ISSN 0239–8044

1
1995

# Techniki Komputerowe

## BIULETYN INFORMACYJNY


INSTYTUT MASZYN MATEMATYCZNYCH
WARSZAWA 1995

# Techniki Komputerowe

## BIULETYN INFORMACYJNY

Rok XXX, Nr 1, 1995

INSTYTUT MASZYN MATEMATYCZNYCH
WARSZAWA 1995

Wydaje:

*INSTYTUT MASZYN MATEMATYCZNYCH*
*UL. KRZYWICKIEGO 34*
*02-798 WARSZAWA*
*TEL. 621.84.41, TLX 81.78.80, FAX 29.92.70*
*E-MAIL imasmat@frodo.nask.org.pl*



Wydanie publikacji dofinansowane
przez Komitet Badań Naukowych

Druk: Zakład Poligrafii Ośrodka Przetwarzania Informacji, Warszawa, al. Niepodległości 186

# TECHNIKI KOMPUTEROWE

Rok XXX Nr 1 1995

## Spis treści

Str.

**Zasady kodowania programów w języku C,** Marek Kotowski ..... 5
**Mikrokomputerowy system dyspozytorski MSD-BUSZ,** Lucjan Szten, Roman Ninard ..... 27
**Pamięci pomocnicze (cache) komputerów osobistych,** Jan Ryżko ..... 41
**Algorytm rozpoznawania grafiki jako struktury pomiarowej,** Jarosław Wójtowicz ..... 47

MAREK KOTOWSKI
INSTYTUT MASZYN MATEMATYCZNYCH WARSZAWA

# Zasady kodowania programów w języku C
## Coding rules for the C language

### Streszczenie

W tekście przedstawiony został standard kodowania programów w języku C, opracowany w Instytucie Maszyn Matematycznych dla potrzeb zespołów programistycznych. Przedstawiono także ogólne zalecenia dotyczące sposobu pisania standardów kodowania w języku C i innych językach programowania.

### Abstract

In the paper the program coding standard for the C language programmers is defined. General rules concerning coding standards for a class of programming languages are also suggested.

### Wstęp

Standardy kodowania programów stanowią integralny element technologii pracy nad dużymi projektami programistycznymi. Bez nich rozwój dużego programu, jego modyfikowanie czy przenoszenie do innych środowisk byłyby bardzo utrudnione, o ile w ogóle możliwe. W Polsce standardy kodowania nie są jeszcze szerzej stosowane i fakt ten jest swoistym odzwierciedleniem słabej kondycji naszego rodzimego oprogramowania. Sytuacja programisty, który zmieniając firmę wie, że w nowym miejscu pracy będzie od niego wymagane przestrzeganie pewnego minimum reguł kodowania, w Stanach Zjednoczonych naturalna, dla nas jest wciąż jeszcze nieco egzotyczna. Niemniej stan ten powoli się zmienia na skutek dynamicznego rozwoju rynku oprogramowania. Bez wątpienia standardy kodowania w języku C i nie tylko pojawią się i w Polsce w ciągu najbliższych lat (wręcz już się pojawiają) jako integralny element towarzyszący rozwojowi produkcji oprogramowania i swoisty przejaw dojrzałości i profesjonalizmu środowiska zawodowego.

W tekście tym przedstawiony jest standard kodowania programów w języku C opracowany w Instytucie Maszyn Matematycznych (podane są także pewne ogólne zalecenia dotyczące tworzenia standardów kodowania). W końcowej części tekstu opisano niektóre reakcje i opinie programistów na temat standardu.

## 1. Standard kodowania: czym jest i po co się go wprowadza?

Standardy kodowania obejmują zazwyczaj pewien zestaw zaleceń i reguł, w większości związanych z szeroko rozumianym dobrym stylem kodowania. Zalecenia te mogą być różne, mniej lub bardziej szczegółowe, a całość standardu mieścić się na jednej kartce lub też wydłużać w wielostronicową listę. Oprócz terminu "standard kodowania" używane są też inne, np. konwencje kodowania (ang. coding conventions). Standard kodowania opracowany w IMM nazwano "zasadami kodowania", uznając, że słowo "standard" lepiej pozostawić na określenie standardowej wersji języka ANSI C.

Standard kodowania z definicji dotyczy pisania kodu. Trzeba powiedzieć, że istnieją też standardy programowania, dotyczące całego procesu tworzenia programu, od określania założeń, aż po testowanie, modyfikację i rozwój programu. Niemniej zdecydowana większość standardów funkcjonujących w programowaniu (zwłaszcza w języku C), to właśnie standardy kodowania.

Po co jest tworzony standard kodowania? Celem konkretnym jest ujednolicenie praktyk stosowanych w kodowaniu. Jeśli coś można zrobić na kilka sposobów, należy wybrać jeden z nich, możliwie najlepszy i trzymać się go zawsze, także wtedy, gdy jest równoprawny z innymi. Cel ogólny standardu, to zwiększenie jakości programu i efektywności pracy programisty. Słowo "jakość", używane często przy omawianiu standardów kodowania, oznacza tu szereg różnych cech. Jako podstawowy element stylistyczny decydujący o jakości tworzonego kodu wymienia się zazwyczaj jego czytelność (programu nieczytelnego nie można testować, poprawiać, ani rozwijać). Inne elementy stylu kodowania, decydujące o jakości tworzonego kodu, to jego przenośność, modyfikowalność, niezawodność (w sensie tworzonych konstrukcji językowych, które powinny dawać jak najmniej okazji do popełniania błędów), wreszcie spójność (program zakodowany w złym stylu czyta się źle, zaś programu zakodowanego w stylu zmiennym w ogóle nie da się czytać).

Standard kodowania ma ułatwić powstawanie kodu o takich zaletach. Oczywiście nie jest on panaceum na wszystko - jego stosowanie nie spowoduje np. usunięcia błędów logicznych istniejących w programie, ani nie zwiększy efektywności zastosowanego algorytmu, nie mówiąc już o spełnieniu przez program oczekiwań użytkownika (sprawa ta rozstrzygana jest przede wszystkim na etapie tworzenia założeń i projektu programu). Niemniej dzięki stosowaniu standardu kodowania pewne elementy pracy programisty, ustalone w formie najbardziej poprawnej, staną się jednocześnie bardziej mechaniczne, rutynowe, a przez to samo kodowanie będzie mniej podatne na błędy, zaś tworzony kod - czytelniejszy i łatwiejszy do modyfikowania.

Z powyższego wynika już w dużej mierze, dla kogo przeznaczony jest standard kodowania. Głównym adresatem zaleceń standardu jest oczywiście sam autor programu, zwłaszcza mniej doświadczony. Ogólnie rzecz biorąc, standard kodowania przeznaczony jest dla wszystkich, którzy pracują - czy też będą pracować - nad programem: usuwają w nim błędy, przenoszą go na inny komputer, kompilator czy do innego środowiska operacyjnego, rozszerzają go o nowe opcje lub wykorzystują jego fragmenty w innych programach. Jeśli zespół programistów jest zgrany i jednocześnie w zespole stosowany jest jeden dobry standard kodowania, w sytuacji awaryjnej każdy programista może modyfikować moduły pozostałych członków zespołu.

Standardy kodowania, będąc de facto ograniczeniami, budzą zazwyczaj zastrzeżenia programistów, a nawet zaciekłe dyskusje. Styl kodowania jest polem, na którym

można zamanifestować swoją odrębność, zwłaszcza że dyskutowany aspekt sprawy często jest trudny do jednoznacznego rozstrzygnięcia. Niezależnie od tego istnieje - i to zarówno wśród programistów, jak i kierowników zespołów - obawa, że wprowadzenie standardu kodowania ograniczy aktywność i inicjatywę programistów, że ludzie będą się opierali stosowaniu zasad standardu i że w rezultacie skutek będzie odwrotny do oczekiwanych: wydajność programistów zmniejszy się miast się podnieść.

Oczywiście wszystko zależy od standardu. Przy stosowaniu dobrego standardu kodowania ogólna produktywność programistów zazwyczaj wzrasta, jako że powstający kod jest bardziej niezawodny, łatwiejszy w testowaniu i w modyfikowaniu. W rezultacie produkt może być ukończony w krótszym czasie i łatwiej go będzie utrzymywać i rozwijać. Niezależnie od tego aspekt twórczy w programowaniu zazwyczaj jest związany z projektowaniem, a nie z samym kodowaniem, którego konstrukcje są zazwyczaj rutynowe, jeśli nie liczyć różnych tricków i sztuczek programistycznych (stosowane bez wyraźnej potrzeby uchodzą za naganne, jako że zmniejszają czytelność kodu).

## 2. Jak pisać standardy kodowania?

To, czy standard kodowania spełni pokładane w nim nadzieje, zależy od wielu czynników, z których szereg nie jest związanych z samym standardem (o wprowadzaniu standardu kodowania jest mowa na końcu tego punktu). Ale oczywiście sam standard odgrywa tu rolę kluczową. Sposób, w jaki jest napisany i w jaki są sformułowane jego zalecenia, może mieć dla jego akceptacji przez programistów znaczenie zasadnicze.

Oto ogólne uwagi na temat tworzenia standardu kodowania:

- Zasada naczelna: standard powinien mieć takie cechy, jak kod, którego tworzenie normuje: być jasny i czytelny.
- W standardzie powinny być sformułowane ogólne zasady i kryteria, jakimi należy kierować się przy kodowaniu (duch standardu). Nie wszystko da się zawrzeć w przykładach.
- W standardzie należy rozdzielić wyraźnie to, co ma w nim być od tego, co pozostawione jest programistom. Wśród spraw, które mają podlegać standaryzacji, trzeba rozdzielić też to, co jest ujęte i sprecyzowane w standardzie, od tego, czego określenie pozostawiono np. kierownikowi projektu.

W istocie jest to problem zasadniczy. Gdzie postawić granicę? Wprawdzie są elementy stylu kodowania wyraźnie dobre i wyraźnie złe, ale bardzo wiele jest dyskusyjnych. Niektórych elementów stylu w ogóle nie należy standaryzować, bo

a) są nazbyt rozmyte, trudne do ujęcia w precyzyjny zapis (np. złożoność wyrażeń),
b) dotyczą spraw, w których osobiste preferencje programisty mogą być bardzo istotne lub które budzą zbyt duże spory, a zatem ich normowanie jest potencjalnym źródłem konfliktu. Przykładem tego może być styl kodowania nawiasów klamrowych (bywa, że autorzy standardów narzucają tu konkretny styl, uznawany przez nich za jedynie słuszny),

c) nie są warte standaryzacji (np. jednolity sposób inkrementacji lub dekrementacji zmiennych w wyrażeniach niezależnych).
d) są zbyt szczegółowe, by je normować,
e) nie można w praktyce weryfikować ich wprowadzenia (np. zgodność komentarzy z komentowanym kodem).

– Standard powinien być spójny i konsekwentny. Jeśli normuje się jeden z kilku podobnych elementów stylu, należy również unormować pozostałe. Nie może być sytuacji, gdy pisząc drobiazgowo o używaniu spacji (gdzie ją wstawiać i ile), pomija się jednocześnie np. wcięcia w kodzie.

– Należy rozdzielić zalecenia różnego rodzaju. Standard nie powinien być jedną monotonną listą, w której przeplatają się zalecenia o faktycznie różnym charakterze. Czym innym są np. sprawy edycji kodu, a czym innym struktura modułu.

– Tekst opisujący standard nie powinien być za długi, a jego struktura logiczna (rozdziały lub punkty) zbyt złożona. Np. zalecenia dotyczące makroinstrukcji można ująć w jeden punkt. Rozbijanie ich na podpunkty (np. zalecenia ogólne, argumenty makroinstrukcji, stosowanie nawiasów, makroinstrukcje wielozdaniowe) tworzy tekst zbyt skomplikowany.

– Należy unikać sformułowań ostrych, typu "zabrania się", czy "nie wolno". Wprawdzie istnieją zalecenia, które można sformułować w postaci zakazu lub nakazu (np. niedefniowanie zmiennych w plikach definicyjnych czy stosowanie skoków (goto) zawsze w przód). Ale w innych przypadkach stosowanie takich ostrych rygorów byłoby niewskazane. Standard stałby się niespójny, a jego interpretacja mogłaby przerodzić się w przesadnie dokładną analizę litery, a nie ducha standardu.

– Ogólną zasadą przy tworzeniu standardu powinno być: żaden element (zdanie, operator) języka ANSI C nie powinien być w standardzie napiętnowany jako całkowicie zły, a jego stosowanie definitywnie zakazane. Są to integralne elementy standardu ANSI języka C i po coś zostały wprowadzone. Owszem, język C jak każdy inny język programowania może być używany w różny sposób. Nie należy używać elementów języka tylko dlatego, że są. Ale też standard kodowania nie powinien tworzyć nowego języka C--, o mniejszych możliwościach.

– Jeśli istnieje wątpliwość, czy ograniczać stosowanie danej konstrukcji językowej, czy nie, lepiej pozostawić wolną rękę programiście (mówiąc obrazowo, dobrze jest stosować zasadę sądową, iż wątpliwości przemawiają na korzyść oskarżonego).

– Nie należy wymuszać rewolucji (to zalecenie dotyczy też metodyki wprowadzania standardu). Jeśli zespół programistów koduje w pewien sposób, nie należy wprowadzać zasad całkowicie sprzecznych z dotychczasowymi praktykami.

Na zakres i formę standardu może wpływać to, dla kogo jest on tworzony i jakiego rodzaju jest projekt, przy pracy nad którym standard ma być wykorzystywany. Np. jeśli pisze się jądro systemu, którego niezawodność musi być wysoka i które będzie rzadko zmieniane, należy to napisać dobrze i czytelnie, z dokładnym objaśnieniem każdej operacji. Różne fragmenty systemu związane z samą aplikacją bywają kodowane bez takiego nacisku na styl.

Należy też brać pod uwagę, co będzie działo się z kodem napisanym przez programistę. Jeśli - co bywa najbardziej prawdopodobne - programista sam będzie ten kod poprawiał i rozwijał, lepiej dać mu więcej swobody, by ograniczyć jego ewentualne frustracje. Jeśli, przeciwnie, zleca się napisanie modułu osobie z zewnątrz, która z zasady po przetestowaniu napisanego przez nią modułu nie będzie więcej pracować nad projektem, to standard tworzony dla takich osób może nakładać bardziej ścisłe ograniczenia, np. na nazewnictwo, czy komentowanie kodu. W istocie rygorystyczność standardu kodowania zależy od zespołu. Jeśli zespół ten jest duży i duża jest mobilność ludzi, wskazane jest wprowadzić standard bardziej rygorystyczny. Można do niego dołączyć też pewien standard dokumentowania tworzonego kodu.

Oddzielną sprawą jest wdrażanie standardu. Jeśli ma on być użyteczny, musi być przestrzegany. Trzeba zdawać sobie sprawę, że standard kodowania - a przynajmniej niektóre jego zalecenia - będą najpewniej budzić u programistów mniejsze lub większe zastrzeżenia, i jest to zjawisko naturalne. Każdy programista ma inne doświadczenia i nawyki zawodowe. Określenie "człowiek to styl" odnosi się również do programowania, i to zwłaszcza w języku C, w którym programista z zasady ma dużo swobody. Sprawa przestrzegania standardu jest związana w dużej mierze z zastosowaną technologią i organizacją pracy nad projektem, relacjami międzyludzkimi w zespole i w całej firmie, itd. Tu wspomnimy tylko o niektórych sprawach, związanych z samym wprowadzaniem standardu kodowania. Należy go wprowadzać - i ewentualnie dyskutować - na początku pracy nad projektem, jako jeden z elementów technologii pracy, nie zaś w połowie rozwoju projektu. Dobrze jest uczynić programistów w firmie współautorami standardu i konsultować z nimi zarówno ogólne zasady, jak i szczegóły techniczne. Można to zrobić w różny sposób, chociażby w trakcie publicznej dyskusji. Czasami bardzo skutecznym argumentem przemawiającym za koniecznością wprowadzenia standardu kodowania może być niespójność stylistyczna programów pisanych w firmie (trzeba je wcześniej przejrzeć i wychwycić w nich szczególnie rażące różnice między stylami kodowania stosowanymi przez różnych programistów). Przed przystąpieniem do tworzenia standardu dobrze jest sformułować ogólne cele, zasady i kryteria budowania standardu, co do których wszyscy będą zgodni.

