.w 1 standard I//O pathlist name offset
M$Extens:   do.w 1 customization module name offset
M$Clock:    do.w 1 clock module name offset
M$Slice:    do.w 1 clock ticks per time slice
M$IPID:     do.w 1 interprocessor ID
M$Site:     do.l 1 installation site code
M$Instal:   do.w 1 installation name offset
M$CPUTyp:   do.l 1 expected cpu type: 68000//68010//68020//etc.
M$OS9Lvl:   do.b 4 operating system level//version//edition
M$OS9Rev:   do.w 1 OS-9 level//revision string offset
M$SysPri:   do.w 1 initial system priority
M$MinPty:   do.w 1 initial system minimum executable priority
M$MaxAge:   do.w 1 initial system maximum natural age
M$MDirSz:   do.l 1 reserved:    maximum module directory entries allowed
M$Events:   do.w 1 initial system event table size (0)
M$Compat:   do.b 1 compatibility byte #1
M$Compat2:  do.b 1 compatibility byte #2
M$MemList:  do.w 1 offset to memory definitions (if any)
M$IRQStk:   do.w 1 size of IRQ stack (in longwords)
M$ColdTrys: do.w 1 number of retries to attempt if initial chd fails
            do.w 10 reserved

********************************
```

```
* Module Field Definitions

**  ID Field and Format values
M$ID12: equ $4AFC Module Header ID

**  Device Type values
 org 0
DT_SCF:     do.b 1 sequential character file type
DT_RBF:     do.b 1 random block file type
DT_Pipe:    do.b 1 pipe file type
DT_SBF:     do.b 1 sequential block file type
DT_NFM:     do.b 1 network file type
DT_CDFM:    do.b 1 compact disc file type
DT_UCM:     do.b 1 user communications manager
DT_SOCK:    do.b 1 socket communication manager
DT_PTTY:    do.b 1 pseudo-keyboard manager
DT_INET:    do.b 1 internet interface manager
DT_NRF:     do.b 1 non-volatile ram file manager (CD-I variety)
DT_GFM:     do.b 1 graphics file manager

** Access permission bits
* xxxx xxxx xxxx ?ewr   owner permission (same as file system)
* xxxx xxxx ?ewr xxxx   group permissions
* xxxx ?ewr xxxx xxxx   world permissions

**  CRC Result Constant
CRCCon: equ $00800FE3

********************************
* Module Directory Entry Definitions
ModDir: equ 1<<8 module directory format revision (for D_Cigar)

 org 0
MD$MPtr:    do.l 1 module ptr
MD$Group:   do.l 1 module group ptr
MD$Static:  do.l 1 module group memory size
MD$Link:    do.w 1 module link count
MD$MChk:    do.w 1 module header checksum
MD$ESize:   equ . module directory entry size
 ends
 opt l
```

module.h

```
/* OS-9//68k module header definitions */

#define VHPCNT  (sizeof(struct modhcom)-2)  /* sizeof common header */
#define MODSYNC 0x4afc      /* module header sync code */
#define CRCCON  0x800fe3    /* crc polynomial constant */

/* Module access permission values */
#define MP_OWNER_READ   0x0001
#define MP_OWNER_WRITE  0x0002
#define MP_OWNER_EXEC   0x0004
#define MP_GROUP_READ   0x0010
#define MP_GROUP_WRITE  0x0020
#define MP_GROUP_EXEC   0x0040
#define MP_WORLD_READ   0x0100
#define MP_WORLD_WRITE  0x0200
#define MP_WORLD_EXEC   0x0400
#define MP_OWNER_MASK   0x000f
#define MP_GROUP_MASK   0x00f0
#define MP_WORLD_MASK   0x0f00
#define MP_SYSTM_MASK   0xf000

/* Module Type//Language values */
#define MT_ANY      0
#define MT_PROGRAM  1
#define MT_SUBROUT  2
#define MT_MULTI    3
#define MT_DATA     4
#define MT_CSDDATA  5
#define MT_TRAPLIB  11
#define MT_SYSTEM   12
#define MT_FILEMAN  13
#define MT_DEVDRVR  14
#define MT_DEVDESC  15

#define ML_ANY      0
#define ML_OBJECT   1
#define ML_ICODE    2

#define mktypelang(type,lang)   (((type)<<8)|(lang))

/* Module Attribute values */
#define M/AREENT    0x80
#define M/AGHOST    0x40
#define M/ASUPER    0x20
```

```
#define mkattrevs(attr, revs)   (((attr)<<8)|(revs))

/* configuration module:  version smoothing definitions (_mcompat field) */
#define CP_SLOWIRQ      (1<<0)  /* save all regs during IRQ processing */
#define CP_NOSTOP       (1<<1)  /* don't use STOP instruction */
#define CP_NOGHOST      (1<<2)  /* don't retain ghost//sticky modules */
#define CP_NOBURST      (1<<3)  /* don't enable 68030 burst-fill */
#define CP_ZAPMEM       (1<<4)  /* patternize memory when allocated & freed */
#define CP_NOCLOCK      (1<<5)  /* don't start system clock during coldstart */

typedef unsigned short ushort;

struct modhcom {
    short       _msync,     /* sync bytes ($4afc) */
                _msysrev;   /* system revision check value */
    long        _msize,     /* module size */
                _mowner,    /* owner id */
                _mname;     /* offset to module name */
    short       _maccess,   /* access permission */
                _mtylan,    /* type//lang */
                _mattrev,   /* rev//attr */
                _medit;     /* edition */
    long        _musage,    /* comment string offset */
                _msymbol;   /* symbol table offset */
    short       _mident;    /* ident code for ident program */
    char        _mspare[12];/* reserved bytes */
    short       _mparity;   /* header parity */
};

/* Executable memory module */
typedef struct {
    struct modhcom _mh;     /* common header info */
    long        _mexec,     /* offset to execution entry point */
                _mexcpt,    /* offset to exception entry point */
                _mdata,     /* data storage requirement */
                _mstack,    /* stack size */
                _midata,    /* offset to initialized data */
                _midref;    /* offset to data reference lists */
} mod_exec;

/* device driver module */
typedef struct {
    struct modhcom _mh;     /* common header info */
    long        _mexec,     /* offset to execution entry point */
                _mexcpt,    /* offset to exception entry point */
                _mdata;     /* data storage requirement */
```

