MICROINFORMÁTICA

Antonio Geraldo da Rocha Vidal

CLIPPER®

VERSÃO SUMMER 87

Vol. 1

COMANDOS
FUNÇÕES
COMPILAÇÃO
EXECUÇÃO

LIVROS TÉCNICOS E CIENTÍFICOS EDITORA LTDA.

CLIPPER®

VERSÃO SUMMER 87

Vol. 1

CONHEÇA TAMBÉM

Albrecht — Análise Numérica
Amaral — Projeto Estruturado: Fundamentos e Técnicas
Andrade — Introdução ao Paradox
Arakaki/Arakaki — Turbo C 2.0
Arakaki/Salles — Fundamentos de Programação C
Araribóia — Inteligência Artificial
Araújo — Xenix
Arthur — Unix: Programação Shell
Ashley/Fernandez — Excel Avançado
Avery — Logo e o Apple
Back — CP/M: Sistema Operac. para Microcomputadores
Barbuto — Programas Basic para IBM PC
Bastos — Programação Cobol
Beason — MS Word 5.0
Bennett/Randall — LaserJet
Bianchi — Introdução aos Microcomputadores
Bolocan — Conquistando o Symphony
Boratto — Basic para Engenheiros e Cientistas
Borges — Basic: Aplicações Comerciais
Borges/Schmitz — Projeto de Circuitos Integrados
Bove/Rhodes/Thomas — A Arte de Desktop Publishing
Bui — Planejamento Executivo com Basic
Busch — PC/MS-DOS 4.0
Camarão — Glossário de Informática
Campbell/Siminoff/Yates — Micros de Lógica Siclair
Cardoso — Programação Estruturada em Cobol
Caruso/Steffen — Segurança em Informática
Carvalho — Automação de Escritórios
Cerícola — Banco de Dados
Chinelato — O & M Integrados à Informática
Coburn — Informática na Educação
Cordeiro Paulo — Cobol: Técnicas e Dispositivos Especiais
Cruz — Guia Prático de Sistemas Gráficos Apple II
Dachs — Estatística Computacional
Dantas — Guerrilha Tecnológica
David/Ziegler/David — Desktop Publishing com LaserDesk
Davis — Análise e Projetos de Sistemas
DeLucchi — AutoCAD
Dias — O Sistema de Informação e a Empresa
Dias/Gazzaneo — Projetos e Sistemas de Proc. de Dados
Dobler — Linguagem C
Eadie — Minicomputadores: Teoria e Prática
Erskine — Linguagem Assembly 8086 e 8088
Evans — Norton Utilities Versão 4.0
Evans — Norton Utilities Versão 4.5
Fernandes/Kugler — Gerência de Projetos de Sistemas
Furtado — Teoria dos Grafos: Algoritmos
Gaines/Shaw — A Interação Computador-Usuário
Gane — CASE: O Relatório Gane
Gane — Desenvolvimento Rápido de Sistemas
Gane/Sarson — Análise Estruturada de Sistemas
Garcia/Nogueira — dBase Total
Genaro — Sistemas Especialistas
Gillenson/Goldberg — Planej. Estrat. Análise de Sist. e Proj. de Banco de Dados
Grillo — Turbo Pascal V.5.0 e 5.5
Grillo — Turbo Pascal V.4.0
Grillo — Turbo Pascal V.3.0
Grillo — Programação Estruturada com Fortran e Watfiv
Gruenberger — Programando no Apple
Guimarães/Lages — Algoritmos e Estruturas de Dados
Guimarães/Lages — Introdução à Ciência da Computação
Helt/Bonfim/Martins — PECS: Estatística em Microcomputadores
Hollins — WordStar 5 e 5.5
Holtz — Dominando o Ventura
Holzgang — PostScript
Hursch/Hursch — SQL
Hyman — OS/2
Jantz — Ventura Publslisher 2.0
Kamin — Disco Rígido no PC
Katzan — Segurança de Dados em Computação
Kramer/Parker — Aldus PageMaker 4.0
Larsen — Computação para Crianças
Leventhal — 80386 Guia de Programação
Lomuto/Lomuto — Introdução ao Unix
Maciel — StoryBoard Plus
Maciel/Moreira/Pinheiro — Transcrição de Dados
McCaleb — A Microinformática na Empresa
McNitt — Simulação em Basic
Melendez — Prototipação de Sistemas de Informações
Mercês — Introdução aos Sistemas On-Line
Miller — Dominando o TK-90X e TK-95
Módulo — Linguagem C: Programação e Aplicações
Módulo — Toda a Verdade Sobre o Vírus no PC
Neto — Introdução à Compilação
Niva — ZIM
Nogueira/Garcia — Avaliação e Seleção de Sistemas
Nunes — Comunicação de Dados
Nunes — Introdução aos Sistemas Operacionais
Pacitti — Programação: Princípios
Pacitti/Atkinson — Programação e Métodos Computacionais
Parikh — Reengenharia de Software
Parker — Editoração Eletrônica com WordPerfect
Passos — Inteligência Artificial
Persiano/Oliveira — Introdução à Computação Gráfica
Person/Rose — Microsoft Windows
Pimentel — Introdução ao Turbo Basic
Praça — Cobol para IBM PC
Prado — Administração de Projetos com PERT/CPM
Prates — Basic Aplicado: Um Enfoque Profissional
Press — IBM PC e suas Aplicações
Puccini/Pizzolato — Programação Linear
Rettig/Moody — FoxPro
Ribeiro — Introdução aos Sistemas Especialistas
Roberts — Turbo Prolog
Rubel — Programando em dBase III Plus
Sá Carvalho — Análise de Sistemas
Schmitz/Teles — Pascal e Técnicas de Programação
Schneider — Multiplan: Manual do Usuário
Seabra — Meu Primeiro Encontro com o Microcomputador MSX
Seabra — WordPerfect
Silva/Heibel — SuperCalc 2
Silva/Heibel — SuperCalc Total
Smith — IBM PC AT: Guia de Programação
Soares Neto/Porto/Lima — Introdução à Teleinformática
SofCAD — Clipper com Gráficos
Souza — Chart 3.0
Spiegel — Rotinas Prontas MSX
Stevens — C++
Strack — GPSS: Modelagem e Simulação de Sistemas
Strack — Sistemas de Processamento Distribuído
Swensson — Programando em Lotus 1-2-3: Macros
Swensson — Lotus 1-2-3 V. 2.2 & Allways
Swensson/Figueira — Quattro: A Planilha Moderna
Swensson/Pombeiro — Lotus 1-2-3
Szajnberg — Eletrônica Digital
Toigo — Recuperação de Sistemas de Informação
Tori/Arakaki — Fundamentos de Computação Gráfica
Townsend — Programando em dBase IV
Trevisan — Curso de Programação Basic
Vasconcellos — Componentes Eletrônicos e Processamento
Vega — Turbo Pascal 5.5

Vidal — Clipper, Vols. 1 e 2
Vidal — Clipper 5.0 Vols. 1, 2, 3 e 4
Von Staa — Engenharia de Programas
Ward — Desenvolvimento de Sistemas sem Complicação
Weiss/Kulikowski — Sistemas Especialistas
Wiese — Home Video
Winston — Inteligência Artificial
Wirth — Modula 2
Yallouz — Programas Residentes no IBM PC
Yu/Harrison — Lotus 1-2-3 Versão 3 Avançado
Zakir — Redes Locais
Zucchi — Transmissão de Dados

CARTÕES DE REFERÊNCIA

Alves — Apple II
Alves — dBase II
Cintra Leite — 8086/8088
Cintra Leite — MS-DOS/BIOS: Interrupções
Cintra Leite/Prates — Turbo Pascal Toolbox V.4.0
Cintra Leite/Prates — Turbo Pascal Toolbox V.3.0
Cintra Leite/Prates — Turbo Debugger V.1.0
Cintra Leite/Prates — Turbo Pascal V.4.0
Cintra Leite/Prates — Turbo Pascal V.5.0
Conart — Xenix
Dichaune — WordStar V.3.45
Inoue/Prates — Word V.5.0
Inoue/Prates — Word V.4.0
Inoue/Prates — Lotus 1-2-3 V. 2.0
Inoue/Prates — Lotus 1-2-3 V. 2.2
Inoue/Prates — Lotus 1-2-3 V. 3
Inoue/Prates — Quattro V.1.0
Inoue/Prates — SuperCalc 4
Inoue/Prates — SuperCalc 5
Kleiber — Cobol 80
Maciel — PC StoryBoard
Prates — Clipper V.Summer 87
Prates — Clipper V.Autumn 86
Prates — CP/M
Prates — dBase III
Prates — dBase III Plus
Prates — dBase IV
Prates — FoxBase+ V.2.0
Prates — GW Basic
Prates — MBasic
Prates — MS-DOS até V.2.1
Prates — MS-DOS até V.3.2
Prates — MS-DOS V.3.3
Prates — MS-DOS V. 4.01
Prates — Norton Utilities V.4.0
Prates — Norton Utilities V. 4.5
Prates — PC Basic
Prates — Sidekick
Prates — Turbo Basic
Prates — WordStar V.4.0
Prates/Cintra Leite — PC Tools Deluxe V.4.2
Prates/Cintra Leite — QuickBasic V.4
Prates/Cintra Leite — QuickBasic V. 4.5
Prates/Cintra Leite — Turbo C V.1.5
Prates/Cintra Leite — Turbo C V. 2.0
Prates/Cohen — Dataflex V.2.3
Ramalho/Prates — Turbo Pascal V.3.0
Silva/Heibel — Supercalc/Supercalc 2
SoftCAD — CLBC V. 2.0
Taveira — Chart 3.0
Tortello — Autocad Release 9.0
Valença — Open Acess
Vieira — Ventura V. 2.0
Ziegler — Pagemaker V.3.0

CLIPPER®

VERSÃO SUMMER 87

Vol. 1

Antonio Geraldo da Rocha Vidal

- Professor da Faculdade de Economia e Administração da USP, lecionando a disciplina Processamento de Dados.
- Coordenador de Projetos do Instituto de Administração da FEA/USP, na área de Processamento de Dados e Informática, especialmente em microcomputação.
- Consultor de Empresas nas áreas de Microinformática e Sistemas de Informação.

EDITORA

CLIPPER é marca registrada da Nantucket, Inc.

Revisão de originais: Sandra Pássaro

Arte/Fotolito: Paulo Andrade/Amauri Antunes

Capa: AG Comunicação Visual, Assessoria e Projetos

Produção gráfica: Gilcinei Gomes dos Santos

1ª edição: 1989
Reimpressão — 1989 (duas), 1990 (quatro), 1991 (seis), 1992 (quatro) e 1993

CIP-Brasil. Catalogação-na-fonte.
Sindicato Nacional dos Editores de Livros, RJ.

Vidal, Antonio Geraldo da Rocha
V691c Clipper / Antonio Geraldo da Rocha Vidal. — Rio de Janeiro: LTC — Livros Técnicos e Científicos Ed., 1989.
2v (Microinformática)

Bibliografia.
Apêndice.

1. Clipper (Gerência de banco de dados). 2. dBaseIII (Programa de computador). I. Título. II. Série.

89-0449 CDU — 001.64.25

ISBN: 85-216-0602-8 (Obra completa)
ISBN: 85-216-0639-7

Matriz:
Rua Vieira Bueno, 21. São Cristóvão. CEP 20920 — Rio de Janeiro — RJ
Telex: (21) 36909 LTCE BR
FAX: (021) 580.0187 LTCE RJ
Tels.: (021) 580-6055 / 580-9174 / 580-7147 (Vendas)

Filial:
Rua Euclides Andrade, 15. CEP 05030 — Sumarezinho — SP
Telex: (11) 81430 LTCE BR
FAX: (011) 580.5633 LTCE SP
Tels.: (011) 864-6066 / 864-5799

Dedicatória

Aos meus pais, que fazem de sua vida a realização de seus filhos.

Agradecimentos

A todos os meus amigos e em especial ao Prof. Nicolau Reinhard pelo apoio recebido.
Ao pessoal da LTC, pelo incentivo, revisão e coordenação.

Marcas Registradas

Clipper e Nantucket são marcas registradas da Nantucket Corporation.

Foxbase é marca registrada da Fox Software, Inc.

QuickSilver é marca registrada da Wordtech Systems, Inc.

Framework, dBASE Tools, dBASE II, dBASE III e dBASE III Plus são marcas registradas da Ashton Tate, Inc.

Lotus 1-2-3 é marca registrada da Lotus Development Corporation.

SideKick é marca registrada da Borland International, Inc.

WordStar é marca registrada da MicroPro International Corporation.

IBM PC, IBM PC/XT e IBM PS/2 são marcas registradas da International Business Machines, Inc.

Microsoft C. 5.0, MS-Windows e MS-DOS são marcas registradas da Microsoft Corporation.

PLINK86 é marca registrada da Phoenix Software Associates, LTD.

Quickcode é marca registrada da Fox & Geller, Inc.

Genifer é marca registrada da Bytel Corporation.

dFLOW é marca registrada da WallSoft Systems, Inc.

Prefácio

Este livro é destinado a profissionais e a usuários de microinformática que já possuem alguma experiência no desenvolvimento de Sistemas de Informações com a utilização do software dBASE III e gostariam em implementá-los utilizando o poder do Clipper.

Nosso objetivo é tentar fornecer aos que já utilizam a linguagem de programação **dBASE III**, suficientes conhecimentos do **Clipper**, o compilador do dBASE III, que permitirão a construção de sistemas mais eficientes, sofisticados e de alto nível profissional. Para tanto organizamos este livro introdutório em doze capítulos, em dois volumes, cujos conteúdos descrevemos sucintamente a seguir.

VOLUME 1

Capítulo 1: Apresenta uma introdução ao Clipper incluindo: uma visão geral do dBASE III e do Clipper, as principais diferenças entre um Compilador e um Interpretador, e uma breve consideração sobre as diferentes versões do Clipper.

Capítulo 2: Aborda as principais diferenças entre o Clipper e o dBASE III, assunto de especial interesse para aqueles que desejarem converter seus programas escritos em dBASE para Clipper, implementando-os com os vários recursos disponíveis apenas no Clipper. Nele incluímos: principais implementações e restrições do Clipper em relação ao dBASE III Plus e principais equivalências, ou seja, recursos que existem no dBASE III, mas são implementados de maneira diferente no Clipper.

Capítulo 3: Neste capítulo apresentamos algumas noções simples e úteis sobre Sistemas de Informações. Abordamos Arquivos e Bancos de Da-

dos, elementos fundamentais para a criação de Sistemas de Informação utilizando o Clipper.

Capítulo 4: Apresenta a "Linguagem Clipper", detalhando seus aspectos técnicos, estrutura de dados, operadores disponíveis e recursos para seleção e edição de dados.

Capítulo 5: Apresenta todos os comandos do Clipper, classificados de acordo com seu objetivo ou categoria. Cada comando é apresentado de acordo com seu propósito, sintaxe, forma de utilização, biblioteca de rotinas onde se encontra, exemplos e outros comandos e funções à ele relacionados.

Capítulo 6: De forma semelhante aos comandos, apresenta todas as funções do Clipper, classificadas de acordo com seu propósito ou categoria. No Clipper, as funções são tão importantes quanto os comandos.

Capítulo 7: Aborda a facilidade de criação de ajuda ao usuário (Help) implementada no Clipper. Através da tecla F1 é acessado um programa denominado HELP.prg (criado por você) que poderá fornecer, "on-line", informações de ajuda ao usuário das suas aplicações desenvolvidas em Clipper.

Capítulo 8: Aborda a compilação, a "link-edição" e a execução dos programas escritos em Clipper. São também sugeridas algumas técnicas simples para torná-las mais rápidas, organizadas e eficientes.

VOLUME 2

Capítulo 9: Apresenta os utilitários do Clipper, entre eles e o seu "Debugador" (utilitário que auxilia na detecção e correção de erros) e alguns "softwares" utilitários internos e externos, que podem ser utilizados, com vantagem, para complementar os recursos já oferecidos pelo Clipper.

Capítulo 10: Apresenta um exemplo real de um Sistema de Informação desenvolvido em Clipper: o Sistema MLD, de emissão de Mala-Direta. Além de descrevê-lo, incluímos a estrutura de seus arquivos e o código-fonte dos programas. Você poderá estudá-lo, para verificar a aplicação prática dos diversos comandos e funções do Clipper, e também utilizá-lo como base para o desenvolvimento de seus próprios sistemas. Neste capítulo fazemos também inúmeras reco-

mendações para o aprimoramento do desenvolvimento dos seus sistemas em Clipper. Caso seja de seu interesse, poderá ser fornecido um disquete contendo o código-fonte e os arquivos do Sistema MLD.

Capítulo 11: Aborda a utilização do Clipper no desenvolvimento de aplicações em ambiente de Rede Local de Microcomputadores (LAN). São discutidos os comandos de rede do Clipper, técnicas de programação e efeitos do ambiente compartilhado nos comandos. São também apresentados exemplos de funções-de-usuário que facilitam o desenvolvimento de aplicações destinadas ao processamento compartilhado em rede.

Capítulo 12: São apresentados apêndices, onde se encontram, de forma sintética, os erros mais comuns que podem ocorrer utilizando-se o Clipper: durante a compilação, durante a "link-edição" e durante a execução dos programas. Também aqui estão incluídos os programas-fonte do Sistema MLD, apresentado no Cap. 10 e alguns exemplos de relatórios por ele emitidos.

Apesar de não ser um livro completo, mas apenas introdutório, esperamos que, com as informações contidas nele, você consiga utilizar o Clipper com facilidade, extraindo o máximo de seus recursos e criando sofisticados Sistemas de Informação. Aproveitamos a oportunidade para agradecer qualquer contribuição que você possa nos trazer, no sentido de correção de falhas ou de sugestões para o aprimoramento da obra.

O Autor

Sumário

1

Introdução

1.1. O QUE É O CLIPPER

Uma das aplicações mais freqüentes e úteis para os computadores é a organização de coleções de dados (ou arquivos), com a finalidade de se elaborar consultas, efetuar cálculos, gerar listagens e relatórios, análises estatísticas etc. Muitas aplicações em Processamento de Dados têm essas características em comum, sendo por isso chamadas de aplicações de **Base de Dados** (ou Banco de Dados), pois baseiam-se em coleções ou arquivos de dados. Por outro lado, as Bases de Dados constituem o ingrediente fundamental dos Sistemas de Informação, cujo objetivo principal é gerar informações para a tomada de decisões.

O Clipper é um completo **Sistema Gerenciador de Base de Dados**, tendo originado-se de um dos softwares de maior sucesso para microcomputadores, o dBASE III.

O dBASE III (cuja denominação significa "data base" ou base de dados), assim como o Clipper, permite criar, manipular e gerenciar Bases de Dados de pequeno e médio porte utilizando microcomputadores da linha IBM-PC/AT/PS-2, baseados em discos magnéticos. Com ele pode-se criar e utilizar rapidamente arquivos de dados das mais variadas naturezas, conforme sua necessidade de gerar informações. O dBASE III é bastante poderoso, flexível, simples de ser utilizado e capaz de produzir verdadeiros Sistemas de Informação, sendo por isso um dos mais vendidos em todo o mundo dentro de sua categoria.

O enorme sucesso alcançado pelo dBASE praticamente determinou o aparecimento do Clipper, que torna as aplicações (programadas) originalmente desenvolvidas em dBASE, depois de compiladas, mais rápidas, eficientes e sofisticadas.

Utilizando o Clipper é possível:

- Criar, organizar, classificar, copiar, selecionar e relacionar conjuntos de arquivos que formam a Base de Dados;

- Adicionar, alterar, eliminar, exibir e listar global ou seletivamente as informações contidas nos arquivos de dados;
- Gerar relatórios padronizados, efetuar automaticamente somas, agregações, contagens e operações aritméticas sobre os valores dos dados armazenados nos arquivos;
- Formatar telas de entrada de dados no vídeo e gerar relatórios, tabelas e listagens complexas na impressora, de acordo com as necessidades do usuário;
- Produzir Sistemas de Informação completos e integrados, com recursos e sofisticações encontrados apenas nos mais modernos softwares que hoje disputam o fabuloso mercado da microinformática.

Em resumo, o Clipper permite a dinamização de aplicações com arquivos de dados, tornando-as mais fáceis e rápidas que as desenvolvidas em uma linguagem de programação tradicional, como Cobol, Basic ou Pascal. Com uma simples, moderna e eficiente linguagem de programação, baseada na linguagem dBASE, permite o encadeamento ordenado e lógico de seus comandos possibilitando rapidamente a definição de programas com alto grau de complexidade e sofisticação, permitindo inclusive interações com outras linguagens como "C" e Assembler, que lhe confere a flexibilidade necessária para a utilização profissional.

1.2. OS COMPILADORES DO dBASE III

Embora nunca formalmente definido como uma linguagem de programação, o dBASE III possui praticamente todos os requisitos de uma. Em termos de popularidade é mais bem sucedido que muitas linguagens consideradas de "sangue-azul". Por isso, não é surpreendente que o dBASE tenha dado origem a **compiladores** como o Clipper, característica típica de uma verdadeira linguagem.

"Clipper" em inglês significa "cortador veloz". Na verdade Clipper é uma alusão aos grandes navios à vela denominados "Clipper's", que eram construídos para cortarem as águas a grande velocidade. Na gíria atual, Clipper significa aquele que é veloz; no nosso caso, "Clipper" é o compilador que torna as aplicações desenvolvidas em dBASE III mais velozes.

São três os principais compiladores da "linguagem dBASE": o **Clipper** da Nantucket Software, o mais tradicional e popular de todos, o Foxbase$^+$ da Fox Software e o Quicksilver da WordTech Systems.

O principal atrativo de um compilador é principalmente a alta velocidade de execução dos programas, que não pode ser alcançada por um interpretador, como é o caso do dBASE III original. Os usuários familiares com os recursos adicionais e implementações que um compilador BASIC dá sobre o conhecido interpretador BASICA podem esperar recursos parecidos de um compilador dBASE como o Clipper. Contudo, vale ressaltar, que no caso de um Gerenciador de Base de Dados, o ganho com a compilação não é tão grande quanto numa linguagem de processamento intensivo, como é o caso do BASIC. A principal razão para isso

é que o Gerenciador de Base de Dados gasta relativamente mais tempo lendo arquivos em disco do que lendo o programa em si. Se em um particular processamento o interpretador gasta, digamos, 50% do tempo esperando a leitura e gravação em disco, o tempo total do processamento não pode ser acelerado pela compilação mais do que um fator 2, não importando quão rápido seja a execução do código compilado nos outros 50% do processamento. Neste caso, a melhor solução é melhorar a velocidade de I/O (leitura e gravação no disco) do equipamento.

Isto não significa, entretanto, que o Clipper não possui muitas vantagens. Mesmo utilizando uma "linguagem de quarta-geração", como o dBASE, muitas operações ainda precisam ser codificadas em detalhe. Apresentações na tela, validações de entrada de dados e operações com variáveis de memória são extremamente beneficiadas com a compilação. Em particular, o usuário irá perceber uma altíssima melhora no tempo de resposta do processamento quando houverem muitas entradas e saídas de tela com validações e checagens de dados.

Por outro lado, quanto à comercialização e distribuição de aplicações desenvolvidas por você, o Clipper fornece duas importantes vantagens ao lado da velocidade de execução.

A primeira é a segurança do código-fonte. Quando se produz uma aplicação com o dBASE III, o código-fonte dos programas tem que ser fornecido ao usuário, o que não acontece com um compilador; ele protege os métodos do programador, seus algoritmos, enfim, sua criação, de alteração e utilização intensional (ou não) por outras pessoas.

A segunda é o baixo custo de licenciamento de cópias da aplicação realizada. Para se executar uma aplicação produzida com o dBASE III é necessária uma cópia do mesmo, o que não acontece com o compilador. O Clipper, é vendido com uma licença para a distribuição ilimitada de cópias de aplicações escritas por você. Cada cópia do Clipper permite que você desenvolva aplicações e as distribua ou comercialize livremente, pois para serem executadas não dependerão mais do compilador.

O Clipper é um software que contém todas as facilidades necessárias para a criação de um Sistema de Informações. Utilizando o Clipper, você pode criar e compilar programas aplicativos sem a necessidade de nenhum outro software, a não ser um "Editor de Textos", para escrever os programas-fonte. Após isso, um programa executável é gerado e pode ser fornecido aos usuários sem a necessidade de qualquer parte do compilador.