## 3. Standard kodowania

W punkcie tym przedstawiony jest standard kodowania programów w języku C (zwany dalej ZASADAMI), opracowany dla potrzeb zespołów programistycznych Instytutu Maszyn Matematycznych w Warszawie.

### 3.1. Cele i zakres stosowania ZASAD

Celami niniejszych ZASAD kodowania programów w języku C jest zwiększenie czytelności tworzonego kodu, jego niezawodności, łatwości modyfikowania i przenośności.

ZASADY dotyczą samego kodowania w języku C (projektowanie i testowanie programu nie są ZASADAMI objęte). U podstaw ZASAD leży założenie, że ich czytelnik zna język C i jego podstawowe konstrukcje i idiomy.

Oprócz zaleceń ogólnych ZASADY obejmują także komentowanie, nazewnictwo, układ kodu, używanie danych i używanie kodu. Opisana jest też struktura, jaką powinien mieć moduł, a także podane pewne zalecenia dotyczące plików definicyjnych. W ZASADACH pominięto obsługę błędów czasu wykonania, jako że sprawa ta zależy w dużej mierze od technik zastosowanych w projekcie.

ZASADY nie obejmują dokumentacji technicznej programu ani nie zakładają, że istnieje jakiś związek między nią, a np. zasadami komentowania programu. Niemniej ZASADY narzucają określony format nagłówka komentarzowego modułu oraz zawierają - jako swoją integralną część - program drukujący nagłówki modułów wchodzących w skład projektu.

W kilku miejscach w tekście ZASAD występują odwołania do kierownika projektu jako instancji decydującej. Oznacza to, że ostateczna decyzja w sprawie, której dotyczy dany punkt, może być podjęta niezależnie od ZASAD.

### 3.2. Zalecenia ogólne

- Zakładaj, że czytelnik twojego programu zna język C i jego podstawowe idiomy.

- Zakładaj, że czytelnik twojego programu nie zna dokładnie aplikacji i może wiedzę o niej czerpać z Twojego kodu.

- Dbaj o prostotę programu. Nie stosuj złożonych konstrukcji bez wyraźnej potrzeby.

- Staraj się, by kod odzwierciedlał Twoje intencje. Nie stosuj konstrukcji, które je ukrywają lub przeinaczają.

- Dbaj o przenośność kodu. Wybieraj konstrukcje, które dają kod przenośny na inne komputery, czy do innych kompilatorów. Tam, gdzie jest wybór, używaj konstrukcji ANSI C (funkcje biblioteczne, makroinstrukcje). Zakładaj, że Twój program będzie działał zawsze w środowisku obsługującym standard ANSI C (nie stosuj konstrukcji używanych przed wprowadzeniem standardu ANSI C, np. kopiowania struktur za pomocą funkcji memcpy).

- Stosuj zasadę decyzji możliwie wysokiego poziomu, tzn. koduj tak, by np. zmiana typu zmiennych zewnętrznych, atrybutu funkcji prywatnej modułu, modelu pamięci, czy stałej symbolicznej mogła być realizowana na poziomie modułu czy projektu.

- Jeśli w konkretnej sytuacji między zaleceniami jest konflikt, ostatecznym kryterium powinna być czytelność kodu.

- Jeśli ZASADY nie regulują stosowania danej konstrukcji, wybierając jej określoną wersję, staraj się zachować ducha ZASAD (czytelność, przenośność, niezawodność kodu). Stosuj ten sam sposób zapisu w całym module.

- O ile kierownik projektu nie zdecyduje inaczej, pisz identyfikatory po angielsku, a komentarze po polsku lub po angielsku. Przy wyborze komentarzy polskich w gestii kierownika projektu pozostaje wybór kodu polskich znaków narodowych (o ile to możliwe, preferowany powinien być kod Latin 2).

- Dbaj o spójność stylistyczną kodu, tj. staraj się stosować wszędzie te same zasady.

## 3.3. Komentowanie

- Każdy moduł powinien mieć nagłówek komentarzowy, w którym powinny znaleźć się:
  - nazwa modułu oraz nazwa i wersja projektu,
  - opis ogólny modułu (jakie operacje realizuje),
  - lista funkcji publicznych,
  - lista zmiennych publicznych (o ile są zdefiniowane w module),
  - lista modyfikowanych zmiennych globalnych,
  - nazwisko autora modułu,
  - data utworzenia modułu,
  - lista dokonanych modyfikacji, z opisem każdej z nich, datą dokonania i nazwiskiem programisty realizującego tę zmianę,
  - kompilator, za pomocą którego kompilowano moduł.

  Konkretna postać nagłówka modułu podana jest w punkcie 3.8.1. Nagłówek może zawierać także dodatkowe elementy, których charakter i miejsce w nagłówku określa kierownik projektu.

- Wszystkie funkcje publiczne w module powinny mieć nagłówki komentarzowe, w których podane są:
  - nazwa funkcji,
  - opis ogólny funkcji,
  - opis argumentów funkcji,
  - opis wartości zwracanej,
  - opis modyfikowanych zmiennych prywatnych modułu oraz zmiennych globalnych, jeśli takie dopuszcza się w projekcie.

- Małe jedno- lub dwuliniowe funkcje prywatne w module, z dobrze dobranymi nazwami funkcji i argumentów, mogą nie mieć nagłówków komentarzowych. Funkcje prywatne większych rozmiarów powinny mieć nagłówki komentarzowe takie, jak funkcje publiczne.

- Komentarze w kodzie powinny być pomocą w zrozumieniu sensu operacji, a nie opisywać je, np.

```
Counter++; /* ZWIĘKSZENIE ZMIENNEJ Counter */  - ŹLE
Counter++; /* KOLEJNY DOKUMENT PRZETWORZONY */ - DOBRZE
```

– Komentarze w kodzie powinny być wyraźnie oddzielone od kodu. Nie należy mieszać kodu i komentarzy (np. przeplatać ze sobą na przemian linie kodu i linie komentarza). Komentarze dotyczące poszczególnych zdań powinny być w miarę możliwości pisane za nimi, w tej samej linii, i justowane pionowo, jednolicie w obrębie bloku lub funkcji.

– Jeśli komentowana ma być grupa zdań, blok komentarzowy należy umieścić na samym początku tej grupy, z wyrównaniem stosownie do komentowanego kodu, np.:

```
...
{
 ...
 {
  /* ------------------------------------------------ |
  | TU NASTĘPUJE OPIS SENSU OPERACJI REALIZOWANYCH    |
  | PRZEZ ZDANIA NASTĘPUJĄCE PONIŻEJ. ZALECA SIĘ,     |
  | BY BLOK KOMENTARZOWY BYŁ OTWARTY OD DOŁU.       */
  ...
 }
}
```

Konkretna postać bloku komentarzowego może być ustalana przez kierownika projektu. W szczególności może to być jedna linia komentarza.

– Nie zagnieżdżaj komentarzy. Jest to nieprzenośne.

– Stosuj klasyczny komentarz /* */, a nie //, nawet jeśli kompilator na to pozwala i nie generuje ostrzeżeń.

– Nie stosuj komentarzy do maskowania fragmentów kodu przed kompilacją. Zamiast tego wyłączaj fragmenty kodu z kompilacji za pomocą dyrektywy warunkowej #ifdef z odpowiednią niezdefiniowaną stałą symboliczną lub z wartością 0

```
#ifdef XXXXXXXX
  /* Tu kod wyłączony z kompilacji */
  ...
#endif
```

### 3.4. Nazewnictwo

– Nie redefiniuj słów kluczowych i operatorów języka.

– Jeśli w projekcie stosuje się atrybuty funkcji lub zmiennych zależne od kompilatora (np. near, far, huge, pascal, fastcall w kompilatorach dla IBM PC), należy je definiować symbolicznie, np.

```
#define USER pascal near
```

i używać tylko tych definicji (patrz p. 3.8, 3.9).

- Nazwy jednoznakowe powinny być kodowane małymi literami i zarezerwowane tylko dla zmiennych lokalnych lub argumentów funkcji. Wskazane jest stosować się do ogólnie przyjętych zasad nadawania zmiennym nazw jednoliterowych. I tak jednoznakowe zmienne zawierające indeksy pętli powinny być nazywane i, j, k; wskaźniki - p, q, r, s, t; znaki - c, d.

- Nazwy funkcji powinny być czasownikowe (np. SearchString lub search_string, ReadFile itp.).

- Nazwy zmiennych powinny być rzeczownikowe (np. Counter, Line).

- Wskazane jest, by długość nazwy identyfikatora była proporcjonalna do jego zasięgu. Dotyczy to zwłaszcza nazw zmiennych. Np. nazwy zmiennych lokalnych w funkcji powinny być w miarę możliwości krótsze od nazw zmiennych prywatnych modułu, czy globalnych projektu.

- Nazwy definiowane za pomocą typedef, #define, enum powinny być kodowane wielkimi literami.

- Nie używaj "liczb magicznych". Poza liczbami 0, 1, -1, resztę stałych w programie definiuj symbolicznie.

- Nie stosuj identycznych nazw o pokrywających się zasięgach. Innymi słowy: nie przesłaniaj zmiennej inną zmienną o tej samej nazwie.

- Staraj się kodować tak, by nazwy o pokrywającym się zasięgu, jakkolwiek syntaktycznie różne, nie brzmiały podobnie (np. Index, Indeks), nie wyglądały podobnie (np. LostWord, LastWord) ani żeby różnica między nimi nie sprowadzała się tylko do wielkości tworzących je liter (np. Counter, counter), lub też do wstawionego znaku podkreślenia (np. Wordcounter, Word_counter).

- Jeśli jest grupa funkcji realizujących podobne operacje i posiadających identyczne argumenty, to nazwy tych argumentów powinny być identyczne (podobne zalecenie odnosi się do zmiennych lokalnych).

- Nie stosuj tej samej nazwy dla obiektów różnej natury, ale o nierozłącznym zasięgu, np. etykiety, etykietki struktury czy zmiennej. Kompilator dopuszcza, by obiekty te miały tę samą nazwę, ale nie należy tego wykorzystywać.

- Zmienne globalne, o ile wprowadzono je do projektu, powinny być łatwo identyfikowalne, np. za pomocą określonego prefiksu, np. GL_ lub dwóch pierwszych liter nazwy projektu (konkretna decyzja pozostaje w gestii kierownika projektu).

- Tam, gdzie istnieje wybór, używaj enum zamiast #define.

- Wybór i sprecyzowanie konwencji nazewniczej są pozostawione do decyzji kierownika projektu (czy zaczynać nazwę z wielkiej litery, czy słowa w nazwie oddzielać podkreśleniami, etc.). Według konwencji przyjętej w ZASADACH domyślnie, wszystkie nazwy liczące co najmniej 2 znaki należy zaczynać wielką literą. Jeśli nazwa składa się z kilku wyrazów lub skrótów, należy łączyć je bez podkreśleń i każde słowo zaczynać z wielkiej litery, np. ClearArray, EmpNumber.

- Wprowadzenie prefiksów semantycznych (odmiana notacji węgierskiej), a także zakres ich stosowania, jest również pozostawione w gestii kierownika projektu.

### 3.5. Układ kodu

- Należy stosować adjustację pionową kodu - wszystkie zdania należące do tego samego bloku powinny zaczynać się w tej samej kolumnie.

- Wcięcia w kodzie powinny być jednolite, np. po 2 znaki (dobrze jest, jeśli jest to ustalone na poziomie projektu).

- Staraj się, by linia kodu nie przekraczała 80 znaków, razem z miejscami pustymi, tak by mieściła się w całości na standardowym ekranie znakowym 24x80.

- Zdania wykonywane w obrębie jednej etykiety case oddzielaj od zdań odpowiadających innym etykietom case linią pustą, np.:

```
case PUGUP:
  PageUp();
  break;

case ESCAPE:
  ...
```

Wyjątkiem są przypadki, gdy między dwiema etykietami case nie ma zdania break.

- Staraj się kodować jedną operację w linii. Odstępstwa od tej zasady stosuj rzadko i tylko w sytuacji, gdy zdania umieszczane w jednej linii nie są trudne do odczytania, np. w przypadku prostych operacji w etykietach case:

```
case PGUP: PgUpFun(); break;
case ESCAPE: EndFunc(); break;
```

- Łamanie zarówno wywołań funkcji, jak i wyrażeń arytmetycznych, powinno być jednolite. Jest ono w zasadzie pozostawione w gestii programisty, chyba że kierownik projektu narzuci tu określone zasady. Domyślnie zalecane jest, by wcięcie przy łamaniu zdań było zawsze takie samo i równe wielkości standardowego wcięcia (np. 2 znaki).

```
func(a,b,c,
 x,y,z);
```

– Wyrażenia arytmetyczne należy - o ile to możliwe - łamać według operatora o najmniejszej hierarchii (wcięcie linii przeniesionej powinno być standardowe). Operator powinien zostawać w linii przenoszonej, np.

```
Lines = NumberOfPars * ParsLength +
 HeadersLines;
```

– Stałe ciągi znaków, tj. ujęte w podwójne cudzysłowy, nie powinny być przenoszone, jako że może to psuć układ kodu. Lepiej pisać je w postaci oddzielnych i skończonych ciągów, sklejanych przez kompilator, np.

```
printf("Tekst ten jest bardzo długi. Dlatego "
 "zostanie pocięty na trzy ciągi różnej "
 "długości");
```

– Unikaj kodowania jednej nieprzerwanej kolumny kilkunastu czy kilkudziesięciu zdań w funkcji. Staraj się ją rozdzielać pustymi liniami, stosując np. zasadę siedmiu, tj. ograniczając długość ciągłej kolumny zdań do 7 (z marginesem 2 linii w obie strony).

– Deklaracje zmiennych lokalnych w funkcji powinny być oddzielone od kodu wykonywalnego pustą linią.

– Ciała zdań if, for, while, do-while ujmuj w nawiasy klamrowe także wtedy, gdy składają się tylko z jednego zdania, np.

```
if (x > y)
{
 Func(x, y);
}
```

– Wybór stylu nawiasów klamrowych jest w zasadzie pozostawiony programiście. Jeśli programista nie jest przywiązany do określonego stylu, wskazane jest kodować nawiasy klamrowe w stylu Allmana (jest on stosowany w tym tekście). Dwoma wyjątkami, w których należy stosować określony styl są a) same funkcje (styl Allmana) oraz while w zdaniu do-while, które powinno być poprzedzone nawiasem klamrowym:

```
do
{
 ...
} while (Expression);
```

– Operatorów jednoargumentowych nie oddzielaj spacjami od ich argumentów, np. a++, *p, &Counter, !Flag. Operatorów dwuargumentowych struktury ., -> nie rozdzielaj od identyfikatorów znajdujących się po ich obydwu stronach (podobnie z lewostronnym nawiasem kwadratowym). Pozostałe operatory dwuargumentowe rozdzielaj spacjami od ich argumentów, z wyjątkiem sytuacji, gdy wymóg czytelności może sugerować inny zapis. Np.

```
x = a * b;
x = a*b + c;

if (a + x*y < b - f*g)
{
  a++;
}

while((c=getchar()) != EOF)
{
 ...
```

- Opcje edycji, a zwłaszcza ustawienia tabulatorów, powinny być ustalane na poziomie projektu.

### 3.6. Używanie danych

- Minimalizuj zasięg danych.

- Unikaj stosowania zmiennych globalnych. Jeśli wprowadzenie ich jest już konieczne, powinny one być definiowane na poziomie projektu w oddzielnym pliku źródłowym, i mieć oddzielny plik definicyjny (jego nazwa powinna go wyróżniać, np. być skrótem nazwy projektu z dodaną końcówką GLB, np. VIRGLB.C).