```
    short       _mdinit,    /* offset to init routine */
                _mdread,    /* offset to read routine */
                _mdwrite,   /* offset to write routine */
                _mdgetstat, /* offset to getstat routine */
                _mdsetstt,  /* offset to setstat routine */
                _mdterm,    /* offset to terminate routine */
                _mderror;   /* offset to exception error routine */
} mod_driver;

/* Device descriptor module */
typedef struct {
    struct modhcom  _mh;     /* common header info */
    char *          _mport;     /* device port address */
    unsigned char   _mvector;   /* trap vector number */
    unsigned char   _mirqlvl;   /* irq interrupt level */
    unsigned char   _mpriority; /* irq polling priority */
    unsigned char   _mmode;     /* device mode capabilities */
    short           _mfmgr;     /* file manager name offset */
    short           _mpdev;     /* device driver name offset */
    short           _mdevcon;   /* device configuration offset */
    unsigned short  _mdscres[4];/* (reserved) */
    unsigned short  _mopt;      /* option table size */
    unsigned char   _mdtype;    /* device type code */
} mod_dev;

/* Configuration module */
typedef struct {
    struct modhcom _mh;         /* common header info */
    long        _mmaxmem;       /* top limit of free ram (unused) */
    ushort      _mpollsz,       /* number of IRQ polling tbl entries */
                _mdevcnt,       /* number of device table entries */
                _mprocs,        /* number of process table entries */
                _mpaths,        /* number of path table entries */
                _msysparam,     /* offset to parameter string for _msysgo */
                _msysgo,        /* offset to initial module name */
                _msysdrive,     /* offset to system device name */
                _mconsol,       /* offset to system consol terminal name */
                _mextens,       /* offset to customization module name list */
                _mclock,        /* offset to clock module name */
                _mslice,        /* number of clock ticks per time slice */
                _mip_id;        /* interprocessor identification */
    long        _msite;         /* installation site code */
    ushort      _minstal;       /* installation name offset */
    long        _mcputyp;       /* cpu class (68000//010//020//030//3XX//040//070) */
    char        _mos9lvl[4];    /* operating system level//version//edition */
    ushort      _mos9rev,       /* offset to OS-9 level//revision string */
```

```
                _msyspri,       /* initial system priority */
                _mminpty,       /* initial minimum executable priority */
                _maxage;        /* initial maximum natural process age */
    long        _mmdirsz;       /* number of module dir entries (unused) */
    ushort      _mevents;       /* number of system event table entries */
    char        _mcompat,       /* version change smooth byte #1 */
                _mcompat2;      /* version change smoothing byte #2*/
    ushort      _mmemlist,      /* offset to (colored) memory list */
                _mstacksz,      /* IRQ stack size (in longwords) */
                _mcoldretrys,   /* coldstart's ćhd" retry counter */
                _mreserved[10]; /* reserved space */
} mod_config;
```

Dodatek B

Definicje zmiennych globalnych (plik: **/dd/DEFS/sysglob.a**)

sysglob.a

```
 opt -l
 nam System Global Data Definitions
*******************************
* Edition History
*  #   Date    Changes Made                                                by
* -- -------- ------------------------------------------------------------ ---
* 01 11//09//84 Split from process.a file.                                   rfd
* 02 01-11-85 Added event queue variables.                                  rfd
* 03 05-13-85 Changed active queue to doubly-linked.                        rfd
*             Added D_ActAge, changed process aging technique.              rfd
* 04 05-31-85 Rearranged slightly; added event tbl end ptr.                 rfd
*               and ActMem ptr, for initialized acct mod data.              rfd
* 05 06-19-85 Added D_MPUTyp (to be passed from boot rom).                  rfd
*               ---- OS-9//68k V1.2 released ----
* 06 08-12-85 Eliminated P$Queue and D_WProcQ.                              rfd
* 07 08-22-85 Added D_MinBlk & D_BlkSiz for new mem allocation.             rfd
* 08 09-09-85 Removed D_FreRAM and D_FreeCn.                                rfd
*             Renamed D_LowMem, D_HiMem to D_FreMem.                        rfd
* 09 09-16-85 Added SPU definitions.                                        rfd
* 10 01-20-86 Made System Process descr part of global static.              rfd
* 11 01-31-86 Added block and fragment sizes (powers of 2).                 rfd
* 12 03-13-86 Added D_Sieze process ID.                                     rfd
* 13 07-02-86 Added FPU defs: D_68881, D_FProc.                             rfd
*             Removed obsolete DAT conditionals.                            rfd
* 14 09-03-86 Rearranged for better (020) long-word alignment.              rfd
* 15 09-11-86 Added D_ProcSz to encourage its use.                          rfd
* 16 09-16-86 Added 'T_' defs (from process.a).                             rfd
*             Added D_Cigar: gross version of defs used.                    rfd
* 17 09-17-86 Added FPU trap 'T_' names.                                    rfd
* 18 09-25-86 Added D_Compat for smoothing upgrade problems.                rfd
*               ---- OS-9//68k V2.0 released ----
* 19 01-27-87 Added D_Flags for customization flags.                        rfd
* 20 03-25-87 Added D_Cache.                                                rfd
* 21 04-09-87 Removed D_SProc.                                              rfd
*    04-16-87 Added D_MMinLim D_MMaxLim.                                    rfd
* 22 05-05-87 Changed (removed) SPU definitions.                            rfd
* 23 05-11-87 Removed user accounting module definitions.                   rfd
*             Removed D_Flags.                                              rfd
*             Added D_Compat definitions.                                   rfd
* 24 05-21-87 Added D_SStkLm.                                               rfd
*               ---- OS-9//68k V2.1 released ----
* 25 08-25-87 Added D_Thread, and later D_AlarTh.                           rfd
* 26 09-15-87 Changed D_Tick, D_Slice, D_TSlice bytes to words.             rfd
*               ---- OS-9//68k V2.2 released ----
* 27 03-11-88 Moved D_PolTbl; added 68070 on-chip vectors.                  rfd
```