1.3. O QUE É UM INTERPRETADOR

Para um computador executar qualquer operação, é necessário que as instruções fornecidas em uma linguagem de alto nível, parecida com a humana, seja interpretada (traduzida) para códigos de uma linguagem de baixo nível, linguagem

de máquina, que o computador entende. Usando um software interpretador, como o dBASE III, cada linha de um programa é lida, interpretada, convertida para código de máquina, e então executada individualmente cada vez que o programa é "rodado". O interpretador "passa" por cada linha do programa-fonte e se nenhum erro é encontrado todas as linhas são executadas.

Para desenvolver um programa usando um interpretador, você cria um arquivo contendo, de forma ordenada e lógica, os vários comandos e funções que compreendem suas instruções. Depois de haver digitado o programa através de um editor de textos, você o executa sob o controle do interpretador. Este "interpreta" o programa (chamado de código-fonte) linha por linha até a última. Após uma linha ser verificada de acordo com a sintaxe (regras dos comandos) o interpretador converte-a do código-fonte para a linguagem de máquina que é realmente entendida e executada pelo computador, produzindo os resultados desejados. Este processo sempre se repete da primeira à última linha do programa.

Na verdade, o código-fonte somente é verificado e convertido após uma porção do código já ter sido executada. Em qualquer caso, sempre que um programa é escrito numa linguagem interpretada de alto nível, como o dBASE III ou o BASIC, o código-fonte precisa ser analisado e convertido para a linguagem de máquina antes que cada comando possa ser executado. Se um erro de sintaxe é encontrado em alguma linha, a execução do programa cessa imediatamente e o interpretador mostra uma mensagem de erro apropriada na tela. Você deve então retornar ao editor de textos, corrigir o problema e novamente executar o programa sob o controle do interpretador. Este imediato retorno à execução do programa é de grande ajuda para a correção de erros durante o desenvolvimento inicial do programa. Porém, depois de você ter corrigido o primeiro problema, o interpretador novamente analisa, converte e executa cada linha, iniciando da primeira. Se um novo problema é encontrado algumas linhas abaixo do primeiro problema, a execução novamente será cancelada e nova mensagem de erro será apresentada. Este processo se repete até que todo o programa esteja perfeito, isto é, não contenha erros.

Infelizmente você não tem a opção de salvar o código de máquina correto que é criado em cada "rodada" do programa, ou de desligar o verificador de erros das linhas. Cada vez que você executa o programa, cada linha é minuciosamente examinada pelo interpretador, o que aumenta significativamente o tempo de execução.

A grande vantagem dos interpretadores é, entretanto, a facilidade com que os programas são desenvolvidos, uma vez que os erros podem ser corrigidos assim que forem detectados, pois execução é interrompida exatamente na linha onde ocorrem. Uma vez corrigido, o programa pode ser imediatamente rodado. Este é exatamente o caso do dBASE III.

1.4. O QUE É UM COMPILADOR

Da mesma forma que um interpretador, um compilador checa cada linha do programa mas, ao contrário do interpretador, apresenta apenas o número da linha

e a respectiva mensagem de erro para cada problema encontrado, até que seja alcançado o final do programa.

Um compilador efetua um passo intermediário de conversão do **código do programa-fonte** (declarações escritas numa linguagem de alto nível) para o chamado **código-objeto** (código em linguagem de máquina), gravando todas as linhas assim convertidas em um arquivo que recebe a extensão ".obj".

O código-objeto não pode ser executado até ser unido, ou melhor, encadeado ("linked" ou "link-editado") com uma série de rotinas de biblioteca pré-programadas por quem desenvolveu o compilador. Estas rotinas são necessárias para a criação de um módulo auto-executável, conhecido como módulo de carga ou executável, que recebe a extensão ".exe". Após o código-objeto ter sido encadeado ou "link-editado" com sucesso, o arquivo executável passa a conter todas as instruções necessárias (já em código de máquina) para executar as instruções do programa a partir do sistema operacional, bastando para isso digitar o seu nome.

O programa é agora executado com a máxima velocidade possível, pois não é mais necessário checar e converter cada linha como no caso do interpretador.

Para gerar um programa utilizando um compilador, você inicialmente cria, através de um editor de textos, um arquivo contendo os vários comandos e funções que compreendem o seu programa. Este é o programa-fonte ou código-fonte.

Após haver terminado de escrever o **código-fonte** de seu programa você o submete ao compilador. Este é analisado pelo compilador linha por linha, de forma semelhante como faz um interpretador. A diferença é que o compilador traduz cada linha para a linguagem de máquina, até o final do programa, não as executando, mas gravando um arquivo com o código traduzido, chamado **código-objeto**. Para cada linha, se um problema é encontrado, uma mensagem de erro associada é apresentada, e o arquivo contendo o código-objeto não é gravado.

Após o fim da compilação você terá uma lista, linha por linha, de todos os erros que ocorreram, e deverá, então, retornar ao editor de textos para corrigí-los, tomando cuidado para não criar novos erros.

Uma vez corrigidos os erros, o programa deverá ser novamente compilado, e o processo repetido até que não sejam acusados mais erros. Uma operação demorada, portanto.

Após a compilação, o código-fonte permanece intacto, e poderá ser reutilizado para alterações e implementações no programa, que exigem novas compilações. Somente através dele isto será possível.

O código-objeto é praticamente indecifrável, e constitui um método seguro de gravar o código-fonte convertido. Portanto, ao contrário do que ocorre com o interpretador, não será mais necessário repetir o demorado processo de checagem e tradução de cada linha do programa.

O código-objeto não possui, entretanto, todas as instruções necessárias à execução do programa. Na verdade, cada comando de uma linguagem de alto nível é uma rotina em uma linguagem de nível mais baixo. Estas rotinas (já em código de máquina) estão gravadas em bibliotecas que devem ser encadeadas com o código-objeto gerado pelo compilador para produzir o programa auto-executável. O Clipper (como o dBASE III) é escrito em linguagem "C", assim, após a obtenção

do código-objeto de seus programas, estes deverão ser encadeados com as bibliotecas de rotinas escritas em "C", que também fazem parte do Clipper.

A operação de encadeamento ou "link-edição" é feita por um programa especial chamado "link-editor", que gera um módulo ou programa que pode ser executado sob o sistema operacional. O programa executável gerado é consideravelmente maior que o código-objeto que lhe deu origem, devido à inclusão das rotinas de bibliotecas e outras, que permitirão que ele seja executado diretamente a partir do Sistema Operacional. Um comando do Clipper (como por exemplo um IF) pode requerer que 20 a 30 instruções em linguagem de máquina sejam incorporadas ao código-objeto original para que o comando seja realmente executado pelo computador.

Durante o processo de encadeamento ("link-edição"), muitos erros também podem ocorrer. Por exemplo, podem haver referências duplicadas a rotinas ou roti-

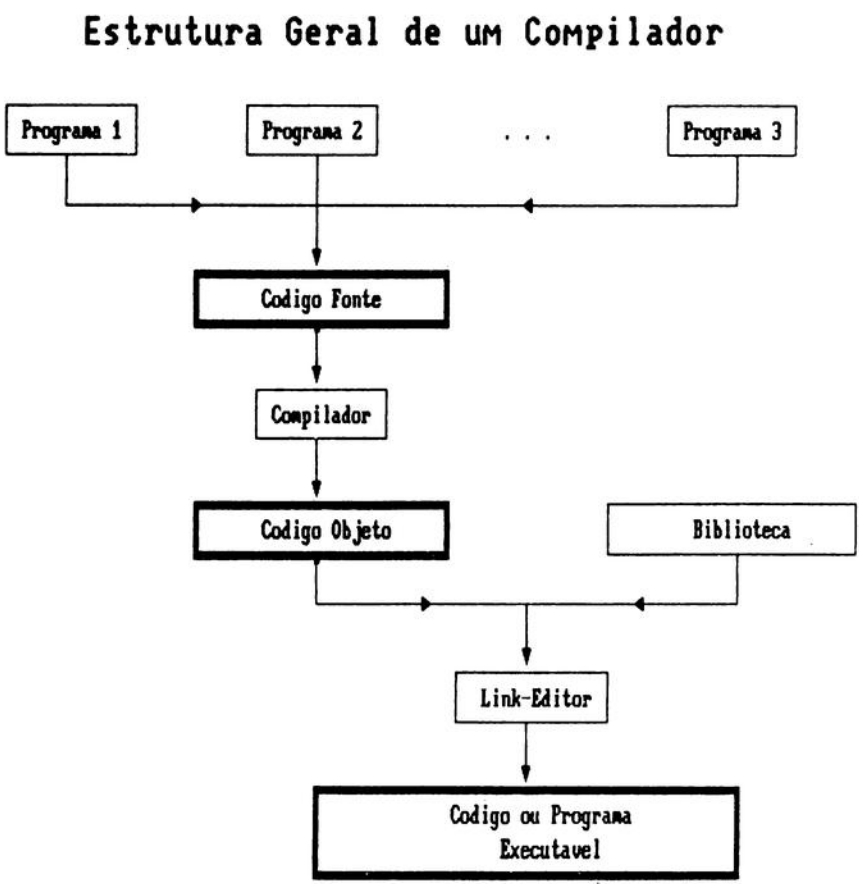


Fig. 1.1.

nas indefinidas. Estes erros precisam ser corrigidos, ou alterando o código-fonte, ou incluindo novos módulos-objeto que não tenham sido incluídos durante a operação de encadeamento. Nestes casos, tanto a compilação como o processo de encadeamento ("link-edição") deverão ser repetidos, até que não existam mais erros e o módulo executável seja corretamente gerado.

O programa executável é auto-suficiente. Nele estão contidas (em linguagem de máquina) todas as instruções necessárias à sua execução sob o Sistema Operacional. Não será mais necessário para sua utilização o Clipper, o código-fonte, o código-objeto, o "link-editor" ou as bibliotecas de rotinas, mas apenas os arquivos de dados, de índices ou de variáveis de memória. Com uma linguagem interpretada, como o dBASE III, é sempre necessária a presença do interpretador para executar um programa, no caso o próprio dBASE.

A Fig. 1.1 ilustra a estrutura geral de um compilador como o Clipper.

1.5. VANTAGENS E DESVANTAGENS DE UM COMPILADOR

Vantagens:

- Maior velocidade e eficiência na execução dos programas;

  - O código executável é literalmente livre de erros de sintaxe, pois já foi verificado durante a compilação;
  - Total proteção ao código-fonte, que fica em poder de seu criador, pois é virtualmente impossível decifrar-se o módulo executável;
  - Possibilidade de utilização de vários recursos avançados, normalmente não disponíveis nos interpretadores;
  - Possibilidade de distribuição ou comercialização de um ilimitado número de cópias do módulo executável das aplicações desenvolvidas;
  - Possibilidade de integração direta com outras linguagens também compiladas, possibilitando total flexibilidade de desenvolvimento de aplicações, garantindo um nível altamente profissional;
  - Possibilidade de executar programas e acessar arquivos em ambiente multiusuário sem licenças adicionais.

Desvantagens:

- Programação em geral mais sofisticada e difícil;

  - Maior dificuldade e demora na detecção e correção de erros;
  - Exigência de maior conhecimento técnico e experiência para sua utilização.

1.6. AS VERSÕES DO CLIPPER

O Clipper foi lançado no mercado americano em 1985, logo após o lançamento do dBASE III em meados de 1984. De lá para cá ambos sofreram modificações e implementações que resultaram em novas e mais poderosas versões.

A versão Summer 85 do Clipper é compatível com a versão 1.0 do dBASE III. Nesta versão o Clipper é bem próximo do dBASE, possuindo, entretanto, alguns recursos adicionais:

- A facilidade de criação de "Help" ao usuário;
- Maior capacidade de arquivos e variáveis de memória;
- Múltiplo relacionamento entre arquivos;
- Possibilidade de criação de funções-de-usuário;
- E uma série de novos comandos e funções que faltavam ao dBASE.

A versão Winter 85 (uma das mais populares), ainda mantém uma certa compatibilidade com o dBASE III versão 1.0. As principais implementações efetuadas foram:

- Variáveis indexadas: vetores;
- Menus de barras através dos comandos @...PROMPT e MENU TO;
- E novas funções para a manipulação de campos memo.

Entretanto, logo após a versão Winter 85 é lançado o dBASE III Plus que, além de incluir vários recursos (comandos e funções) que o Clipper já possuía, pode agora ser utilizado em ambiente multiusuário.

Para acompanhar a evolução do dBASE, é lançada a versão Autumn 86 do Clipper. Nela o Clipper também se torna multiusuário, ganha novos comandos e funções mas muitos dos novos recursos do dBASE III Plus são implementados de forma provisória, através de rotinas auxiliares, escritas em linguagem "C" e Assembler. A preocupação de se manter a compatibilidade com o dBASE III Plus ainda é grande.

Uma grande mudança ocorre, entretanto, na versão Summer 87, que recebemos já durante o processo de preparação do livro. Dois são os principais motivos dessa grande evolução:

- Mudança do compilador "C" através do qual o Clipper é construído e;
- Aparece decisão de não mais obrigatoriamente "amarrar" o Clipper ao dBASE III.

Até a versão Autumn 86 o Clipper era construído com o compilador Lattice "C", um dos primeiros e mais tradicionais compiladores da linguagem "C". A versão Summer 87, entretanto foi construída com o Microsoft "C" 5.0, o mais moderno e completo compilador "C" disponível na atualidade.

O Microsoft "C" 5.0 transformou dramaticamente o Clipper. As velocidades de compilação e de execução tornaram-se incrivelmente maiores; para quem conhece o Turbo Pascal, a versão Summer 87 é o "Turbo Clipper".

O Clipper agora "escreve" diretamente no "buffer" da tela, o que faz com que todas as operações de "desenho na tela" sejam simplesmente instantâneas. A velocidade de manipulação de arquivos aumentou impressionantemente: um sistema construído pelo autor com a versão Autumn 86 levava 52 minutos para indexar todos os seus enormes arquivos; com a versão Summer 87, o mesmo sistema, sem sofrer nenhuma alteração e sendo executado no mesmo equipamento leva agora apenas 12 minutos para efetuar a mesma operação.

O outro motivo da grande evolução do Clipper é o "descolamento" ocorrido em relação ao dBASE III. Apesar de manter as mesmas características das versões anteriores, na versão Summer 87 o Clipper "decolou", transformando-se numa ferramenta destinada realmente à construção de sistemas profissionais. Além de uma quantidade considerável de novos comandos e funções, que oferecem recursos para a construção de sistemas mais sofisticados do que um "Assist" do dBASE III Plus, a arquitetura do Clipper foi praticamente aberta. Agora é possível escrever uma função qualquer utilizando o Microsoft "C" e "link-editá-la" diretamente com as bibliotecas e módulos-objeto do Clipper, sem utilizar o comando CALL. Você cria sua própria biblioteca de funções em "C", para serem utilizadas com o Clipper.

Você poderá conhecer, no Cap. 2, e estudar, nos Caps. 5 e 6, as novas funções e os novos comandos. Deixamos, para uma próxima oportunidade, entretanto, os recursos que "abrem" o Clipper para a linguagem "C", devido à sua relativa complexidade e à necessidade de se conhecer profundamente esta linguagem.

A versão Summer 87 do Clipper possui uma arquitetura aberta e é, na verdade, uma grande e sofisticada biblioteca de rotinas escritas em linguagem "C", destinada ao desenvolvimento de Sistemas de Gerenciamento de Bases de Dados. Esta versão, embora revele altíssima performance nos PC/XT, é visivelmente destinada à máquinas mais poderosas como os "286", "386" ou PS/2 e às versões mais atuais do MS-DOS.

Mesmo assim, se você está preocupado com o dBASE IV, utilize a versão Summer 87 do Clipper e faça como eu, tente imaginar como serão as próximas.

2

Principais Diferenças Entre o Clipper e o dBase III

2.1. PRINCIPAIS IMPLEMENTAÇÕES DO CLIPPER

Além das características e vantagens já mencionadas, devido ao fato de ser um compilador, o Clipper possui uma série de implementações que não existem no dBASE III Plus – versão 1.1:

- Maior capacidade de manipulação de dados;
- Criação e manipulação de vetores;
- Novos comandos e funções;
- Processamento multiusuário integrado;
- E outras.

Estas implementações, resumidamente apresentadas nesta seção, colocarão à sua disposição novos e poderosos recursos permitindo que você as sofistique, dotando-as de alto nível profissional, encontrado apenas nos mais modernos "softwares" atualmente disponíveis no mercado.

2.1.1. Implementações de Capacidade

- Número de variáveis ativas:

  Clipper: 2.048
  dBASE: 256

- Número máximo de campos por arquivo:

  Clipper: ilimitado
  dBASE: 128

– Tamanho máximo de uma variável tipo caractere:

Versão Summer 87	64 Kbytes
Versões anteriores	254 bytes
dBASE	254 bytes

– Número de vetores: 2.048 (cada vetor equivale a uma única variável de memória).

– Número de elementos por vetor:

Versão Summer 87	4.096
Versões anteriores	2.048

– Número de arquivos abertos simultaneamente (áreas de trabalho):

Versão Summer 87	255 (com o DOS 3.3)
Versões anteriores	15
dBASE	10

2.1.2. A Facilidade de Criação de "HELP" Para o Usuário

Uma das mais interessantes implementações do Clipper é o recurso de se criar, com facilidade, ajuda ou "Help" "on-line" ao usuário. Pressionando-se a tecla **F1**, a partir de qualquer estado de espera do programa, o Clipper automaticamente invoca o programa HELP.prg que, se existir, poderá apresentar explicações ao usuário, adequadas à situação específica em que este se encontra. O programa HELP.prg é criado por você, de acordo com a necessidade de "Help" ao usuário de sua aplicação.

Com a utilização da "facilidade de Help"do Clipper, você poderá dotar suas aplicações, de Help's tão sofisticados quando os do próprio dBASE III, Lotus 1-2-3, SideKick etc.

Para maiores detalhes veja o Cap. 7 – A Facilidade de Ajuda – HELP – ao Usuário.

2.1.3. Maior Número de Arquivos Relacionados Pelo SET RELATION

O Clipper, através do comando SET RELATION, permite o relacionamento simultâneo entre um arquivo principal com até oito arquivos secundários relacionados, por área de trabalho selecionada (relacionamento de 1 para 8). O dBASE III permite apenas o relacionamento de 1 para 1. Com isso, a capacidade e a facilidade de relacionamento entre arquivos ganha novas dimensões, tornando ainda mais fácil e ágil o desenvolvimento de aplicações.

Clipper: um arquivo principal e oito relacionados.
dBASE: um arquivo principal e apenas um relacionado

Para maiores detalhes veja o comando SET RELATION na seção 5.11 — Comandos Para a Configuração de Ambiente.

2.1.4. Novos Tipos de Variáveis

Além das variáveis normais, no Clipper é possível a criação e utilização de variáveis indexadas de uma dimensão (vetores).

Um vetor é criado pelo comando DECLARE (ou PRIVATE na versão Summer 87), que além do seu nome define número de elementos que este possuirá. Um vetor pode possuir até 2.048 elementos (ou 4.096 na versão Summer 87). Cada elemento de um mesmo vetor poderá armazenar dados de qualquer tipo (numérico, caractere, data ou lógico).

Por exemplo:

```
DECLARE vetor1 [10], vetor2 [20]

* Declara dois vetores: o vetor1 com 10 elementos e o vetor2 com 20.

vetor 1 [1] = 100

* Atribui 100 como conteúdo do primeiro elemento do vetor1

vetor1 [2] = "Clipper"

* Atribui "Clipper" como conteúdo do segundo elemento do vetor1.

vetor 2 [1] = 2*vetor1 [1]

* Atribui como conteúdo do primeiro elemento do vetor2 2 vezes o conteúdo
do vetor1, ou seja, o valor 200.
```

Além do comando DECLARE, que cria os vetores, há uma série de novas funções especiais, exclusivas para a manipulação de vetores:

ACHOICE( ), ACOPY( ), ADEL( ), ADIR( ), AFIELDS( ), AFILL( ), AINS( ), ASCAN( ) e ASORT( ).

Para maiores detalhes veja as seguintes seções.

4.1.3 — Variáveis de Memória;
5.2 — Comandos para Manutenção de Variáveis;
6.4 — Funções para Manipulação de Vetores.

2.1.5. Recurso Para a Manipulação e Manutenção de Campos Memo

Para melhorar as possibilidades de edição e manipulação de campos tipo **memo**, o Clipper possui uma série de novas funções não encontradas no dBASE III, que entre outras coisas permitem:

- Edição de campos memo através de janelas predefinidas na tela, com grande capacidade de controle. A função MEMOEDIT( ) é um "editor de textos" completo, embutido no Clipper; com ela é possível até mesmo construir uma função para acentuar o texto editado.
- Leitura ou gravação de arquivos-texto padrão ASCII diretamente de e para campos memo;
- Capacidade de armazenar campos memo em variáveis normais tipo caracter, permitindo a realização de pesquisas, concatenações e REPLACE's em campos memo como se fossem cadeias de caracteres;
- Capacidade de formatar campos memo de acordo com as necessidades de apresentação de dados ou de impressão da aplicação desenvolvida;
- Capacidade de rolagem vertical e horizontal ("scroll") para a apresentação de campos memo dentro de uma janela de qualquer dimensão na tela.

As novas função para a manipulação de campos memo são:

HARDCR( ), MEMOEDIT( ), MEMOLINE( ), MEMOREAD( ), MEMOTRAN( ), MEMOWRIT( ) e MLCOUNT( ).

Para maiores detalhes veja a seção 6.6 — Funções Para Controle de Arquivos de Dados.

2.1.6. Novos Recursos Para o Uso de Macros (&)

O Clipper permite a utilização de macros como condições nos comandos DO WHILE...ENDDO e também o uso recurssivo de macros, ou seja "macros dentro de macros". Tais utilizações não são possíveis no dBASE III, mas facilitam o desenvolvimento de rotinas parametrizadas. Para maiores detalhes veja a função Macro — & — na seção 6.7 — Funções Especiais.

2.1.7. Definição de Funções Pelo Usuário (Funções-de-Usuário)

Uma implementação notável do Clipper é a possibilidade de você criar suas próprias funções, através do comando FUNCTION. Uma vez definidas, essas funções podem ser utilizadas em qualquer parte do programa. Você pode, por exemplo, construir uma biblioteca particular de funções, que será incorporada aos seus programas durante o processo de link-edição. Essa biblioteca fica disponível para todas as aplicações que você desenvolver.

A sintaxe do comando FUNCTION é a seguinte:

FUNCTION <nome da função>

PARAMETERS <lista de argumentos da função>

"

<comandos>

"

"

RETURN(<valor a ser retornado como resultado>)

Uma função deve ter um nome único e sempre retornar um valor. Por exemplo, a função Dpc( ), definida abaixo, retorna a partir de uma data qualquer, a sigla do mês e o ano em português:

```
**********************
FUNCTION Dpc
**********************
PARAMETERS dd    && dd recebe a data
aux= 3 * MONTH(dd) - 2
dc=SUBSTR("JANFEVMARABRMAIJUNJULAGOSETOUTNOVDEZ",aux,3) +;
SUBSTR(DTOC(dd),6,8)
RETURN(dc)
```

Exemplo de aplicação:

Dpc("20/02/88") retornará "FEV/88"

Na verdade, o comando FUNCTION "abre" a arquitetura do Clipper, pertindo que você a amplie através da definição de novas funções, de acordo com suas necessidades.

Para maiores detalhes veja o comando FUNCTION na seção 5.9 — Comandos de Controle de Fluxo de Programação — Estruturas de Programação.

2.1.8. O Recurso da Variável CLIPPER

Através da criação de uma variável pública chamada CLIPPER, você poderá utilizar os novos comandos e funções do Clipper em seus programas dBASE, sem prejudicar a execução dos mesmos no dBASE interpretado.

Executando o programa compilado pelo Clipper, a variável "clipper" assume automaticamente o conteúdo .T. (verdadeiro), enquanto que executando o mesmo programa interpretado pelo dBASE III, a variável "clipper" assume o valor .F. (falso). Desta forma, no Clipper, os novos comandos serão executados, enquanto que no dBASE III não. Isso permitirá a execução do mesmo programa tanto em um como no outro.

Por exemplo:

```
PUBLIC clipper
* declaração da variável pública "clipper"
IF clipper
   FOR i=1 TO 24
       @ i,40 SAY "*"    && Será executado no Clipper
   NEXT
ELSE
   i=1
   DO WHILE i<=24
      @ i,40 SAY "*"     && Será executado no dBASE III
      i=i+1
   ENDDO
ENDIF
```

Para maiores detalhes veja a seção 5.2 — Comandos Para a Manutenção de Variáveis.