- Zamiast czynić zmienne globalnymi, ukrywaj je (czyń je prywatnymi w module) i dostęp do nich realizuj za pomocą odpowiednich funkcji.

- Staraj się nie używać struktur bitowych, chyba że aplikacja to wymusza (np. przetwarzane są rekordy zapisane na bitach). Jeśli musisz ustawiać poszczególne bity jako sygnalizatory określonych stanów programu, używaj do tego zmiennych integralnych typu unsigned int lub unsigned long.

- Jeśli istnieje wybór, staraj się preferować notację indeksową (tablice) nad wskaźnikową.

- Definiując struktury lub unie, zamiast korzystania z etykietki definiuj typ za pomocą typedef.

- Stosuj domyślnie typ int dla zmiennych liczbowych (typ o mniejszej długości, np. char, stosuj tylko wtedy, gdy trzeba oszczędzić pamięć, a long - gdy zakres wartości zmiennej wyraźnie tego wymaga).

- Klasę pamięci register stosuj tylko w deklaracjach zmiennych typu int.

- Nie używaj unii w sposób nieprzenośny, np. do uzyskiwania dostępu do różnych bajtów zmiennych integralnych.

- Nie zakładaj nic o wewnętrznym uporządkowaniu bajtów w słowie.

- Nie zakładaj nic o ułożeniu składowych w strukturze. Jeśli w programie potrzebna jest wartość przesunięcia składowej w strukturze, używaj standardowej makroinstrukcji offsetof.

- Gdy w wyrażeniu porównywane są lub przypisywane wskaźniki różnego typu, zawsze stosuj odpowiednie rzuty. Staraj się zawsze rzutować NULL na właściwy typ wskaźnika, nawet jeśli jest to typ void *.

- Jeśli dany argument w funkcji ma nie być modyfikowany, opatruj go kwalifikatorem const.

- O ile to możliwe, używaj zmiennych przekazywanych w wywołaniu funkcji jako zmiennych roboczych.

- O ile wśród argumentów funkcji wydzielone są wyraźnie dane wejściowe i/lub wyjściowe, pamiętaj o zasadzie, że dane wejściowe dla funkcji powinny być przez nią tylko czytane, a dane wyjściowe - tylko pisane.

## 3.7. Używanie kodu

### 3.7.1. Zalecenia ogólne

- Unikaj głębokich zagnieżdżeń kodu (staraj się nie przekraczać trzech). Głębsze zagnieżdżenia stosuj tylko w sytuacjach, gdy wynika to w sposób naturalny z samego algorytmu, np. przy operacjach na tablicach wielowymiarowych.

- Zdanie goto staraj się stosować tylko do skoków w przód, i tylko do wyjścia z pętli w sytuacji błędnej lub nienormalnej.

- Unikaj sytuacji mogących dać złe skutki uboczne np.:
  - użycie w jednym wyrażeniu, przed punktem sekwencji, zmiennej i wskaźnika do niej (np. jako argumentu w wywołaniu fukcji),
  - użycie w jednym wyrażeniu, przed punktem sekwencji, zmiennej bez i z inkrementacją (dekrementacją),
  - zakładanie określonej kolejności oszacowania argumentów w wywołaniu funkcji.

- Nie optymalizuj operacji przez zmianę operatorów (np. nie realizuj dzielenia argumentu przez potęgę liczby 2 poprzez przesuwanie jego zawartości za pomocą operatora >>). Uogólniając: staraj się nie optymalizować poszczególnych konstrukcji wprost, jeśli powstający kod jest sprzeczny z ZASADAMI. Stosuj zasadę informowania kompilatora o zamierzeniach (np. stosowanie klasy register, używanie rzutu (unsigned) dla zmiennych signed, które są dzielone przez potęgę liczby 2, itp.).

- Nie porównuj (operator ==) zmiennych typu zmiennoprzecinkowego (używaj tylko operatorów >, <, <=, >=).

- Unikaj wyrażeń warunkowych zawierających wiele operatorów || i &&, nawet ujętych w nawiasy.

- Staraj się nie stosować zbędnych nawiasów zwykłych. Wstawiaj je tam, gdzie:
  - ich brak powoduje pojawienie się ostrzeżeń kompilatora,
  - przy ich braku musisz zaglądać do tablicy hierarchii operatorów, by stwierdzić, czy wyrażenie jest poprawne.

- Staraj się nie mieszać operatorów, np. binarnych i arytmetycznych. Przy mieszaniu operatorów należy stosować nawiasy, nawet jeśli kompilator nie generuje ostrzeżeń, np.

```
x = (x + y) ^ z;
```

- Bierz pod uwagę nadmiar, jaki może się pojawić w wyrażeniach arytmetycznych (używaj stosownych rzutów zabezpieczających przed nadmiarem).

- Eliminuj repetycję wyrażeń. Stosuj operatory skrótowe (+=, -=, itd.). Jeśli w kodzie powtarzają się kilkakrotnie długie wyrażenia, zdefiniuj je jako nowe wyrażenie za pomocą dyrektywy #define lub wprowadź wskaźnik, np.

```
char *p = Window->Text->Par->Line;
```

- W konstrukcji if-else warunek spełniony częściej wstawiaj do if. Jeśli obydwa warunki są równouprawnione, wstawiaj do if ten warunek, któremu odpowiadający kod jest krótszy.

- Tam, gdzie jest wybór, preferuj rekursje nad iteracje.

- Minimalizuj użycie operatora przecinka (stosuj go tylko tam, gdzie jest wyraźnie przydatny, np. w częściach inicjującej i przyrostowej pętli for, jeśli wymagają one więcej niż jednego wyrażenia), np.:

```
for(i=0, j=0; Func(i,j); i++, j++)
{
  ...
```

- Używając w kodzie zmiennych symbolicznych, staraj się w miarę możliwości, by w wyrażeniach nie pojawiały się ze stałymi addytywnymi (np. w wyrażeniu limitującym pętli for stosuj MAX_VALUE, a nie MAX_VALUE-1).

- Minimalizuj użycie wyrażeń warunkowych, np. nie używaj ich w wywołaniach funkcji. Nie zagnieżdżaj wyrażeń warunkowych.

- Kody opatrzone etykietami domyślnymi case (zdanie switch) staraj się w miarę możliwości zawsze czynić niezależnymi (wstawiaj po każdej etykiecie break). "Opadanie" sterowania między etykietami case realizuj tylko w sytuacjach, w których dla wszystkich etykiet wykonywane są te same operacje, np.:

```
case F1:
case ENTER:
case ESCAPE:
 Func();
 return(k)

/* DOBRZE */
```

```
case F1:
case ENTER:
  Func1();
  Func2();
case ESCAPE:
  return(k)

/* ŹLE */
```

### 3.7.2. Makroinstrukcje

- Jeśli makroinstrukcje mają zawierać kilka zdań języka, ujmuj je w blok do-while(0).

- Makroinstrukcje własne powinny tylko raz oszacowywać argument. Odstępstwa od tej zasady należy wyraźnie opisać w komentarzu przy definicji makroinstrukcji.

- Jeśli makroinstrukcja jest wyrażeniem, bierz w nawiasy całą makroinstrukcję i każdy z jej argumentów z osobna.

- Jeśli makroinstrukcja jest wyrażeniem, używaj w niej operatora potrójnego ?:, a jeśli zdaniem lub zdaniami - konstrukcji if-else.

- Staraj się, by wywołanie makroinstrukcji zawsze wymagało zakończenia jej średnikiem.

- Dla celów testowania programu używaj makroinstrukcji uaktywnianych/maskowanych za pomocą stałych symbolicznych (np. assert).

### 3.7.3. Funkcje

- Rozmiar funkcji nie powinien przekraczać kilkudziesięciu linii (np. 60), łącznie z nagłówkiem komentarzowym (jedna strona wydruku). Funkcje o większym rozmiarze powinny być tworzone tylko w przypadkach wyjątkowych, gdy rozmiar taki jest naturalnym odbiciem algorytmu (np. funkcje ze zdaniem switch zawierającym wiele etykiet case).

- Każda funkcja powinna mieć prototyp. W prototypie funkcji, oprócz typu jej i jej argumentów, należy podawać także nazwy argumentów. Jeśli funkcja nie ma argumentów, pisz argument pusty void.

- W definicji funkcji zawsze podawaj explicite jej typ, nawet w przypadku funkcji typu int. Jeśli funkcja nie zwraca wartości, zaznaczaj to pustą wartością void.

- Nie pisz funkcji pasożytujących tj. takich, których działanie zależy od wewnętrznego sposobu działania innych funkcji.

- Liczba argumentów w funkcji powinna być ograniczona do kilku (np. do 6). Przekraczaj tę liczbę tylko w wyjątkowych przypadkach.

- W wywołaniach funkcji zawsze dbaj o to, by typy argumentów formalnych i aktualnych pokrywały się. Nie polegaj na promocji typów realizowanej przez kompilator - stosuj odpowiednie rzuty.

- Nie stosuj metody zwracania danych przez funkcję w jej buforach statycznych. Staraj się również nie przekazywać danych w buforach zdefiniowanych zewnętrznie.

- Staraj się w miarę możliwości nie mieszać - a przynajmniej ograniczać mieszanie - sensu wartości zwracanych przez funkcję (np. część wartości liczby całkowitej zwracanej przez funkcję reprezentuje wynik jej działania, a niektóre wybrane informują o błędzie). Staraj się ograniczać takie sytuacje do przypadków klasycznych (wskaźnik NULL, wartość całkowita EOF).

- Staraj się, by funkcja realizowała tylko jedną operację, ale za to realizowała ją w sposób pełny i skończony. Jeśli funkcja nie jest w stanie jej wykonać, niech w miarę możliwości anuluje skutki swojej działalności (np. niech zwolni alokowaną pamięć).

- Tam, gdzie jest wybór, używaj standardowych funkcji bibliotecznych i makroinstrukcji ANSI C.

- W funkcjach ze zmienną liczbą argumentów używaj standardowych makroinstrukcji inicjowania i pobierania argumentów. Unikaj używania argumentów o typach, których rozmiar nie jest wielokrotnością słowa maszyny, np. unsigned char.

### 3.8. Struktura i układ modułu

- Minimalizuj połączenia między modułami. Jako interfejs powinny być stosowane tylko funkcje publiczne z argumentami i zmienne globalne (także potoki, o ile środowisko na to pozwala).

- Funkcji publicznych powinno być w module możliwie mało.

- Wszystkie funkcje w module, które nie mają być publiczne, powinny być prywatne (mieć klasę pamięci static).

- Funkcje prywatne modułu nie powinny wywoływać jego funkcji publicznych.
- Układ modułu powinien być następujący:
  - nagłówek komentarzowy modułu,
  - lokalne definicje sterujące kompilacji,
  - włączane pliki definicyjne:
    - pliki systemowe (nazwy ujęte w nawiasy <>),
    - pliki własne projektu (nazwy ujęte w cudzysłowy),
  - deklaracje i definicje wewnętrzne:
    - definicje danych publicznych,
    - definicje danych prywatnych,
    - deklaracje funkcji prywatnych,
  - definicje funkcji
    - definicje funkcji publicznych,
    - definicje funkcji prywatnych.

  W obrębie każdego rodzaju deklaracji i definicji powinien być w miarę możliwości stosowany porządek alfabetyczny. Każda kolejna część modułu powinna być wyraźnie oddzielona od poprzedniej linią komentarzową. Jeśli danej części w module brak, zaznaczaj to linią komentarzową.

- Deklaracje funkcji i danych publikowanych przez moduł powinny znajdować się w pliku definicyjnym modułu.

- Każda funkcja powinna być oddzielona od poprzedniej co najmniej dwiema pustymi liniami.

- Rozmiar modułu w miarę możliwości nie powinien przekraczać 1000 linii, łącznie z nagłówkami, komentarzami i liniami pustymi.

- Nie stosuj arbitralnych atrybutów dla funkcji w module (patrz p.3.4). To, jakie atrybuty mają funkcje publiczne, a jakie prywatne, jest zależne od wymogów projektu. Ich typ powinien być zdefiniowany w pliku definicyjnym projektu (decyzja w tej sprawie należy do kierownika projektu).

### 3.8.1. Szkielet modułu

- Wszystkie moduły w projekcie powinny mieć znormalizowany nagłówek i strukturę, zgodną z opisaną wyżej. W projekcie powinien istnieć szkielet modułu, który programista wypełnia treścią (deklaracjami i definicjami funkcji i zmiennych). Konkretna postać szkieletu modułu jest określana przez kierownika projektu. Poniżej podana jest postać przykładowa:

```
/*-------------------------------------------------------|
| MODUŁ: WORDSEL.C                                      |
| PROJEKT: WORDS II (V23)                               |
|-------------------------------------------------------|
```

```
| OPIS: MODUŁ EKSTRAKCJI SŁÓW KLUCZOWYCH                          |
| ---------------------------------------------------------------- |
| FUNKCJE PUBLICZNE:                                               |
| - OpenKeywordFile(char FileName)                                 |
| - GetKeyword(char *Buffer)                                       |
| - CloseKeywordFile(void)                                         |
| ---------------------------------------------------------------- |
| ZMIENNE PUBLICZNE: BRAK                                          |
| ---------------------------------------------------------------- |
| WYWOŁYWANE FUNKCJE GLOBALNE:                                     |
| - CheckGarbage                                                   |
| ---------------------------------------------------------------- |
| UTWORZONY:    10-12-1994                                         |
| AUTOR: M.KOTOWSKI                                                |
| ---------------------------------------------------------------- |
| MODYFIKACJA 1:  5-1-1995                                         |
| AUTOR: M.KOTOWSKI                                                |
| OPIS: FUNKCJA GetKeyword ZOSTAŁA POSZERZONA O OPCJĘ               |
|           DWUSŁOWOWĄ - WYBIERANE SĄ TAKŻE HASŁA SKŁA-            |
|           DAJĄCE SIĘ Z DWÓCH SŁÓW                                |
| ---------------------------------------------------------------- |
| KOMPILATOR: BORLAND C 3.1.                                       |
| ---------------------------------------------------------------- */

/* ---------------------------------------------------------------- |
| 1.      LOKALNE DEFINICJE STERUJĄCE KOMPILACJI                   |
| ---------------------------------------------------------------- */
...

/* ---------------------------------------------------------------- |
| 2.       WŁĄCZANE SYSTEMOWE PLIKI DEFINICYJNE                    |
| ---------------------------------------------------------------- */
...

/* ---------------------------------------------------------------- |
| 3.        WŁĄCZANE PLIKI DEFINICYJNE PROJEKTU                    |
| ---------------------------------------------------------------- */
...

/* ---------------------------------------------------------------- |
| 4.           DEFINICJE DANYCH PUBLICZNYCH                        |
| ---------------------------------------------------------------- */
...
```

```
/*-------------------------------------------------------|
| 5.          DEFINICJE DANYCH PRYWATNYCH               |
|-------------------------------------------------------*/
...

/*-------------------------------------------------------|
| 6.          DEKLARACJE FUNKCJI PRYWATNYCH             |
|-------------------------------------------------------*/
...

/*-------------------------------------------------------|
| 7.          DEFINICJE FUNKCJI PUBLICZNYCH             |
|-------------------------------------------------------*/
...

/*-------------------------------------------------------|
| 8.          DEFINICJE FUNKCJI PRYWATNYCH              |
|-------------------------------------------------------*/
...
```

– Zakłada się, że nagłówek znajduje się zawsze na początku modułu i jest łatwy do wydzielenia (wiadomo dokładnie, gdzie jest jego początek i gdzie koniec). Zakłada się również, że w projekcie istnieje program dokumentacyjny DOCUMOD.C, który pobierając z pliku znakowego listę modułów wchodzących w skład projektu, wczytuje z każdego z nich nagłówek i drukuje go, tworząc w ten sposób nagłówkową dokumentację modułów tworzących projekt. Kod źródłowy najprostszej wersji programu DOCUMOD.C jest publicznie dostępny (działa on tylko na plikach źródłowych C). Kierownik projektu może utworzyć jego nową wersję, poszerzając jego możliwości np. o wypisywanie nazwisk autorów modułów, przeszukiwanie modułów modyfikowanych po określonej dacie, przez określoną osobę, itd.