```
* 28 05-27-88 Added D_ClkMem.                                          rfd
* 29 05-31-88 Added D_IPCmd and D_FreeMem.                             rfd
* 30 07-05-88 Added (warren's) Cache control definitions.              rfd
* 31 07-18-88 Added D_IPID.                                            rfd
* 32 08-11-88 Added D_Forks.                                           rfd
*               ---- CD-RTOS V0.97 released ----
* 33 09-19-88 Added a few Memory color definitions (for CD-I).         rfd
* 34 10-06-88 Changed compat byte names slightly, added B_ZapMem.      rfd
*    10-07-88 Added D_ID.                                              rfd
*               ---- CD-RTOS V0.99 released ----
* 35 11-22-88 Added D_IRQFlag.                                         rfd
*               ---- OS-9//68k V2.2 edition #50 released ----
*               ---- OS-9//68k V2.3 released ----
* 36 08-23-89 Added D_NoSleep.                                         wwb
*               ---- OS-9//68k V2.3 edition #66 released ----
* 37 03-09-90 Added D_Compat2, D_SnoopD.  Added 68040 comments.        wwb
*               ---- CD-RTOS V1.1 released ----
*               ---- OS-9//68k V2.4 released ----
* 38 01-29-91 Added T_CProto, T_StkFmt, T_FPUndata, T_MMUConf,         wwb
*             T_MMUIlleg, T_MMUAcces.
*               ---- OS-9//68k V2.4 68040 released ----
* 39 06-09-91 Added D_FPUSize.                                         wwb
* 40 06-13-91 Added D_FPUMem (for FPU emulator).                       wwb
*               ---- OS-9//68k V2.4.3 released ----
*
edition equ 40 current edition number

 psect sysglob,0,0,edition,0,0
 pag
*******************************
* System Global Variable Definitions
SysGlob:    equ 1<<24 system global format revision (for D_Cigar)

 org 0
D_ID:       do.w 1 set to modsync code after coldstart has finished
D_NoSleep:  do.w 1 set to non-zero to prevent sysproc from sleeping.
            do.w 14 reserved
D_Init:     do.l 1 initialization module ptr ("init")
D_Clock:    do.l 1 address of system tick routine
D_TckSec:   do.w 1 number of ticks per second
D_Year:     do.w 1 year
D_Month:    do.b 1 month
D_Day:      do.b 1 day
D_Compat:   do.b 1 reserved for version compatability problems (#1)
D_68881:    do.b 1 FPU Type (68020//030//040 systems)
*                   0 = no FPU, 1 = 68881, 2 = 68882, 40 = 68040
```

```
D_Julian:   do.l 1 julian day number
D_Second:   do.l 1 system time: seconds left until midnight
            do.b 2 reserved (old D_Tick, T_TSlice, D_Slice)
D_IRQFlag:  do.b 1 IRQ flag
D_UnkIRQ:   do.b 1 number of times unknown IRQ occurred in a row
D_ModDir:   do.l 2 module directory (start & end ptrs)
D_PrcDBT:   do.l 1 Process Descriptor Block Table ptr
D_PthDBT:   do.l 1 Path Descriptor Block Table ptr
D_Proc:     do.l 1 Current Process descriptor ptr
D_SysPrc:   do.l 1 System Process Descriptor ptr
D_Ticks:    do.l 1 ever-increment system tick
D_FProc:    do.l 1 Process whose context is in FPU registers
D_AbtStk:   do.l 1 System state bus trap panic abort sp, return pc
D_SysStk:   do.l 1 System IRQ stack ptr
D_SysROM:   do.l 1 Bootstrap ROM execution entry point
D_ExcJmp:   do.l 1 Exception Jump Table ptr
D_TotRAM:   do.l 1 total RAM found by BootROM at startup
D_MinBlk:   do.l 1 process minimum allocatable block size
D_FreMem:   do.l 2 system free memory list head
D_BlkSiz:   do.l 1 system minimum allocatable block size
D_DevTbl:   do.l 1 I//O Device Table ptr
            do.l 1 reserved
D_AutIRQ2:  do.l 7 68070 on-chip I//O AutoVector Polling table heads
D_VctIRQ:   do.l 192 Vectored Interrupt device tbl ptrs
D_SysDis:   do.l 1 System Service dispatch table ptr
D_UsrDis:   do.l 1 User Service dispatch table ptr
D_ActivQ:   do.l 2 Active process queue head
D_SleepQ:   do.l 2 Sleeping process queue head
D_WaitQ:    do.l 2 Sleeping process queue head
D_ActAge:   do.l 1 Active queue age
D_MPUTyp:   do.l 1 MPU type (68000//68010//68020//68030//68070//68300//68040)
D_EvTbl:    do.l 2 ptr to system Event table start, end
D_EvID:     do.l 1 next (incrementing) event ID number
D_SPUMem:   do.l 1 ptr to SPU global variables (null == not enabled)
D_AddrLim:  do.l 1 highest address found during startup
D_Compat2:  do.b 1 reserved for version compatibility problems (#2)
D_SnoopD:   do.b 1 all data caches are coherent//snoopy (if non-zero)
D_ProcSz:   do.w 1 size of a process descriptor
D_PolTbl:   do.l 8 I//O AutoVector Polling table heads
D_FreeMem:  do.l 2 head of system free memory color node list
D_IPID:     do.w 1 interprocessor identification number
            do.w 1 reserved
D_CPUs:     do.l 1 ptr to array of cpu descriptor list heads
D_IPCmd:    do.l 2 head of inter-processor command queue
D_SProc     do.l 210 reserved (old system process descriptor stub)
D_CachMode: do.l 1 68020//68030//68040 CACR mode
```

```
D_DisInst:  do.l 1 instruction cache disable depth
D_DisData:  do.l 1 data cache disable depth
D_ClkMem:   do.l 1 ptr to clock tick thief's static storage
D_Tick:     do.w 1 current tick
D_TSlice:   do.w 1 ticks per slice
D_Slice:    do.w 1 current time slice remaining (ticks)
            do.w 1 reserved
D_Elapse:   do.l 1 time (ticks) to elapse before sys proc is awakened
D_Thread:   do.l 2 system Thread queue head (immediate, or at absolute time)
D_AlarTh:   do.l 2 system timed alarm threads (relative times)
D_SStkLm:   do.l 1 System IRQ stack low bound
D_Forks:    do.l 1 Number of actively forked processes.
D_BootRAM:  do.l 1 Ram found during bootrom search (for integrity check)
D_FPUSize:  do.l 1 FPU (max) state frame size
D_FPUMem:   do.l 1 FPU Emulator global data
D_IOGlob:   do.b 256 System Hardware dependent I//O flags
            do.w 1 reserved
D_MinPty:   do.w 1 system minimum process priority
D_MaxAge:   do.w 1 system priority maximum age limit
D_Sieze:    do.w 1 process ID of process that has siezed cpu
D_Cigar:    do.l 1 gross estimate of sysglob, process, and module defs
D_MMinLim   do.l 1 former minimum memory address allocatable
D_MMaxLim   do.l 1 former maximum memory address allocatable (+1)
            do.l 11 reserved
D_SysLst:   equ . last user changable global
            do.l 1 reserved (former D_ActMod: user accounting module ptr)
            do.l 1 reserved (former D_ActMem: user accounting memory ptr)
D_SysDbg:   do.l 1 system debugger entry pt address
D_DbgMem:   do.l 1 system debugger memory ptr
D_DbgFlg:   do.b 1 system debugger active flag
            do.b 3 reserved
D_Cache:    do.l 1 disk cache buffer head
            do.b $1000-. reserved for system stack
D_End:      equ . End of system global RAM