2.1.9. <Control — U> — Abortando Alterações

Ao editar o conteúdo de uma variável ou de um campo na tela, através do comando @...SAY...GET, o pressionamento simultâneo das teclas <Ctrl> e <U> retorna o valor original do dado alterado. Por exemplo, se o conteúdo original do campo for "CLIPPER" e tiver sido alterado para "FLIPPER", pressionando <Ctrl-U>, antes de sair da edição, você poderá trazer o valor original "CLIPPER" de volta.

Para maiores detalhes dos recursos para controle de edição do Clipper veja a seção 4.4 — Recursos Para a Edição de Dados.

2.1.10. Validação Automática de Entrada de Dados

A cláusula VALID do comando @...GET permite que sejam especificadas validações para a entrada de dados no próprio GET, sem ser necessário finalizar o comando READ. A cláusula VALID avalia uma expressão lógica ou uma função-de-usuário que define a validade ou não dos dados digitados. Se for retornado falso (.F.), significando digitação inválida, o GET atual não será completado; se for retornado verdadeiro (.T.), significando dado válido, o GET será completado e a execução do programa passa para o GET seguinte. Com a cláusula VALID os GET's serão completados apenas quando forem digitados dados corretos ou for pressionada a tecla <Esc>.

Para maiores detalhes veja a seção 5.3 — Comandos Para Saída de Dados na Tela ou Impressora e 5.4 — Comandos Para Entrada de Dados.

2.1.11. Novos Recursos Para a Passagem de Parâmetros

O Clipper não exige que o número de parâmetros passados por um programa (parâmetros reais) seja igual ao número de parâmetros esperados (parâmetros formais) pelo programa chamado. Com isso a parametrização de aplicações torna-se muito mais fácil e flexível. A verificação do número de parâmetros passados pode ser feita através da função PCOUNT( ), que retorna o número de parâmetros realmente passados (parâmetros reais), permitindo assim o adequado controle dos dados recebidos.

Além disso, o comando PARAMETERS pode ser utilizado para receber parâmetros (através de cadeias de caracteres) passados diretamente a partir do DOS, quando o módulo executável (.EXE) é carregado.

Para maiores detalhes veja o comando PARAMETERS na seção 5.9 – Comandos de Controle do Fluxo de Programação – Estruturas de Programação.

2.1.12. Novos Operadores

Além dos operados matemáticos normais, o Clipper possui o operador módulo, representado pelo sinal de porcentagem – % – que, operando dois valores numéricos, retorna o resto da divisão do primeiro pelo segundo. Este operador é idêntico e substitui a função MOD( ) do dBASE III Plus.
Foram implementados também novos operadores relacionais:

- Duplo-igual " # ": compara duas cadeias de caracteres, retornando .T. (verdadeiro) apenas quando ambas forem exatamente iguais; como se o SET EXACT estivesse em ON (ligado).
- Exclamação "!": equivalente ao operador lógico .NOT..
- Exclamação-igual "!=": equivalente aos operados relacionais "não-igual": "< >" ou "#".

Para maiores detalhes veja a seção 4.2 – Operadores do Clipper.

2.1.13. Criação de Menus de Barra com Legendas

O Clipper possui os comandos @...PROMPT...MESSAGE, SET MESSAGE TO e MENU TO, que permitem a criação de menus de opções com barras luminosas, semelhantes aos do Lotus 1-2-3.

As opções podem ser dispostas em qualquer posição na tela, sendo apresentadas por palavras que as identificam ("prompts"). Através das setas de direção, uma barra luminosa percorre cada uma delas. A escolha da opção desejada é feita teclando-se <Enter> sobre a opção desejada ou a teclando sua primeira letra. Para cada opção oferecida, pode ser apresentada uma mensagem explicativa em uma determinada linha da tela.

Através desses comandos podem ser facilmente criados em seus programas menus que funcionam exatamente como os do Lotus 1-2-3, do SideKick, e outros "softwares" consagrados.

Além desses comandos, na versão Summer 87 foi implementada a função ACHOICE( ) que possibilita, com grande facilidade, a criação de menus instantâneos em janelas ("pop-up menus"), idênticos aos do comando "Assist" do dBASE III Plus ou do FrameWork.

Para maiores detalhes veja:

- @...PROMPT...MESSAGE – seção 5.3 – Comandos Para Saída de Dados na Tela ou Impressora;
- MENU TO – seção 5.4 – Comandos Para a Entrada de Dados.
- SET MESSAGE TO – seção 5.11 – Comandos Para Configuração de Ambiente;
- ACHOICE( ) – seção 6.4 – Funções Para a Manipulação de Vetores.

2.1.14. Comandos Para Desenhar Molduras na Tela

Além do comando @...TO... DOUBLE , disponível também no dBASE III, o Clipper possui o comando @...BOX, que permite desenhar molduras em qualquer posição da tela, com qualquer caractere desejado como borda, não os limitando apenas às linhas simples ou duplas.

Para maiores detalhes veja a seção 5.3 – Comandos Para Apresentação de Dados na Tela ou na Impressora.

2.1.15. Opção de Ligar e Desligar o Cursor

O comando SET CURSOR ON/off, disponível na versão Summer 87, permite ligar ou desligar a apresentação do cursor na tela. Na verdade, em uma aplicação de alto nível, o cursor somente deve ser apresentado quando o usuário está efetuando uma entrada ou alteração de dados. Durante todo o restante do processamento, como construção e apresentação de telas, impressão de relatórios etc., o cursor, de preferência, não deve aparecer.

Para maiores detalhes veja a seção 5.11 – Comandos Para Configuração de Ambiente.

2.1.16. Salvar e Recuperar Telas Formatadas Para a Criação de Janelas

O Clipper permite que sejam armazenadas, em variáveis de memória normais (tipo caractere), telas formatadas inteiras ou apenas determinadas regiões especificadas. Estas telas ou regiões de telas, podem depois ser instaneamente recuperadas e reapresentadas, sem a necessidade de serem reconstruídas.

O comando SAVE SCREEN TO <variável> pode salvar múltiplas telas formatadas, que depois podem ser instantaneamente recuperadas pelo comando RESTORE SCREEN FROM <variável>.

A função SAVESCREEN( ) pode salvar múltiplas regiões de telas formatadas, que depois podem ser instantaneamente recuperadas pela função RESTSCREEN( ) (estas funções somente estão disponíveis a partir da versão Summer 87).

Com estes comandos e funções você poderá conseguir efeitos de "pop-up menus" e de apresentação de diversas janelas (menus, Help ao usuário, vista parcial de dados, efeitos especiais etc.) como no SideKick, MS-Windows etc.; ou seja, os mais modernos softwares atualmente desenvolvidos para microcomputadores.

Para maiores detalhes veja a seção 5.3 — Comandos Para Saída de Dados na Tela ou Impressora e a seção 6.7 — Funções Especiais.

2.1.17. Rolagem ou "Scroll" Automático de Linhas na Tela

A nova função SCROLL( ) (disponível apenas a partir da versão Summer 87), permite realizar o rolamento vertical instantâneo de linhas na tela, melhorando sensivelmente as aplicações onde a apresentação seqüencial de dados ou registros é muito utilizada.

Para maiores detalhes veja a seção 6.7 — Funções Especiais.

2.1.18. Edição de Arquivos de Dados (BROWSE)

A função DBEDIT( ) (que sofreu diversas implementações na versão Summer 87) torna possível a edição completa de um ou vários arquivos de dados simultâneos em uma janela especificada da tela, possuindo mais recursos de controle que os disponíveis no comando BROWSE do dBASE III. Além disso pode ser construída uma função-de-usuário, através do comando FUNCTION, que permitirá a implementação de qualquer recurso ou processamento adicional desejado.

Para maiores detalhes veja a seção 6.6 — Funções Para Controle de Arquivos de Dados.

2.1.19. Capacidade de Efetuar Pesquisa Relativas

A partir da versão Summer 87, o comando SET SOFTSEEK on/OFF, permite que através do comando SEEK sejam efetuadas pesquisas relativas, isto é, se uma chave idêntica à pesquisada não for encontrada em nenhum registro, o Clipper posicionará o apontador de registros no primeiro registro que possua uma chave superior à chave pesquisada.

Para maiores detalhes veja a seção 5.11 — Comandos Para Configuração de Ambiente.

2.1.20. Utilização das Cláusulas FOR e WHILE Simultaneamente nos Comandos

O Clipper permite que sejam simultaneamente utilizadas as cláusulas condicionais FOR e WHILE para especificar condições de seleção de registros em um comando. Por exemplo:

```
vestado="SP"
SEEK vestado
LIST FOR salario>10000 WHILE estado=vestado
```

Para maiores detalhes veja a seção 4.3 — Seleção de Registros Para o Processamento dos Comandos.

2.1.21. Definição de Teclas de Função, de F2 a F40 e Novas Funções Para o Controle da Digitação

Através do comando SET KEY, o Clipper permite que as teclas de função, de F2 a F40 (F1 é reservada para o Help.prg) sejam configuradas para a execução de uma determinada rotina.

A partir de qualquer estado de espera do programa, quando a tecla de função designada for pressionada a rotina correspondente será executada. Com este recurso, suas aplicações poderão fazer largo uso das teclas de função, e tornarem-se ainda mais sofisticadas.

Para maiores detalhes veja o comando SET KEY na seção 5.11 — Comandos Para Configuração de Ambiente.

Além do comando SET KEY, outros comandos e funções também permitem controlar e verificar as teclas digitadas: LASTKEY( ), INKEY( ) e KEYBOARD. Para maiores detalhes veja a seção 6.7 — Funções Especiais.

2.1.22. Novos Comandos Para o Controle do Fluxo de Execução dos Programas

Além do comando FOR...NEXT, que permite a construção de "loops" finitos controlados por um contador, os novos comandos BEGIN SEQUENCE...END e IF...ELSEIF...ENDIF (disponíveis a partir da versão Summer 87) lhe darão maior flexibilidade na estruturação de seus programas. Nenhum deles está disponível no dBASE III.

Para maiores detalhes veja a seção 5.9 — Comandos de Controle do Fluxo de Programação — Estruturas de Programação.

2.1.23. Maior Flexibilidade na Definição e Construção de "Procedures" e Funções-de-Usuário

O Clipper permite que sejam declarados, através do comando SET PROCEDURE TO, tantos arquivos de "procedimentos" quanto forem necessários para a sua aplicação. Todas as rotinas ou funções-de-usuário neles definidas estarão disponíveis a todos os outros programas do sistema desenvolvido.

Além desse recurso, as "procedures" (ou rotinas) e as funções-de-usuário poderão ser definidas nos próprios arquivos (programas) em que são chamadas.

Para maiores detalhes veja as seções 5.9 — Comandos de Controle de Fluxo de Programação e 5.11 — Comandos Para Configuração de Ambiente.

2.1.24. Uso de Funções Diretamente em uma Linha de Comandos

O Clipper permite que uma linha de programa seja iniciada diretamente por uma função. Isto é muito útil quando a função em questão causa uma ação, e não necessariamente o retorno de um resultado. Por exemplo:

```
@ 12,25 SAY "Digite uma tecla quando pronto...!"
INKEY(0)
```

ou

```
SETCOLOR(W+/R)
```

2.1.25. Integração Direta com Rotinas Escritas em Linguagem "C" ou Assembler

O Clipper permite a integração direta de um número ilimitado de rotinas escritas em "C" ou Assembler, com o seu código fonte em Clipper. Além do comando CALL, que "chama" estas rotinas, são fornecidas outras funções auxiliares escritas em "C" que fazem toda a conversão de parâmetros necessária para o intercâmbio adequado de dados entre o seu programa em Clipper e rotinas em "C".

Infelizmente, por exigir conhecimento aprofundado de "C" e Assembler, consideramos este assunto além dos objetivos deste primeiro trabalho. Por favor, para maiores detalhes consulte as extensões do seu manual do Clipper.

2.1.26. Acesso Direto à Arquivos do DOS

A partir da versão Summer 87 o Clipper possui uma série de funções que permitem o acesso de baixo nível (byte a byte) a qualquer arquivo do DOS. Para

utilizá-las, entretanto, é necessário conhecimento aprofundado do DOS. Estas funções são:

FCLOSE( ), FERROR( ), FOPEN( ), FREAD( ), FREADSTR( ), FSEEK( ) e FWRITE( )

Para maiores detalhes veja a seção 6.7 — Funções Especiais.

2.1.27. Processamento Multi-Usuário

As aplicações desenvolvidas em Clipper podem ser diretamente utilizadas, sem nenhum custo adicional, em ambiente multiusuário através de uma Rede Local de Microcomputadores que opere no padrão do DOS 3.1 ou maior.

Os arquivos de dados podem ser compartilhados por um número ilimitado de usuários conectados à rede. Para administrar este ambiente, o Clipper possui uma série de comandos e funções específicos para processamento multiusuário, que garantem segurança e integridade.

Para maiores detalhes veja as seções 5.5 — Comandos Para Manutenção de Arquivos de Dados, 5.11 — Comandos Para a Configuração de Ambiente, 6.5 — Funções Para Controle do Ambiente, 6.6 — Funções Para Controle de Arquivos de Dados e o Cap. 11 — Utilizando o Clipper em Rede Local de Microcomputadores.

2.1.28. Utilitários do Clipper

O Clipper vem acompanhado de diversos utilitários que o ajudarão muito na construção de suas aplicações. O mais interessante deles talvez seja o "DEBUG" ou o "depurador de erros" do Clipper. "**Bug**" em "programês" significa erro no programa; "debugar" significa eliminar os erros.

Através do DEBUG é possível obter-se, durante a execução dos programas, uma enorme quantidade de informações que ajudarão a corrigir erros ("debugá-los") com grande rapidez e eficiência.

Além do "Debug" outros utilitários permitem a criação e manutenção de arquivos (DBU.exe), a criação de relatórios (REPORT.exe ou RL.exe) e a criação de relatórios de etiquetas (LABEL.exe ou RL.exe).

Para maiores detalhes veja a seção 9.1 — Utilitários do Clipper.

2.1.29. Lista dos Comandos e Funções Implementados no Clipper, que não existem no dBASE III Plus

Em adição às implementações mais importantes, apresentadas nos itens anteriores, existem outros recursos, comandos e funções no Clipper que não estão dis-

poníveis no dBASE III Plus. A seguir apresentamos, para orientação do leitor, a lista completa de todos os comandos e funções implementados no Clipper, versão Summer 87, mas que não existem no dBASE III Plus versão 1.1. Alguns comandos ou funções da lista existem no dBASE III, mas foram incluídos por possuírem novos recursos no Clipper. Para maiores detalhes sobre os mesmos, consulte os Caps. 5 — Comandos do Clipper e 6 — Funções do Clipper:

Seção 5.2 — Comandos Para Manutenção de Variáveis:

DECLARE	— cria vetores (variáveis indexadas).
PUBLIC e PRIVATE	— permitem declarar vetores públicos ou privados

Seção 5.3 — Comandos Para Manipulação de Tela ou Impressora:

CLEAR SCREEN	— permite limpar a tela sem cancelar os GET's pendentes
@...BOX	— criação de molduras na tela
@...GET...VALID	— validação da entrada de dados
@...PROMPT..MESSAGE	— criação de Menus de Barras Luminosas
@..SAY...PICTURE	— novas máscaras de formatação de dados
RESTORE SCREEN	— recuperação de telas formatadas
SAVE SCREEN	— armazenamento de telas formatadas

Seção 5.4 — Comandos Para Entrada de Dados:

@...GET...VALID	— validação da entrada de dados
MENU TO	— criação dé Menus de Barras Luminosas
KEYBOARD	— entrada forçada de caracteres no buffer de teclado

Seção 5.5.4 — Comandos Para Alterar Registros:

REPLACE	— utilização do ALIAS dos arquivos
UNLOCK	— permitir acesso aos arquivos em ambiente multiusuário
USE	— abertura exclusiva de um arquivo

Seção 5.5.6 — Comandos Para Posicionamento nos Registros:

SKIP	— utilização do ALIAS dos arquivos

Seção 5.9 — Comandos Para Controle do Fluxo de Programação-Estruturas de Programação:

BEGIN SEQUENCE	— execução de uma seqüência de comandos com a possibilidade de executar um desvio.

CALL	– novas opções para chamar programas externos
EXTERNAL	– declaração de rotinas externas
FOR...NEXT	– "Loop" com contador, tipo BASIC
FUNCTION...RETURN	– cria funções-de-usuário
IF...ELSEIF...ENDIF	– equivalente ao DO CASE...ENDCASE.

Seção 5.11 – Comandos Para Configuração de Ambiente:

SET CLIPPER	– configuração de memória de trabalho
SET CURSOR	– liga ou desliga o cursor
SET EXCLUSIVE	– restringe o acesso aos arquivos em ambiente multiusuário
SET KEY	– definição e execução de rotinas através de teclas de função
SET MESSAGE	– define a linha de apresentação das mensagens em Menus de Barras
SET RELATION	– permite o relacionamento 1 para 8 e adicionar novos relacionamentos aos existentes
SET SOFTSEEK	– permite a pesquisa relativa de dados através do comando SEEK
SET WRAP	– define a atuação de teclas no MENU TO

Seção 6.1 – Funções Numéricas:

DAYS( )	– converte segundos para número de dias
LENNUM( )	– retorna comprimento de valores numéricos

Seção 6.2 – Funções Alfanuméricas ("Strings"):

ALLTRIM( )	– tira brancos do início e do fim
SOUNDEX( )	– gera código numérico para nomes
STRTRAN( )	– pesquisa e troca sub-cadeias
STRZERO( )	– coloca zeros na frente de números transformando-os em caractere
RAT( )	– pesquisa caracteres à direita retornando a posição

Seção 6.3 – Funções de Data e Hora:

DTOS( )	– transforma data no caractere "aaaammdd"
ELAPTIME( )	– tempo decorrido em segundos
SECONDS( )	– retorna a hora do sistema em segundos
TSTRING( )	– transforma segundos em hora "string"

Seção 6.4 — Funções Para Manipulação de Vetores:

ACHOICE( )	— construção de "pop-up menus"
ACOPY( )	— copia um vetor para outro
ADEL( )	— deleta elementos de um vetor
ADIR( )	— atribui o diretório do disco ao vetor
AFIELDS( )	— coloca as características dos campos de um arquivo em um vetor
AFILL( )	— preenche os elementos de um vetor com qualquer dado
AINS( )	— insere elemento em um vetor em qualquer posição
ASCAN( )	— pesquisa dados em um vetor
ASORT( )	— classifica vetores em ordem ascendente

Seção 6.5 — Funções Para Controle de Ambiente:

ALTD( )	— invoca o "debugador" do Clipper
DOSERROR( )	— retorna o número do erro do DOS
EMPTY( )	— verifica conteúdo de variáveis/campos
ERRORLEVEL( )	— retorna o nível de erro do DOS
GETE( )	— acessa variáveis ambientais do DOS
ISPRINTER( )	— verifica se a impressora está pronta
NETERR( )	— acusa erro no acesso à rede
NETNAME( )	— retorna o nome do usuário na rede
PCOUNT( )	— retorna o número de parâmetro reais passados de um programa para outro
PROCLINE( )	— retorna o número da linha do programa atual
PROCNAME( )	— retorna o nome do programa atual
READEXIT( )	— permite redefinir a função das setas no comando READ
READINSERT( )	— permite alterar o modo de inserção da edição de dados
READVAR( )	— nome da varivél editada/esperada
SETCANCEL( )	— liga ou desliga interrupção <Alt-C>
SETCOLOR( )	— redefine as cores da tela
SETPRC( )	— permite a redefinição dos valores de PROW( ) e PCOL( )
TYPE( )	— avalia o tipo de dados (novos tipos)

Seção 6.6 — Funções Para Controle de Arquivos de Dados:

ALIAS( )	— retorna o "alias" de área
DBEDIT( )	— permite a edição de arquivos de dados
DBFILTER( )	— retorna a expressão do comando SET FILTER
DBRELATION( )	— retorna as expressões do comando SET RELATION

DBRSELECT( ) – retorna as áreas relacionadas pelo comando SET RELATION
DESCEND( ) – permite a criação de índices em ordem descendente
FCOUNT( ) – retorna o número de campos do arquivo
FIELDNAME( ) – retorna o nome dos campos do arquivo
FLOCK( ) – inibe acesso ao arquivo na rede
HARDCR( ) – troca controles de mudança de linha
HEADER( ) – retorna o tamanho do cabeçalho dos arquivos de dados
INDEXEXT( ) – acusa o tipo de índice usado (NTX ou NDX)
INDEXKEY( ) – retorna a chave do índice
INDEXORD( ) – retorna o número do índice mestre
LOCK( ) – inibe o acesso a arquivos na rede
LUPDATE( ) – retorna a data da última atualização do arquivo
MEMOEDIT( ) – permite a edição de campos memo
MEMOLINE( ) – formata uma linha de campo memo
MEMOREAD( ) – leitura de arquivos texto para variáveis ou campos memo
MEMOTRAN( ) – substitui os caracteres de controle em campos memo
MEMOWRIT( ) – grava arquivos texto a partir de campos memos
MLCOUNT( ) – conta o número de linhas formatadas de um campo memo
RLOCK( ) – inibe o acesso a registros na rede
SELECT( ) – retorna o número da área atualmente selecionada
UPDATED( ) – verifica se os dados foram atualizados no último comando READ executado
USED( ) – verifica se um arquivo está em uso na área de trabalho selecionada.

Seção 6.7 – Funções Especiais:

FCLOSE( ) – fecha um arquivo DOS
FCREATE( ) – cria um arquivo DOS
FERROR( ) – retorna um erro de arquivo DOS
FOPEN( ) – abre um arquivo DOS
FREAD( ) – lê caracteres de um arquivo DOS
FREADSTR( ) – lê caracteres de um arquivo DOS e retorna uma cadeia de caracteres
FSEEK( ) – movimenta o ponteiro de arquivos DOS
FWRITE( ) – grava um arquivo DOS
LASTKEY( ) – retorna o código da última tecla digitada
MEMORY(0) – informa a quantidade de memória disponível
NEXTKEY( ) – lê a próxima tecla digitada
READVAR( ) – nome da variável que está sendo lida
RESTSCREEN( ) – recupera uma região de tela que tenha sido salva

SAVESCREEN( )	— salva uma região da tela para uma variável
SCROLL( )	— permite realizar rolagem vertical de linhas na tela
TONE( )	— permite a emissão de sons com várias freqüências e durações
WORD( )	— converte parâmetros no comando CALL

2.2. PRINCIPAIS RESTRIÇÕES DO CLIPPER

O Clipper não suporta uma série de comandos e funções do dBASE III Plus. Entre eles estão principalmente os comandos destinados ao processamento interativo, alguns comandos para correção de erros, algumas funções e comandos destinados à criação e modificação de arquivos, relatórios, estruturas e catalogues. Além disso, uma das maiores restrições do Clipper, até a versão Autunm 86 e anteriores é a de utilizar arquivos de índice (.ntx), diferentes dos (.ndx) utilizados pelo dBASE III. A versão Summer 87 já permite que os arquivos de índice do dBASE sejam utilizados e criados pelo Clipper; veja a seção 2.3 — Equivalências para maiores detalhes.

A seguir discutiremos as principais restrições do Clipper em relação ao dBASE III Plus:

2.2.1. Comandos do dBASE III Plus não Suportados Pelo Clipper:

2.2.1.1. Comandos Interativos

O Clipper não suporta nenhum comando para processamento eminentemente interativo, ou seja, aqueles comandos utilizados diretamente a partir do "prompt" do dBASE e não em programas. A razão é óbvia, pois o Clipper não possue a forma interativa, apenas a programada.

Os comandos interativos do dBASE III Plus que não estão disponíveis no Clipper são os seguintes:

ASSIST, HELP, CLEAR FIELDS

APPEND, BROWSE, CHANGE, EDIT, INSERT

MODIFY COMMAND

SET, SET HELP, SET HEADING, SET MENUS
SET SAFETY, SET TALK (estão sempre OFF)
SET CATALOG, SET FIELDS, SET MEMO WIDTH
SET CARRY, SET STATUS, SET TITLE,
SET TYPEAHEAD, SET VIEW, SET COLOR ON/OFF

2.2.1.2. Comandos Para Depuração de Erros ou Verificação de "Status":

Embora o Clipper não suporte vários comandos de checagem de ambiente e correção de erros do dBASE III ("debugging commands"), há sempre uma forma a eles equivalente, disponível no "debugador" do Clipper. Veja a seção 9.2 — O "Debugador" do Clipper, para maiores destalhes a respeito.