## 3.9. Pliki definicyjne

– Parametry określające wersje projektu powinny znajdować się w wydzielonym pliku definicyjnym o nazwie np. XXXver.h, a podstawowe typy ogólne (np. BYTE, WORD, typy funkcji prywatnych, itp.) - w innym pliku o nazwie np. XXXtyp.h (XXX oznacza trzy pierwsze litery projektu, np. dla projektu o nazwie VIRT nazwami takimi są VIRVER.H i VIRTYP.H).

– Z każdym modułem powinien być związany jeden plik definicyjny. Jego nazwa powinna być nazwą modułu lub jej skrótem, a rozszerzenie - literą H. Długość nazwy pliku definicyjnego w miarę możliwości nie powinna przekraczać 6 znaków.

– W pliku definicyjnym modułu powinny znajdować się tylko deklaracje i makrodefinicje publikowane (makrodefinicje i deklaracje prywatne modułu powinny być

umieszczone w nim samym). Przy czym plik ten powinien być włączony także do modułu, z którym jest związany, jako ostatni z włączanych własnych plików definicyjnych projektu.

- Do pliku definicyjnego nie wolno wprowadzać definicji zmiennych czy funkcji (innymi słowy: plik definicyjny nie może powodować alokowania pamięci).

- Plik definicyjny modułu powinien być związany tylko z jednym modułem i - o ile to możliwe - nie wymagać żadnych innych plików definicyjnych. Jeśli do poprawnej analizy przez kompilator zawartości pliku definicyjnego niezbędne jest włączenie innych plików definicyjnych projektu, to kwestia, czy pliki te włączać do modułu za pomocą zdań #include, czy też do owego pliku definicyjnego jako zagnieżdżone zdania #include, pozostaje w gestii kierownika projektu.

- Zawartość pliku definicyjnego powinna być w miarę możliwości pogrupowana logicznie: obiekty podobnego rodzaju (definicje struktur, stałych symbolicznych) i prototypy funkcji realizujących podobne operacje, powinny być grupowane razem.

- W nazwie pliku włączanego za pomocą dyrektywy #include nie należy podawać katalogów (wyjątkiem mogą być katalogi względem katalogu bieżącego projektu).

- Definicje stałych informujących o włączeniu danego pliku definicyjnego do kodu powinny mieć ten sam format, np. wszystkie składać się z nazwy pliku definicyjnego zakończonej przyrostkiem _H (stała taka dla pliku FILEOP.H ma postać FILEOP_H).

## 3.10. ZASADY kodowania a projekt programistyczny

Opisane ZASADY kodowania dotyczą samego języka C. Przy pracy w konkretnym środowisku musi być branych pod uwagę także szereg innych elementów jak np.

- rodzaj i wersja kompilatora,
- poziom ostrzeżeń kompilatora,
- model pamięci (o ile się stosuje),
- znaki zmiennych znakowych,
- rodzaj procesora,
- zakres optymalizacji,
- rozmiar stosu.

Sprawy te muszą być w projekcie określone. Łącznie z parametrami środowiska oraz z precyzyjnymi ustaleniami odnoszącymi się do punktów ZASAD, w których decyzję pozostawiono kierownikowi projektu, tworzą one standard lokalny projektu. Jego konkretny zakres (w jakich sprawach podjęto decyzje, a w jakich pozostawiono wolną rękę programistom) i forma są sprawą wewnętrzną projektu.

## 4. Wnioski

Cechą charakterystyczną przedstawionego standardu kodowania jest pozostawienie w nim stosunkowo wielu spraw otwartych. Rozstrzygnięcie części z nich scedowano na kierownika projektu. W innych sprawach standard, sugerując pewne konkretne rozwiązania, ostateczną decyzję pozostawia programiście (dotyczy to m.in. stylu stosowania nawiasów klamrowych). Nic nie zostało zabronione (teoretycznie kierownik projektu mógłby podnieść wprawdzie niektóre zalecenia do rangi kategorycznych zakazów lub nakazów, ale byłoby to sprzeczne z duchem standardu). Intencją autorów ZASAD było uniknięcie kontrowersji, jakie mogłyby pojawić się na skutek przyjęcia z góry takich czy innych wersji niektórych zaleceń. Niezależnie od tego, jak i przez kogo uregulowane zostaną owe kontrowersyjne elementy stylu kodowania, sam fakt, że zostały w standardzie wymienione, wskazuje na ich wagę w kodowaniu i konieczność ich standaryzacji.

Zastosowane w ZASADACH rozbicie zaleceń kodowania na główne i lokalne, związane z konkretnym projektem, zwiększa elastyczność stosowania standardu i czyni go bardziej niezależnym od konkretnego środowiska. Ale ma i wadę: mnoży byty. Programista musi stosować się nie do jednego, ale do dwóch standardów. Przy dobrej organizacji projektu standard lokalny może być mały i sprowadzać się do kilku definicji, nie licząc ewentualnie wprowadzenia jednolitego pliku konfiguracyjnego projektu.

Na koniec słowo o reakcjach ludzkich. Przedstawione ZASADY były dyskutowane w zespole programistów, którzy mieli później ich przestrzegać. Na podstawie tych dyskusji ZASADY poddano pewnym zmianom, korektom i uzupełnieniom (wersja przedstawiona wyżej jest wersją finalną). Pouczająca była różnorodność zgłaszanych zastrzeżeń. Większość uwag zgłaszanych przez każdego z programistów nie pokrywała się z uwagami zgłaszanymi przez pozostałych. Największa zgoda panowała w krytyce wszelkich ograniczeń liczbowych, np. długości funkcji, liczby zagnieżdżeń, wielkości modułu. Wprawdzie przyznawano, że i tak rzadko się je przekracza, niemniej - twierdzono - takie formalne ograniczenia mogą być krępujące. Kilku programistów miało zastrzeżenia do pewnego ograniczenia stosowania skoków niejawnych, break i continue (sugestia taka znajdowała się w pierwotnej wersji ZASAD i została usunięta). Zastrzeżenia budziło też zalecenie, by długość nazw wiązać z ich zasięgiem (zalecenie to zostało w ZASADACH złagodzone). Również ustalenie konkretnych jednolitych reguł tworzenia nazw okazało się nader trudne i w końcu uznano za najrozsądniejsze pozostawić decyzję w tej sprawie kierownikowi projektu.

W dyskusji dość wyraźnie odnalazły się doświadczenia zawodowe programistów. Osoby, które pracowały nad dużymi programami, tworzonymi przez zespoły programistów i przenoszonymi do różnych środowisk, bardziej zwracały uwagę na sprawy ogólniejsze: np. organizację plików definicyjnych i ich wzajemne powiązania czy dokumentowanie plików źródłowych. Z kolei programiści, którzy większość programów pisali dla środowiska WINDOWS, krytykowali te zasady, których przestrzeganie w tym środowisku jest szczególnie kłopotliwe (np. zalecenia co do ograniczenia długości ciągłej kolumny zdań języka czy długości funkcji), oraz postulowali, by wprowadzić notację węgierską jako obligatoryjną.

Ujawniły się też wyraźnie preferencje osobiste. Ujmowanie bloku składającego się z jednego zdania w nawiasy klamrowe jedna osoba potraktowała jako przesadę i zbędne poszerzanie pliku źródłowego, podczas gdy inna uznała to za praktykę szczególnie

zalecaną, chroniącą przed popełnianiem subtelnych i trudnych do wykrycia błędów. Duże różnice występowały w opiniach na temat stosowania spacji w wywołaniach funkcji i w wyrażeniach, a także sposobów przenoszenia wyrażeń (zarówno wersja z operatorem arytmetycznym pozostawianym jak i przenoszonym znalazły swoich zwolenników). Jedna osoba uznała za wskazane narzucić wykorzystywanie w edycji znaków tabulacji, inna przeciwnie - sugerowała stosowanie spacji, itd.

Swoistym paradoksem jest fakt, że przy ostrych protestach na temat ograniczeń liczbowych, zgodzono się, więcej: zasugerowano nawet, by włączyć do ZASAD jako ich integralny element inne ograniczenie: szkielet modułu, określający dokładnie, co i w jakiej kolejności znajduje się w module (w pierwotnej wersji ZASAD nie było go). Zważywszy, że programiści wypowiadający się na temat ZASAD posiadali dość duże i różnorodne doświadczenie w programowaniu w języku C, można uznać to za jeszcze jeden dowód znaczenia właściwej struktury modułów dla całego projektu.

## Literatura:

[1] AT&T Indian Hill Coding Standards - Dostępny w Internecie, Komputer: cs.washington.edu, plik: pub/cstyle.tar.Z.

[2] D.Straker - C Style: Standards and Guidelines, Appendix A: Heledd's Company Coding Standards, Prentice Hall, Englewood Cliffs, 1992.

[3] H.Ballay, R.Storn - A Tool for Checking C Coding Conventions, C/C++ Users Journal, July 1994.

LUCJUSZ SZTEN
ROMAN NINARD
INSTYTUT KOMPUTEROWYCH SYSTEMÓW
AUTOMATYKI I POMIARÓW WROCŁAW

# Mikrokomputerowy system dyspozytorski MSD-BUSZ
# Microcomputer Dispatcher System MSD-Busz

## Streszczenie

Mikrokomputerowy system dyspozytorski (MSD-Busz) przeznaczony jest do wizualizacji stanu sieci trakcyjnej (kolejowej i komunikacji miejskiej) oraz zdalnego sterowania jej poszczególnymi urządzeniami. W artykule przedstawiono opis techniczny systemu wraz ze schematem blokowym, jego działanie, a także strukturę oprogramowania.

## Abstract

Microcomputer Dispatcher System (MSD-Busz) provides a tool for graphical visualization of the state of - both railway and municipal transport services - traction network as well as for remote control the network elements. The paper presents specification of the system including its block diagram, operation and software structure.

## 1. Wstęp

Opracowany w Instytucie Komputerowych Systemów Automatyki i Pomiarów we Wrocławiu, wdrożony w Dolnośląskiej DOKP. Mikrokomputerowy System Dyspozytorski (MSD-BUSZ) przeznaczony jest do monitorowej wizualizacji stanu sieci trakcyjnej oraz zdalnego sterowania poszczególnymi urządzeniami wchodzącymi w jej skład.

System ten zrealizowany w ramach modernizacji Centralnej Nastawni współpracuje z 87-ma urządzeniami stacyjnymi, rozłożonymi wzdłuż 8-miu linii kolejowych w promieniu od 30 do 100 km od Wrocławia.

Kolejowa sieć trakcyjna, zasilana napięciem stałym 4 kV, podzielona jest na odcinki, które zasilane są z indywidualnych zasilaczy umieszczonych na podstacjach trakcyjnych. Wyjścia poszczególnych zasilaczy posiadają zabezpieczenia wyłączające je w przypadku przekroczenia założonej wielkości natężenia prądu. Wszystkie urządzenia podstacji trakcyjnej są pod kontrolą sterownika BUSZ, który cyklicznie przekazuje informacje o ich stanach do Centralnej Nastawni (CN), poprzez łącze telegraficzne, za pomocą urządzeń telegrafii wielokrotnej (TgFM).

Poszczególne odcinki sieci trakcyjnej lub ich części (sekcje) można z sobą łączyć i rozłączać, przełączać ich źródła zasilania z jednej podstacji na drugą lub wyłączać zasilanie poszczególnych sekcji. Konfiguracja sieci trakcyjnej dokonywana jest za pomocą zdalnie sterowanych odłączników. Urządzenia sterujące poszczególnymi grupami odłączników (umieszczone na terenie stacji kolejowych lub podstacji trakcyjnych) kontrolowane są przez analogiczne sterowniki jak urządzenia podstacji trakcyjnych.

Sterowniki BUSZ, umieszczone na stacjach i podstacjach trakcyjnych, przyjmują z CN, za pośrednictwem urządzeń TgFM, polecenia zmiany stanu poszczególnych urządzeń (ZAŁĄCZ lub WYŁĄCZ), realizują je i informują CN o rezultatach wykonania polecenia.

## 2. Opis techniczny

W skład MSD wchodzą następujące bloki funkcjonalne:
- Stacja dyspozytorska I,
- Stacja dyspozytorska II,
- Stacja serwisowa,
- Blok separatorów wejściowych,
- Nadajnik poleceń.

Stacje dyspozytorskie i stacja serwisowa składają się z części centralnej i części peryferyjnej. Urządzenia części centralnej Systemu oraz urządzenia pośredniczące umieszczone są w szafie. Aparatura peryferyjna, tj. monitory kolorowe, terminale, manipulatory i drukarki, znajdują się z na poszczególnych stanowiskach dyspozytorskich.

Część centralna każdej stacji zabudowana w oddzielnej kasecie wyposażonej we własny zasilacz sieciowy stanowi odrębny zestaw mikrokomputerowy.

Z urządzeniami peryferyjnymi oraz z pozostałymi stacjami połączone jest poprzez złącza RS (monitory poprzez kable RGS).

W skład stacji dyspozytorskiej wchodzą:
- Moduł procesora VSBC-2 wyposażony w sześć portów transmisji szeregowej RS-232, pamięć RAM o pojemności 1 MB z własnym buforowym zasilaniem bateryjnym, pamięć ROM o pojemności 512 kB, zegar czasu rzeczywistego,
- Moduł pamięci VMEM-S1 o pojemności 2 MB,
- Moduł zasilania bateryjnego VBAT2 zapewniający podtrzymanie zawartości w module pamięci przy braku zasilania sieciowego,
- 2 Moduły sterownika monitora graficznego BGPM umożliwiające sterowanie kolorowymi monitorami o rozdzielczości 128x1024,
- 3 Moduły wejść dwustanowych VIOP zawierające po 32 kanały wejściowe oraz lokalny mikroprocesor,
- Moduł zasilacza kasety SVE-3 zasilający wszystkie moduły umieszczone w kasecie.

Stacja serwisowa zawiera następujące moduły:
- moduł procesora VSBC-2,
- moduł pamięci VMEM-S1,
- moduł zasilania bateryjnego VBAT2,

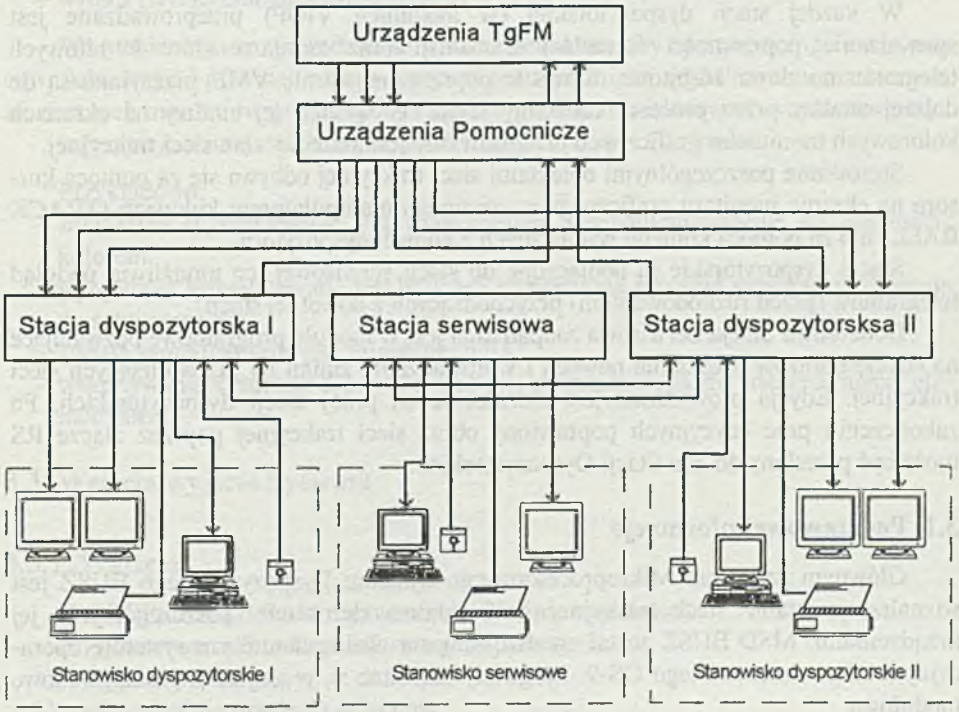
- moduł sterownika monitora graficznego VGPM,
- moduł sterownika napędu pamięci dyskowej,
- napęd pamięci dyskowej 3,5",
- moduł zasilacza SVE-3.