NoCRC:      equ 28523 avoid CRC chk if D_ID = No_CRC (for warmstart)

 pag
*******************************
* Service Dispatch Table special entries
SysTrap:    equ $8000 system state entry

 org 0 Free Memory Block offsets
M_NxtPtr:   do.l 1 ptr to next higher free block
M_PrvPtr:   do.l 1 ptr to next lower free block
M_BlkSiz:   do.l 1 size of this block
```

```
** Exception Vector address definitions
 org 0
T_ColdSP:   do.l 1 (0) reset initial SSP
T_ColdPC:   do.l 1 (1) reset initial PC
T_BusErr:   do.l 1 (2) bus error
T_AddErr:   do.l 1 (3) address error
T_IllIns:   do.l 1 (4) illegal instruction
T_ZerDiv:   do.l 1 (5) integer zero divide
T_CHK:      do.l 1 (6) CHK//CHK2 instruction
T_TRAPV:    do.l 1 (7) TRAPV//TRAPcc//FTRAPcc instruction
T_Priv:     do.l 1 (8) privilege violation
T_Trace:    do.l 1 (9) trace
T_1010:     do.l 1 (10) line 1010 emulator
T_1111:     do.l 1 (11) line 1111 emulator
            do.l 1 (12) reserved
T_CProto:   do.l 1 (13) coprocessor protocol violation
T_StkFmt:   do.l 1 (14) stack frame format error
T_UnIRQ:    do.l 1 (15) unitialized interrupt vector
            do.l 8 (16-23) reserved
T_SpurIO:   do.l 1 (24) spurious interrupt
T_AutIRQ:   do.l 7 (25-31) level 1-7 interrupt autovectors
T_TRAP:     do.l 16 (32-47) TRAP #n instruction vectors
T_FPUnordC: do.l 1 (48) FP bra or set on unordered condition
T_FPInxact: do.l 1 (49) FP inexact result
T_FPDivZer: do.l 1 (50) FP divide by zero
T_FPUndrFl: do.l 1 (51) FP underflow
T_FPOprErr: do.l 1 (52) FP operand error
T_FPOverFl: do.l 1 (53) FP overflow
T_FPNotNum: do.l 1 (54) FP signaling not a number
T_FPUnData: do.l 1 (55) FP unimplemented data type
T_MMUConf:  do.l 1 (56) PMMU configuration error
T_AutIRQ2:  do.l 7 (57-63) 68070 level 1-7 on-chip interrupt autovectors
T_MMUIlleg: equ T_AutIRQ2 (57) PMMU illegal operation
T_MMUAcces: equ T_AutIRQ2+4 (58) PMMU access level violation
T_VctIRQ:   do.l 192 vectored interrupt vectors
T_End:      equ .

AutOffs:    equ D_PolTbl-T_AutIRQ+4 constant needed in kernel excpt.a & poll.a
VctOffs:    equ D_AutIRQ2-T_AutIRQ2 constant needed in kernel excpt.a & poll.a

* process catchable error traps
MINTRAP:    equ T_BusErr minimum user catchable TRAP number
MAXTRAP:    equ T_1111 maximum user catchable TRAP number
FPCPMIN:    equ T_FPUnordC lowest catchable FPCP eror (68020//68030//68040)
FPCPMAX:    equ T_FPNotNum highest (68020//68030//68040)
```

```
* D_Compat flag bit numbers
B_SlowIRQ:  equ 0 xxxxxxx1 save all regs during IRQ processing if set
B_NoStop:   equ 1 xxxxxx1x don't use stop instruction if set
B_NoGhost:  equ 2 xxxxx1xx don't retain Ghost memory modules if set
B_NoBurst:  equ 3 xxxx1xxx don't enable cache burst mode (68030) if set
B_ZapMem:   equ 4 xxx1xxxx wipe out memory that is allocated//freed
B_NoClock:  equ 5 xx1xxxxx don't start system clock during coldstart

*D_Compat2 flag bit numbers
* Note:  a śnoopy" cache is a cache that maintains its integrity
*        through hardware means, or is non-existant.
B_SEI:      equ 0 xxxxxxx1 external instruction cache is snoopy
B_SED:      equ 1 xxxxxx1x external data cache is snoopy
B_SOI:      equ 2 xxxxx1xx on-chip instruction cache is snoopy
B_SOD:      equ 3 xxxx1xxx on-chip data cache is snoopy

B_DDIO:     equ 7 1xxxxxxx don't disable data caching when in I//O

* memory type access bit definitions
B_USER:     equ 1<<0    memory allocatable by user procs
B_PARITY:   equ 1<<1    parity memory; must be initialized
B_ROM:      equ 1<<2    read-only memory; searched for modules
B_NVRAM:    equ 1<<3    non-volatile RAM; searched for modules
B_SHARE:    equ 1<<4    shared memory

* Memory color definitions
SYSRAM: equ $01 generic system ram
VIDEO1: equ $80 CD-I video bank 1
VIDEO2: equ $81 CD-I video bank 2

 pag
 use ioglob.a include system specific I//O global definitions
 ends
 opt l
```

Dodatek C

Definicje biblioteki gniazdek (plik: **/dd/DEFS/socket.h**)

socket.h

```
/*
 * Copyright (c) 1982, 1985, 1986 Regents of the University of California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms are permitted
 * provided that this notice is preserved and that due credit is given
 * to the University of California at Berkeley. The name of the University
 * may not be used to endorse or promote products derived from this
 * software without specific prior written permission. This software
 * is provided ‘‘as is’’ without express or implied warranty.
 *
 *  @(#)socket.h    7.2 (Berkeley) 12//30//87
 */
/*-----------------------------------------------------------------\
| Copyright (c) 1988 by Microware Systems Corporation
| Reproduced Under License
|
| This source code is the proprietary confidential property of
| Microware Systems Corporation, and is provided to licensee
| solely for documentation and educational purposes. Reproduction,
| publication, or distribution in any form to any party other than
| the licensee is strictly prohibited.
|
|   socket.h header file
|
| Edition History
|  ver  Comment                                                 who  Date
|  ---  ------------------------------------------------------- --- --------
|   01  Ported                                                  aln 88//11//10
\-----------------------------------------------------------------------------*/