Os comandos para depuração de erros ("debug") ou verificação de "status" não disponíveis no Clipper são:

DISPLAY MEMORY,	LIST MEMORY
DISPLAY STATUS,	LIST STATUS
DISPLAY STRUCTURE,	LIST STRUCTURE
DISPLAY HISTORY,	LIST HISTORY
DISPLAY FILES,	LIST FILES
DISPLAY USERS,	LIST USERS

RESUME, SUSPEND

SET DEBUG, SET ECHO, SET STEP
SET DO HISTORY, SET HISTORY, SET ENCRYPTION

2.2.1.3. Comandos Para Criação/Modificação de Arquivos:

O Clipper não suporta o comando CREATE do dBASE da mesma maneira. No Clipper o comando CREATE (veja a seção 5.5.1 — Comandos Para a Criação de Arquivos de Dados), apenas cria uma estrutura vazia a partir de um arquivo de estrutura extendida.

Além disso, o Clipper não suporta os seguintes comandos para a criação ou modificação de arquivos, embora possua utilitários que permitem substituí-los (veja a seção 9.1 — Utilitários do Clipper)

CREATE LABEL,	MODIFY LABEL
CREATE REPORT,	MODIFY REPORT
CREATE SCREEN,	MODIFY SCREEN
CREATE,	MODIFY STRUCTURE
CREATE QUERY,	MODIFY QUERY
CREATE VIEW,	MODIFY VIEW
	MODIFY COMMAND

IMPORT FROM, EXPORT TO

SET CATALOG TO

2.2.1.4. Comandos de Estruturas de Programação

Existem alguns comandos de específicos para programação que não são suportados pelo Clipper, mas que poderão ser substituídos por rotinas desenvolvidas pelo próprio usuário.

Os comandos de programação não suportados são:

LOAD

LOGOUT, MESSAGE

RETURN TO MASTER
RETRY, RESUME
ON ERROR / ESCAPE / KEY

2.2.2. Funções do dBASE III Plus não Suportadas Pelo Clipper

Assim como os comandos, existem algumas funções que existem no dBASE III Plus e não são suportadas pelo Clipper. Estas funções estão relacionadas a seguir:

ERROR( )
FKLABEL( )
FKMAX( )
GETENV( ) (equivalente à GETE( ) do Clipper)
MESSAGE( )
NDX( )
OS( )
READKEY( )
VERSION( )

2.3. PRINCIPAIS EQUIVALÊNCIAS

Além das implementações e restrições entre o Clipper e o dBASE III, existem alguns recursos, comandos e funções comuns a ambos, que são suportados de maneira diferente, mas equivalente. A seguir relacionamos os principais.

2.3.1. Comandos Equivalentes

Os comandos DISPLAY, LIST, AVERAGE e SUM são equivalentes, contudo o Clipper exige que os campos dos arquivos a serem por eles processados sejam explicitamente definidos e, no caso dos comandos AVERAGE e SUM, as variáveis

que receberão os resultados das operações também devem ser obrigatoriamente definidas. Por exemplo:

DISPLAY é válido no dBASE mas não é no Clipper, pois será necessário especificar os campos a serem processados:

```
DISPLAY ALL codigo,nome,endereco,cidade,cep
```

SUM é válido no dBASE mas não é no Clipper, pois será necessário especificar os campos a serem somados e as variáveis que receberão os resultados:

```
SUM saldo,valor TO tsaldo,tvalor.
```

Além destes o comando SET ESCAPE ON/off ao invés de desabilitar a interrupção de um programa através da tecla <Esc>, no dBASE III, desabilita a utilização das teclas <Alt-C> para interromper os programas do Clipper.

2.3.2. Funções Equivalentes

Existem algumas funções do dBASE e do Clipper que, apesar de possuirem nomes diferentes, são totalmente equivalentes. Para maiores detalhes veja o Cap. 6 — Funções do Clipper.

A seguir relacionamos as funções do Clipper e a correspondente função equivalente no dBASE III Plus.

Clipper	dBASE III
FIELDNAME( )	FIELD( )
GETE( )	GETENV( )
IIF( )	IF( )
LASTKEY( )	READKEY( )
LASTREC( )	RECCOUNT( )

Além destas existem outras duas funções do Clipper, relacionadas a seguir, que apesar de equivalentes, funcionam de forma diferente.

A função & Macro Substituição pode ser utilizada em qualquer lugar de comandos DO WHILE...ENDDO, pode ser utilizada de forma recursiva ou encadeada. Contudo a função macro não pode conter comandos, ou qualquer parte de um comando (como USE por exemplo), mas poderá ser utilizada para substituir parâmetros de comandos. Veja mais detalhes na seção 6.7 — Funções Especiais do Clipper.

Na função STR( ) (string) do Clipper, o comprimento e o número de casas decimais são opcionais e, se não incluídos, todo o resultado da expressão, incluindo as casas decimais, será convertido para caractere.

A função MOD( ) do dBASE III, que retorna o resto da divisão entre dois números, no Clipper pode ser substituída pelo operador módulo (%), que é equivalente.

2.3.3. Indexação de Arquivos de Dados

Tanto o Clipper quanto o dBASE III, trabalham com os mesmos arquivos de dados. Um arquivo (.dbf) criado no dBASE III ou no Clipper (através de seu utilitário CREATE), são totalmente equivalentes e compatíveis. Portanto, os arquivos de dados criados no dBASE III, poderão ser diretamente utilizados pelos programas compilados pelo Clipper. O mesmo, entretanto, já não ocorre com os arquivos de índice.

O comando INDEX, tanto no Clipper como no dBASE III cria arquivos de índice associados a arquivos de dados, permitindo ordená-los da forma desejada e acessá-los aleatoriamente. Contudo, os arquivos de índice criados pelo dBASE, não são compatíveis com os criados pelo Clipper.

No dBASE III, o comando INDEX cria arquivos com a extensão (.ndx), enquanto que no Clipper são produzidos arquivos com a extensão (.ntx). Estes arquivos são diferentes entre si, embora sua função seja exatamente a mesma. Portanto, para trabalhar com arquivos de dados do dBASE III que estejam indexados, é necessário indexá-los novamente através do comando INDEX do Clipper ou pelo utilitário INDEX.prg (programa de indexação) que o acompanha. Dessa forma serão criados os índices (.ntx) com o qual o Clipper trabalha.

O programa utilitário INDEX.prg (veja a seção 9.1.2 — Index) é escrito em Clipper, e após compilado e "linkeditado" gera um programa executável (INDEX.EXE) que permite que os arquivos de dados (.dbf) sejam indexados pelo Clipper. É interessante, portanto, que você compile e utilize este programa para indexar arquivos de dados a serem utilizados pelos programas compilados com o Clipper, quando a indexação não for realizada pelo próprio programa desenvolvido.

A melhor forma de criar e manter os arquivos de índice do Clipper é incluir dentro de seus programas uma opção para a indexação de arquivos. Nela basta incluir o comando INDEX para indexar os arquivos que a sua aplicação utiliza, de acordo com as chaves de classificação desejadas. O comando INDEX irá automaticamente criar os arquivos de índice (.ntx), que serão utilizados pelos arquivos e programas de sua aplicação.

Apesar da diferença entre os índices, os arquivos de dados não são afetados e a compatibilidade entre o dBASE e o Clipper continuará total.

Para maiores detalhes a respeito seja o comando INDEX na seção 5.5.7 — Comandos Para Organização dos Arquivos.

2.3.4. Utilizando Índices do dBASE III com o Clipper

A partir da versão Summer 87, o Clipper é capaz de utilizar os mesmos índices do dBASE III. Para isso basta incluir na lista de módulos-objeto passada ao link-editor o arquivo NDX.obj que acompanha esta versão.

Se o módulo executável de uma aplicação for gerado utilizando-se o arquivo NDX.obj, o Clipper passará automaticamente a criar e a trabalhar com arquivos de índice compatíveis com os (.ndx) do dBASE III, sem ser necessária nenhuma modificação adicional no código-fonte dos programas. Poderão, neste caso, ser utilizados os mesmos arquivos de índice já criados pelo dBASE III, pois o Clipper os aceitará normalmente.

O arquivo NDX.obj é um módulo-objeto que acompanha a versão Summer 87 e pode ser link-editado juntamente o módulo-objeto de sua aplicação, permitindo que você crie e utilize índices compatíveis com os do dBASE III nos programas compilados pelo Clipper. Para utilizar o módulo-objeto NDX.obj em seus programas siga o seguinte esquema durante a fase de "link-edição":

1 – Utilizando o Link-editor do DOS:

```
C>LINK PROG+NDX,,,\Clipper\CLIPPER
```

2 – Utilizando o PLINK86:

```
C>PLINK86 FI PROG,NDX LIB \Clipper\CLIPPER
```

onde, PROG é o nome do módulo-objeto dos seus programas e NDX é o módulo-objeto que produz e trabalha com índices (.ndx). \Clipper\Clipper é a localização e o nome da biblioteca a ser utilizada (CLIPPER.lib no diretório \CLIPPER).

Através deste esquema de link-edição será gerado o módulo executável PROG.EXE que utilizará índices (.ndx) compatíveis com os do dBASE III Plus.

Atenção: Não é possível utilizar simultaneamente índices (.ndx) e (.ntx) em uma mesma aplicação.

Apesar da possibilidade de se utilizar índices (.ndx), compatíveis com o dBASE III Plus, recomenda-se que sua utilização pelo Clipper seja feita apenas durante a fase de testes ou desenvolvimento de programas. Assim que a aplicação estiver funcionando devem ser utilizados os arquivos de índice (.ntx) do Clipper. Com eles o Clipper trabalha mais eficientemente e rapidamente, não sendo, portanto, interessante utilizar os arquivos (.ndx) se a aplicação desenvolvida sempre for utilizada compilada através do Clipper.

Evite utilizar arquivos de índice do dBASE III nas aplicações compiladas pelo Clipper, a menos que você também necessite acessar os arquivos através do dBASE.

Para verificar o tipo de índice que está sendo utilizado pela aplicação, ou seja, se o módulo-objeto NDX.obj foi incluído na link-edição, a versão Summer 87 possui a função INDEXEXT( ) (ou índice externo). Esta função retornará "NTX" se os índices utilizados forem do Clipper e "NDX" se os índices utilizados forem compatíveis com o dBASE III. Por exemplo, para verificar a existência de arquivos de índice no disco, quando se está utilizando tanto (.ndx) como (.ntx) pode-se utilizar as seguintes instruções:

```
IF FILE("Indcod."+INDEXEXT) .OR. FILE("Indnome."+INDEXEXT)
   USE Mala INDEX Indcod,Indnome
ELSE
   USE Mala
   INDEX ON codigo TO Indcod
   INDEX ON nome   TO Indnome
   SET INDEX TO Indcod,Indnome
ENDIF
```

2.3.5. Teclas de Movimentação do Cursor

Tanto o Clipper como o dBASE III utilizam algumas teclas especiais para a movimentação do cursor, principalmente quando da edição de dados. Apesar de parecidas e equivalentes, algumas destas teclas possuem diferenças de utilização entre ambos. A seguir apresentamos as teclas utilizadas pelo Clipper.

Tecla	Função
Home	Move o cursor para o início do campo sendo editado pelo comando GET.
^Home	Move o cursor para o início do primeiro GET de uma lista de dados editados pelos GET's que precedem um comando READ.
End	Move o cursor para o último caractere do campo sendo editado pelo comando GET.
^End	Move o cursor para o início do último GET de uma lista de GET's que precedem um comando READ.
Seta para Cima	Move o cursor para o GET anterior, e não irá completar um READ quando pressionada no último GET.
Seta para Baixo	Move o cursor para o GET seguinte, e não irá completar um READ quando pressionada no último GET.

Tecla	Função
——>	Move o cursor um caractere para a direita. Quando pressionada no fim de um GET, o cursor irá se mover para o próximo GET. Não irá completar um GET quando pressionada no último caractere do último GET que precede um comando READ.
^——>	Move o cursor para o início da próxima palavra.
<——	Move o cursor um caractere para a esquerda, mas não irá mover o cursor, como no dBASE, para o GET anterior.
^<——	Move o cursor para o início da palavra atual.
^T	Deleta a palavra à direita do cursor
^Y	Deleta o resto do conteúdo de um campo, à direita da posição do cursor.
^C ou ^W PgUp ou PgDw	Finaliza um comando READ no GET onde atualmente está posicionado o cursor e salva os valores dos dados editados.
Esc	Finaliza um comando READ no GET onde atualmente está posicionado o cursor sem salvar os novos valores dos dados editados.
^U	Retorna o GET corrente para o seu valor original desde que este não tenha sido finalizado.

3

Sistemas de Informação

3.1. A NECESSIDADE DO ARMAZENAMENTO DE DADOS

Ao procurar um número de telefone, conferir seu saldo bancário, escrever um resumo, cuidar da velocidade do seu carro etc., você está **processando dados.** O resultado de suas atividades em processamento de dados é a informação que você usa para dirigir suas atividades. Quanto melhor informado você estiver, mais facilmente alcançará seus objetivos.

Ao menos três grandes motivos combinados tornam o processamento de informações tão importante em organizações:

- A complexidade crescente da sociedade moderna;
- A administração científica;
- E a tecnologia da informática.

O constante crescimento das empresas, ao mesmo tempo e na mesma proporção em que afasta os administradores de alto nível da supervisão mais direta das operações, tende a tornar cada vez mais crítico o recurso da **informação**, e conseqüentemente necessário o processamento eletrônico de dados.

As organizações podem ser consideradas como sistemas, criados em função de um objetivo que em geral está ligado à uma transformação e satisfação de necessidades, gerando ou não lucro. Em princípio as atividades na organização existem e podem ser avaliadas em função da sua contribuição para que aqueles objetivos sejam atingidos. Isto fornece base para a definição de subsistemas, tais como os **Sistemas de Informação** que compreendem as atividades de comunicação na organização, cujo objetivo é fornecer a informação requerida pela organização como um todo.

Haveria muitas formas de conceituarmos **informação**, dependendo do ângulo de observação e do campo de conhecimento em que se procure tal conceito. Do ponto de vista mais específico de Sistemas de Informação construídos através do

Clipper, em que estamos interessados, examina-se o conceito a partir do entendimento da informação como resultado do tratamento de **dados**. Pode-se entender um **dado** como um item elementar da informação (um conjunto de idéias ou fatos expressos através de letras, dígitos ou outros símbolos) que, tomado isoladamente, não transmite nenhum conhecimento, ou seja, não possue significado intrínseco.

Podemos, partindo do conceito acima, definir informação como o resultado de fatos e idéias relevantes, ou seja dados, que foram transformados (processados) numa forma inteligível para quem os recebe e tem valor (utilidade) real ou aparente para a tomada de decisões presentes ou futuras.

Dentro do enfoque acima, Sistema de Informação é um componente do sistema organizacional. É uma rede espalhada pela organização inteira, utilizada por todos os outros componentes da mesma. Seu propósito é obter informações dentro e fora da organização, torná-las disponíveis para os outros componentes, quando necessitarem, e apresentar as informações exigidas pelos que estão fora da organização.

Os Sistemas de Informação, em geral, são utilizados para orientar a tomada de decisão em três níveis diferentes da administração de uma organização: o operacional, o tático e o estratégico.

Para permitir uma tomada de decisão adequada, a informação deve estar sempre disponível a quem a requereu em uma forma facilmente utilizável e compreensível, a tempo de auxiliar nas tomadas de decisões e a custo razoável, caso contrário as pessoas não poderão utilizá-la. Manter a informação prontamente acessível para uso futuro é, portanto, um dos principais e mais comuns objetivos de um Sistema de Informação.

3.2. COMPONENTES BÁSICOS DE UM SISTEMA DE INFORMAÇÃO

Um Sistema de Informação possui três componentes principais: Dados, Sistemas de Processamento de Dados e os Canais de Comunicação.

O total de unidades vendidas, datas, descrição de produtos e nomes de clientes são exemplos de **Dados**. Os dados são adquiridos inicialmente pelo Sistema de Informação de seu ambiente (por exemplo a empresa e suas operações) e referidos como entradas.

O **Sistema de Processamento de Dados** manipula (processa) e transforma os dados em conjuntos de informações relevantes, por exemplo: o relatório das vendas do mês de julho (com a data de cada venda, descrição e quantidade dos produtos vendidos, valor unitário e valor total, por cliente, por vendedor etc.), a folha de pagamento dos funcionários etc. Este sistema é responsável pelo armazenamento, processamento e recuperação, em geral utilizando computadores, dos dados necessários ao funcionamento do Sistema de Informação. Ele deve ser projetado para servir ao Sistema de Informação, do qual é um subsistema. Trata, portanto, do processamento (armazenamento, rearranjo, reestruturação, classificação, agregação, reordenação, cálculo etc.) de dados, com a finalidade de aumentar sua utilidade e conseqüentemente seu valor, transformando-os em **informação**.

O terceiro componente do Sistema de Informação é o **Canal de Comunicações**. Ele fornece os meios de transmissão de informações de um componente do sistema para outro; por exemplo: relatórios impressos, telefone, telex, correio, reuniões, circulares, teleprocessamento, terminais de computador etc.

Um Sistema de Informações deve atingir o mais rapidamente possível seus objetivos de armazenamento e fornecimento de informações para a organização, em formato, tempo e custos apropriados. Identificar as fontes de dados, os componentes e a forma do processamento dos dados que serão utilizados, além de especificar o formato, custo e tempo mínimo para a apresentação da informação são os procedimentos básicos que governam o desenvolvimento dos Sistemas de Informação.

O ciclo vital dos Sistemas de Informação automatizados compreende as seguintes fases:

- **Definição dos Requisitos do Sistema:** definição dos problemas, dados e informações a serem obtidas e fornecidas pelo sistema.
- **Projeto Lógico:** elaboração do projeto do sistema, ou seja, como ele irá funcionar e como será construído.
- **Projeto Físico:** construção do sistema: elaboração dos programas de computador, formulários e telas de entrada de dados, relatórios de saída, telas de consulta, descrição dos procedimentos etc.
- **Codificação e Depuração:** codificação dos programas do sistema através de linguagens de programação de computadores, no nosso caso o Clipper, e depuração (correção e eliminação) de erros.
- **Teste e Implantação:** testes do sistema, em termos de equipamentos (hardware) e programas (software) verificando se estão de acordo com os projetos lógico e físico, para então proceder a implantação.
- **Operação e Manutenção:** funcionamento do sistema: operação e manutenção necessários ao atendimento permanente de seus objetivos de geração de informação, com as características especificadas nas definições dos requisitos.

Um Sistema de Informações eficaz nos moldes por nós compreendidos deveria:

- Produzir informações realmente necessárias, confiáveis, em tempo hábil e com custo condizente, atendendo aos requisitos operacionais e gerenciais de tomada de decisões a que tais informações devem suprir.
- Ter por base diretrizes capazes de assegurar o atingimento dos objetivos, de maneira direta, simples e eficiente.

- Integrar-se à estrutura da organização e auxiliar na coordenação entre as diferentes unidades organizacionais (departamentos, divisões, diretorias etc.) por ele interligadas.
- Ter um fluxo de procedimentos (internos e externos ao processamento) racional, integrado, rápido e de menor custo possível.
- Contar com dispositivos de controle interno que garantam a confiabilidade das informações de saída e adequada proteção aos dados controlados pelo sistema.
- Finalmente ser simples, seguro e rápido em sua operação.

3.3. O COMPUTADOR E OS SISTEMAS DE INFORMAÇÃO

O computador, pela sua capacidade de armazenar enormes quantidades de dados e de processá-los a grandes velocidades, pelos recursos que oferece para aumentar a confiabilidade da informação e pelas possibilidades que introduz de retenção, recuperação, pesquisa e transmissão de informações, tem muito a ver com os pontos já mencionados e que distingue um sistema de alta qualidade.

Entretanto, a simples introdução de recursos de processamento eletrônico de dados nos Sistemas de Informação de uma organização, não representa uma garantia de solução de seus problemas. Por si só, o computador não assegura que a organização passe a contar com sistemas de alta qualidade.

Ao mesmo tempo, porém, sem o seu emprego, certas necessidades e benefícios objetivados no planejamento dos sistemas podem não ser factíveis, e até mesmo pode não ser possível encontrar soluções viáveis para determinados problemas.

Convém rapidamente comentar as capacidades específicas que um software como o Clipper fornece a um microcomputador, no sentido de atender as necessidades dos Sistemas de Informação:

- Mantém dados em arquivos que refletem entidades ou eventos de interesse para a organização;
- Recupera dados dos arquivos a qualquer momento;
- Modifica a seqüência dos dados, intercalando-os ou classificando-os da forma desejada;
- Executa cálculos com os dados;
- Mostra os resultados do processamento através de relatórios em impressoras ou através do monitor de vídeo;
- Permite o acesso aos dados em locais remotos e transmite dados para locais remotos quase que instantaneamente através de sistemas de comunicação;
- Todas as operações são executadas numa velocidade impossível de ser obtida de outra forma, o que permite que considerável quantidade de trabalho seja executada em espaço de tempo relativamente curto;

— Com sua capacidade de seguir instruções detalhadas, permite que o computador processe automaticamente grandes volumes de dados mediante uma longa série de instruções (programas).

3.4. ENTIDADES E EVENTOS

Os fatos e idéias que descrevem entidades e eventos são os dados que formam a estrutura da informação. Os dados, em geral, se referem a mais de um fato. Um único fato é referido como um **item.**

Informação é um conjunto de dados significativos e relevantes que descrevem eventos e entidades para o órgão ou organização que os utilizará. Para serem significativos, os dados devem ser representados por símbolos compreensíveis, devem ser completos e expressar idéias não ambígüas. Para serem relevantes, os dados devem ser úteis na resolução dos problemas propostos, ou seja, para a tomada de decisões.

Para por exemplo, se efetuar uma transação comercial, é essencial a obtenção de informações sobre as entidades servidas ou utilizadas pela organização. Entre os tipos característicos de entidades encontradas em um ambiente comercial estão incluídos clientes, proprietários, produtos, serviços, valores, suprimentos, funcionários, equipamentos, matérias-primas etc. Em outras palavras, uma **entidade** é uma pessoa, lugar ou coisa; um objeto de interesse para um Sistema de Informação.

Um aspecto fundamental de uma transação empresarial é a obtenção de informações sobre os eventos que ocorrem durante a transação. A informação sobre estes eventos é necessária para objetivos financeiros, legais, administrativos etc. Um **evento** é algo que acontece em um certo tempo e lugar, uma ocorrência significativa para um Sistema de Informação.

As Entidades e Eventos de interesse de um Sistema de Informação podem ser reconhecidas e descritas em termos de seus atributos. Atributos são fatos ou dados sobre eventos e entidades. Existem diversos tipos de atributos, incluindo-se:

— **Identificadores:** dados especialmente úteis na distinção de objetos de informação entre si (por exemplo: código, nome, número do RG etc.).

— **Descritores:** dados relacionados com a percepção sensorial dos objetos de informação (por exemplo: tamanho, peso, forma, cor etc.).

— **Localizadores:** dados que nos permitem determinar onde ocorreu um evento ou onde uma entidade está situada (por exemplo: endereço, caixa postal, prateleira, armazém, departamento etc.).

— **Temporais:** dados que permitem determinar quando ocorreu um evento (por exemplo: horário, data etc.).

— **Relacionais:** dados que descrevem a relação existente entre eventos ou entidades individuais (por exemplo: pais, criança, chefe, subordinado, item, subitem etc.).

— **Condicionais:** dados que descrevem o estado da entidade (por exemplo: casado, empregado, ativo, pendente, em manutenção, em produção etc.).

— **Classificadores:** dados que determinam o modo de relacionamento entre eventos ou entidades e a organização, por categoria ou por tipo (por exemplo: cliente, funcionário, transação de venda, transação de compra, produção etc.).

— **Quantificadores:** dados que descrevem quantidades em relação a eventos e entidades (por exemplo: montante de cruzados, número de produtos adquiridos, número de produtos produzidos etc.).

Estes dados ou atributos são usados em combinação, descrevendo completamente eventos e entidades. Em geral, quanto maior o número de dados disponíveis, mais útil e compreensível se torna a informação.

Quando é desenvolvido um novo Sistema de Informação, deve-se prever as necessidades futuras (entidades, eventos e os dados que as descrevem), além das que determinaram a necessidade do sistema. Contudo, deve-se levar em consideração o custo da obtenção e armazenamento da informação que não possuirá uso imediato.

O exemplo a seguir mostra entidades, eventos e dados de interesse de uma escola:

Exemplo: Escola

Entidade	Atributos	Informação
Aluno →	Nome Sexo Idade Matrícula Curso Etc.	→ Secretaria

Evento	Atributos	Informação
Prova →	Data/Hora Local Professor Matéria Etc.	→ Aluno

3.5. O REGISTRO DA INFORMAÇÃO

Tirar conclusões e tomar decisões exigem normalmente o acesso a fatos e idéias passadas, ou seja dados. Estes dados são mais confiáveis se forem **registrados** e guardados de forma tal que lembrem facilmente as entidades e eventos que representam. Portanto, o registro de dados sobre eventos e entidades é um aspecto muito importante dos Sistemas de Informação.