Kaseta z urządzeniami pomocniczymi zawiera:

- sześć modułów separatorów sygnałów dwustanowych SIP-1 każdy po 32 kanały separujące,
- moduł nadajnika poleceń INP-01 wyposażony w dwa niezależne kanały transmisyjne do TgFM sterowane lokalnym mikroprocesorem,
- zasilacz MPS-7OEU-3/1.

## 3. Opis działania Systemu

Schemat blokowy MSD-Busz pokazany na rys. 1 przedstawia wzajemne połączenia pomiędzy poszczególnymi stacjami Systemu oraz pomiędzy Systemem a urządzeniami TgFM.




Rys. 1. Schemat blokowy SYSTEMU DYSPOZYTORSKIEGO MSD-Busz

Połączenie pomiędzy poszczególnymi obiektami a Centralną Nastawnią zrealizowane jest za pomocą urządzeń telegrafii wielokrotnej w następujący sposób:

Każdy obiekt przekazuje stan swoich urządzeń indywidualnych kanałem transmisyjnym (kanały meldunkowe), natomiast Centralna Nastawnia przesyła polecenia do

wszystkich obiektów (podzielonych na dwa obszary) dwoma kanałami wspólnymi dla wszystkich obiektów danego obszaru (kanał poleceniowy).

Sygnały wyjściowe (telegramy) z poszczególnych stacji i podstacji podawane są poprzez urządzenia TgPM na wejścia modułów separatorów wejściowych, które oprócz oddzielania galwanicznego zamieniają sygnały prądowe na sygnały napięciowe o wspólnym zerze sygnałowym.

Sygnały te podawane są szeregowo na dwa układy separacji, z wyjść których przekazywane są niezależnie do dwu stacji dyspozytorskich.

Każdej ze stacji przypisany jest programowo do obsługi jeden z dwu części obszaru podległego Centralnej Nastawni.

Stacje dyspozytorskie są ze sobą połączone i wzajemnie kontrolują swoją obecność. W przypadku wyłączenia jednej z nich jej funkcje przejmuje druga stacja, umożliwiając obsługę obu obszarów.

Sygnały sterujące - polecenia wypracowane w stacji dyspozytorskiej przekazywane są do nadajnika poleceń gdzie tworzony jest standardowy telegram i wysyłany w odpowiednią linię. Konstrukcja i oprogramowanie nadajnika poleceń pozwalają na wysyłania poleceń z dowolnej stacji dyspozytorskiej do dowolnego obszaru.

W każdej stacji dyspozytorskiej (w modułach VIOP) przeprowadzane jest sprawdzanie poprawności formalnej transmisji oraz zamiana słów 20-bitowych telegramu na słowa 16-bitowe. Słowa te poprzez magistralę VME przesyłane są do dalszej analizy przez procesor centralny stacji. W wyniku tej analizy na ekranach kolorowych monitorów graficznych przedstawiony jest aktualny stan sieci trakcyjnej.

Sterowanie poszczególnymi obiektami sieci trakcyjnej odbywa się za pomocą kursora na ekranie monitora graficznego sterowanego manipulatorem kulowym (TRACK BALL) lub za pomocą komend wydawanych z konsoli dyspozytora.

Stacje dyspozytorskie są podłączone do stacji serwisowej, co umożliwia podgląd telegramów (przed rozkodowaniem) przychodzących z dowolnej stacji.

Dodatkowo Stacja Serwisowa zaopatrzana jest w moduły programowe pozwalające na edycję obrazów (tworzenie nowych i wprowadzanie zmian na już istniejących sieci trakcyjnej. Edycja prowadzona jest niezależnie od pracy stacji dyspozytorskich. Po zakończeniu prac edycyjnych poprawiony obraz sieci trakcyjnej poprzez złącze RS może być przesłany do obu Stacji Dyspozytorskich.

### 3.1. Podstawowe informacje

Głównym zadaniem Mikroprocesorowego Systemu Dyspozytorskiego BUSZ jest wizualizacja stanu sieci trakcyjnej oraz zdalne sterowanie poszczególnymi jej urządzeniami. MSD BUSZ został zrealizowany na wielozadaniowym systemie operacyjnym czasu rzeczywistego OS-9. Programy napisane są w języku C i mają budowę modułową.

Poszczególne fragmenty programu są odpowiedzialne za:
- wysyłanie komend/poleceń operatora, co jest głównie związane ze sterowaniem urządzeniami sieci;
- odbiór komunikatów;
- obsługę kolorowych monitorów, co jest związane z wizualizacją stanu sieci;
- obsługę terminala, co jest związane m.in. z obsługą kroniki zdarzeń;
- obsługę manipulatora;

– obsługę połączeń pomiędzy komputerami.

## 3.2. Obsługa programu

Po włączeniu systemu należy podać kod (hasło) dyspozytora; jeżeli hasło jest prawidłowe, nazwisko lub inicjały dyspozytora będą umieszczone w każdym komunikacie, który pojawi się w kronice zdarzeń.

Po wstępnym przetestowaniu konfiguracji sprzętowej, system rozpoczyna zbieranie da nych o sieci na podstawie odbieranych telegramów.

Faza uaktualniania danych kończy się wyświetleniem menu obszaru I i II.

Przeglądanie aktualnego stanu sieci sterowania jej elementami polega na wybraniu z menu odpowiedniej stacji, podstacji, kabiny sekcyjnej, a następnie elementu sieci i komendy za pomocą manipulatora lub klawiatury.

Podstawowymi komendami są:
- blokuj/odblokuj (uniemożliwienie/umożliwienie sterowania danym elementem);
- załącz (włączanie danego elementu);
- wyłącz (wyłączenie danego elementu).

Wszystkie zmiany spowodowane wykonaniem komendy operatora, ingerencją pracowników stacji i podstacji oraz sytuacjami awaryjnymi, są sygnalizowane na bieżąco, na ekranach monitorów graficznych, a także sygnałem dźwiękowym (wybrane awarie).

Informacje o wszystkich zderzeniach zapisywane są też do kronik.

Przykładowo:
- załączenie sygnalizowane jest wypełnieniem symbolu urządzenia na schemacie kolorem,
- wyłączenie sygnalizowane jest brakiem wypełnienia kolorem symbolu urządzenia na schemacie,
- awaria sygnalizowana jest migotaniem tła,
- wykonywanie komendy sygnalizowane jest migotaniem liczby identyfikującej element sieci.

## 3.3. Wejścia/wyjścia Systemu

### 3.3.1. Wejście

(3xVIOP) - 64 kB RAM + przerwanie, 32 linie wejściowe)

Przyjmowane są 3 rodzaje informacji wejściowych:

a) telegram podstawowy,
b) potwierdzenie przyjęcia polecenia,
c) meldunek szybki,
d) informacje dodatkowe.

**ad a) Telegram podstawowy**

Telegram przynosi informację o stanie elementów stacji lub podstacji.

Stacja (podstacja) jest identyfikowana na podstawie numeru linii wejściowej. Telegramy są różnej długości, w zależności od ilości elementów na stacji lub podstacji.

Zawartość telegramu jest pamiętana w RAM porcie pod określonym adresem.

W przypadku zmiany zawartości (zmienione wartości są wstawiane do telegramu) ustawiany jest bit w słowie statusowym i bit odpowiadający numerowi linii, z której przyszedł telegram w 4-bajtowym rejestrze zmian, a następnie wstawiane jest przerwanie do procesora głównego.

**ad b) Potwierdzenie przyjęcia polecenia**

Gdy przyjęte zostanie potwierdzenie o przyjęciu polecenia, ustawiany jest bit w słowie statusowym i bit odpowiadający numerowi linii, z której przyszedł meldunek w 4-bajtowym rejestrze potwierdzeń, a następnie wystawiane jest przerwanie do procesora głównego.

**ad c) Meldunek szybki - informacja o zmianie stanu elementu**

Informacja te jest wpisywana w pole telegramu w RAM porcie.

Dodatkowo ustawiany jest bit w słowie statusowym i bit odpowiadający numerowi linii, z której przyszedł meldunek w 4-bajtowym rejestrze meldunków i bit w tablicy meldunków (którego słowa telegramu dotyczy meldunek), a następnie wystawiane jest przerwanie do procesora głównego.

**ad d) Informacje dodatkowe**

– przekłamany telegram

W tym przypadku ustawiany jest bit w słowie statusowym, bit w rejestrze przekłamań (w którym kanale nastąpiło przekłamanie), bit lub bity w tablicy przekłamań), a następnie wystawiane jest przerwanie do procesora głównego.

– brak telegramu

W tym przypadku ustawiany jest bit w słowie statusowym i bit w rejestrze braków informujący o numerze linii, z której nie otrzymano telegramu

– brak S1

System zapisuje ilość kanałów obsługiwanych przez VIOP w rejestrze ilości kanałów (linii) i długość telegramu ( w słowach).

Postać RAM portu pakietu VIOP:

0000h - rejestr statusowy
- bit 0 zmiana zawartości telegramu
- bit 1 przyjęcia potwierdzania
- bit 2 meldunek szybki
- bit 3 przekłamany telegram
- bit 4 brak telegramu
- bit 5 brak S1

- rejestr braków S1
- rejestr ilości kanałów
- rejestr zmian
- rejestr potwierdzeń
- rejestr meldunków

16 słów - tablica meldunków
- rejestr braków
- rejestr przekłamań

16 słów - tablica przekłamań

16 słów - telegram linii 1
16 słów - telegram linii 2
*
*
16 słów - telegram linii 32

### 3.3.2. Wyjście

Za wysyłanie poleceń odpowiada nadajnik poleceń INP-01, który jest podłączony do wyjścia RS-232.

Nadajnik poleceń przyjmuje polecenia do systemu głównego, odpowiednio je przekształca i wysyła do obiektu sterowanego.

— Postać polecenia D15 D8 D6 D7 D0

D0-D6 - adres zespołu stacyjnego

D7 = 0 - stacje obsługiwane przez system 1
1 - stacja obsługiwana przez system 2

D8-D15 - adres obiektu z informacją sterującą (załącz/wyłącz)

— Postać potwierdzenia D7 D8 D4 D0

D0-D3 = 0000 - transmisja poprawna
xxxx - kod błędu (jeśli różny od 0)

D4-D6 = 000 - transmisja poprawna
xxx - kod błędu (jeśli różny od 0)

— Sposób wysyłania poleceń

System główny wysyła polecenia do INP-01 i czeka na potwierdzenie. INP-01 po przyjęciu polecenia wysyła do systemu głównego XOFF i potwierdzenie przyjęcia polecenia, nadaje polecenie do obiektu, a następnie wysyła XON do systemu głównego. W przypadku braku potwierdzenia po czasie 3 sekund wysyła komunikat o błędzie. W przypadku potwierdzenia z komunikatem o błędzie dwukrotnie powtarza polecenia. Po trzech błędach wysyłany jest komunikat do systemu głównego.

## 4. Struktura oprogramowania

System pracuje na bazie opisującej rzeczywisty obiekt (BUSZ_BAZA.DAT), zawierającej dane o elementach obiektu, sieci połączeń, dane wykorzystywane podczas wizualizacji obiektu, dane opisowe wykorzystywane podczas tworzenia komunikatów do operatora oraz dane potrzebne do wygenerowania poleceń wysyłanych do elementów sterowanych obiektu.

Uzupełniane są one i uaktualniane podczas pracy systemu. Po włączeniu dane o stanach poszczególnych elementów sieci są zbierane w oparciu o otrzymywane telegramy. Uaktualnienie następuje po odebraniu telegramu z informacją o zmianie stanu danego elementu oraz po zaakceptowaniu komendy lub polecenia operatorskiego.

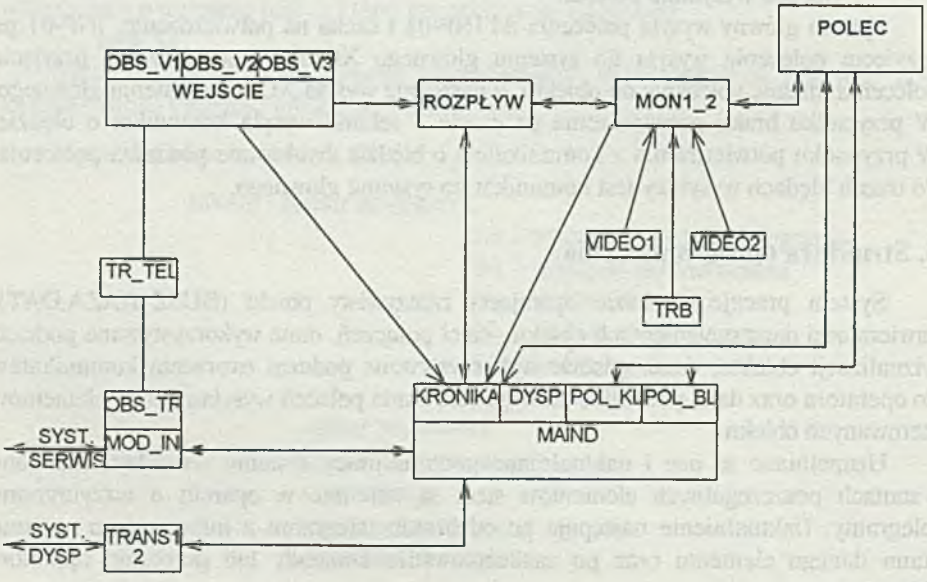
Oznaczenia pojęć:
Gałąź - element sterowany,
Węzel - element stanowiący połączenie między gałęziami.

Moduły programowe, za pomocą których tworzona jest baza danych zostały krótko omówione w części dotyczącej systemu serwisowego.

Poniżej zostały wymienione główne tablice z danymi:

ADRESYB - tablica zawierająca wielkości i adresy poniższych tablic w module BUSZ_BAZA,
TKWE - tablica korelacji odłączników (VIOP, linia, słowo telegr., bitu słowa) - nr systemowy elementu,
TK - tablica kolorów potencjałów,
TMENU - tablica korelacji pozycji z nr stacji (obrazu),
SL - tablica korelacji - nazwa stacji, nr VIOP, nr linii,
GF - tablica danych funkcjonalnych opisujących gałęzie,
TEW - tablica adresów elementów gałęziowych podłączonych do węzłów,
WF - tablica danych funkcjonalnych opisujących węzły,
TABGS - tablica adresów rekordów gs dla danego obrazu,
GS - tablica danych opisujących symbole graficzne,
TNAP - tablica napisów (stringów),
NZ - tablica nazw zwyczajowych elementów (odłączników, wyłączników, itd.),
BAZANZ - baza danych wykorzystywana do tworzenia nazw zwyczajowych.

Poniżej przedstawiony jest wykaz modułów (zadań, procesów) systemu dyspozytorskiego (patrz rys. 2) i serwisowego z krótkim opisem działania.




Rys. 2. SYSTEM DYSPOZYTORSKI - schemat blokowy oprogramowania

Objaśnienie niektórych słów użytych w opisie, specyficznych dla wywołań systemu operacyjnego CS-B:

sygnał - przerwanie programowe do innego zadania,
alarm - sygnał alarmowy do siebie (zadania wywołującego),
komunikat - ciąg znaków wysłanych/odebranych do/z "pipe".