/*
 * Definitions related to sockets: types, address families, options.
 */

/*
 * Types
 */
#define SOCK_STREAM 1       /* stream socket */
#define SOCK_DGRAM  2       /* datagram socket */
#define SOCK_RAW    3       /* raw-protocol interface */
#define SOCK_RDM    4       /* reliably-delivered message */
#define SOCK_SEQPACKET  5       /* sequenced packet stream */

/*
```

```
 * Option flags per-socket.
 */
#define SO_DEBUG        0x0001      /* turn on debugging info recording */
#define SO_ACCEPTCONN   0x0002      /* socket has had listen() */
#define SO_REUSEADDR    0x0004      /* allow local address reuse */
#define SO_KEEPALIVE    0x0008      /* keep connections alive */
#define SO_DONTROUTE    0x0010      /* just use interface addresses */
#define SO_BROADCAST    0x0020      /* permit sending of broadcast msgs */
#define SO_USELOOPBACK  0x0040      /* bypass hardware when possible */
#define SO_LINGER       0x0080      /* linger on close if data present */
#define SO_OOBINLINE    0x0100      /* leave received OOB data in line */

/*
 * Additional options, not kept in so_options.
 */
#define SO_SNDBUF   0x1001      /* send buffer size */
#define SO_RCVBUF   0x1002      /* receive buffer size */
#define SO_SNDLOWAT 0x1003      /* send low-water mark */
#define SO_RCVLOWAT 0x1004      /* receive low-water mark */
#define SO_SNDTIMEO 0x1005      /* send timeout */
#define SO_RCVTIMEO 0x1006      /* receive timeout */
#define SO_ERROR    0x1007      /* get error status and clear */
#define SO_TYPE     0x1008      /* get socket type */

/*
 * Structure used for manipulating linger option.
 */
struct  linger {
    int l_onoff;        /* option on//off */
    int l_linger;       /* linger time */
};

/*
 * Level number for (get//set)sockopt() to apply to socket itself.
 */
#define SOL_SOCKET  0xffff      /* options for socket level */

/*
 * Address families.
 */
#define AF_UNSPEC   0       /* unspecified */
#define AF_UNIX     1       /* local to host (pipes, portals) */
#define AF_INET     2       /* internetwork: UDP, TCP, etc. */
#define AF_IMPLINK  3       /* arpanet imp addresses */
#define AF_PUP      4       /* pup protocols: e.g. BSP */
#define AF_CHAOS    5       /* mit CHAOS protocols */
```

```
#define AF_NS       6       /* XEROX NS protocols */
#define AF_NBS      7       /* nbs protocols */
#define AF_ECMA     8       /* european computer manufacturers */
#define AF_DATAKIT  9       /* datakit protocols */
#define AF_CCITT    10      /* CCITT protocols, X.25 etc */
#define AF_SNA      11      /* IBM SNA */
#define AF_ETHER    12      /* Ethernet 2.0 */
#define AF_DLI      13      /* Direct data link interface */
#define AF_LAT      14      /* LAT */
#define AF_HYLINK   15      /* NSC Hyperchannel */
#define AF_APPLETALK    16      /* Apple Talk */
#define AF_NIT          17      /* Network Interface Tap */
#define AF_802          18      /* IEEE 802.2, also ISO 8802 */
#define AF_OSI          19      /* umbrella for all families use
                          * by OSI  */
#define AF_X25          20      /* Apple Talk */

#define AF_MAX          21

/*
 * Structure used by kernel to store most
 * addresses.
 */
struct sockaddr {
    u_short s/afamily;      /* address family */
    char    s/adata[14];        /* up to 14 bytes of direct address */
};

/*
 * Structure used by kernel to pass protocol
 * information in raw sockets.
 */
struct sockproto {
    u_short sp_family;      /* address family */
    u_short sp_protocol;        /* protocol */
};

/*
 * Protocol families, same as address families for now.
 */
#define PF_UNSPEC   AF_UNSPEC
#define PF_UNIX     AF_UNIX
#define PF_INET     AF_INET
#define PF_IMPLINK  AF_IMPLINK
#define PF_PUP      AF_PUP
#define PF_CHAOS    AF_CHAOS
```

```
#define PF_NS       AF_NS
#define PF_NBS      AF_NBS
#define PF_ECMA     AF_ECMA
#define PF_DATAKIT  AF_DATAKIT
#define PF_CCITT    AF_CCITT
#define PF_SNA      AF_SNA
#define PF_DECnet   AF_DECnet
#define PF_DLI      AF_DLI
#define PF_LAT      AF_LAT
#define PF_HYLINK   AF_HYLINK
#define PF_APPLETALK    AF_APPLETALK

#define PF_MAX      AF_MAX

/*
 * Maximum queue length specifiable by listen.
 */
#define SOMAXCONN   5

/*
 * Message header for recvmsg and sendmsg calls.
 */
struct msghdr {
    char *msg_name;     /* optional address */
    int msg_namelen;        /* size of address */
    struct  iovec *msg_iov;     /* scatter//gather array */
    int msg_iovlen;     /* # elements in msg_iov */
    char *msg_accrights;        /* access rights sent//received */
    int msg_accrightslen;
};

#define MSG_OOB     0x1     /* process out-of-band data */
#define MSG_PEEK    0x2     /* peek at incoming message */
#define MSG_DONTROUTE   0x4     /* send without using routing tables */

#define MSG_MAXIOVLEN   16
```

Dodatek D

Definicje procesów (pliki: **/dd/DEFS/process.a** i **/dd/DEFS/procid.h**)

process.a