Um exemplo de informação registrada é o formulário preenchido por candidatos a um emprego. Ao preencher um formulário para se candidatar a um emprego, uma pessoa está fornecendo informação ao seu provável empregador. Cada item ou campo de informação como nome, endereço, telefone, emprego anterior etc., fornece uma descrição ou dado relevante e significativo. Este tipo de documento é referido como documento-fonte, pois serve como fonte de dados, que serão processados posteriormente. Uma vez registrados, nestes campos do formulário, tornam-se dados processáveis, ou seja, podem ser armazenados, manipulados, analisados e expostos por um Sistema de Processamento de Dados.

Um documento a ser preenchido por indivíduos possui geralmente os nomes dos **campos** escritos ao longo dos espaços para preenchimento. Desta forma, o mesmo documento é usado na descrição do contexto e captação dos dados apropriados.

O processamento de dados requer afirmações explícitas de contexto para todo dado a ser processado. As pessoas devem fornecer **contexto** apropriado. Por exemplo: os números 260288 escritos em um papel, não havendo mais nada que forneça qualquer contexto, podem representar tanto a data de 26 de fevereiro de 1988, como a quantidade numérica de 260.288, ou o cheque de número 2-60-288, e assim por diante.

Para responder e usar os dados corretamente é necessário obter-se o contexto destes símbolos de dados.

Contexto

Nome de Dado	Valor do Dado
Número do RG	6.406.564
Sexo	Masculino
Nome	José da Silva
Salário	Cz$ 20.000,00
Departamento	Contabilidade

O valor máximo da informação ocorre quando os dados relacionados são reunidos em um conjunto de campos. Este conjunto de campos relacionados que divide o contexto comum sobre certo evento ou entidade denomina-se **registro**.

Os registros de processamento de dados dividem entre si um contexto comum pela sua utilização, tal como computar folha de pagamento (registro de pessoal),

matricular alunos numa escola (registro de matrícula) etc.; além disso devem conter somente campos relevantes ao processamento.

Agrupar campos de dados logicamente relacionados e colocá-los em registros garante que os itens sejam associados de forma relevante quando forem necessários para gerar informações.

3.6. FORMATO DE REGISTROS

A forma estabelecida para um registro de dados denomina-se formato de registro. As especificações de formato devem incluir o nome do registro, a lista de nomes de todos os campos de dados contidos no registro e devem especificar certas características dos campos para que os dados estejam representados consistentemente nos registros.

As características dos campos são estabelecidas em uma especificação de formato de registro:

- **Nomes dos Campos:** quando possível, deverão identificar ou lembrar o contexto específico de cada item.
- **Seqüência de Campos:** ordem na qual os campos aparecem em todas as ocorrências do registro.
- **Valores Válidos:** listagem específica de valores válidos ou declaração de como determinar se um certo valor é válido.
- **Tamanho dos Campos:** número de posições requeridas para os símbolos (letras, dígitos etc.) representarem completamente os dados.
- **Tipo de Símbolos de Dados:** numéricos (representando valores matemáticos), alfanuméricos (combinação de caracteres alfabéticos, números não utilizados na representação de valores matemáticos e outros símbolos), datas e lógicos (representando apenas sim ou não ou falso e verdadeiro).

Exemplo de Registro

Nome do Registro: Conta Bancária
Itens ou Campos: Especificados em ordem de apresentação:

Número da conta:

- Comprimento de 6 posições
- Símbolos de dados alfanuméricos
- Intervalo válido: inteiros de 200.000 a 800.000

Nome do cliente:

- Comprimento de 30 posições
- Símbolos de dados alfanuméricos
- Pode ser utilizado qualquer caractere alfanumérico

Tipo de conta:

- Comprimento 12 posições
- Símbolos de dados alfanuméricos
- Tipos válidos: NORMAL, ESPECIAL, MASTER etc.

Saldo Atual:

- Comprimento de 12 posições
- Símbolos de dados numéricos com duas casas decimais
- Total máximo: 999999999.99

Data inicial:

Dia inicial:

- Comprimento de 2 posições
- Símbolos de dados numéricos
- Intervalo válido: inteiro p/ meses:
  - 04, 06, 09 e 11 = 01 a 30
  - 02 = 01 a 28 *
  - demais meses = 01 a 31
  - * 1+28, se o ano for divisível por 4

Mês inicial:

- Comprimento de 2 posições
- Símbolos de dados numéricos
- Valores válidos de 01 a 12

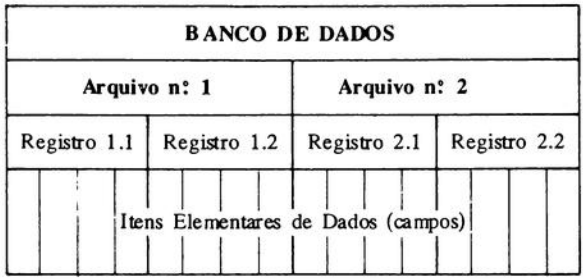
Ano inicial:

- Comprimento de 2 posições
- Símbolos de dados numéricos
- Valores válidos: de 00 a 99

Embora as especificações acima pareçam muito detalhadas, correspondem exatamente ao que é necessário para definir o formato de um registro ou a estrutura de um arquivo no Clipper.

3.7. ARQUIVOS E BANCOS DE DADOS

Assim como os dados são organizados em registros, os registros são organizados em **Arquivos** e estes, por sua vez, podem ser organizados em **Bancos ou Base de Dados**. Isto forma a uma hierarquia de estruturas de dados.



Quando todos os registros de um mesmo contexto são reunidos em um único conjunto de informações relacionadas, o conjunto denomina-se **arquivo.** Por exemplo: o conjunto de todos os registros de contas correntes de um banco denomina-se Arquivo Contas Corrente.

No Clipper, os arquivos são conjuntos de registros altamente estruturados, podendo ser recuperados rapidamente por "campos-chave" que os identifiquem. Os registros de um mesmo arquivo possuem sempre o mesmo formato e são, em geral, armazenados em ordem cronológica de inclusão no arquivo. As técnicas de recuperação e classificação de registros baseiam-se na utilização de campos especiais contidos no registro, referidos como **campos-chave.** Os campos armazenam os dados ou atributos descritores de eventos ou entidades.

Existem essencialmente dois meios de pesquisa que tornam os registros acessíveis para a recuperação de informações: seqüencial (ler um registro após o outro até encontrar o desejado) e direto (ir diretamente ao registro desejado). O arquivo de acesso direto (**indexado**) é mais útil que o seqüencial quando a maioria dos acessos a registros deste arquivo for aleatória e feita a registros isolados em tempos imprevisíveis. Os arquivos seqüenciais são mais úteis quando todos, ou quase todos, os registros do arquivo forem processados como um conjunto.

Um **Banco de Dados** pode ser definido como um conjunto de arquivos relacionados entre si, dividindo a mesma relação geral de contexto. As entidades e eventos descritos em cada arquivo guardam relações entre si, podendo ser ligadas através dos campos chave.

Um Banco de Dados armazena informações usadas pelos diversos subsistemas ou unidades de uma organização. Todos os dados necessários podem ser mantidos em um conjunto de arquivos de fácil acesso, de forma que se minimize sua duplicação e redundância e, conseqüentemente, a inconsistência de informações. Em síntese, um Banco ou Bases de Dados pode ser entendido como uma coleção de arquivos estruturados, não redundantes e inter-relacionados, que proporciona uma fonte única de dados para uma grande variedade de aplicações.

4

A "Linguagem Clipper"

4.1. ESPECIFICAÇÕES TÉCNICAS

Arquivos de Dados:

Número Máximo de Registros: 1 bilhão

Registros: Seqüenciais com tamanho fixo

Tamanho: ilimitado (versão Summer 87)
4.000 bytes (versões anteriores)

Número de Campos: ilimitado (Summer 87)
1.024 (versões anteriores)

Campos:	Caractere:	máximo 32 Kbytes	(Summer 87)
		máximo 254 bytes	(anteriores)
	Numérico:	máximo 19 dígitos	(19 bytes)
	Data:	máximo 8 números	(8 bytes)
	Lógico:	máximo 1 caractere	(1 byte)
	Memo:	máximo 64 Kbytes	(Summer 87)
		máximo 5.000 bytes	(anteriores)

Índices:

Técnica da árvore binária (B-tree)
Chaves com até 250 caracteres por índice

Versão Summer 87: (.ntx) não compatíveis com o dBASE III ou opcionalmente (.ndx), compatíveis
Versões Anteriores: (.ntx), não compatíveis com o dBASE III.

Variáveis de Memória:

Número Máximo: 2.048

Tamanho: Tipo caractere: 64 Kbytes (versão Summer 87)
254 bytes (versões anteriores)
Tipo numérica: 19 dígitos (19 bytes)
Tipo data: 8 dígitos (8 bytes)
Tipo lógica: 1 caractere (1 byte)

Vetores:

Número Máximo: 2.048 (cada um equivale à uma variável)

Número Máximo de Elementos: 4.096 (versão Summer 87)
2.048 (versões anteriores)
Tamanho: idêntico às variáveis para cada elemento.

Arquivos:

Arquivos Abertos: Até 255 (Summer 87 com DOS 3.3)
Até 15 (versões anteriores)
Áreas de Trabalho: Até 255 (Summer 87 com DOS 3.3)
Até 10 (versões anteriores)
Índices Ativos: número ilimitado por área de trabalho.

Rotinas (procedures) por arquivo:

Número ilimitado

Precisão Numérica Não Incluindo o Ponto Decimal:

- 18 casas decimais (Summer 87)
- 9 casas decimais (versões anteriores)

Linha de Programa:

Tamanho máximo: 256 caracteres (256 bytes)

Equipamentos:

Microcomputador: IBM-PC/XT, AT, PS/2 ou outro 100% compatível.
Memória RAM mínima: 256 Kbytes.
Acionadores de Disco: dois acionadores de disquetes ou, de preferência, um acionador de disquetes e um acionador de disco rígido (winchester).
Co-processador matemático: (8087, 80287 ou 80387): se presente será utilizado pelo Clipper em todas as operações aritméticas, agilizando o processamento.
Impressoras: Matriciais, de Linha ou Laser.
Capacidade de Comunicação pode ser interessante.

Sistema Operacional:

MS-DOS, PC-DOS versões 2.0 e superiores ou outros 100% compatíveis.
O MS-DOS 3.3 e superiores é recomendado para a versão Summer 87.

4.2. ARQUIVOS UTILIZADOS OU GERADOS PELO CLIPPER

A nomenclatura dos arquivos utilizados pelo Clipper segue as regras gerais de nomenclatura utilizada pelo DOS. O Clipper requer que cada rotina, programa ou função possua um único nome.

Para a maioria dos comandos não é necessário especificar a extensão no nome dos arquivos, a menos que seja dada outra que não a padrão. Entretanto, os comandos COPY FILE, DELETE FILE, ERASE, RENAME e TYPE requerem que a extensão dos arquivos a serem processados seja especificada.

Arquivos de Dados: extensão .DBF

Contêm a estrutura dos registros e dos campos e os dados de cada registro. Porem ser criados através dos utilitários CREATE.exe ou DBU.exe e dos comandos CREATE e CREATE FROM.

Arquivos de Campos Memo: extensão .DBT

Contêm todos os campos memo do arquivo de dados associado, recebendo o mesmo nome deste arquivo. São automáticamente criados quando da criação de um arquivo de dados que possua um ou mais campos memo.

Arquivos de índice: extensão .NTX

São arquivos auxiliares criados para permitir acesso ordenado ou direto (aleatório) aos registros de um arquivo de dados. Vários arquivos de índice podem ser associados a um arquivo de dados de acordo com múltiplas ordens de classificação. São criados através do comando INDEX ou do utilitário INDEX.exe. A versão Summer 87 permite que os arquivos de índice sejam compatíveis com o formato dos arquivos de índice do dBASE III Plus. Neste caso recebem a extensão .NDX.

Arquivos de Variáveis de Memória: extensão .MEM

São arquivos que contêm a identificação e o conteúdo de variáveis de memória que foram "salvas" no disco durante a execução de um programa. São criados através do comando SAVE.

Arquivos para Emissão de Etiquetas: extensão .LBL

Contêm definições e instruções requeridas para a impressão de etiquetas através do comando LABEL FORM. São criados através dos utilitários LABEL.exe ou RL.exe (versão Summer 87).

Arquivos para Emissão de Relatórios: extensão .FRM

Contêm definições e instruções requeridas para a impressão de relatórios através do comando REPORT FORM. São criados através dos utilitários REPORT.exe ou RL.exe (versão Summer 87).

Arquivos de Formatação de Telas: extensão .FMT

São utilizados pelo comando SET FORMAT TO e contêm comandos @...SAY...GET para a apresentação formatada de dados na tela. Um arquivo de formatação é um arquivo-texto tipo ASCII podendo ser criado com o editor de textos de sua preferência.

Arquivos "Alternate" (documentação): extensão .TXT

São criados pelos comandos SET ALTERNATE TO e SET ALTERNATE ON; contêm as saídas de tela geradas por vários comandos do Clipper que não os @...SAY.

Arquivos de Programas: extensão .PRG

São arquivos-texto padrão ASCII, podendo ser criados ou modificados pelo editor de textos de sua preferência. Estes arquivos contêm o código-fonte

de seus programas, ou seja, os comandos e funções do Clipper e manipulações de dados (nomes e conteúdos de variáveis e campos) que constituem suas instruções.

Arquivos de Rotinas (procedures): extensão .PRG

Contêm uma ou mais rotinas ou programas identificados pelos comandos PROCEDURE ou FUNCTION. Um arquivo de rotinas é um arquivo-texto tipo ASCII, podendo ser criado ou modificado pelo editor de texto de sua preferência. Arquivos de Rotinas são incluídos em suas aplicações através do comando SET PROCEDURE TO. No Clipper podem ser incluídos tantos arquivos de rotinas ("procedures") quantos forem necessários à uma aplicação.

Arquivos Objeto: extensão .OBJ

Contêm o código-objeto dos programas fonte (.prg) após terem sido compilados pelo Clipper. Estes arquivos são criados pelo compilador do Clipper (CLIPPER.exe).

Arquivos de Compilação: extensão .CLP

São arquivos-texto padrão ASCII, podendo ser criados ou modificados pelo editor de textos de sua preferência. Estes arquivos contêm uma lista dos nomes dos programas fonte a serem compilados pelo compilador do Clipper gerando um único módulo-objeto.

Arquivos de "Link-Edição": extensão .LNK

São arquivos-texto padrão ASCII, podendo ser criados ou modificados pelo editor de textos de sua preferência. Estes arquivos contêm os nomes dos módulos-objeto a serem "link-editados" (encadeados) pelo link-editor PLINK86-Plus para gerar o módulo executável (.exe).

Arquivos Executáveis: extensão .EXE

São os arquivos gerados após o processo de "link-edição". Contêm o código ou programa executável de sua aplicação. Estes arquivos podem ser executados diretamente sob o Sistema Operacional e seguirão as instruções definidas em seus programas-fonte.

Arquivos de Overlay: extensão .OVL

São arquivos que podem ser gerados após o processo de "link-edição", quando se utilizam "overlays" externos. Estes arquivos contêm parte do código

executável necessário à execução de suas aplicações. Serão carregados na memória durante a execução do módulo executável apenas quando as rotinas neles contidas forem solicitadas.

4.3. A ESTRUTURA DOS ARQUIVOS DE DADOS DO CLIPPER

O Clipper pode ser classificado como um Sistema de Gerenciamento de Bancos de Dados do tipo **relacional**. A organização relacional é uma das mais naturais e flexíveis, do ponto de vista do usuário, pois trata os dados como se estivessem organizados em relações ou tabelas. Nestas tabelas as colunas são identificadas por nomes descritivos (os campos) e as linhas são os registros, onde são armazenados os dados, desta forma equivalem às fichas de um arquivo tradicional.

Esquema Lógico da Estrutura dos Arquivos Clipper

Campos →	Nome	Endereço	Telefone	Cidade	Estado
Registro 1	Antonio	Rua xxxx	65-1111	Rio	RJ
Registro 2	Ricardo	Rua zzzzz	66-6666	S.Paulo	RJ
Registro 3	Ana Maria	Rua zzzzz	61-1752	S.Paulo	SP
Registro 4	Severino	Rua wwwww	45.2345	Franca	SP
Registro 5	Cristina	Rua kkkkk	55-2233	Guaruja	SP
Registro N	Wilson	Rua ssssss	88-9900	Natal	RN

Um arquivo é, portanto, um conjunto de registros do mesmo tipo, e, por sua vez, cada registro é formado por um conjunto de campos onde estão armazenados os dados que constituem a informação desejada.

Os registros de um arquivo são numerados seqüencialmente, de acordo com a ordem cronológica de sua inclusão no arquivo. Nas versões Summer 87 e posteriores, um arquivo pode conter um número ilimitado de campos (limitado apenas à memória do equipamento). Nas versões anteriores o número máximo de campos por registro é 1.024.

Os nomes dos campos dos arquivos podem conter letras, números ou o símbolo sublinhado (_). A primeira posição do nome de um campo deve ser obrigatóriamente uma letra. Os nomes dos campos podem possuir até 10 caracteres, mas não podem possuir espaços em branco entre eles.

Para formar os registros dos arquivos de dados do Clipper, podem ser definidos os seguintes tipos de campos:

- Caractere (alfa-numéricos)
- Numérico

- Data
- Lógico
- Memo

A definição do tipo e características dos campos faz parte da estrutura dos arquivos de dados. A seguir são descritos os tipos de campos possíveis de serem definidos nos arquivos de dados do Clipper.

- **Campo Caractere (tipo C):** quando os dados a serem nele armazenados possuírem, na sua composição, caracteres numéricos (0-9), caracteres alfabéticos (a-z ou A-Z) ou especiais (+ * % $). Seu tamanho máximo é de 32 kbytes (32.768 posições) na versão Summer 87 e 254 bytes (254 posições) nas versões anteriores.
- **Campo Numérico (tipo N):** quando os dados a serem nele armazenados possuírem, na sua composição, somente caracteres numéricos, os sinais + ou − e o ponto decimal (o ponto decimal, e não vírgula, pois o Clipper utiliza o sistema americano). Estes campos destinam-se, ao armazenamento de dados com os quais se realizarão cálculos matemáticos. Um campo numérico pode conter até 19 dígitos ou posições, com até 18 casas decimais para a versão Summer 87 ou 9 para as anteriores.
- **Campo Lógico (tipo L):** quando os dados a serem armazenados representarem apenas dois estados possíveis, refletindo as condições de sim ou não ou falso ou verdadeiro. Seu conteúdo poderá ser somente .Y. (yes/sim), .N. (no/não), .T. (true ou verdadeiro) ou .F. (false ou falso), sendo entretanto somente gravados os símbolos T ou F. Um campo lógico terá apenas uma posição. Os símbolos lógicos devem estar sempre delimitados entre pontos: .T. ou .F..
- **Campo Data (tipo D):** quando os dados armazenados possuírem na sua composição 8 caracteres com números e a barra "/" no formato 99/99/99. Este campo armazena apenas datas, e permite a realização de uma série de operações com elas (como diferenças em número de dias, por exemplo). Um campo data possuirá sempre 8 posições e o Clipper não irá permitir o armazenamento de datas inválidas como 30/02/88. Campos data quando usados como chave de classificação, ordenarão o arquivo cronológicamente. O formato padrão de datas no Clipper é o americano, ou seja mes/dia/ano. Para utilizar o formato de datas adotado no Brasil, dia/mes/ano, utiliza-se o comando SET DATE BRITISH ou SET DATE FRENCH.
- **Campo Memo (tipo M):** um campo memo pode conter textos longos, como por exemplo observações ou comentários, específicos para cada registro do arquivo de dados. Estes textos serão armazenados em um arquivo separado identificado pelo mesmo nome do arquivo de dados a ele associado e pela extensão (.dbt). O Clipper define um comprimento de 10 posições para os campos memo, contudo o tamanho máximo que um campo memo poderá ter por registro é de 64 Kbytes (65.535 caracteres) na versão Summer 87 ou 5 Kbytes nas versões anteriores. Os campos memo são edi-

tados e manipulados no Clipper através de um conjunto de funções especiais: as funções MEMO ( ).

4.4. ALIASES – NOMES ALTERNATIVOS PARA OS ARQUIVOS DE DADOS

Um "alias" é um nome alternativo para um arquivo de dados. Ele é definido quando o arquivo é aberto através do comando USE, em uma determinada área de trabalho. Na verdade o alias do arquivo identifica a área de trabalho na qual o mesmo foi aberto.

Um "alias" é usado principalmente para permitir o acesso ao campo de um arquivo aberto em uma outra área de trabalho que não a atualmente selecionada. O Clipper não permite o acesso a um campo de um arquivo aberto em outra área de trabalho a menos que o respectivo alias seja informado.

A partir da versão Summer 87, podem ser abertos até 255 arquivos de dados simultaneamente, cada um em uma área de trabalho diferente (deve ser utilizada a versão 3.3 ou superiror do DOS, caso contrário o limite será de apenas 15).

O comando SELECT é utilizado para selecionar a área de trabalho desejada, através a especificação do "alias" do arquivo nela aberto. Os "aliases" de A a L ou de 1 a 255 são automaticamente atribuídos às áreas de trabalho nas quais forem abertos arquivos de dados. Por exemplo:

```
SELECT 1        && seleciona a área de trabalho 1
USE Mala        && abre-se nela o arquivo Mala.dbf
SELECT 5        && seleciona a área de trabalho 5
USE Clientes    && abre-se nela o arquivo Clientes.dbf
```

Se o nome para um "alias" não for especificado quando da abertura de um arquivo, o Clipper irá assumir o próprio nome do arquivo como nome do "alias", além da letra ou números atribuídos à área na qual o arquivo foi aberto.

Para especificar um nome alternativo, ou seja, um "alias" ao arquivo e consequentemente à área de trabalho na qual o mesmo foi aberto, deve ser utilizada a cláusula ALIAS, na abertura do arquivo através do comando USE. Por exemplo:

```
SELECT 1
USE Mala ALIAS Mld
* atribui o "alias" Mld para o arquivo Mala.dbf e para
* a área de trabalho 1.
SELECT 5
USE Clientes ALIAS Cli
* atribui o "alias" Cli para o arquivo Clientes.dbf e
* para a área de trabalho 5
```

Tanto a letra, o número, o "alias" automático assumido pelo Clipper, como o "alias" especificado, podem ser utilizados para identificar um campo de um arquivo aberto na respectiva área de trabalho.

Os campos de um arquivo de dados abertos em outra área de trabalho, que não a atualmente selecionada, devem ser identificados através da seguinte "sintaxe":

<alias>→<nome do campo>

onde <alias> é o nome, letra ou número que identifica a área de trabalho onde o arquivo desejado está aberto e o <nome do campo> é o nome do campo desejado daquele arquivo.

O uso do "alias" é fundamental nos comandos SET RELATION, REPLACE, SKIP. Aconselha-se, que sempre se faça a atribuição de "aliases próprios" (através da cláusula ALIAS do comando USE) aos arquivos abertos em qualquer sistema. Esse nome alternativo facilitará muito a criação e a manutenção dos programas.

4.5. VARIÁVEIS DE MEMÓRIA

4.5.1. Tipos de Variáveis

Durante o processamento de um programa existem determinadas informações que necessitam ser armazenadas na memória do computador para poderem ser utilizadas. Estas informações são armazenadas através de nome chamados de variáveis, pois podem ter seu conteúdo modificado (variado) durante o fluxo do processamento.

Uma variável é, portanto, qualquer dado armazenado temporariamente na memória do computador através de um nome, e que pode ter seu valor variado durante um processamento.

Os nomes definidos para variáveis podem conter letras, números ou o símbolo sublinhado (_), sendo formados por até 10 caracteres, entretanto o primeiro caractere deve obrigatoriamente ser uma letra. Uma variável poderá possuir o mesmo nome que o campo de um arquivo de dados, contudo, quando este nome for utilizado simultaneamente, o campo do arquivo terá precedência sobre a variável. Para especificar uma variável quando ela possuir nome idêntico ao campo de um arquivo deve ser utilizada a letra "M" como "ALIAS":

M→<nome da variável>.