## 4.1. System dyspozytorski

* MAIND

Moduł główny systemu dyspozytorskiego. Zadaniem tego modułu jest inicjalizacja systemu - uruchomienie zadań systemu z odpowiednimi parametrami oraz umożliwienie dyspozytorowi uruchomienia zadań występujących w menu systemu i działających na terminalu dyspozytorskim.

Menu dyspozytorskie zawiera następujące opcje:
- wprowadzenie aktualnej daty i godziny,
- uruchomienie podglądu kroniki zdarzeń,
- wybór reżimu pracy z rozpływem lub bez rozpływu potencjałów,
- wybór obszaru, na którym pracuje dany dyspozytor,
- wysyłanie poleceń z klawiatury,
- wysyłanie bloku poleceń,
- rejestracja dyspozytora,
- akceptacja błędów na stacji (dyspozytor może zaakceptować z klawiatury terminala błędy, które wystąpiły na określonej stacji).

Na menu dyspozytorskim wyświetlane są:
- data i godzina,
- aktualny reżim pracy,
- numer aktualnego obszaru pracy,
- nazwisko lub inicjały aktualnie pracującego dyspozytora,
- komunikat o braku połączenia z drugim systemem dyspozytorskim (jeżeli taka sytuacja zaistnieje).

* OBS_V1, OBS_V2, 0BS_V3

Moduły wejściowe - obsługują dane nadchodzące z pakietów VIOP. Tu następuje ich wstępna analiza oraz wysyłany jest odpowiedni komunikat do modułu WEJŚCIE.

* WEJŚCIE

Obsługa danych wejściowych. Zawiera obsługę komunikatów od modułów OBS_V? i POLEC, analizuje nadchodzące informacje i wysyła komunikaty do zadań KRONIKA, POLEC, ROZPŁYW.

Po uzyskaniu komunikatu z POLEC o nadaniu polecenia zmiany stanu ustawia alarm na określony czas - jest to czas, po którym powinno nadejść potwierdzenie przyjęcia polecenia. Jeżeli potwierdzenie nie nadejdzie, wysyła komunikat do POLEC w celu powtórzenia polecenia. Jeżeli potwierdzenie przyjdzie w odpowiednim czasie, WEJŚCIE nadaje komunikat do KRONIKI zdarzeń i POLEC, jednocześnie ustawia alarm na określony czas, jako czas na przyjście meldunku szybkiego lub meldunku o zmianie stanu danego obiektu.

Przyjmuje komunikaty o zmianach stanu od OBS_V?. Jeżeli meldunek szybki przyszedł w odpowiednim czasie, nadaje komunikat do modułów KRONIKA i ROZPŁYW. Jeżeli oczekiwany meldunek nie nadszedł, nadaje komunikat do kroniki. W przypadku przyjęcia komunikatu o zmianie samoczynnej, nadaje komunikat do KRONIKI i ROZPŁYW. Do KRONIKI wysyłany jest też komunikat o nadejściu nieoczekiwanego potwierdzenia.

WEJŚCIE obsługuje również polecenia kontrolne i polecenie wspólne - polecenia, po których ma przyjść jedynie potwierdzenie bez komunikatu o zmianie stanu.

Podczas inicjalizacji WEJŚCIE na podstawie danych w bazie BUSZ_BAZA.DAT dokonuje konfiguracji i uruchamia oprogramowanie pakietów VIOP.

* POLEC

Wysyłanie poleceń. Przyjmuje komunikaty z poleceniami od MON 12, POL_TERM, testuje prawidłowość polecenia, dokonuje konwersji poleceń na postać zrozumiałą przez INP-01 i wysyła je. Jeśli są błędy, nadaje komunikat do KRONIKI oraz modułu, z którego przyjął polecenie. Polecenia są ustawiane w kolejce poleceń, kolejne polecenie wysyłane do danej stacji musi czekać na przyjście potwierdzenia po poleceniu wysłanym do danej stacji, jeżeli takie polecenie czeka w kolejce.

Moduł POLEC obsługuje również polecenie wspólne - polecenia kontrolne do wszystkich stacji wysyłane automatycznie co 15 minut, a także polecenia do elementów sprzężonych (informuje WEJŚCIE o nadejściu dwóch zmian stanów po jednym poleceniu).

* ROZPŁYW

Uaktualnienie stanu sieci trakcyjnej. Na podstawie otrzymywanych komunikatów wyznacza rozpływ potencjałów w sieci energetycznej (kolory trakcji na obrazach graficznych), dokonuje aktualizacji bazy danych BUSZ_BAZA.DAT i wysyła sygnał do MON12.

* TRANS1_2

Obsługa transmisji System1 - System2. Przyjmuje komunikaty o stanie drugiego systemu oraz o jego działaniach. W przypadku zerwania kontaktu z drugim systemem nadaje komunikat do KRONIKI i MAIND (awaria drugiego systemu).

* TRB

Obsługa manipulatora. Obsługuje dane przychodzące z manipulatora kulowego, współpracuje z modułem MON12.

* KRONIKA

Obsługa kroniki zdarzeń. W kronice zdarzeń rejestrowane są wszystkie polecenia operatora, zmiany stanów obiektu i stany awaryjne.

Format zapisu: data, godzina, kod dyspozytora, nazwa stacji, kod elementu, stan elementu, opis zdarzenia (np. zmiana samoczynna, brak potwierdzenia, zmiana po poleceniu), opis elementu (np. odłącznik sieci trakcyjnej nr 107).

Wielkość kroniki jest ograniczona ilością dostępnej pamięci, dane nadmiarowe są kasowane (mogą być przedtem wydrukowane - sygnalizacja komunikatem w KRONICE i na menu MAIND).

Moduł obsługi kroniki zdarzeń umożliwia zapis danych, wyświetlenie kroniki w porządku chronologicznym lub komunikatów wybranych wg wybranych kluczy (np.

tylko awarie na określonej stacji, które wystąpiły w danym przedziale czasu), drukowanie wybranych fragmentów kroniki.

* POL_KL

Wysyłanie poleceń z klawiatury. Umożliwia wysłanie polecenia dla wybranego elementu określonej stacji z klawiatury terminala dyspozytorskiego.

* POL_BL

Wysyłanie bloku poleceń. Umożliwia przygotowanie i wysłanie bloku (sekwencji) poleceń. Za pomocą edytora zawartego w module można przygotować kilka bloków zawierających po kilkanaście poleceń. Dane dotyczące bloków są przechowywane w pamięci nieulotnej. Podczas wykonywania bloku dyspozytor może śledzić wykonywanie poleceń zarówno na ekranie terminala (komunikaty), jak i na ekranie monitora kolorowego (obraz ze sterowanym elementem jest automatycznie wyświetlany). Dyspozytor może wysłać cały blok lub wybraną jego część, w każdym momencie może przerwać wysyłanie bloku poleceń.

Jeżeli jedno z poleceń bloku nie zostanie wykonane, program przerywa wysyłanie poleceń, decyzję o wysłaniu następnego polecenia zostawiając dyspozytorowi.

* MON12

Moduł koordynujący pracę modułów podsystemu graficznego obsługującego dwa monitory. Pośredniczy przy przesyłaniu informacji z manipulatora do obydwu monitorów VIDEO oraz zbiera od nich informacje o wydanych poleceniach i przekazuje je odpowiednim zadaniom. Pośredniczy także w wymianie informacji między samymi modułami graficznymi, steruje systemem wygaszania ekranów monitorów i obsługuje informacje o błędach i alarmach.

* VIDEO1, VIDEO2

Moduły grafiki sterujące wyświetlaniem menu i obrazów odzwierciedlających aktualny stan sterowanych obiektów na ekranie monitora graficznego pracującego z rozdzielczością 1024x768 punktów.

Umożliwiają także dyspozytorowi wydawanie poleceń do sterowanych obiektów na podstawie aktualnego stanu obiektów uwidocznionych na obrazach, a także akceptację zaistniałych zdarzeń.

Rodzaje obrazów:
- menu systemów I i II z podziałem na kierunki,
- stacje i podstacje,
- trakcja elektryczna,
- kabiny sekcyjne.

Rodzaje menu:
- menu obszaru I i obszaru II (wybór stacji),
- menu sterowania (wybór polecenia),
- menu dodatkowe (czułość manipulatora, ręczny/automatyczny).

W specyficznych sytuacjach w menu pojawiać się będą także i inne komendy lub polecenia.

Kursor, sterowany manipulatorem, porusza się po ekranach monitorów graficznych. Wyboru systemu, stacji, podstacji, elementu lub komendy dokonuje się

poprzez najechanie kursorem na odpowiednie pole lub wciśnięcie odpowiedniej kombinacji z trzech klawiszy.

Wybór elementów z obrazu:
- najazd kursorem na symbol elementu i odpowiedniego klawisza powoduje:
  - wyświetlenie obrazu danej stacji lub podstacji,
  - wyświetlenie obrazu kabiny,
  - wyświetlenie numeru elementu oraz zestawu komend, które można przesłać do tego elementu.

Przesunięcie kursora na brzeg ekranu powoduje przesunięcie obrazu na ekranie, wciśnięcie odpowiedniej sekwencji klawiszy manipulatora umożliwia powiększenie części obrazu znajdującej się wokół kursora.

Jeżeli przez pewien określony czas nie będą wykonywane żadne operacje na monitorze graficznym, to zostanie on automatycznie wygaszany. Ponowne wyświetlenie obrazu następuje po wykonaniu ruchu manipulatorem lub w przypadku nastąpienia zmiany stanu obiektu.

* TR_TEL

Transmisja telegramów i poleceń. Umożliwia transmisję w czasie rzeczywistym postaci telegramów nadchodzących z dowolnej wybranej stacji i postaci wysyłanych z systemu dyspozytorskiego poleceń, do systemu serwisowego. Moduł ten jest uruchamiany ze stacji serwisowej.

* OBS_TR

Obsługa transmisji między systemem dyspozytorskim i serwisowym. Umożliwia wymianę komunikatów, transmisję telegramów do stacji serwisowej i bazy danych do stacji dyspozytorskiej.

* MOD_IN

Obsługa transmisji bazy. Moduł ten uruchamiany jest na żądanie dyspozytora, podczas transmisji danych praca systemu dyspozytorskiego jest zatrzymywania (pracę na danym obszarze przejmuje drugi system dyspozytorski). Po poprawnej transmisji bazy danych system jest inicjalizowany.

* DYSP

Rejestracja dyspozytora. Moduł ten umożliwia, po podaniu hasła, wprowadzenie kodów (haseł) i nazwisk wszystkich dyspozytorów uprawnionych do pracy na danym systemie. Baza kodów dyspozytorskich jest przechowywana w pamięci nieulotnej i wprowadza ją (zna hasło) tylko osoba uprawniona.

Każdy dyspozytor przed rozpoczęciem pracy wprowadza swój kod (hasło) i od tej pory wszystkie komunikaty w kronice zdarzeń będą opatrzone jego nazwiskiem. W przypadku wprowadzenia błędnego hasła, w miejscu nazwiska ukaże się komunikat: "DYSP???".

## 4.2. System serwisowy

* MAINS

Program główny. Menu systemu serwisowego zawiera następujące opcje:
- sprawdzenie połączenia z systemem dyspozytorskim,

- podgląd telegramów i poleceń,
- inicjalizacja polskich znaków na drukarce,
- edycja bazy danych,
- kompilacja bazy danych,
- operacje dyskowe,
- nadanie początkowych potencjałów,
- zmiana kolorów sieci,
- transmisja bazy danych.

Ponieważ baza danych jest kluczowym elementem systemu dyspozytorskiego, system "pilnuje" operacji na niej wykonywanych, np. nie można przesłać bazy nieskompilowanej do systemu dyspozytorskiego.

* POL_ZN
Instalacja polskich znaków na drukarce.

* GRAF
Edytor bazy danych, umożliwia wprowadzanie zmian do istniejącego schematu synoptycznego (kasowanie/dodawanie elementów, linii, obrazów), a także danych wykorzystywanych przez wejście i wyjście systemu dyspozytorskiego.

*OBS_TR
Obsługa połączenia z systemem dyspozytorskim - przesyłanie komunikatów do/z systemu dyspozytorskiego.

* WIZ_TEL
Wizualizacja telegramów i poleceń w formie pierwotnej na ekranie terminala stacji serwisowej. Sygnalizuje też zaistniałe błędy.

* DYSKI
Obsługa stacji dyskietek - zapis/odczyt bazy danych na dyskietki, porównanie bazy zawartej w PAO i na dyskietkach, kasowanie zbiorów, testowanie i formatowanie dyskietek.

* TYPK, KON
Analiza sieci trakcyjnej i nadanie początkowych potencjałów. Z danych powstałych za pomocą tych modułów korzysta ROZPŁYW.

* PRZETWORZ
Kompilacja bazy danych - wypełnianie i kasowanie rekordów, sortowanie w celu umożliwienia szybszej pracy, testowania danych.

* KOLOR
Zmiana kolorów (oznaczeń potencjałów) sieci trakcyjnej wg życzenia użytkownika.

* MOD_OUT
Obsługa transmisji bazy danych do systemu dyspozytorskiego.

* TRB
Obsługa manipulatora kulowego jak w systemie dyspozytorskim.

Opisany System Dyspozytorski pracuje poprawnie od stycznia 1983 r.

Podkreślić należy dwie cechy systemu, które znacznie ułatwiają jego eksploatację:

– trwałe przypisanie wyjściom poszczególnych zasilaczy podstacyjnych kolorów, które następnie są 'prowadzone' poprzez wszystkie zamknięte odłączniki do poszczególnych sekcji trakcji, pozwala to na szybkie określenie przez dyspozytora źródła zasilania interesującej go sekcji;

– możliwość tworzenia sekwencji sterowania i uruchamiania ich na żądanie, jest to szczególnie istotne w przypadkach prowadzenia remontów, kiedy trzeba wykonać operacje załączenia zasilania sekcji w celu przepuszczenia pociągu, a następnie jego wyłączenia w celu kontynuowania remontu.

System MSD-Busz uzupełniony o nowoczesne urządzenia stacyjne (stacje obiektowe) jest wyrobem oferowanym przez IKSAiP do bezpośredniego zainstalowania przy nadzorze i sterowaniu urządzeniami zasilania trakcji elektrycznej w kolejnictwie i komunikacji miejskiej. Po wprowadzeniu zmian może pracować również w dyspozytorniach nadzoru gazociągów, wodociągów itp.

JAN RYŻKO
INSTYTUT MASZYN MATEMATYCZNYCH WARSZAWA

# Pamięci pomocnicze (cache) komputerów osobistych
## Cache memories of personal computers

### Streszczenie

Artykuł zawiera analizę rozwoju pamięci pomocniczych komputerów osobistych, przewidywane kierunki zmian konstrukcyjnych oraz porównanie cen niektórych typów pamięci.

### Abstract

The paper contains the analysis of development cache memories of personal computers, forecast of trends change construction as well as comparison of prices some types' memories.