```
 opt -l
 nam Process Descriptor Format
********************************
* Edition History
*  #   Date    Changes Made                                                by
* -- -------- ------------------------------------------------------------ ---
* 00 04-11-83 Converted to 68000 from OS-9 Level II Edition 7.             rfd
* 00 05-24-83 Removed DAT routines with conditional assembly.              rfd
* 00 06-21-83 Changed from "ds" to "do" (define offset).                   rfd
* 01 02-09-84 Converted for linkage use.                                   lac
* 02 03-23-84 Added D_TckSec word.                                         rfd
* 03 05-21-84 Rearranged D_ variables for convenience.                     rfd
* 04 08-24-84 Added new codes for scheduler changes.                       rfd
* 05 09-06-84 Increased Process Local stack size.                          rfd
* 05 10-16-84 Added D_UnkIRQ in reserved byte.                             rfd
* 06 10-24-84 Added P$QueuID, P$SCall, and D_Debug variables.              rfd
* 07 11-05-84 Added D_ActMod.                                              rfd
*             Added machine characteristics.                               rfd
* 08 11-09-84 Removed system globals, rearranged.                          rfd
* 09 01-08-85 Reworked P$Queue pointers.                                   rfd
* 10 05-15-85 Added P$Sched variable, changed process aging.               rfd
* 11 05-31-85 Added P$Acct space for accounting module.                    rfd
*               ---- OS-9//68k V1.2 released ----
* 12 08-12-85 Eliminated P$Queue, moved P$QueueN, P$QueueP.                rfd
* 13 09-09-85 Added P$Frags.                                               rfd
* 14 09-16-85 Added SPU//MMU definitions.                                   rfd
* 15 01-17-86 Added P$Data and P$DataSz for new memory alloc.              rfd
* 16 07-16-86 Added 68881 FPU definitions.                                 rfd
* 17 09-03-86 Reorganized for better (020) long-word alignment.            rfd
* 18 09-16-86 Removed 'T_' names, moved to sysglob.a.                      rfd
*               ---- OS-9//68k V2.0 released ----
* 19 03-20-87 Added signal mask and queue.                                 rfd
*             Removed P$Size; D_ProcSz(a6) must be used instead.           rfd
* 20 04-21-87 Reworked the signal variables.                               rfd
* 21 05-05-87 Changed (removed) SPU definitions.                           rfd
* 22 05-27-87 Added cache control definitions.                             rfd
*               ---- OS-9//68k V2.1 released ----
* 23 08-25-87 Added thread block definition and "Q_None".                  rfd
* 24 09-25-87 Added P$Thread queue head.                                   rfd
* 25 01-19-87 Added P$BkPtCnt.                                             rfd
*               ---- OS-9//68k V2.2 released ----
* 26 06-13-88 Added 68030 cache control definitions.                       rfd
* 27 07-05-88 Changed cache control definitions for wwb.                   rfd
* 28 08-11-88 Added P$Baked.                                               rfd
*               ---- CD-RTOS V0.97 released ----
```

```
* 29 10-01-88 Added P$MOwn.                                          rfd
*               ---- CD-RTOS V0.99 released ----
* 30 01-09-89 Nervously added P$ExpStk for temporary use of SockMan.  rfd
*               ---- OS-9//68k V2.2 edition #50 released ----
*               ---- OS-9//68k V2.3 released ----
* 31 06-15-89 Added FPU$Busy field in fpu save area.                  rfd
*               ---- CD-RTOS V1.1 released ----
*               ---- OS-9//68k V2.4 released ----
* 32 91//01//29 Added 68040 FP state frame definitions.                wwb
*               ---- OS-9//68k V2.4 68040 released ----
*               ---- OS-9//68k V2.4.3 released ----
*
edition set 32 current edition

 psect process,0,0,edition,0,0
 pag

MemBlks:    equ 32 number of separate memory blocks per process
NumPaths:   equ 32 number of local I//O paths
DefIOSiz:   equ 32 default I//O data area size (RBF requires >=20)
ExecDir:    equ DefIOSiz//2 offset of default execution directory data

*******************************
* Process Descriptor Definitions
ProcDsc: equ 1<<16 process descriptor format revision (for D_Cigar)

 org 0
P$ID:       do.w 1 process ID
P$PID:      do.w 1 parent's ID
P$SID:      do.w 1 sibling's ID
P$CID:      do.w 1 child's ID
P$sp:       do.l 1 system stack ptr
P$usp:      do.l 1 user stack ptr
P$MemSiz:   do.l 1 total combined non-module memory consumption
P$User:     do.w 2 user index (group, user)
P$Prior:    do.w 1 priority
P$Age:      do.w 1 age
P$State:    do.w 1 status
P$Task:     do.w 1 process task number
P$QueuID:   do.b 1 current queue; what process is doing
P$SCall:    do.b 1 last system call executed
P$Baked:    do.b 1 non-zero if process was created by Fork
            do.b 1 reserved
P$DeadLk:   do.w 1 dominant Process ID if I//O locked
P$Signal:   do.w 1 signal code
P$SigVec:   do.l 1 signal intercept vector
```

```
P$SigDat:   do.l 1 signal intercept data address
P$QueueN:   do.l 1 next process queue ptr
P$QueueP:   do.l 1 previous process queue ptr
P$PModul:   do.l 1 primary module
P$Except:   do.l 10 Program error exception vectors
P$ExStk:    do.l 10 program error exception stack frame ptrs
P$Traps:    do.l 15 user's TRAP vectorltable
P$TrpMem:   do.l 15 trap handler static memory block ptrs
P$TrpSiz:   do.l 15 trap handler static memory block sizes
P$ExcpSP:   do.l 1 system state exception recovery stack
P$ExcpPC:   do.l 1 system state exception recovery program counter
P$DIO:      do.b DefIOSiz default I//O data
P$Path:     do.w NumPaths I//O path table
P$MemImg:   do.l MemBlks allocated memory block ptrs
P$BlkSiz:   do.l MemBlks size of each allocated memory block
P$DbgReg:   do.l 1 debugged process register stack frame
P$DbgPar:   do.l 1 debugged process parent Proc Desc ptr
P$DbgIns:   do.l 1 debugged process instruction count
P$UTicks:   do.l 1 user state ticks elapsed
P$STicks:   do.l 1 system state ticks elapsed
P$DatBeg:   do.l 1 fork system julian date
P$TimBeg:   do.l 1 fork julian time when forked
P$FCalls:   do.l 1 number of function calls executed
P$ICalls:   do.l 1 number of I//O system calls executed
P$RBytes:   do.l 1 number of bytes read
P$WBytes:   do.l 1 number of bytes written
P$IOQP:     do.w 1 previous I//O queue (process ID) link
P$IOQN:     do.w 1 next I//O queue (process ID) link
P$Frags:    do.l 2 free memory fragment list head
P$Sched:    do.l 1 active queue scheduling constant
P$SPUMem:   do.l 1 ptr to process' SPU data
P$BkPtCnt:  do.l 1 number of breakpoints set
P$BkPts:    do.w 16 reserved for contents of breakpoint instructions
P$Acct:     do.l 8 reserved for user accounting module
P$Data:     do.l 1 ptr to process primary data area
P$DataSz:   do.l 1 size of process primary data area
P$FPUSave:  do.l 1 ptr to FPU save area
P$FPExcpt:  do.l 7 floating point error exception vectors
P$FPExStk:  do.l 7 floating point error exception stack frame ptrs
P$SigLvl:   do.b 1 signal interrupt level
P$SigFlg:   do.b 1 signal flag
P$Sigxs:    do.w 1 number of excess signal blocks allocated
P$SigMask:  do.l 1 mask to disable signals 2-31
P$SigCnt:   do.l 1 number of signals pending
P$SigQue:   do.l 1 ptr to head of signal queue
P$DefSig:   do.l 4 default initial signal queue element
```