No Clipper podem ser definidos diferentes tipos de variáveis, conforme a característica do dado nelas armazenado:

Alfa-Numérica (Caractere ou "String")
Numérica
Lógica
Data

- **Variável Caractere:** quando os dados armazenados possuem, na sua composição, caracteres numéricos (0-9), caracteres alfabéticos (a-z ou A-Z) ou especiais (+ * %). Os valores ou constantes a serem armazenados devem ser sempre delimitados por aspas (""), apóstrofos (')ou colchetes ([ ]). O tamanho máximo de variáveis deste tipo é 32 Kbytes (32.768 caracteres), na versão Summer 87, e de 254 bytes (254 caracteres), nas versões anteriores. O valor nulo para variáveis ou constantes tipo caractere é o código ASCII 0, representado por dois delimitadores contíguos, por exemplo:

  nulo = ""

- **Variável Numérica:** quando os dados ou constantes nela armazenados possuem, na sua composição, somente caracteres numéricos, os sinais + ou - e o ponto decimal (o ponto decimal, e não vírgula, pois o Clipper utiliza o sistema americano). Essas variáveis destinam-se, portanto, à realização de cálculos matemáticos. Uma variável numérica pode conter até 19 dígitos ou posições, com até 18 casas decimais de precisão na versão Summer 87 ou 9 nas versões anteriores. Um dado ou uma constante numéricos não devem ser delimitados por nenhum símbolo, caso contrário serão considerados como caractéres ou ocorrerá algum erro. O valor nulo para variáveis ou constantes numéricas é o zero.
- **Variável Lógica:** quando o dado armazenado representar apenas dois estados possíveis: falso ou verdadeiro. Seu conteúdo poderá ser: .Y. (yes), .N. (no), .T. (true ou verdadeiro) ou .F. (false ou falso). Os valores armazenados devem ser sempre delimitados por pontos, por exemplo: .T. ou .F..
- **Variável Data:** quando os dados nela armazenados possuírem na sua composição 8 posições constituídas por números e "/", no formato 99/99/99. Esta variável representa datas e permite a realização de uma série de operações com elas (como a obtenção da diferença entre duas datas em número de dias). Para a definição de variáveis tipo DATA é necessário utilizar a função CTOD( ), que transforma dados tipo caractere em dados tipo data. Por exemplo:

```
aniver=CTOD("08/03/58")
```

O formato padrão de datas do Clipper é o americano, ou seja, mes/dia/ano. Para convertê-lo para o formato adotado no Brasil, dia/mes/ano, utiliza-se o comando SET DATE BRITISH ou SET DATE FRENCH.

O valor nulo para as variáveis ou constantes data é a data em branco ou data vazia, por exemplo:

```
datanula = CTOD(SPACE(8)) ou
datanula = CTOD(" / / ")
```

As variáveis podem ser declaradas (criadas) através do comando STORE ou do operador " = " (igual). O número máximo de variáveis que podem ser criadas é 2.048, ou o limite da memória do equipamento utilizado.

4.5.2. Classes de Variáveis

Uma variável pode ser, de acordo com sua classe, normal, pública ou privada, dependendo da abrangência de sua utilização dentro das rotinas e programas:

- **Variáveis Normais:** são aquelas criadas em uma rotina, através do comando STORE ou do operador "=". Elas retêm o seu conteúdo durante a execução desta rotina e de outras por ela chamadas (rotinas de nível inferior). Ao finalizar-se a rotina que as criou são automaticamente canceladas.
- **Variáveis Públicas:** são aquelas declaradas pelo comando PUBLIC e criadas em qualquer rotina através do comando STORE ou do operador "=". Elas retêm o seu conteúdo durante a execução de todos os programas, do sistema, independendo da rotina. Ao contrário das variáveis normais, as públicas não são canceladas ao final da rotina na qual foram criadas; seu conteúdo é acessado por qualquer rotina em todo o sistema desenvolvido, evitando a necessidade de passagem de parâmetros de uma rotina para outra.
- **Variáveis Privadas:** são, em oposição às públicas, aquelas declaradas pelo comando PRIVATE, criadas em uma determinada rotina através do comando STORE ou do operador "=". Elas, como as variáveis normais, retêm o seu conteúdo apenas durante a execução desta rotina ou de rotinas por ela chamadas. Contudo, as variáveis privadas podem ter nomes idênticos a outras variáveis que sejam públicas ou normais, pois são reconhecidas apenas pela rotina que as criou, não afetando o conteúdo das outras variáveis em outras rotinas ou partes do programa.

Resumindo: **variáveis normais** são criadas em uma rotina e canceladas no final desta; podem ser utilizadas na rotina que as criou e em todas as outras por ela chamadas. **Variáveis Públicas** são declaradas e criadas em uma rotina; podem ser utilizadas por qualquer outra, pois não são canceladas ao seu final. E finalmente, **variáveis privadas** são declaradas, criadas e canceladas ao final de uma determinada rotina; podem ser usadas apenas nela, não afetando outras variáveis que possuam o mesmo nome criadas em outras rotinas.

4.5.3. Variáveis Indexadas — Vetores

Vetores são "variáveis" que podem conter diversos elementos, identificados por um índice numérico. Em cada um destes elementos podem ser armazenados dados de qualquer tipo. São também denominados "variáveis indexadas".

O Vetor lista

Elementos →	1	2	3	4	5	6	....	n
Conteúdos →	10	58	AB	.T.	231	0	....	VID

Cada um dos elementos de um vetor é identificado pelo nome do vetor e, entre colchetes, pelo seu índice.

Por exemplo: **lista[5]** é o elemento número cinco do vetor lista; seu conteúdo é o valor 231. A seguinte expressão é válida: var = lista[1] + lista[2]; var assumirá o valor 68.

Vetores são declarados pelo comando **DECLARE**, que define seu o nome e o número elementos, que poderá ser no máximo 4.096 (na versão Summer 87 e posteriores) ou 2.048 (nas versões anteriores). Por exemplo:

DECLARE v[100], cria um vetor que possui 100 elementos; v[10]=1000, armazena o valor 1000 no décimo elemento do vetor v.
v[20]="Clipper", armazena a palavra Clipper no vigésimo elemento do vetor v.

Nas versões anteriores à Summer 87, os vetores são sempre variáveis de classe privada, e seus elementos devem ser tratados para todas as operações como tal. A partir da versão Summer 87 os vetores podem ser declarados como públicos (através do comando PUBLIC) ou privados (através dos comando DECLARE ou PRIVATE), e seus elementos passarão a ser tratados de acordo com a classe do vetor.

4.6. OPERADORES DO CLIPPER

O Clipper possui quatro tipos de operadores que permitem a construção de expressões sobre valores, variáveis e campos de arquivos:

- Matemáticos
- Relacionais
- Lógicos
- Caracteres

4.6.1. Operadores Matemáticos

Os operadores matemáticos operam sobre expressões ou valores numéricos e geram números como resultado. São os seguintes:

```
+   — Adição
—   — Subtração
*   — Multiplicação
/   — Divisão
^   — Exponenciação
%   — Módulo (resto de divisão)
( ) — Agrupamento
```

Em operações de cálculo a ordem de precedência para a execução das operações é a seguinte:

1 — Os sinais unários (+ ou —)
2 — A exponenciação (^)
3 — O módulo (%), a multiplicação (*) e a divisão (/)
4 — A adição (+) e a subtração (—)

O operador módulo calcula o resto da divisão de uma expressão numérica por outra. Assim, por exemplo, 4%3 resulta 1.

Os parênteses são utilizados para agrupar operações e determinar a precedência de umas sobre as outras. Por exemplo:

```
? 4*2+1     && resulta 9
? 4*(2+1)   && resulta 12
```

Podem ser definidos tantos níveis de parênteses quantos forem necessários, desde que seu balanceamento esteja correto.

4.6.2. Operadores Relacionais

Os operadores relacionais geram resultados lógicos (verdadeiro ou falso) a partir do relacionamento (comparação) de duas expressões. Podem ser utilizados em expressões numéricas, alfanuméricas (caractere) ou data. As expressões relacionadas devem ser obrigatoriamente do mesmo tipo.

Os operadores relacionais do Clipper são os seguintes:

```
<    — Menor que
>    — Maior que
=    — Igual a
==   — Duplo igual a
```

< >, # ou !=	— Não igual a
<=	— Menor ou igual a
>=	— Maior ou igual a
$	— Comparação de subcadeia

Quando se relacionam duas expressões através destes operadores, a expressão da direita é comparada (relacionada) com a da esquerda, resultando dessa comparação o valor lógico verdadeiro (.T.) ou falso (.F.), dependendo se o relacionamento (encarado como um afirmação) for verdadeiro ou falso. Por exemplo:

```
a=100
b=200
? a > b                      && resulta falso       (.F.)
? b > a                      && resulta verdadeiro (.T.)
? a < > b                    && resulta verdadeiro (.T.)
? "Clipper" = "Cli"          && resulta             (.T.)
? "Clipper" < > "Clipper"    && resulta             (.T.)
```

Os operadores relacionais possuem apenas uma ordem de precedência, da esquerda para a direita, através da qual as operações são realizadas.

O operador $ subcadeia ou "substring" opera apenas sobre expressões alfanuméricas e compra a expressão da esquerda com a da direita, retornando verdadeiro se a expressão da esquerda está contida ou é idêntica à expressão da direita.

Por exemplo:

```
a="Clipper"
b="dBASE III"
c="Clipper, o Compilador do dBASE III"
? a $ b && resulta falso       (.F.)
? a $ c && resulta verdadeiro (.T.)
? b $ c && resulta verdadeiro (.T.)
? c $ a && resulta falso       (.F.)
```

O operador ==, duplo igual, atua de maneira idêntica ao operador = (igual) para expressões numéricas ou datas. Contudo, no caso de expressões alfanuméricas (caracteres), resultará verdadeiro apenas se a expressão da direita for exatamente idêntica expressão da esquerda. Por exemplo:

```
a="Cli"
b="Clipper"
? b = a      && resulta verdadeiro (.T.)
? b == a     && resulta falso       (.F.)
```

4.6.3. Operadores Lógicos

Os operadores lógicos resultam valores lógicos (falso ou verdadeiro) a partir da comparação de duas expressões lógicas. Os operadores lógicos do Clipper são os seguintes:

```
.AND.       — Conector lógico E
.OR.        — Conector lógico OU
.NOT. ou ! — Conector lógico NÃO
( )         — Parênteses para agrupamentos
```

A ordem de precedência para os operadores lógicos é a seguinte:

1 — Negação .NOT. ou !
2 — Conector .AND.
3 — Conector .OR.

Os operadores lógicos relacionam duas expressões lógicas; por exemplo:

```
a=100
b=200
c=300
? b > a .AND. c > b && resulta verdadeiro (.T.)
? b > a .AND. b = c && resulta falso       (.F.)
? b > a .OR.  b = c && resulta verdadeiro (.T.)
```

O operador .NOT. ou ! (negação) atua sobre uma única expressão. Se a expressão for verdadeira, irá tranformá-la em falsa e vice-versa. Por exemplo:

```
x= .T.
?  .NOT. x          && resulta falso          (.F.)
?  ! x              && resulta falso          (.F.)

SEEK codigo         && pesquisa o código
IF .NOT. FOUND( ) && se NÃO foi encontrado
   RETURN           && finaliza
ENDIF
```

4.6.4. Operadores de Caracteres

Os operadores de caracteres apenas atuam sobre expressões alfanuméricas; existem apenas dois:

```
+  — Concatenação com brancos
—  — Concatenação sem brancos
```

O operador +, concatenação com brancos, une duas ou mais cadeias de caracteres em uma única. Por exemplo:

```
a="Clipper "
b="- O compilador do dBASE III"
c="a+b
? c  && resulta "Clipper - O compilador do dBASE III"
```

O operador -, concatenação sem brancos, também une duas ou mais cadeias de caracteres em uma única, contudo os espaços em branco que precedem o operador são movidos para o final da cadeia de caracteres que o segue. Por exemplo:

```
a="Clipper"
b="     O compilador do dBASE III"
c=a+b
? c  && resulta "Clipper     O compilador do dBASE III"
d=a-b
? d  && resulta "ClipperO compilador do dBASE III
```

4.6.5. Ordem Geral de Precedência

Quando vários operadores distintos são utilizados em uma mesma expressão, a ordem de precedência para a execução das operações é a seguinte:

1 - Caracteres (string)
2 - Matemáticos
3 - Relacionais
4 - Lógicos

Quando as operações possuirem a mesma precedência serão executadas da esquerda para a direita. Os parênteses podem ser utilizados para modificar e determinar a ordem na qual as operações deverão ser efetuadas. As operações que estiverem dentro dos parênteses mais internos serão executadas em primeiro lugar e assim por diante.

De forma geral, são efetuadas as operações contidas no parênteses mais interno para o mais externo, sendo inicialmente executadas as operações matemáticas, seguidas das comparações relacionais e finalmente são avaliados os conectores lógicos. Por exemplo, o resultado da expressão abaixo é verdadeiro (.T.)

```
(4*(3+5))/2 > 100 .OR. (5*(3+4))/2 <100
```

4.7. SINTAXE GERAL DOS COMANDOS E FUNÇÕES DO CLIPPER

No Clipper, como em qualquer outra linguagem de programação, existem regras que determinam a forma de utilização de seus comandos e funções. Estas regras são normalmente chamadas de "**sintaxe**".

Um comando, de forma geral, consiste em uma palavra chave e outras que especificam parâmetros adicionais a serem supridos pelo usuário. Por exemplo: USE <arquivo>.

Uma função, de forma geral, consiste em uma palavra chave seguida pelo abre parênteses "(", argumentos e o "fecha parênteses ")". Por exemplo: SQRT(x).

As palavras chaves que identificam os comandos e funções do Clipper não podem ser utilizadas como nomes de arquivos, campos ou variáveis. São palavras reservadas. Em todo o livro serão sempre identificadas através de letras maiúsculas.

Os símbolos relacionados abaixo serão utilizados para determinar a sintaxe correta dos comandos e funções, indicando, em cada caso, se os parâmetros ou argumentos são obrigatórios ou opcionais. Estes símbolos apenas indicam a sintaxe dos comandos não fazendo parte dela, portanto, não devem ser utilizados nos programas.

[ ] — Colchetes indicam que a parte do comando ou função entre eles é opcional, podendo ou não ser especificada pelo usuário.

< > — Menor e Maior indicam que a informação entre eles deve ser fornecida pelo usuário (programador), tal como o nome de um arquivo, uma expressão, o nome de um campo etc.

... — Três pontinhos indicam que podem ser especificadas inúmeras cláusulas, como por exemplo no comando

DO CASE...CASE...ENDCASE.

/ — A barra indica que podem ser alternativamente especificadas as informações ou parâmetros por ela separados.

Vários comandos do Clipper, principalmente aqueles destinados à consulta ou visualização dos dados, possuem uma estrutura de parâmetros semelhantes, consistida de:

COMANDO [OFF] [escopo] FIELDS <campos>
[FOR/WHILE >expressão lógica>]

Onde,

COMANDO é o nome do comando do Clipper, uma palavra chave, em geral um verbo em inglês.

OFF omite o número do registro para os comandos DISPLAY e LIST.

<escopo> é a quantidade de registros a ser processada pelo comando.

RECORD n – determina o número físico do registro dentro do arquivo.
NEXT n – determina quantos registros serão processados, a partir do registro corrente (inclusive).
ALL – processa todos os registros do arquivo em uso.

FIELDS informa o nome dos campos dos registros que deverão ser processados.

FOR permite a utilização de uma expressão lógica para selecionar os registros a serem processados. FOR "varre" o arquivo inteiro e processa todos os registros para os quais a expressão é verdadeira.

WHILE também utiliza uma expressão para selecionar os registros a serem processados. WHILE não "varre" o arquivo inteiro, processa a partir do registro corrente apenas ENQUANTO a expressão permanecer verdadeira.

Quando as cláusulas FOR e WHILE forem utilizadas em um comando, não é necessário especificar o escopo dos registros a serem processados. As cláusulas FOR e WHILE podem ser usadas simultaneamente na construção de uma expressão condicional para a seleção dos registros.

Por exemplo, a sintaxe do comando LIST é a seguinte:

```
LIST [OFF] [escopo] FIELDS <lista de campos>
     [FOR <condição> [WHILE <condição>]]
     [TO PRINT] [TO FILE <nome do arquivo>]
```

A seguir apresentamos alguns exemplos de sua utilização:

```
LIST FIELDS codigo,nome,salario FOR salario > 100000
LIST ALL codigo,nome,salario
LIST WHILE codigo>"8000" nome,salario,depto
LIST FOR salario< 50000 WHILE depto="CONTABILIDADE"
```

Para maiores detalhes consulte os comandos específicos no Cap. 5.

4.8. TECLAS PARA CONTROLE DE EDIÇÃO DE DADOS

Ao editar (incluindo ou alterando) dados através do comando **GET**, que permite a edição em telas formatadas, o cursor poderá ser movimentado e os dados poderão ser alterados através de teclas de controle especiais. Estas teclas permitem deslocar o cursor no campo sendo editado e entre os diversos campos, eliminar ou inserir caracteres ou dados e gravar ou cancelar as alterações efetuadas.

A seguir apresentamos cada uma das teclas de controle de edição e sua respectiva função.

Tecla	Movimento
Seta Esquerda	Move uma posição para a esquerda. Não move o cursor para o GET anterior.
Seta Direita	Move uma posição para a direita. No final de um GET o cursor é movido para o próximo GET.
Ctrl e Seta Esquerda	Move para a palavra à esquerda.
Ctrl e Seta Direita	Move para a palavra à direita.
Seta para Cima	Move para o GET anterior.
Seta para Baixo ou Enter	Move para o próximo GET.
Home	Move para o início do campo do GET.
End	Move para o fim do campo do GET.
Ctrl e Home	Move para o início do primeiro GET.
Ctrl e End	Move para o início do último GET.

Tecla	Ação
Del	Apaga o caractere sobre o cursor.
Backspace	Move o cursor um caractere para a esquerda apagando-o.
Ctrl e T	Apaga a palavra à direita do cursor.
Ctrl e Y	Apaga os caracteres à direita do cursor até o final do GET.
Ctrl e U	Recupera o valor original dos dados no GET atual.
Ins	Liga ou desliga o modo de inserção de caracteres.
Tecla	**Ação**
Ctrl e W Ctrl e End Ctrl e C PgUp e PgDn	Finalizam um comando READ, salvando os dados editados nos GET's.
Enter	Finaliza um comando READ se o cursor estiver posicionado no último GET.
Esc	Finaliza um comando READ, sem salvar as alterações efetuadas nos GET's.

5

Comandos do Clipper

5.1. NOTAÇÃO UTILIZADA PARA A APRESENTAÇÃO DOS COMANDOS

Este capítulo apresenta de forma completa os comandos da "linguagem Clipper", classificados de acordo com seu propósito ou função principal, para facilitar a localização de acordo com as necessidades do leitor. Cada comando será apresentado através dos seguintes tópicos:

– Comando:	palavra-chave ou nome que o identifica
– Sintaxe:	regras de utilização do comando
– Propósito:	sua função ou propósito principal
– Utilização:	forma e conceitos para sua aplicação
– Biblioteca:	biblioteca de rotinas que contém o comando
– Exemplo:	exemplos genéricos de aplicação
– Veja também:	outros comandos ou funções relacionadas

Em todo livro usaremos um tipo de notação já tradicionalmente padronizada, para a apresentação e explicação dos comandos e funções do Clipper. As convenções e símbolos desta notação são apresentados a seguir:

- As palavras-chave que identificam os comandos e funções do Clipper estarão sempre escritas em letras maiúsculas; por exemplo: LIST;
- A parte a ser suprida pelo usuário aparecerá entre os sinais < > e estará em letras minúsculas; por exemplo: USE <nome do arquivo>;
- As cláusulas opcionais de alguns comandos, ou argumentos opcionais de algumas funções serão apresentados entre colchetes [ ], significando que o usuário poderá ou não incluir esta parte do comando ou função, pois ela é opcional; por exemplo: CLEAR [SCREEN];

- As opções alternativas de alguns comandos serão indicadas através da barra /; por exemplo: SET DATE AMERICAN/BRITISH/FRENCH/ ANSI/ITALIAN/GERMAN;
- Cláusulas repetitivas ou extensas serão indicadas por retissências ...; por exemplo: IF...ELSEIF... ELSEIF...ENDIF;
- Os nomes das funções ou rotinas definidas pelo usuário, ou seja, rotinas ou funções não pertencentes ao Clipper, terão a primeira letra do seu nome apresentada em letra maiúscula e as restantes em letras minúsculas; por exemplo: Dpc( ) ou Aviso;
- Os nomes de arquivos serão apresentados em letras maiúsculas.

Exemplo de Notação utilizada para apresentação do comando DISPLAY:

```
DISPLAY    [OFF] <lista de campos>
           [FOR <condição>]
           [WHILE <condição>]
           [TO PRINT/TO FILE <nome do arquivo>]
```

Onde:

DISPLAY —é a palavra-chave (nome) que indentifica o comando.

OFF —é uma cláusula opcional para apresentação ou não do número dos registros na tela.

<lista de campos> —é uma lista obrigatória a ser suprida pelo usuário, com os nomes dos campos do arquivo em uso, separados por vírgula.

FOR e WHILE —são cláusulas opcionais, para a seleção de registros a serem processados.

<condição> —é uma expressão lógica usada para especificar a seleção dos registros a serem processados.

TO PRINT/TO FILE — são cláusulas opcionais alternativas para direcionar a saída do comando para a impressora ou para um arquivo em disco, ao invés da tela.

<nome do arquivo> —caso a cláusula TO FILE seja utilizada o nome do arquivo a ser gravado deverá ser especificado pelo usuário.

Ao contrário do que ocorre com as funções, todos os comandos do Clipper estão contidos na Biblioteca **CLIPPER.LIB**, que deverá estar disponível ao link-editor (através da lista de bibliotecas) quando for efetuado o encadeamentos dos módulos-objeto da aplicação desenvolvida.

5.2. COMANDOS PARA MANUTENÇÃO DE VARIÁVEIS

Comando CLEAR MEMORY:

Sintaxe:

CLEAR MEMORY

Propósito:

O comando CLEAR MEMORY cancela todas as variáveis de memória, públicas ou privadas, que estiverem ativas.

Utilização:

CLEAR MEMORY cancela todas as variáveis, quer sejam elas públicas (veja o comando PUBLIC) ou privadas (veja o comando PRIVATE). O comando RELEASE ALL, em contraste com o comando CLEAR MEMORY, cancela apenas as variáveis privadas pertencentes à rotina atual.

Biblioteca:

CLIPPER.LIB

Veja Também:

RELEASE ALL, CLEAR ALL

Comando DECLARE:

Sintaxe:

DECLARE **<nome 1>[<exp.numérica>**
[, <nome 2>[<exp.numérica>]] ...
[, <nome n>[<exp.numérica>]]

Propósito:

O comando DECLARE cria um ou mais vetores (variáveis indexadas de uma dimensão), com o número de elementos definido pela expressão numérica.