Pamięci pomocnicze stały się ostatnio istotnym podzespołem komputerów o dużej wydajności. W początkowym okresie rozwoju komputerów osobistych ich pamięć operacyjna nadążała za szybkością mikroprocesorów (Tablica 1, utworzona z rys. w pracy [1] na str. 87). W pierwszej połowie lat osiemdziesiątych jednakże, począwszy od mikroprocesora 80386, coraz bardziej zaczęła się ujawniać rozbieżność pomiędzy tymi wielkościami, osiągając obecnie (wrzesień 1995) ponad 100 MHz - 133 MHz dla aktualnie oferowanej wersji Pentium i około 20 MHz dla powszechnie stosowanych pamięci RAM. Częściową przynajmniej poprawę sytuacji przynosi tu zastosowanie pamięci pomocniczej (ang. cache), w której wykorzystuje się szybsze od dynamicznych (DRAM) elementy pamięci statycznych SRAM. Są one wytwarzane w różnych odmianach jako synchroniczne, asynchroniczne, o małej mocy, nieulotne, FIFO, wieloportowe, z etykietą. Pojemność ich zawiera się od kilku bajtów do 128 kB i realizowane są w różnych konfiguracjach. Rozwój komputerów osobistych przyspiesza znacznie poprawę parametrów SRAM i prowadzi do obniżenia ich ceny. Procesory o wysokiej wydajności, jak Pentium i PowerPC, zwiększają zapotrzebowanie na pomocnicze pamięci operacyjne, zwłaszcza synchroniczne SRAM. Zapotrzebowanie jest tak znaczne, że prawie każdy większy dostawca pamięci skupia swe wysiłki nad wytwarzaniem bardzo szybkich SRAM, które szybko stają się dostępnym produktem. Powoduje to obniżenie ceny, przynosząc korzyść innym ich użytkownikom.

Tablica 1. Częstotliwość procesorów i pamięci operacyjnych [MHz] w latach 1975-98

| lata | 1975 | 1976 | 1977 | 1980 | 1982 | 1983 | 1985 |
|---|---|---|---|---|---|---|---|
| procenty | 2.5 | 4.7 | | | 8 | | 16 |
| pamięci | | | 6 | 7 | | 8 | |
| lata | 1986 | 1989 | 1992 | 1993 | 1994 | 1995 | 1998 |
| procenty | | 25 | | 66 | 100 | 133 | |
| pamięci | 11 | 14 | 16.5 | | | 20 | 26 |

Początkowo rynek SRAM niewiele różnił się od rynku DRAM. Były one szybsze, ale o parametrach tego samego rzędu (np. 100 ns wobec 150 ns dla DRAM). Rynek na te standardowe, wolne SRAM nadal się rozszerza. Stosowane są one np. w stacjach dysków twardych, modemach i układach zbierania danych. Znacznie bardziej jednak rozwijają się szybkie SRAM, potrzebne do rozbudowanych komputerów osobistych i stacji roboczych.

Kilka lat temu, kiedy prędkość przesyłania informacji na szynach procesorów osiągnęła 33 MHz, a różnica prędkości pomiędzy SRAM i DRAM wzrosła, wystąpiła konieczność stosowania pamięci pomocniczych. Obecnie pamięci te stały się istotnym podzespołem w systemach opartych na procesorze Pentium, a rola ich jeszcze wzrośnie w wielozadaniowych systemach operacyjnych, takich jak Windows 95. Szybkie SRAM wykorzystywane są również w wyspecjalizowanych zastosowaniach, takich jak szybkie testery i systemy akwizycji danych, ale stanowi to małą część zastosowań. Dostawcy pamięci przewidują, że w bieżącym roku co najmniej 95% systemów z Pentium zawierać będzie minimum 256 kB pamięci pomocniczej (koszt elementów 50-60 dolarów).

Gdy zegar systemu osiąga 150-200 MHz, stacje robocze, które poprzednio używały układów SRAM o czasach dostępu 6-12 ns, wymagają teraz wartości tego parametru w zakresie 4,5 do 8 ns. Podobnie, gdy prędkość szyny osiąga 75 MHz, pamięć pomocnicza komputerów osobistych musi zmniejszyć swój czas dostępu z 12-20 ns do 8-12 ns.

Tablica 2 (utworzona z rys 1. [2]) pokazuje przewidywane częstotliwości szyny pamięci pomocniczych w złożonych komputerach osobistych i stacjach roboczych, jak również tego parametru dla pamięci DRAM w latach 1992-99. Wynika z niej, że częstotliwości procesorów wzrastają ponad możliwości DRAM. To właśnie powodować będzie rozwój SRAM o zwiększonych szybkościach.

Tablica 2. Częstotliwość pamięci DRAM i szyny pamięci pomocniczych

| Rok | 1992 | 1993 | 1994 | 1995 | 1996 | 1997 | 1998 | 1999 |
|---|---|---|---|---|---|---|---|---|
| DRAM | 12 | 24 | 41 | 56 | 68 | 82 | 94 | 109 |
| złoż. PC | 53 | 65 | 82 | 110 | 142 | 180 | 237 | 335 |
| złoż. st. rob. | 71 | 100 | 136 | 181 | 237 | 309 | 395 | 518 |

Różnica częstotliwości pomiędzy stacjami roboczymi a komputerami osobistymi powoduje, że pamięci pomocnicze tych systemów wykorzystują różne techniki SRAM. Np. potokowe, synchroniczne układy CMOS spełniają wymagania wolniejszych systemów (50 do 66 MHz). Kosztują one około 15-20 dolarów za układ 32k x 32 bity. Systemy średniej prędkości (75-100 MHz) mogą wykorzystywać technikę BiCMOS, której analogiczne układy są w cenie 25-30 dolarów. Najszybsze systemy (150-200 MHz) wymagają potokowych systemów BiCMOS wykorzystujących dwa różne zegary. Kosztują one około 100 dolarów za układ 4 Mbity.

Proces BiCMOS wymaga mieszanych technik bipolarnej i CMOS dla poprawienia czasu dostępu, co podraża cenę tych układów. Za wyjątkiem najszybszych stacji roboczych, szybkie pamięci SRAM w technice CMOS wypadają korzystniej od układów BiCMOS. Według przewidywań firmy In-Stat w ciągu najbliższych kilku lat średnia cena BiCMOS ustali się na około 12 dolarów za układ, podczas gdy ceny synchronicznych układów CMOS będą spadać.

Do niedawna większość sterowników pamięci pomocniczych w systemach 486 wykorzystywało protokół strumieniowy poprzez przeplatanie dwóch zespołów standardowych, asynchronicznych SRAM jako pamięci pomocnicze poziomu drugiego (level-2 caches). Ze wzrostem zagęszczenia pamięci pomocniczych rozwiązanie to będzie coraz rzadziej używane, zwłaszcza dla szybkich systemów. Ze względu na cenę, droższe, synchroniczne układy SRAM będą najpierw używane w złożonych stacjach roboczych. Ale również wkrótce dla systemów z Pentium i 486 staną się one najbardziej opłacalnymi pamięciami pomocniczymi.

Synchroniczne pamięci statyczne zawierają rejestry do przechowywania informacji (zarówno danych, jak i sygnałów sterujących), które uwalniają inne elementy pamięci od przygotowywania następnego cyklu dostępu. Rejestry te pozwalają układom synchronicznym na 20% przyspieszenie działania w stosunku do układów asynchronicznych. Ponadto pamięci asynchroniczne wymagają odpowiednich kształtów impulsów i ich usytuowania w funkcji czasu. Natomiast poprawna praca układów synchronicznych zależy tylko od odpowiedniego rozmieszczenia krawędzi impulsów zegarowych w stosunku do innych sygnałów i często wykorzystują ten sam zegar w całym systemie, co upraszcza projektowanie.

Pamięci synchroniczne dostępne są w wersji zwykłej i potokowej, gdzie pewne operacje przebiegają równolegle. Te ostatnie posiadają rejestr wyjściowy, wpływający na wydajność. Pamięć taka o czasie dostępu 8 ns działa przy częstotliwości szyny 66 MHz, podczas gdy pamięć zwykła (12 ns) przy częstotliwości 50 MHz.

Również rodzaj obudowy wpływa na wydajność tych układów pamięciowych. Coraz mniej pamięci cache sprzedaje się w tradycyjnej obudowie dwurzędowej (DIP),

a stosuje się takie obudowy jak SOJ, TQFP i TSOP. Gdzie szczególnie istotna jest szybkość, a więc przede wszystkim w stacjach roboczych, tam stosowana jest 119-nóżkowa obudowa PGPA (plastic ball-grid-array). Obecnie najpopularniejsza jest 100-nóżkowa obudowa TQFP, która jest również standardem dla synchronicznych pamięci SRAM.

Opracowując pamięci synchroniczne inni wytwórcy wzorują się na Intelu, która to firma miała na celu bezpośrednią współpracę z Pentium, a więc układ 32k x 32 bity (bez bitu parzystości), stosowaną tu kolejność impulsów i napięcie zasilające 3,3 V. Przewiduje się, że układy te staną się powszechne w 1996 roku. Według In-Stat w roku 1993 układy synchroniczne stanowiły tylko 2,2% rynku SRAM. W roku 1994 ten udział wzrósł do 5%, a w roku 1998 ma wynieść 22,8%. Ilościowy wzrost ma wynieść w tym roku 141% w stosunku do roku 1994. Jest to wynikiem wzrostu zapotrzebowania na pamięci pomocnicze drugiego stopnia w układach z Pentium i PowerPC, przy czym wzrost pamięci cache w stacjach roboczych i serwerach plików idzie w parze z rozwojem szybkich układów telekomunikacyjnych, gdzie również wykorzystywane są te pamięci. Jednocześnie przewiduje się, że różnica cen pomiędzy asynchronicznymi a synchronicznymi układami SRAM, wynosząca obecnie około 30%, spadnie w 1996 roku do 15-20%.

Tablica 3 (uzyskana z rys. 2 [2]) pokazuje, jak kształtował się dotąd i jak przewiduje się na najbliższe lata przebieg kosztów jednostki (w tym przypadku kbita) dla synchronicznych pamięci SRAM przy konstrukcji układów o różnej pojemności. Widzimy, że najkorzystniejsze są obecnie układy o pojemności 1 Mbita. Cena 1 kbita w tych układach spadnie z około 3,5 centa w 1994 roku do 1 centa w 1997 roku i później nie będzie się już zmieniać. Natomiast parametr ten dla układów 4 Mb wynosił w 1994 roku ok. 8,5 centa, a w roku 1998 ma spaść nieco poniżej 1 centa.

Tablica 3. Koszt kbita synchronicznej pamięci SRAM w centach

| Rok wprowadzenia / Poj. pamięci | 1994 | 1995 | 1996 | 1997 | 1998 |
|---|---|---|---|---|---|
| 64 k | 11,5 | 9,6 | 8,7 | 8 | 7,3 |
| 256 k | 4,5 | 3,5 | 3 | 2,4 | 2,2 |
| 512 k | 5,3 | 3 | 2,2 | 1,4 | 1,4 |
| 1 M | 3,7 | 2 | 1,4 | 1,1 | 1,1 |
| 4 M | 8,3 | 6,6 | 3 | 1,8 | 0,9 |

Nie ma obecnie standardowego układu synchronicznej SRAM do systemów opartych na Pentium. Wiele płyt głównych tych systemów wykorzystuje konfigurację 32k x 8 bitów o czasie dostępu 15-20 ns. Nowe opracowania obejmują w pełni zintegrowane moduły pozwalające na wytwarzanie wspólnych płyt głównych dla różnych

zastosowań, na których to płytach można wymieniać jednostkę centralną i pamięć pomocniczą. Na przykład firma Integrated Device Technology (IDT), będąca pionierem na rynku modułów pamięci pomocniczych, oferuje szereg opcji dla systemów 486 i Pentium. Ceny tych opcji znacznie się różnią - od 64 dolarów za asynchroniczną pamięć o pojemności 256 kB do 559 dolarów za 512 kB system potokowy z układem parzystości (podawane wartości cen dotyczyły początku 1995 roku). Projektanci liczą na osiągalność tych modułów o różnych konfiguracjach po cenach możliwych do zaakceptowania.

Większość tych modułów ma wbudowane układy sterowania pamięcią pomocniczą. Jednakże w przenośnych komputerach jest nieco inaczej i można tu korzystać z pamięci pomocniczej na pojedynczym układzie, co pozwala na najbardziej zwarte projektowanie. Obecnie takie synchroniczne układy oferuje tylko firma Sony. Natomiast IDT uważa, że najwięcej miejsca zajmuje pamięć operacyjna danych i ta część pamięci powinna być oddzielona, by wykorzystać układy standardowe. Firma ta opracowała układ z etykietą i sterownikiem w jednym układzie, a projektant systemu może dodać żądaną wielkość pamięci danych. Również firma Cypress Semiconductor, która ostatnio zakupiła wytwórcę zespołów układów Contaq Microsystems, bada możliwość umieszczenia pamięci pomocniczej w tych zespołach. Moduły IDT dostosowane są do zespołu układów (chip sets) oferowanych przez znane firmy jak Intel, OPTi, PicoPower i VLSI. Oczekuje się oferty podobnych modułów ze strony innych firm jak Alliance, Cypress, Motorola i Samsung.

Wydaje się, że o ile jeszcze obecnie synchroniczne pamięci statyczne stanowią małą część całego rynku SRAM, to rola ich będzie rosła ze względu na prosty interfejs projektowania i wkrótce staną się standardem zarówno w komputerach, jak i innych zastosowaniach.

Na zakończenie prześledźmy, jak zwiększał się udział pamięci pomocniczych w nowych opracowaniach mikroprocesorów [3]. W procesorze 80486 mieliśmy scaloną pamięć pomocniczą o pojemności 8 KB, która wykorzystywana była zarówno do rozkazów jak i danych. W Pentium występowały już dwie niezależne pamięci pomocnicze: jedna dla rozkazów, a druga dla danych - każda o pojemności 8 KB. Wynikało to z rozdzielenia ścieżek przetwarzania rozkazów i danych, a także dwukrotnego zwiększenia (z 32 do 64) ilości wejść i wyjść procesora. Wreszcie w projektowanym mikroprocesorze nazywanym roboczo P6 [4] rozdzielone pamięci pomocnicze mają tak samo po 8 KB każda, ale oprócz tego występuje tu pamięć pomocnicza drugiego poziomu (L2) o pojemności aż 256 KB, która jest połączona z jednostką centralną specjalną 64-bitową szyną. Pamięć ta zawiera aż 15,5 miliona tranzystorów i jest umieszczona wraz ze strukturą półprzewodnikową zawierającą jednostkę centralną (5,5 mln tranzystorów) w ceramicznej obudowie 387 nóżkowej.

## Literatura

[1] Tom R. Halfhill: New Memory Architectures to Boost Performance; Byte, July 1993, str. 86.

[2] Markus Levy: A thumbnail sketch of Cache memory; EDN, January 19 1995, str. 30.

[3] Bob Ryan: Inside the Pentium, Byte, May 1993, str. 102.

[4] Tom R. Halfhill: Intel's P6, Byte, April 1995, str. 42.

JAROSŁAW WÓJTOWICZ
INSTYTUT MASZYN MATEMATYCZNYCH WARSZAWA

# Algorytm rozpoznawania grafiki jako struktury pomiarowej
# The algorithm for recognition of graphic as a measuring structure

## Streszczenie

W artykule opisano struktury danych i algorytm przejścia od graficznej prezentacji struktury pomiarowej do procedur pomiarowych w programie VIRTWin.

## Abstract

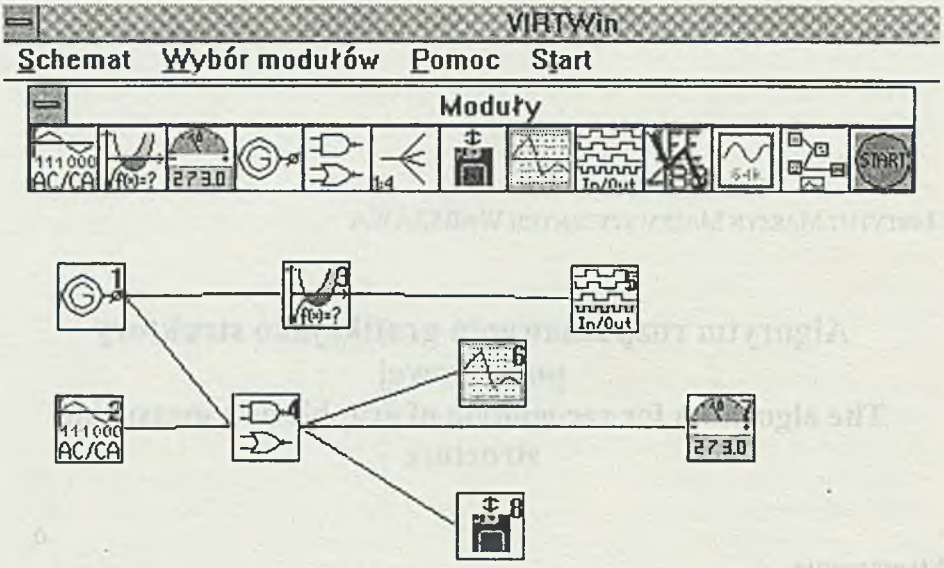
This paper describes data structures and the transformation algorithm from graphical visualization of measuring structure to measuring procedures in VIRTWin program

Program VIRTWin służy do obsługi kart pomiarowych zainstalowanych w komputerze klasy IBM PC, obróbki danych, ich zobrazowaniu i rejestracji. Został on napisany w języku Borland C 3.1 pod systemem operacyjnym Windows 3.1. Cechą charakterystyczną programu jest łatwość budowy dowolnej struktury pomiarowej przez użytkownika. Prostotę budowy struktury uzyskano dzięki graficznemu wprowadzaniu danych.