```
P$Thread:   do.l 2 doubly linked thread queue
P$frag:     do.l 2 new-style memcolor lists
P$MOwn:     do.l 1 original owner of primary module (for security)
P$ExpStk:   do.l 1 reserved for more stack space
            do.l 1 reserved for more stack space (size)
P$Last:     equ . non-stack data requirement
StackRoom   do.b $800-. Local stack
P$Stack:    equ . Top of Stack
P$ProcSz:   equ . Size of Proc Desc (for coldstart - others must use D_ProcSz)

 pag
*******************************
* Process State Flag Bits
SysState:   equ 7 1xxx xxxx executing system state routine
TimSleep:   equ 6 x1xx xxxx timed sleep
TimOut:     equ 5 xx1x xxxx time slice has expired
ImgChg:     equ 4 xxx1 xxxx SPU//MMU protection map has changed
*               3 xxxx 1xxx (unused)
*               2 xxxx x1xx (unused)
Condemn:    equ 1 xxxx xx1x process is condemned
Dead:       equ 0 xxxx xxx1 process has terminated

*******************************
* Process Queue ID codes
Q_None:     equ ' ' not in any queue
Q_Dead:     equ '-' no queue:       dead process
Q_Active:   equ 'a' active process queue
Q_Debug:    equ 'd' no queue:       inactively debugging
Q_Event:    equ 'e' event queue
Q_Sleep:    equ 's' sleep queue
Q_Wait:     equ 'w' waiting queue
Q_Currnt:   equ '*' no queue:       currently running

*******************************
* Signal queue format
 org 0
Sig_Nxt:    do.l 1 ptr to next entry in queue
Sig_Prv:    do.l 1 ptr to previous entry in queue
            do.w 1 reserved
Sig_Code:   do.w 1 signal code
            do.l 1 reserved
Sig_Size:   equ .

* Signal flag bits (in P$SigFlg)
B_SIGMASK:  equ 0 signals are masked when set
B_WAKEUP:   equ 7 wakeup is pending when set
```

```
*********************************
* User Register Stack Image names
        org 0
R$d0:   do.l 1 data registers
R$d1:   do.l 1
R$d2:   do.l 1
R$d3:   do.l 1
R$d4:   do.l 1
R$d5:   do.l 1
R$d6:   do.l 1
R$d7:   do.l 1
R$a0:   do.l 1 address registers
R$a1:   do.l 1
R$a2:   do.l 1
R$a3:   do.l 1
R$a4:   do.l 1
R$a5:   do.l 1
R$a6:   do.l 1
R$a7:   do.l 1 (user stack pointer)
R$sr:   do.w 1 status register
R$cc:   equ R$sr+1 condition codes portion
R$pc:   do.l 1 program counter register
R$fmt:  do.w 1 68010 exception format & vector
R$Size: equ . total register package size

** Status Register Fields
TraceBit:   equ 7 trace      bit in system byte
SupvrBit:   equ 5 supervisor bit in system byte

Trace:      equ %10000000<<8 trace mode
Supervis:   equ %00100000<<8 supervisor state
IntMask:    equ %00000111<<8 interrupt mask
IntEnab:    equ ^IntMask
Extend:     equ %00010000 Extend bit
Negative:   equ %00001000 Negative flag
Zero:       equ %00000100 Zero flag
TwosOvfl:   equ %00000010 Two's Comp Overflow flag
Carry:      equ %00000001 Carry bit
NoCarry:    equ ^Carry

Sign:       equ $80 sign bit of a byte
SignBit:    equ 7 bit number of sign bit

 pag
*********************************
```

```
* F$CCtl cache control definitions

* F$CCtl parameter defs:
* d0.l = 0 --> flush cache  (OLD COMPAT METHOD)

* control bit numbers:
b_endata:   equ 0 enable data cache
b_disdata:  equ 1 disable data cache
b_fldata:   equ 2 flush data cache
b_eninst:   equ 4 enable instruction cache
b_disinst:  equ 5 disable instruction cache
b_flinst:   equ 6 flush instruction cache

endata:     equ 1<<b_endata
disdata:    equ 1<<b_disdata
fldata:     equ 1<<b_fldata
eninst:     equ 1<<b_eninst
disinst:    equ 1<<b_disinst
flinst:     equ 1<<b_flinst

*******************************
* 68881//68882//68040 FPU context save area definitions
 org 0
FPU$MemS:   do.l 1 memory block size
FPU$Busy:   do.l 1 fpu save area is busy (used during signal handling)
* WARNING:  "fpumacs.d" assumes FPU$Regs and FPU$Cntrl are contiguous.
FPU$Regs:   do.l 8*3 FPU register save
FPU$Cntrl:  do.l 3 FPU control reg save
* end of order dependant defs
FPU$Contxt: do.b $60+4 68040 FPU context save area
FPU$Siz040: equ . size of 68040 context area
            do.b $b4-$60 68881 FPU context area
FPU$Siz881: equ . size of 68881 FPU context area
            do.b $20 68882 FPU extras (but towards the front!)
FPU$Siz882: equ . size of 68882 FPU context area
FPU$Size:   equ . max size of FPU context area


*******************************
* Thread Execution Block
 org 0
T_ID:       do.w 1 reserved
T_Proc:     do.w 1 owner process ID (zero if none)
T_MSiz:     do.l 1 thread block memory size
T_User:     do.l 1 owner's user number
T_Next:     do.l 1 next thread in doubly linked list
```

```
T_Prev:     do.l 1 previous thread in doubly linked list
T_Link:     do.l 2 doubly linked list of associated threads (owner link)
T_Sys:      do.l 4 reserved for activation conditions
T_Regs:     do.b R$Size register image
T_Size:     equ . total size of entry

            org T_Sys alarm condition definitions
T_Cycle:    do.l 1 wakeup cycle period
T_WkTime:   do.l 1 wakeup time
T_WkDate:   do.l 1 wakeup date
            do.l 1 reserved
 ends
 opt l
```

procid.h