Utilização:

O comando DECLARE deve ser utilizado sempre que se desejar trabalhar com vetores (variáveis indexadas). Ele "declara" o nome e o número de elementos que o vetor possuirá, permitindo que neles sejam armazenados dados de qualquer tipo.
Os elementos do vetor podem armazenar dados de diferentes tipos (numéricos, caractere, data e lógicos). Cada vetor pode possuir até 2.048 elementos nas versões anteriores à Summer 87 e até 4.096 nesta versão. Cada vetor é considerado, contudo, como sendo uma única variável (lembre-se que o Clipper permite a criação de até 2.048 variáveis).
No caso particular deste comando, os colchetes envolvendo a expressão numérica não indicam uma cláusula opcional, mas fazem parte da sua sintaxe.
Os dados de um vetor são acessados através do seu nome e do número do elemento, especificado entre os colchetes.
Vários vetores podem ser declarados em um mesmo comando DECLARE, bastando especificá-los através de seus nomes e do respectivo número de elementos entre colchetes, separados por vírgulas.

```
DECLARE vetor1[10], vetor2[20],...vetorN[100]
```

A função LEN( ) pode ser utilizada para determinar o número de elementos de um vetor, tendo como argumento o nome do vetor.
Se o argumento da função TYPE( ) for o nome de um vetor ela irá retornar "A", indicando que se trata de um "Array" – vetor.
Não pode ser dado o mesmo nome à uma variável e à um vetor. A criação de uma variável com o mesmo nome de um vetor irá destruir o vetor e apagar todos os dados armazenados em seus elementos.
Vetores não podem ser gravados no disco ("salvos") em arquivos de variáveis do tipo (.mem) (veja os comandos SAVE e RESTORE).
Vetores são sempre declarados como variáveis privadas nas versões Autunm 86 e anteriores. Na versão Summer 87 um vetor já pode ser declarado como variável pública.
A função macro-substituição & pode ser utilizada com vetores, mas de acordo com as seguintes regras:

```
x = "ve1"
eleme = "ve[10]"

DECLARE &x[10]
* equivalente a DECLARE ve1[10]
&eleme = 1000
* equivalente a ve[10]=1000
&x[5] = 500
* equivalente a ve1[5] =500
```

```
? &x[5], &eleme
* equivalente a ? vel[5], ve[10]
```

DECLARE &eleme **Não é válido!**

Passagem de Parâmetros: Vetores inteiros e elementos de vetores podem ser passados como parâmetros a programas ou rotinas e funções-de-usuário escritas em Clipper e para programas externos escritos em outras linguagens como C ou Assembler. Vetores são sempre passados por referência, e elementos de vetores são passados por valor (veja o comando PARAMETERS para maiores detalhes).
Não é possível, contudo, passar vetores inteiros a programas externos usando o comando CALL. Neste caso somente pode ser passado um elemento de cada vez, e por valor.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
DECLARE vetor[10]
vetor[1]="Clipper"
vetor[2]=DATE()
vetor[3]=10000
FOR i=1 TO 3
    ? vetor[i]
NEXT
* Resulta: Clipper
           20/03/88
           10000
```

A rotina a seguir cria dois vetores com 50 elementos cada um. O primeiro conterá o nome dos alunos segundo conterá suas notas, que serão digitadas:

```
CLEAR
DECLARE aluno[50], nota[50]
* Declara os vetores aluno e nota
* O "Loop" a seguir entra todos os nomes e notas
FOR i=1 TO 50
    ACCEPT "Aluno Número "+STR(i,2)+": " TO aluno[i]
    INPUT  "Nota: " TO nota[i]
NEXT
CLEAR

* O "Loop" a seguir mostra os dados digitados
FOR i=1 TO 50
    ? aluno[i], nota[i]
NEXT
RETURN
```

Veja Também:

PRIVATE, STORE, DO, PARAMETERS, AINS( ), ADEL( ), AFILL( ), ASCAN( ), ADIR( ), ASORT( ), TYPE( ), LEN( ), ACHOICE( )

Comando PRIVATE:

Sintaxe:

PRIVATE <lista de variáveis/vetores>

Propósito:

O comando PRIVATE permite que variáveis de memória ou vetores contidos em uma rotina (ou programa) tenham o mesmo nome que outras variáveis criadas em rotinas de maior nível (que chamaram a rotina atual) ou que tenham sido previamente declaradas como públicas.
A declaração de vetores através do comando PRIVATE está disponível apenas a partir da versão Summer 87.

Utilização:

O comando PRIVATE é utilizado para declarar vetores ou variáveis privadas, isto é, restritas a uma única rotina ou programa, mesmo que existam outras rotinas com variáveis ou vetores com o mesmo nome.
Alterações feitas no conteúdo de uma variável privada não afetam os valores de outras variáveis que possuam o mesmo nome em outras rotinas de maior nível. Quando uma rotina contendo variáveis privadas finaliza, os valores das variáveis que possuiam o mesmo nome em outras rotinas é restabelecido e os valores privados (válidos apenas para a rotina em questão) perdem o efeito.
A <lista de variáveis/vetores> é a lista de nomes de vetores (com o respectivo número de elementos) ou variáveis a serem declaradas privadas. As declarações PRIVATE ALL, LIKE ou EXCEPT não são suportadas pelo Clipper.
O comando PRIVATE, assim também como o comando PUBLIC não cria variáveis, apenas define sua classe. Uma variável declarada privada ou pública permanece indefinida até que algum conteúdo seja a ela atribuído. Entretanto, no caso de vetores, o comando PRIVATE é equivalente ao comando DECLARE, definindo o número de elementos que o vetor conterá, sem entretanto, atribuir conteúdo aos seus elementos.
A utilização mais comum de vetores ou variáveis privadas é na passagem de parâmetros de programas para rotinas auxiliares, onde todas as variáveis

podem ser privadas, não influindo no programa principal, mesmo que possuam nomes idênticos a outras variáveis de maior nível.

Biblioteca:

CLIPPER.LIB

Exemplo:

* **Programa Principal** – Chama a Rotina1

```
CLEAR
var1="Clipper - Compilador do dBASE III"
var2=1000.00
@ 04,10 SAY "Valores Originais das Variáveis"
@ 06,10 SAY "Variável 1: "
@ 07,10 SAY "Variável 2: "
@ 06,22 SAY var1
@ 07,22 SAY var2 PICTURE "@E 999,999.99"
* Os  valores apresentados serão: "Clipper - Compilador
* do dBASE III" e 1.000,00
DO rotina1   && Executa a Rotina1
@ 16,10 SAY "Valores Após a Execução da Rotina1"
@ 18,10 SAY "Variável 1: "
@ 19,10 SAY "Variável 2: "
@ 18,22 SAY var1
@ 19,22 SAY var2 PICTURE "@E 999,999.99"
* Os  valores apresentados  serão os mesmos: "Clipper -
* Compilador  do dBASE III" e 1.000,00, pois na Rotina1
* var1     e    var2    foram  declaradas privadas, não
* alterando,  portanto  o valor original das variáveis
* do programa principal.
RETURN
```

Rotina1 – Declara privadas as variáveis var1 e var2

```
PRIVATE var1,var2
var1="Eu sou uma variável privada da Rotina1"
var2=10000.00
@ 10,10 SAY "Valores das Variáveis na Rotina1"
@ 12,10 SAY "Variável 1: "
@ 13,10 SAY "Variável 2: "
@ 12,22 SAY var1
@ 13,22 SAY var2 PICTURE "@E 999,999.99"
* Os  valores apresentados  serão: "Eu sou uma variável
* privada da Rotina1" e 10.000,00.
RETURN
```

O exemplo a seguir mostra como devem ser declarados vetores e variáveis PRIVADOS:

```
PRIVATE vetor[30], var1, var2

PRIVATE vetor1[10] é equivalente a DECLARE vetor1[10]
```

Veja Também:

DO, PARAMETERS, PUBLIC, DECLARE

Comando PUBLIC:

Sintaxe:

PUBLIC <lista de variáveis/vetores>

Propósito:

O comando PUBLIC declara vetores ou variáveis como sendo globais, tornando-as disponíveis (para serem utilizadas ou alteradas) a qualquer rotina ou sub-rotina do sistema. Ao contrário das variáveis normais e privadas, as variáveis declaradas públicas não são canceladas ao final da rotina ou do programa que as criou, mantendo sempre o último conteúdo armazenado.
A declaração de vetores como públicos foi implementada a partir da versão Summer 87.

Utilização:

Variáveis ou vetores devem ser declarados públicos quando forem utilizados ou alterados por diversos programas, rotinas e sub-rotinas dentro do sistema desenvolvido, pois sempre estarão disponíveis a todos os módulos do sistema.
Ao se declarar uma variável como pública, quando a variável ainda não existir, o Clipper cria uma variável do tipo lógica, com o conteúdo falso (.F.). Uma vez atribuído um valor a ela, sua abrangência torna-se global e a variável passa a ser disponível para todas as rotinas ou programas de uma aplicação.
Ao se declarar um vetor como público define-se apenas o número de elementos que o vetor possuirá, não atribuindo nenhum conteúdo aos mesmos, contudo sua abrangência será global.
Os dados armazenados nas variáveis ou vetores públicos podem ser alterados por qualquer programa ou rotina que os utilizem, evitando assim a passagem de parâmetros de uma rotina para outra, quando estes parâmetros são comuns a todas elas.
O conteúdo de um vetor ou uma variável pública pode ficar suspenso temporariamente através da declaração de variáveis ou vetores privados com o mesmo nome em outras rotinas de menor nível. Esses vetores ou variáveis privadas são em geral utilizados para receber os parâmetros passados, através do comando DO WITH <lista de parâmetros>, para uma rotina de menor nível.

Não é permitido declarar um vetor ou uma variável privada já existente como pública. Neste caso o Clipper irá simplesmente ignorar a declaração e o vetor ou variável permanecerão privados.
A declaração de vetores públicos foi implementada a partir da versão Summer 87. Nas versões anteriores os vetores são definidos apenas pelo comando DECLARE, que sempre os declara como privados.

Biblioteca:

CLIPPER.LIB

Exemplos:

* **Programa Principal** – Chama a Rotina2

```
CLEAR
PUBLIC var1,var2
var1="Clipper - Compilador do dBASE III"
var2=1000.00
@ 04,10 SAY "Valores Originais das Variáveis"
@ 06,10 SAY "Variável 1: "
@ 07,10 SAY "Variável 2: "
@ 06,22 SAY var1
@ 07,22 SAY var2 PICTURE "@E 999,999.99"
* Os  valores apresentados serão: "Clipper - Compilador
* do dBASE III" e 1.000,00
DO rotina2 WITH var1,var2  && Executa a Rotina2
@ 19,10 SAY "Valores Após a Execução da Rotina2"
@ 21,10 SAY "Variável 1: "
@ 22,10 SAY "Variável 2: "
@ 21,22 SAY var1
@ 22,22 SAY var2 PICTURE "@E 999,999.99"
* Os  valores apresentados  serão os mesmos: "Clipper
* Compilador do dBASE III" e 1.000,00, pois na Rotina2
* var1     e     var2     foram  declaradas privadas, não
* alterando,   portanto  o valor original das variáveis
* públicas do programa principal.
RETURN
```

* **Rotina2** – Declara privadas as variáveis var1 e var2

```
PARAMETERS var1,var2
@ 09,10 SAY "Valores das Variáveis Públicas na Rotina2"
@ 11,10 SAY "Variável 1: "
@ 12,10 SAY "Variável 2: "
@ 11,22 SAY var1
@ 12,22 SAY var2 PICTURE "@E 999,999.99"
* Os valores apresentados são os mesmos que no programa
* principal.
PRIVATE var1,var2
var1="Eu sou uma variável privada da Rotina2"
var2=10000.00
```

```
@ 14,10 SAY "Valores das Variáveis Públicas na Rotina2"
@ 16,10 SAY "Variável 1: "
@ 17,10 SAY "Variável 2: "
@ 16,22 SAY var1
@ 17,22 SAY var2 PICTURE "@E 999,999.99"
* Os valores apresentados serão: "Eu sou uma variável
* privada da Rotina 1" e 10.000,00.
RETURN
```

Exemplo de declaração de vetores e variáveis públicos:

PUBLIC vetor[20], var1, var2

Veja Também:

DO, PARAMETERS, PRIVATE

Variável CLIPPER:

Sintaxe:

CLIPPER

Propósito:

Permitir que os comandos do Clipper que não existem no dBASE III Plus possam ser incluídos nos programas. No Clipper eles serão executados e no dBASE III serão ignorados, permitindo que o programa possa ser rodado em ambos.

Utilização:

Utiliza-se a variável CLIPPER, que deve ser declarada pública no início do programa principal (veja o comando PUBLIC), para permitir que os comandos que existem no Clipper e não existem no dBASE possam ser incluídos nos programas e utilizados conforme o caso.
No Clipper esta variável será automaticamente inicializada como verdadeira (.T.) e no dBASE III será inicializada com falsa (.F.). Através do comando IF...ENDIF, podem ser construídas estruturas que decidirão por usar ou não os comandos específicos do Clipper, caso o programa esteja sendo executado no Clipper ou no dBASE III.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
CLEAR
PUBLIC CLIPPER
CLEAR
IF CLIPPER
   FOR i=1 TO 23
       @ i,31 SAY "Estou no Clipper !"
   NEXT
ELSE
   i=1
   DO WHILE i<24
      @ i,31 SAY "Estou no dBASE III !"
      i=i+1
   ENDDO
ENDIF
RETURN
```

Obs.: O comando FOR...NEXT não existe no dBASE III veja o Cap. 2 para maiores detalhes.

Comando STORE:

Sintaxe:

STORE <expressão> TO <lista de variáveis>

Propósito:

O comando STORE cria e atribui um conteúdo a uma ou mais variáveis de memória. Uma alternativa para o comando STORE é o operador igual "=":

<nome da variável> = <expressão>

Utilização:

O comando STORE é usado para criar e atribuir conteúdos a variáveis de memória. Quando se deseja criar e inicializar apenas uma variável, é preferível a utilização do operador igual "=". Entretanto, quando se deseja criar várias variáveis, todas inicializadas com o mesmo valor, é mais interessante utilizar-se o comando STORE.

A <expressão> é um valor de qualquer tipo de dado a ser atribuído como conteúdo da variável especificada.

A <lista de variáveis> define os nomes, separados por vírgulas, das variáveis a serem criadas ou a terem novos valores atribuídos. Os nomes de variáveis

podem ter até 10 caracteres e conter letras, dígitos e o sublinhado (_). O primeiro caractere deve, entretanto, ser obrigatoriamente uma letra.
Os nomes das variáveis não poderão conflitar com os nomes dos comandos ou funções do Clipper.
Se uma variável já existe e for novamente criada pelo comando STORE, seu conteúdo original será sobreposto pelo novo valor à ela atribuído.
Para a criação de variáveis numéricas basta atribuir um valor numérico às mesmas:

```
STORE 10 TO var_num
ou
var_num = 10
```

Para a criação de variáveis alfa-numéricas, basta atribuir uma cadeia de caracteres às mesmas:

```
STORE "Clipper" TO var_alfa
ou
var_alfa = "Clipper"
```

Se um campo memo for atribuído à uma variável não existente, a variável criada será do tipo caractere.
Para a criação de variáveis lógicas, basta atribuir os valores .T. (verdadeiro) ou .F. (falso) às mesmas:

```
STORE .T. TO verdade
ou
falso = .F.
```

Para a criação de variáveis tipo data deve ser utilizada a função CTOD( ) ("Caracter TO Date" – caractere para data), para transformar dados caractere em dados data, pois não é possível entrar datas diretamente pelo teclado:

```
STORE CTOD("08/01/88) TO hoje
ou
amanhã = CTOD("09/01/88")
```

Finalmente, para a criação de variáveis indexadas (vetores) deve ser utilizado o comando DECLARE. O comando STORE e o operador igual "=" podem apenas alterar o conteúdo dos elementos de variáveis indexadas.

```
DECLARE vetor[10]
STORE 10 TO vetor[1]
vetor[2]=20
```

As variáveis criadas serão sempre privadas, a menos que antes sejam explicitamente declaradas públicas pelo comando PUBLIC. Variáveis privadas são sempre canceladas quando o programa ou rotina onde foram criadas é finalizado através do comando RETURN.
Uma variável numérica pode ser operada (adicionada, subtraída etc.) dela mesma e uma variável caractere poderá ser concatenada à ela mesma. Por exemplo:

```
var1 = 0
var1 = var1+1
? var1
* Resultará 1.

var2 = "Clipper"
var2 = var2+"— Compilador do dBASE III"
* Resultará Clipper — Compilador do dBASE III"
```

O número máximo de variáveis de memória ativas no Clipper é 2.048 ou o limite da memória do computador utilizado. Um vetor pode ter até 2.048 elementos (4.096 na versão Summer 87), e é considerado, para efeito do limite de variáveis ativas, como uma única variável.
Os comandos RELEASE, CLEAR MEMORY e CLEAR ALL cancelam variáveis de memória criadas pelo comando STORE ou pelo sinal de igual "=".
Se uma variável possuir o mesmo nome que o campo de um arquivo de dados, o campo terá precedência sobre ela em qualquer operação. Para diferenciar uma variável de um campo de arquivo, se ambos possuírem o mesmo nome, basta colocar antes do nome da variável o indicativo: M→<nome da variável>.

```
REPLACE código WITH M→código
        campo      variável
```

Biblioteca:

CLIPPER.LIB

Exemplos:

```
STORE 0 TO var1, var2, var3, var4
* Cria e inicializa com 0 as variáveis var1 a var4.

var1=0
var2=0
var3=0
var4=0
* Equivale ao comando STORE acima.
```

```
STORE .T. to verdade
verdade = .T.
* São equivalentes: criam e inicializam a variável
lógica verdade com .T. (verdadeiro)

STORE 45 TO A
* Armazena 45 na variável A.
B = (A + 3.14) / 2
* Armazena o resultado da expressão na variável B

DECLARE ve[10]   && Cria o vetor "ve" com 10 elementos
STORE 0 TO ve[1] && Armazena 0 no elemento ve[1]
ve[2]="Clipper"  && Armazena Clipper no elemento ve[2]
STORE 2*A TO ve[5]
* Armazena  90 (resultado  da expressão)  no elemento 5
* do vetor ve.

STORE CTOD("07/01/88") TO hoje
* Armazena a data na variável hoje tipo data.
```

Veja Também:

DECLARE, CTOD( ), DTOC( ), SAVE, RESTORE, RELEASE, CLEAR MEMORY, CLEAR ALL, PUBLIC, PRIVATE e a seção 4.6 – Operadores do Clipper.

Comando RELEASE:

Sintaxe:

RELEASE [<**lista de variáveis**>]/
[**ALL** [**LIKE/EXCEPT** <**máscara de seleção**>]]

Propósito:

O comando RELEASE elimina (cancela) variáveis de memória liberando o espaço por elas ocupado na memória do computador.

Utilização:

Utiliza-se o comando RELEASE para cancelar variáveis de memória que tenham sido criadas, liberando, dessa forma, espaço na memória do computador.
A <lista de variáveis> é uma lista contendo os nomes, separados por vírgulas, das variáveis a serem eliminadas.

A <máscara de seleção> é uma máscara tipo DOS para a especificação de um grupo de variáveis a ser eliminado ou excluído da eliminação. Todas as variáveis que tiverem o nome coincidente com a máscara especificada serão selecionadas. Por exemplo: var* seleciona todas as variáveis que possuem o nome começando por var.

Podem ser eliminadas apenas algumas variáveis, através da especificação de seus nomes na lista. Neste caso as variáveis eliminadas poderão ser públicas ou privadas.

Quando se utiliza o escopo ALL, todas variáveis criadas na rotina ou programa atual são canceladas. Neste caso, não são canceladas as variáveis públicas ou as definidas em rotinas ou programas de maior nível.

Podem ser selecionadas automaticamente as variáveis a serem eliminadas através do escopo LIKE, que eliminará todas cujos nomes coincidam com determinada máscara, ou EXCEPT, que eliminará todas, com exceção das que os nomes coincidam com a máscara.

Na máscara de seleção, um ponto de interrogação (?) substitui um único caractere e um asterisco (*) substitui um ou mais caracteres.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
RELEASE var1,var2,var3
* Elimina as variáveis var1, var2 e var3, quer sejam
* públicas ou privadas.

RELEASE ALL
* Elimina todas as variáveis criadas em uma
* determinada rotina ou programa, não afetando as
* demais.

RELEASE ALL LIKE var*
* Elimina todas as variáveis que tenham seus nomes
* iniciados por var.

RELEASE ALL EXCEPT var1*
* Elimina todas as variáveis que não tenham o nome
* iniciado por var1.

RELEASE ALL LIKE var?5
* Elimina todas as variáveis que tenham seu nome
* iniciando por var, qualquer caracter na quarta
* posição e finalizado pelo digito 5.
```

Veja Também:

CLEAR MEMORY, CLEAR ALL, RESTORE, SAVE, STORE, "=".

Comando RESTORE FROM:

Sintaxe:

RESTORE FROM <arquivo de variáveis> [ADDITIVE]

Propósito:

Recuperar (ler) e ativar variáveis de memória armazenadas no disco em um arquivo de variáveis de memória tipo (.mem.), através do comando SAVE.

Utilização:

O comando RESTORE recupera e ativa novamente variáveis de memória (exceto vetores) que tenham sido gravadas no disco em um arquivo tipo (.mem.) através do comando SAVE.
O <arquivo de variáveis> é o nome do arquivo que contêm as variáveis a serem recuperadas (lidas e reativadas).
Quando variáveis são recuperadas através do comando RESTORE, todas as outras variáveis existentes são canceladas, a menos que se utilize a cláusula ADDITIVE. As variáveis recuperadas serão sempre privadas dentro da rotina ou programa que as recuperou, não importando a sua classe na ocasião em que foram "salvas".
No máximo 2.048 variáveis de memória podem estar ativas. Caso esse número seja ultrapassado será apresentada uma mensagem de erro.
Se for usada a cláusula ADDITIVE (aditivas), as variáveis oriundas do disco serão adicionadas às ativas até o limite de variáveis permitido pelo Clipper. Caso as variáveis recuperadas tenham nome igual às existentes, estas serão sobrepostas, contudo se forem públicas permanecerão públicas. Para que as variáveis recuperadas sejam públicas é necessário que sejam declaradas públicas antes de se executar o comando RESTORE.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
var1=3243
var2="Clipper"
var3=CTOD("25/12/87")
SAVE TO Arqvar
```

* Cria o arquivo Arqvar.mem e grava nele as variáveis var1, var2 e var3.

```
var4=34.98
RESTORE FROM Arqvar

    * Recupera as variáveis var1, var2 e var3 do arquivo Arqvar e elimina a
    * variável var4, pois a cláusula ADDITIVE não foi utilizada.

    ? var1, var2, var3
```

Veja Também:

PUBLIC, PRIVATE, RELEASE, SAVE, STORE

Comando SAVE:

Sintaxe:

SAVE TO <arquivo> [ALL [LIKE/EXCEPT <seleção>]]

Propósito:

O comando SAVE armazena variáveis de memória ativas em um arquivo no disco. O arquivo normalmente recebe a extensão <.mem>, por se tratar de um arquivo de variáveis de memória.

Utilização:

O comando SAVE permite gravar no disco variáveis de memória ativas e seus conteúdos para posterior recuperação através do comando RESTORE. Apenas o nome e o conteúdo das variáveis pode ser salvo, referências quanto à classe das variáveis (públicas ou privadas) não são salvas. Além disso, o comando SAVE também permite salvar vetores ou elementos de vetores em arquivos de variáveis de memória.
O <arquivo> é o nome do arquivo onde as variáveis especificadas serão gravadas (salvas). Se não for especificada uma extensão, será assumida a extensão (.mem).
A <seleção> é uma máscara tipo DOS que permite especificar um grupo de variáveis a serem salvas.
Se as cláusulas ALL, LIKE ou EXCEPT não forem usadas, todas as variáveis ativas no programa ou rotina atual (públicas ou privadas) serão gravadas no arquivo.
As variáveis a serem gravadas (salvas) podem ser selecionadas automaticamente através das cláusulas LIKE e EXCEPT. A cláusula LIKE salvará todas as variáveis que possuam o nome coincidindo com determinada máscara e

a cláusula EXCEPT salvará todas as variáveis, com exceção das que coincidam com a máscara. A máscara de seleção deve ser especificada de forma semelhante ao DOS, onde: um ponto de interrogação (?) substitui um único caractere e um asterisco (*) substitui um ou mais caracteres.

Exemplos:

SAVE TO B:arqmem
* Salva todas as variáveis ativas para o arquivo arqmem.mem no drive B.

SAVE TO arqvar ALL LKE var*
* Salva todas as variáveis que tenham o nome iniciando por var para o arqui-
* vo Arqvar.mem no acionador padrão (disco default).

Veja Também:

RESTORE, RESTORE SCREEN, SAVE SCREEN, STORE

5.3. COMANDOS PARA SAÍDA DE DADOS NA TELA OU NA IMPRESSORA

Comando ? ou ??

Sintaxe:

? <lista/expressão> ou ?? <lista/expressão>

Propósito:

Apresentar dados ou resultados de expressões na tela ou na impressora separados por um espaço.

Utilização:

A <lista/expressão> é uma lista de valores de qualquer tipo de dados a serem apresentados. Esta lista pode conter qualquer combinação de tipos de dados, incluindo campos memo.
O comando interrogação possui duas formas. O ponto de interrogação simples avança uma linha antes de apresentar a lista de campos/variáveis especificados. Caso nada seja especificado será apresentada uma linha em branco. O ponto de interrogação duplo apresenta, na mesma linha, a lista de campos/variáveis especificados.

Ambos iniciam apresentação dos dados na primeira coluna da tela ou da impressora, separando cada dado apresentado por um espaço em branco.
Se a lista/expressão contiver cálculos ou operações a serem avaliados, serão apresentados seus respectivos resultados.