## Wprowadzenie danych

Budowa struktury pomiarowej została maksymalnie uproszczona dzięki graficznemu sposobowi wprowadzania danych. Użytkownik wybiera (np. myszką) dowolne moduły i na polu roboczym (ekranie) ustawia ikony wyobrażające te moduły. Następnie łączy owe moduły siecią połączeń. Moduły są trojakiego rodzaju: wejściowe, pośrednie i wyjściowe. Każdy moduł wejściowy (dostarczający danych do przetwarzania) i pośredni mają po cztery wyjścia, a więc do każdego z nich mogą być podłączone cztery następne moduły: pośrednie lub wyjściowe.

Moduły pośrednie i wyjściowe mogą mieć do dwóch wejść. Moduły wyjściowe obsługują z reguły dodatkowe (wspomagające) okno do zobrazowania lub rejestracji danych (np. wykres, wyświetlacz cyfrowy, zapis do pliku).




Rys. 1. Obraz struktury pomiarowej na ekranie

## Struktury danych

Każdy typ modułu jest opisany strukturą zawierająca dane o: nazwie modułu, ilości wejść (od 0 do 2), obecności wyjść ( TRUE lub FALSE ), procedurze obsługującej moduł (wskazanie na procedurę). Takich struktur jest tyle ile różnych typów modułów i tworzą one tablicę o nazwie 'narzędzia'.

Wybranie przez użytkownika modułu powoduje utworzenie w polu roboczym okna-ikony. Struktura danych opisująca okno jest wciągana na listę LISTA_OKIEN. W strukturze opisującej okno znajduje się wskaźnik na inną strukturę DEF_OKNA dodatkowo opisującą przypisany do okna moduł.

Struktura danych DEF_OKNA opisuje każdy moduł w polu roboczym. Zawiera ona najważniejsze informacje dotyczące typu modułu, wskaźniki na moduły (elementy listy LISTA_OKIEN) podłączone do wyjść, wskaźnik na okno wspomagające (o ile takie jest), wskaźnik na strukturę PAR_F zawierającą parametry przekazywane do funkcji obsługującej moduł.

Struktura parametrów przekazywanych do funkcji obsługującej moduł PAR_F zawiera wskaźniki na moduły (elementy listy LISTA_OKIEN) podłączone do wejść (o ile są), wynik z przeprowadzonych operacji zwracany przez funkcję obsługi i parametry dla funkcji obsługi.

Struktury danych LISTA_OKIEN, DEF_OKNA, PAR_F są wypełniane podczas wybierania, ustawiania parametrów działania oraz łączenia modułów w polu roboczym.

## Algorytm działania

Struktura pomiarowa zbudowana przez użytkownika w swej istocie jest drzewem (zabronione są sprzężenia zwrotne). Wystarczy wobec tego w odpowiedniej kolejności

wywołać funkcje obsługujące moduły, aby wykonać całą sekwencję pomiarową, czyli utworzyć drzewo zgodne ze schematem graficznym.

Na początek odszukiwane są moduły będące źródłem danych, czyli moduły wejściowe. Rozpoznawane są one po typie przy przeszukiwaniu listy wszystkich wybranych przez użytkownika modułów. Moduły wejściowe zostają włączone do listy modułów startowych typu START_LINII.

Następnie budowana jest lista struktur danych typu OBSLUGA_OKIEN. Struktura taka składa się ze wskaźnika na funkcję obsługi modułu, wskaźnika na parametry funkcji PAR_F i wskaźnika na następną strukturę typu OBSLUGA_OKIEN.

Dane pierwszego elementu tej listy są wypełniane na podstawie danych o module, którego wskaźnik do danych jest pobierany z listy struktur typu START_LINII. Wskaźnik na następny moduł w strukturze pomiarowej jest pobierany z pierwszej pozycji (o numerze 0) tablicy 'wyjscie' zawartej w danych, opisującej moduł w strukturze typu DEF_OKNA. Analogicznie dołączony jest następny element listy typu OBSLUGA_OKIEN. Jeżeli wskaźnik 'wyjscie[i]' równy jest NULL, to pobierany jest następny wskaźnik z tablicy. Jeżeli wykorzystane zostały już wszystkie wskaźniki z tablicy 'wyjścia', to pobierany zostaje następny wskaźnik z listy struktur typu START_LINII.

W rezultacie tych operacji zostaje zbudowana lista struktur zawierających wskaźniki do funkcji obsługujących moduły. Dla przeprowadzenia jednego cyklu pomiarowego należy kolejno wywoływać funkcje wg wskaźników z listy struktur.

**Definicje struktur danych (fragmenty kodu).**

```
typedef struct{
   HWND   hOkna;            // wskaznik na modul w polu roboczym
          HLOCAL poprzednie_na_liscie;
          HLOCAL nastepne_na_liscie;
          int    flaga;            // dana pomocnicza
} LISTA_OKIEN;
typedef LISTA_OKIEN   near *WSK_LISTE_OKIEN;

typedef struct {
   WSK_LISTE_OKIEN startowe;     // wskaźnik na moduł wejściowy
   void near *nastepne_startowe; // następny element listy
   WSK_OBSLUGE_OKIEN obsluga;
}START_LINII;
typedef START_LINII near *WSK_START;

typedef struct{
  HWND wejscie[2];// wskaźniki na moduły podłączone do wejść
  float wynik;   // wynik zwracany przez funkcje obsługi modułu
  union{
    struct{ int      a,          b,     c,d; }iiii;
    struct{ HWND     a; int      b,     c,d; }hiii;
    struct{ HWND     a; HDC      b; int c,d; }hhii;
    struct{ LONG     a; unsigned b; }lu;
```

```
    struct{ unsigned a,b,c;LOCALHANDLE d; HFILE e;}uuumz;
   }dane;          // parametry dla różnych funkcji
}PAR_F;

typedef struct{
   UINT   typ_okna;        // typ okna
   UINT   nr_okna;         // numer okna w polu roboczym
   POINT  pozycja;         // pozycja okna w polu roboczym
   HWND   wyjscie[4];      // wskaźniki na moduły wyjściowe
   HWND   display;         // wskaźnik na okno wspomagające
   BOOL   flaga;           // dana pomocnicza
   PAR_F  par;             // parametry dla funkcji obsługi
} DEF_OKNA;
typedef DEF_OKNA near *WSK_DEF_OKNA;

typedef struct{
   void (*funkcja)(PAR_F*); // wskaźnik na funkcje obsługi
   PAR_F *wsk_par;          // wskaźnik na parametry funkcji
   void  *nastepne;         // następny element listy
}OBSLUGA_OKIEN;
typedef OBSLUGA_OKIEN near *WSK_OBSLUGE_OKIEN;
```

**Procedura budująca listę ze struktur typu OBSLUGA_OKIEN (fragment kodu).**

```
WSK_OBSLUGE_OKIEN lan(HWND hOkna,WSK_OBSLUGE_OKIEN wskObslugi)
{
 WSK_DEF_OKNA wskDefOkna;
 int          nr_wyjscia;

        // pobranie wskaźnika do struktury opisujacej okno
 wskDefOkna = (WSK_DEF_OKNA)GetWindowWord(hOkna,0);
 wskDefOkna->flaga = TRUE;
 wskObslugi->funkcja = narzedzie[wskDefOkna->typ_okna].funkcja;
 wskObslugi->wsk_par = &(wskDefOkna->parametry);
 for(nr_wyjscia=0;nr_wyjscia<4;nr_wyjscia++)
{
    hOkna1 = wskDefOkna->wyjscie[nr_wyjscia];
    if(hOkna1!=NULL){
       wskObslugi->nastepne =
           (void near*)LocalAlloc(LPTR,sizeof(OBSLUGA_OKIEN));
       wskObslugi = (WSK_OBSLUGE_OKIEN)wskObslugi->nastepne;
       wskObslugi = lan(hOkna1,wskObslugi);
    }
 }
 return wskObslugi;
}
```

**Procedura wykonująca cykle pomiarowe (fragment kodu).**

```
void Pomiar(WSK_START zacznij)
{
  WSK_OBSLUGE_OKIEN wskObslugi = NULL;
  WSK_START         start_linii;

  run = TRUE;     // flaga startu pomiarów
  while(run)      // pętla pomiarowa
  {
    start_linii = zacznij;    // początek pętli
    while(start_linii!=NULL) // pętla dopóki są linie pomiarowe
    {                  // (może być jedna lub kilka równoleglych)
                               // początek linii pomiarowej
       wskObslugi = (WSK_OBSLUGE_OKIEN)(start_linii->obsluga);
       while(wskObslugi!=NULL && run)
       {
// wykonanie funkcji obslugi jednego elementu linii pomiarowej
          (wskObslugi->funkcja)(wskObslugi->wsk_par);
         wskObslugi = (WSK_OBSLUGE_OKIEN) wskObslugi->nastepne;
       }
                                    // następna linia pomiarowa
       start_linii = (WSK_START)start_linii->nastepne_startowe;
    }
   }
 return;
}
```

**Literatura:**

[1] Microsoft Windows Software Development Kit ver.3.0 Reference. Microsoft Corporation.

[2] Jemas McCord. Developing Windows Applications with Borland C++ 3.1 Seconnd Edition SAMS.


**INSTYTUT MASZYN MATEMATYCZNYCH**


OFERUJE

## KURSY KOMPUTEROWE

poranne, popołudniowe, sobotnie,
dla początkujących PODSTAWOWE i UKIERUNKOWANE,
dla zaawansowanych SPECJALISTYCZNE,
*prowadzone przez doświadczonych specjalistów*
*w nowoczesnych laboratoriach IMM,*
gdzie każdy słuchacz pracuje na osobistym komputerze
dostępnym do ćwiczeń także poza godzinami zajęć
i otrzymuje nieodpłatnie materiały szkoleniowe.

**SZKOLIMY W NASTĘPUJĄCYM ZAKRESIE:**

System operacyjny DOS, WORDPERFECT, DRAWPERFECT,
LOTUS, DBASE, CLIPPER, AUTOCAD;
W środowisku WINDOWS: WORD, QR-TEKST, AMI-PRO, WORKS,
EXCEL, ACCESS, CORELDRAW;
System operacyjny UNIX - podstawy, użytkowanie;
Sieci lokalne - NOVELL, WINDOWS FOR WORKGROUPS, poczta elektroniczna;
PC SERVIS, JĘZYK C++;
KOMPUTER W BIURZE POSELSKIM.

Również inne kursy na zamówienie. Istnieje możliwość uzgodnienia zakresu i terminu, a także przeprowadzenia szkolenia u klienta; na zamówienie prowadzimy kursy w języku angielskim.

Polecamy przeprowadzone już w licznych przypadkach
**KURSY DLA KIEROWNICTWA FIRMY.**

Opłata za 41-godzinny, 2-tygodniowy kurs podstawowy - 400 zł.

Instytut prowadzi DORADZTWO KOMPUTEROWE, wykonuje EKSPERTYZY
i inne OPRACOWANIA DOTYCZĄCE KOMPUTERYZACJI
I ZASTOSOWAŃ INFORMATYKI W PRZEDSIĘBIORSTWACH,
przyjmuje zlecenia na OPRACOWANIE OPROGRAMOWANIA.

Informacja:

ul. Krzywickiego 34, 02-078 Warszawa,
**fax** 299270, **tlx** 817880, **tel** 299164 lub 6213351
**e-mail** imasmat@frodo.nask.org.pl


# INSTYTUT MASZYN MATEMATYCZNYCH


*Oferujemy*

## KOMPUTEROWY SYSTEM KONTROLI DOSTĘPU I REJESTRACJI CZASU PRACY SKR

SKR identyfikuje karty magnetyczne, prowadzi ewidencję czasu pracy pracowników, a także może generować informacje dla komputerowych systemów obliczania plac. System składa się z minikomputera IBM PC oraz jednego lub wielu czytników kart magnetycznych (do 32) i oprogramowania CHRONOS działającego pod systemem DOS w wersji 3.10 lub nowszej.

Czytnik kart magnetycznych **AS-1400**:
- zasilanie +5V lub +12V (zasilacz wchodzi w skład wyposażenia),
- karty magnetyczne - wg ISO 2894, ścieżka 2,
- klawiatura dwunastoznakowa,
- przesuw ręczny,
- długość kodu - do 40 znaków,
- pamięć do 5000 rejestracji we/wy,
- temperatura pracy: +5 do +45 °C,
- żywotność głowicy do 1 mln odczytów.

CHRONOS w wersji standardowej pozwala:
- redagować dokumenty opisujące pracowników (do 2000), usuwać je, modyfikować i drukować,
- definiować parametry dotyczące pracowników, w tym czas pracy, różne statusy i in.,
- ustawiać czas i datę w komputerze i czytnikach,
- programować czytnik, np. wprowadzać i usuwać numery kart akceptowanych,
- rejestrować wejścia i wyjścia pracowników i osób obcych (zaproszonych),
- przeszukiwać informacje o poruszaniu się pracowników, np. uzyskać informacje o pracownikach, którzy w określonym czasie wychodzili z zakładu,
- generować informacje dla istniejących w przedsiębiorstwie systemów kadrowo -płacowych,
- drukować na drukarce zestawienia o wejściach i wyjściach pracowników, raporty miesięczne, dzienne i inne.

*Ten najtańszy system rejestracji i kontroli czasu pracy ułatwi pracę Twoim pracownikom .*


# INSTYTUT MASZYN MATEMATYCZNYCH


*Jeśli:*
*- projektujecie lub uruchamiacie systemy mikroprocesorowe,*
*- musicie zaprogramować układ scalony,*
- prowadzicie pomiary wielkości elektrycznych
*wykorzystajcie oferowane przez Instytut profesjonalne narzędzia*

## MIKROPROCESOROWY SYSTEM WSPOMAGANIA PROJEKTOWANIA MSWP-92

(współpracujący z minikomputerami IBM PC)

* emulatory układowe procesorów 8080, Z80, 8085, 8086/88, 80286, 8035/39/40/48/49/50, 8031/51/52/44/154/652/851, 80515/535, 80C552/562
* programator EP-11 dla pamięci EPROM i struktur logicznych PAL
* uniwersalny programator/tester UP-1
* analizator stanów logicznych ASL-24
* symulator pamięci stałych SYM-11
* lampa kasująca EE-12

## SYSTEM POMIAROWY VIRT II

(karty do minikomputera IBM PC)

* RC681 - mostek RLC
* AD611B - przetwornik 12-bitowy A/C, C/A
* IO601A - cyfrowe we/wy
* WD671A - rejestrator przebiegów
* PG621A - generator przebiegów
* ROM-DISK - zastępuje mechanizm dyskowy "A"

*Serwis oraz bezpłatne konsultacje w 12-miesięcznym okresie gwarancyjnym.*
*Pełna dokumentacja eksploatacyjna. Przystępne ceny.*
*Możliwość odpłatnego wypożyczenia.*

**ZAPRASZAMY**