```
/* OS-K process descriptor definitions */
#ifndef VHPCNT
#include <module.h>
#endif

/* The pointer definitions herein are generally (unsigned char *) */

typedef struct pdsc {
    unsigned short
        _id,                /* process id */
        _pid,               /* parent's id */
        _sid,               /* sibling's id */
        _cid;               /* child's id */
    unsigned char
        *_sp,               /* system stack ptr */
        *_usp,              /* user stack ptr */
        *_pagcnt;           /* memory size */
    unsigned short
        _group,             /* group number */
        _user,              /* user number */
        _prior,             /* priority */
        _age,               /* age */
        _state,             /* status */
/* process state flags */
#define PS_SYSSTATE     0x80    /* executing system-state routine       */
#define PS_TIMSLEEP     0x40    /* timed sleep                          */
#define PS_TIMOUT       0x20    /* time slice has expired               */
#define PS_IMGCHG       0x10    /* spu//mmu protection map has changed   */
/*                      0x08    /* reserved                             */
/*                      0x04    /* reserved                             */
#define PS_CONDEMN      0x02    /* process is condemned                 */
#define PS_DEAD         0x01    /* process has terminated               */

        _task;              /* process task number */
    unsigned char
        _queueid,           /* current queue id */
        _scall,             /* last user state system call executed */
        _baked,             /* non-zero if process created by fork */
        _resvd1;            /* (reserved for alignment) */
    unsigned short
        _deadlk,            /* dominant process id if I//O locked */
        _signal;            /* signal code */
    unsigned char
        *_sigvec,           /* signal intercept vector */
        *_sigdat;           /* signal intercept data address */
```

```
struct pdsc
    *_queueN,           /* queue next ptr */
    *_queueP;           /* queue prev ptr */
mod_exec
    *_pmodul;           /* primary module */
unsigned char
    *_except[10],       /* program error exception vectors */
    *_exstk[10],        /* program error exception stack frame ptrs */
    *_traps[15],        /* user's trap vector table */
    *_trpmem[15];       /* user's trap static memory block ptrs */
unsigned int
    _trpsiz[15];        /* trap handler static memory sizes */
unsigned char
    *_excpsp,           /* system state except. recovery stack */
    *_excppc;           /* system state except. recovery program counter */
char
    _dio[32];           /* default i//o data */
unsigned short
    _path[32];          /* i//o path table */
unsigned char
    *_memimg[32];       /* allocated memory block ptrs */
unsigned int
    _blksiz[32];        /* size of each allocated memory block */
unsigned char
    *_dbgreg;           /* debugged process register stack frame */
struct pdsc
    *_dbgpar;           /* debugged process parent pd */
unsigned  int
    _dbgins,            /* debug instruction count */
    _uticks,            /* user state ticks elapsed */
    _sticks,            /* system state tick elapsed */
    _datbeg,            /* julian date when forked */
    _timbeg,            /* julian time when forked */
    _fcalls,            /* number of system calls exec */
    _icalls,            /* number of I//O calls execd */
    _rbytes,            /* number of bytes read */
    _wbytes;            /* number of bytes written */
unsigned short
    _ioqp,              /* I//O queue prev ptr */
    _ioqn;              /* I//O queue next ptr */
unsigned char
    *_frag, *_fragg;    /* free memory fragment ptrs */
unsigned int
    _sched;             /* process scheduling constant */
unsigned char
    *_spuimg;           /* ptr to process SPU image */
```

```
    unsigned int
        _bkptcnt;           /* number of breakpoints set */
    unsigned short
        _bpvalue[16];       /* breakpoint save area */
    unsigned int
        _acct[8];           /* reserved for user accounting module */
    unsigned char
        *_data;             /* ptr to process primary data area */
    unsigned int
        _datasz;            /* size of primary data area */
    unsigned char
        *FPUsave;           /* ptr to 68881 save area */
    unsigned char
        *FPExcpt[7],        /* 68881 exception vectors */
        *FPExStk[7];        /* 68881 exception stack frame ptrs*/
    unsigned char
        _siglvl,            /* signal interrupt level */
        _sigflg;            /* signal flag */
    unsigned short
        _sigxs;             /* number of excess signal blocks allocated */
    unsigned int
        _sigmask,           /* mask to disable signals 2 - 31 */
        _sigcnt;            /* number of signals pending */
    unsigned char
        *_sigque,           /* ptr to head of signal queue */
        _defsig[16],        /* default initial signal queue element */
        *_thread[2],        /* doubly linked thread queue */
        *_cfrag[2];         /* new-style mem color lists */
    unsigned int
        _mown;              /* original owner of primary module (security) */
    unsigned char
        *_expstk;           /* reserved for more stack space */
    unsigned int
        resvd2;             /* reserved for more stack space (size) */
    unsigned char
        _procstk[1108];     /* system state stack for process */
} procid;
```

Dodatek E

Definicje zdarzeń (plik: **/dd/DEFS/signal.h** i **/dd/DEFS/events.h**)

events.h

```
/*
    OS-9 Event and Semaphores

*/

#define EV_SIZE     32      /* event entry size */
#define EV_ALLPROCS 0x8000  /* activate all processes in range */

typedef struct _evstr {
    unsigned short _ev_eid;     /* event id number */
    char _ev_name[12];          /* event name */
    int _ev_value;              /* current event value */
    short _ev_winc;             /* wait increment value */
    short _ev_sinc;             /* signal increment value */
    unsigned short _ev_link;    /* event use count */
    struct _evstr *queN;        /* next event in queue */
    struct _evstr *queP;        /* previous event in queue */
} event;

extern int
    _ev_create(),   _ev_link(),     _ev_signal(),   _ev_pulse(),
    _ev_set(),      _ev_setr(),     _ev_unlink(),   _ev_delete(),
    _ev_info(),     _ev_wait(),     _ev_waitr(),    _ev_read();
```

dr inż. Marek Wnuk
Instytut Cybernetyki Technicznej
Politechniki Wrocławskiej
ul. Janiszewskiego 11/17
50-372 Wrocław

Niniejszy raport otrzymują:

1. OINT ..... - 1 egz.

2. Zleceniodawca ..... - 1 egz.

3. Biblioteka Wydziału Elektroniki ..... -4 egz.

4. Autor ..... - 4 egz.

Razem : 10 egz.

Raport wpłynął do redakcji I-6
w listopadzie 1994 roku.