Bibliotecas:

CLIPPER.LIB

Exemplos:

? "Clipper"	Apresenta Clipper na linha seguinte da tela.
?? "Clipper"	Apresenta Clipper na mesma linha da tela.
USE PESSOAL ? número,nome,salário	Abre o Arquivo PESSOAL.dbf Apresenta o conteúdo dos campos número, nome, salário do registro corrente.
Número Nome 12345 Jose da Silva	Salário 10000
? salário>10000	Avalia a operação, resultando .F. (falso) ou .T. (verdadeiro) caso o campo salário seja maior que 10000.
? 2*30/(1+5)	Calcula a expressão e apresenta o resultado 10.
? DATE( )	Apresenta a data atual do sistema, 05/12/87.

Formatação de uma Tela:

```
? "  Menu Principal"
? "  ---------"
?
? "  1 - Inclusão"
? "  2 - Alteração"
? "  3 - Exclusão"
? "  4 - Relatórios"
? "  5 - Fim"
?
? "  Escolha sua Opção:
```

Veja Também:

— @...SAY, TEXT...ENDTEXT

Comando @...BOX:

Sintaxe:

@<lse,cse,lid,cid> BOX <bordas>>

Propósito:

O comando @...BOX desenha caixas ou molduras retangulares na tela. Podem ser especificados o tamanho e a posição relativa da moldura na tela, e até nove diferentes caracteres podem ser utilizados para formá-la.

Utilização:

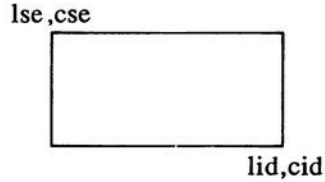
O comando @...BOX é especialmente indicado para o desenho de molduras (ou caixas) na tela, com a função de ressaltar títulos, conjuntos de dados, janelas de visualização etc. O efeito visual conseguido pode ser muito bonito, dependendo de sua criatividade.
O tamanho da caixa e sua posição são determinados por quatro coordenadas numéricas que localizam, através de números correspondentes às linhas e colunas da tela, os cantos superior esquerdo e inferior direito, sendo:

Canto Superior Esquerdo:

lse — linha superior esquerda (valores de 0 a 24)
cse — coluna superior esquerda (valores de 0 a 79)

Canto Inferior Direito:

lid — linha inferior direita (valores de 0 a 24)
cid — coluna inferior direita (valores de 0 a 79)



Para as bordas podem ser especificados 8 caracteres padrão ASCII respectivamente a serem desenhados em cada um dos quatro cantos e cada um dos quadro lados da caixa.
Um nono caractere pode ser especificado para preencher o interior da caixa.
Os caracteres devem ser especificados na seguinte ordem:

1 – canto superior esquerdo
2 – lado superior horizontal
3 – canto superior direito
4 – lado vertical direito
5 – canto inferior direito
6 – lado inferior horizontal
7 – canto inferior esquerdo
8 – lado vertical esquerdo
9 – caractere para preenchimento

Bibliotecas:

CLIPPER.LIB

Exemplos:

Para desenhar uma caixa que moldure toda a tela, com linhas duplas, utilize o seguinte conjunto de caracteres:

```
borda1=CHR(201)+CHR(205)+CHR(187)+CHR(186)+CHR(188)+;
       CHR(204)+CHR(200)+CHR(186)

@ 01,00,24,79 BOX borda1
```

Para desenhar a mesma caixa, só que preenchendo seu conteúdo, utilize:

```
borda2=CHR(201)+CHR(205)+CHR(187)+CHR(186)+CHR(188)+;
       CHR(204)+CHR(200)+CHR(186)+CHR(176)

@ 01,00,24,79 BOX borda2
```

Veja Também:

– @...TO, @...CLEAR, @...SAY...GET

Comando @...CLEAR:

Sintaxe:

@ ⟨lse,cse⟩CLEAR TO ⟨lid,cid⟩

Propósito:

Limpar (apagar) uma região retangular da tela.

Utilização:

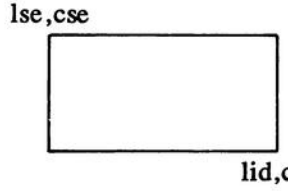
O comando @...CLEAR deve ser utilizado quando se deseja limpar ou apagar apenas uma determinada região da tela, e não toda ela. Este comando é muito interessante na obtenção de efeitos especiais como janelas.
O tamanho da região e sua posição são determinados por quatro coordenadas numéricas. Estas coordenadas localizam, através de números correspondentes às linhas e colunas da tela, os cantos superior esquerdo e inferior direito, sendo:

Canto Superior Esquerdo:

lse – linha superior esquerda (valores de 0 a 24)
cse – coluna superior esquerda (valores de 0 a 79)

Canto Inferior Direito:

lid – linha inferior direita (valores de 0 a 24)
cid – coluna inferior direita (valores de 0 a 79)



A cláusula TO que define o canto inferior direito da região da tela a ser limpa é opcional. Se esta cláusula não for especificada o Clipper assumirá como padrão a linha 24 e a coluna 79, ou seja as coordenadas da extremidade inferior direita da tela.

Bibliotecas:

CLIPPER.LIB

Exemplos:

```
@ 10,20 CLEAR TO 20,40 && limpa uma região retangular

@ 06,00 CLEAR          && limpa da linha 6 para baixo
```

Veja Também:

@...BOX, CLEAR, SCROLL( )

Comando @...PROMPT:

Sintaxe:

@ <linha,coluna> **PROMPT** <exp.caractere 1>
[**MESSAGE** <exp.caractere 2>]

Propósito:

Este comando cria um Menu de Opções na tela. Cada opção é apontada por um cursor luminoso sendo escolhida através das setas de direção e <Enter> ou através da letra inicial de cada opção. Uma mensagem explicativa sobre a função de cada opção pode ser apresentada para facilitar a escolha. Com este comando Menus do tipo do Lotus 1-2-3 podem ser facilmente criados.

Utilização:

O comando @...PROMPT é "incrível", com ele a confecção de menus de "Barras Luminosas" tipo 1-2-3, SideKick etc. é imediata. Você poderá utilizar este comando para sofisticar todos os menus de suas aplicações, fornecendo ao seu usuário um sistema visualmente muito agradável e de fácil operação.

A posição de cada opção do menu é definida através das coordenadas <linha,coluna>.

O dizer de cada opção é definido pela <exp.caractere 1> após a palavra chave PROMPT.

Uma mensagem explicativa sobre a função de cada opção disponível pode ser apresentada pela <exp.caractere 2> após a cláusula MESSAGE. A mensa-

gem associada a cada opção será apresentada cada vez que o usuário posicionar o cursor sobre esta opção.

Cada @...PROMPT...MESSAGE cria apenas uma opção do menu com a respectiva mensagem associada. Portanto, para a criação de um menu completo devem ser utilizados tantos @...PROMPT...MESSAGE quantas forem as opções desejadas, nas posições adequadas da tela.

O comando **SET MESSAGE TO <n>**, onde n variando de 1 a 24 é o número de uma linha da tela, determina a linha onde a mensagem associada a cada opção será apresentada.

O menu de barras luminosas criado pelo comando @...PROMPT...MESSAGE é invocado pelo comando **MENU TO**, que atribui a uma variável um valor numérico correspondente ao número da opção escolhida.

As opções podem ser colocadas em qualquer ordem e posição da tela (horizontalmente, verticalmente ou assimetricamente). O comando MENU TO "navega" a barra luminosa que envolve as opções na ordem em que as mesmas forem colocadas, isto é, do primeiro @...PROMPT para o último.

Durante a execução do programa, as opções podem ser escolhidas através das setas de direção, pelo posicionamento da barra luminosa sobre a opção desejada e a seguir teclando-se <Enter>; ou diretamente pelo simples teclar da primeira letra de cada opção.

Podem ser criados até um máximo de 32 @...PROMPT's por menu.

Quando uma opção possuir a mesma letra inicial que outra, a opção que vier em primeiro lugar, na ordem dos @...PROMPT's será selecionada, caso sua escolha seja feita diretamente através das letras.

A cor das opções será a mesma da definida como padrão para a tela e a cor da barra luminosa será a mesma da definida como realçado através do comando SET COLOR TO.

Biblioteca:

CLIPPER.LIB

Exemplos:

O exemplo a seguir constrói um menu horizontal semelhante aos do Sistema MLD do Cap. 10 (veja neste capítulo as ilustrações):

```
DO WHILE .T.
   CLEAR
   @ 06,01 TO 23,79 DOUBLE
   * Constrói uma moldura dupla na tela
   mp=1
   SET MESSAGE TO 5
   * As mensagens serão apresentadas na linha 5.
   @ 03,02 PROMPT "Inclusões"   MESSAGE "  Inclusões de Registros"
   @ 03,13 PROMPT "Alterações"  MESSAGE "  Alterações de Registros"
   @ 03,25 PROMPT "Exclusões"   MESSAGE "  Exclusões de Registros"
```

```
   @ 03,36 PROMPT "Consultas"   MESSAGE "  Consultas na Tela"
   @ 03,47 PROMPT "Relatórios"  MESSAGE "  Menu de Relatórios"
   @ 03,59 PROMPT "Utilitários" MESSAGE "  Back Up's dos Arquivos"
   @ 03,73 PROMPT " Fim "       MESSAGE "  Fim das Operações"
   * Evite colocar opções que se iniciem pela mesma letra,
   * caso contrário a escolha através de letras será prejudicada.
   MENU TO mp
   * A variável mp recebe o valor da opção escolhida
   @ 24,00 CLEAR
   * A estrutura DO CASE a seguir determina a ação a tomar,
   * em função do opção escolhida.
   DO CASE
      CASE mp=1
        DO Inclui
      CASE mp=2
        DO Altera
      CASE mp=3
        DO Exclui
      CASE mp=4
         DO Consulta
      CASE mp=5
         DO Imprime
      CASE mp=6
         DO Salva
      CASE mp=7
        EXIT
        * Fim
   ENDCASE
ENDDO
CLEAR
USE
RETURN
```

Veja Também:

MENU TO, SET COLOR TO, SET MESSAGE TO, SET WRAP, ACHOICE( )

Comando @...SAY...GET (Arroba):

Sintaxe:

@ <linha,coluna>
[SAY <expressão> [PICTURE <máscara de formato>]]
[GET <expressão> [PICTURE <máscara de formato>]]
[RANGE <exp.numérica 1,exp.numérica 2>]
[VALID <exp.lógica>]

Propósito:

Criação de formatos de entrada ou saída de dados na tela ou na impressora. O comando arroba edita ou apresenta as informações formatadas nas posições desejadas, de acordo com as coordenadas das linhas e colunas especificadas.

Utilização:

O comando @...SAY...GET é o mais importante e mais utilizado comando para a entrada e a saída de dados do Clipper, tanto na tela como na impressora. Através dele podem ser definidas as posições onde um dado deverá ser editado (entrada/alteração) ou apresentado (saída) e a forma como isso irá ocorrer.
As coordenadas <**linha,coluna**> são expressões numéricas que definem a posição da tela ou da impressora onde os dados serão apresentados ou editados.
Para um monitor de 24 linhas × 80 colunas a coordenada da linha poderá variar de 1 a 24 e a da coluna de 0 a 79. Na linha 0 são apresentadas mensagens de status do Clipper, portanto ela não deve ser utilizada.
Para uma impressora normalmente a coordenada da coluna poderá variar de 0 a 256, enquanto que para a coordenada da linha não há limite, sendo o normal de 0 a 66 (número de linhas por página para um formulário de 11 polegadas).
A **cláusula SAY** apresenta uma expressão de qualquer tipo de dado (inclusive campos memo), a partir das coordenadas especificadas na tela ou na impressora.
Normalmente a saída dos dados será a tela. Para direcionar a saída do comando @...SAY para a impressora utiliza-se o comando SET DEVICE TO PRINT; para retornar para a tela utiliza-se SET DEVICE TO SCREEN. O comando @...GET não pode ser direcionado para a impressora.
Quando enviando comandos @...SAY para a impressora, algumas regras devem ser observadas:

- As coordenadas da linha e da coluna não devem ultrapassar 256;
- O decréscimo do número da linha em comandos @...SAY consecutivos causará uma ejeção de página do formulário e a reinicialização da numeração das linhas e colunas da impressão. Pode-se, entretanto, utilizar a função SETPRC( ) para redefiní-las;
- Da mesma forma, se dois comandos @...SAY possuírem o mesmo número de linha, o segundo deverá ter o número da coluna maior que o primeiro, caso contrário ocorrerá sobre-impressão.

Para direcionar a saída dos comandos @...SAY para um arquivo utilizam-se os comandos SET DEVICE TO PRINT e em seguida SET PRINTER TO <nome do arquivo>. A saída será gravada num arquivo padrão texto cujo nome deve ser especificado.

Os dados são apresentados pelo comando @...SAY na cor padrão, ou na cor definida pelo comando SET COLOR TO no vídeo normal.
@ <**linha,coluna**> limpa a linha, iniciando da coordenada indicada.

A **cláusula GET** apresenta e permite a edição (inclusão e alteração) de dados contidos em variáveis de memória ou em campos de arquivos, na posição especificada pelas coordenadas <linha,coluna>. O comando READ deve ser em seguida utilizado em conjunto com um ou mais comandos @...GET para ativar o modo de edição em tela-cheia, permitindo o uso das teclas de controle de edição, e completar a edição efetuando a "leitura" e gravação dos dados.
Podem ser lidos campos de arquivos abertos em outras áreas de trabalho através do comando SELECT, pelo uso do respectivo ALIAS (nome alternativo). Por exemplo:

```
@ 12,20 SAY "Digite um Dado:" GET alias→campo
```

A cláusula GET edita os dados na cor realçada, a menos que esta seja desativada. A cor padrão do vídeo realçado pode ser redefinida pelo comando SET COLOR TO.
Os comandos @...GET ou @...SAY...GET não podem ser redirecionados para a impressora ou para um arquivo. Os dados apenas podem ser editados na tela.

Opções:

- **PICTURE:** Permite, através de uma máscara e ou função, a saída de dados formatados usando-se o comando @...SAY e restringe ou pré-define (para efeito de consistência) os dados que poderão ser editados (entrados ou alterados) em uma variável ou campo de arquivo usando-se o comando @...GET. Uma máscara de formatação deve ser delimitada por aspas (") ou apóstrofes ('). Uma função de formatação, iniciando-se pelo símbolo @, também poderá ser utilizada para dar definições aos dados. Podem ser utilizadas simultaneamente mais de uma função de formatação dentro de uma mesma cláusula PICTURE, contudo, entre uma função e uma máscara deve sempre haver um espaço em branco.

Máscaras disponíveis para formatação ou edição:

9 — aceita apenas dígitos ou sinais para dados numéricos;
— aceita apenas dígitos, brancos e sinais;
A — aceita apenas letras alfabéticas;
L — aceita apenas dados lógicos T ou F;
Y — aceita apenas os dados lógicos Y,y ou N,n;
N — aceita apenas letras e dígitos;
X — aceita qualquer caractere;

! – converte letras minúsculas para maiúsculas;
$ – mostra cifrões à esquerda de valores dados;
* – mostra asteriscos à esquerda de dados numéricos;
. – determina a posição do ponto decimal para dados numéricos;
, – separa milhares para dados numéricos.

Outros caracteres que não os acima especificados em uma máscara sobrepõem o caractere na mesma posição, tanto em entradas quanto em saídas de dados.

Exemplos:

```
valor=5427.45
@ 12,25 SAY valor PICTURE "9,999.99"
* O resultado será 5,427.45

cpf=SPACE(14)
@ 12,25 SAY "CPF:" GET cpf PICTURE "999.999.999-99"
READ
```

* Os pontos e o tracinho já estarão posicionados para a entrada do CPF, bastando digitar os números. Tanto os números como os símbolos serão gravados na variável.

```
nome=SPACE(15)
@ 12,20 SAY "Nome:" GET nome PICTURE "! ! ! ! ! ! ! ! ! ! ! ! ! ! !"
READ
* Converte para maiúscula qualquer nome digitado.

valor=25427.45
@ 12,25 SAY valor PICTURE "$$$,$$$,$$$.$$"
* O resultado será $$$$$25,427.45

valor=25427.45
@ 12,25 SAY valor PICTURE "***,***,***.**"
* O resultado será *****25,427.45
```

Funções para formatação ou edição:

A – aceita apenas caracteres alfabéticos em um @...GET;

B – apresenta os números alinhados à esquerda;

C – mostra CR (crédito) após um número positivo;

D – mostra datas no formato definido pelo comando SET DATE;

E — mostra datas no formato (dd/mm/aa) e dados numéricos no formato europeu, com vírgula separando casas decimais e pontos separando os milhares (999.999,99);

K — limpa os dados tipo caractere inicialmente contidos na variável se a primeira tecla pressionada não for uma tecla de movimentação do cursor (seta por exemplo);

R — permite a inclusão de letras e símbolos nas máscaras de formatação que não farão parte do conteúdo da variável;

S<n>— limita o tamanho do campo de edição a "n" posições, efetuando um deslocamento horizontal para permitir a edição de todo o conteúdo da variável. ("n" deve ser um número inteiro). Usa-se as setas esquerda e direita para o deslocamento;

X — mostra DB (débito) após um número negativo;

Z — transforma o valor zero em brancos;

(— coloca números negativos entre parênteses, inclusive os brancos à esquerda;

) — coloca números negativos entre parênteses, excluindo os brancos à esquerda;

! — converte as letras minúsculas para maiúsculas, aceitando qualquer caractere.

As funções B, C, X, Z, "("e")" somente se aplicam a dados numéricos. As funções C e X podem ser combinadas para se obter os dois efeitos, mas só podem ser utilizadas com o comando@...SAY.
As funções D e E aplicam-se a dados numéricos e a datas e as funções A, R, S<n> e ! aplicam-se apenas a dados caractere. A função K aplica-se a qualquer tipo de dado.

Exemplos:

```
nome=SPACE(40)
@ 12,10 SAY "Nome:" GET nome PICTURE "@!"
READ
```

* Transforma as letras minúsculas para maiúsculas para todas as posições do nome.

```
nome=SPACE(40)
@ 12,25 SAY "Nome:" GET nome PICTURE "@S20"
READ
```

* Permite a edição do nome em uma "janela" de apenas 20 posições, realizando o deslocamento horizontal.

```
val=5427.45
@ 12,20 SAY "Lançamento:" val PICTURE "@CX 99,999.99"
```

* Resulta 5,427.45 CR

```
val=4527.45
@ 12,20 SAY "Valor:" valor PICTURE "@E 999,999.99"
```

* Resulta 5.427,45

```
cod=SPACE(7)
@ 12,20 SAY "Código:" GET cod PICTURE "@R 9.99.99.99"
READ
```

* São apresentados os pontinhos para facilitar a digitação, mas os mesmos não serão gravados como conteúdo da variável.

```
data=CTOD("17/03/88")
@ 12,30 SAY "Digite a Data: GET data PICTURE "@K"
READ
```

* Neste exemplo uma data é sugerida como um valor de entrada, contudo se a tecla digitada pelo usuário não for uma tecla de controle de edição a data sugerida automaticamente será apagada e o usuário poderá digitar a data desejada.

— **RANGE:** A cláusula RANGE é utilizada com dados numéricos ou datas, permitindo que se especifique uma faixa limitada pelos valores mínimo e máximo que poderão ser entrados em resposta à um comando @...GET. O valor mínimo deve preceder o máximo. Se valores menores ou maiores que a faixa especificada forem digitados o Clipper irá apresentar a mensagem "Invalid Range" (faixa inválida) no topo da tela e continuará solicitando os dados, até que valores válidos tenham sido digitados ou seja pressionada a tecla <Esc>; neste caso o GET será finalizado e a variável ou campo editado retornarão ao seu valor original.

Exemplos:

```
var3=0.00
@ 12,30 SAY "Valor:" GET var3 RANGE 100,200
READ
```

```
dat1=CTOD(SPACE(8))
datmin=CTOD("01/01/88")
datmax=CTOD("31/07/88")
@ 12,30 SAY "Data:" GET dat1 RANGE datmin,datmax
READ
```

- **VALID:** A cláusula VALID permite validar a entrada de dados através de um comando @...GET. A entrada não será completada enquanto o dado digitado não tornar a expressão de validação verdadeira ou até que a tecla <Esc> seja pressionada (estando comando SET ESCAPE em ON). A expressão de validação pode ser ou conter uma função definida pelo usuário (função-de-usuário), que poderá efetuar um cálculo a partir do dado inicialmente digitado e retornar o resultado automaticamente.

Exemplos:

```
valor=0.00
@ 12,30 SAY "Valor:" GET valor VALID valor>0.00
READ

op=" "
@ 20,20 SAY "Sua Opção?" GET op VALID (op$"ABCD")
READ
```

Bibliotecas:

CLIPPER.LIB

Veja Também:

?/??, @...TO, @...CLEAR, CLEAR, CLEAR GETS, READ, SET BELL, SET CONFIRM, SET DELIMITERS, SET DEVICE, SET FORMAT, SET INTENSITY, SET COLOR, TEXT...ENDTEXT, COL( ), ROW( ), PCOL( ), PROW( ), SETPRC( )

Comando @...TO:

Sintaxe:

@<lse,cse> [CLEAR] TO <lid,cid> [DOUBLE]

Propósito:

Desenha uma moldura (ou caixa) com linhas gráficas simples ou duplas.

Utilização:

O comando @...TO desenha uma moldura semelhante ao comando @...BOX com linhas gráficas simples ou duplas. A diferença entre os dois é que com @...BOX os caracteres a serem utilizados na moldura podem ser definidos, além de poder ser especificado um caractere para preenchimento do interior da caixa. O comando @...TO, por sua vez, é mais simples de ser utilizado, pois é desnecessário definir-se os caracteres que comporão a moldura.
O comando @...TO é normalmente utilizado para emoldurar títulos, janelas de apresentação de dados etc., fornecendo um ótimo recurso para tornar agradáveis das telas de qualquer sistema.
<lse,cse> são as coordenadas do canto superior esquerdo onde deve se iniciar a moldura ou caixa.
<lid,cid> são as coordenadas do canto inferior direito.
Se as coordenadas das linhas forem as mesmas, apenas uma linha horizontal será desenhada. Por outro lado, se as coordenadas das colunas forem as mesmas será desenhada uma linha vertical.

Opções:

A cláusula **DOUBLE** desenha uma moldura com linhas duplas, ao invés da padrão com linhas simples.

Bibliotecas:

CLIPPER.LIB

Exemplos:

```
vi="Teste do Deslocamento Horizontal e das Caixas"
@ 03,24 TO 05,56 DOUBLE
* Desenha uma moldura de linhas gráficas duplas
@ 08,24 TO 12,56
* Desenha uma moldura de linhas gráficas simples
@ 10,30 GET vi PICTURE "@S20"
@ 20,22 SAY "Pressione as Setas para o Deslocamento"
@ 22,24 SAY "Pressione (Enter) para Finalizar"
READ
@ 08,24 CLEAR TO 12,56
* Limpa a região da tela determinada pelas coordenadas
```

Veja Também:

@...BOX, @...CLEAR, ACHOICE( ), DBEDIT( ), SCROLL( ), SAVESCREEN( ), RETSCREEN( )

Comando CLEAR:

Sintaxe:

CLEAR [SCREEN]

Propósito:

O comando CLEAR limpa toda a tela, reposicionando o cursor no canto superior esquerdo (coordenadas 00,00). Se for utilizado sem a cláusula SCREEN também cancela todos os comandos @...GET pendentes (que ainda não foram lidos por um comando READ).

Utilização:

O comando CLEAR é usado para apagar toda a tela do computador, reposicionando o cursor no canto superior esquerdo (coordenadas 00,00) da tela, e cancelar todos os comandos @...GET que estiverem pendentes, isto é, que ainda não foram lidos por um comando READ.
Caso a cláusula SCREEN (disponível a partir da versão Summer 87) seja utilizada, os comandos @...GET pendentes não serão cancelados, apenas a tela será limpa (equivalente a @ 00,00 CLEAR). Este comando é particularmente útil quando se deseja apagar a tela para mostrar, por exemplo, um HELP ao usuário, e em seguida recuperar a tela de entrads de dados anterior.

Biblioteca:

CLIPPER.LIB

Veja Também:

@...CLEAR, CLEAR GETS

Comando CLEAR GETS:

Sintaxe:

CLEAR GETS

Propósito:

Cancela todos os comandos @...GET que estiverem pendentes desde o último READ, CLEAR ALL ou CLEAR GETS executados.

Utilização:

O número de GETS que podem estar pendentes é limitado pela quantidade de memória disponível no equipamento. CLEAR GETS pode ser utilizado para cancelar GETS pendentes permitindo a utilização simultânea do comando @...SAY...GET conseguindo o efeito de vídeo reverso para a apresentação dos dados.

Biblioteca:

CLIPPER.LIB

Veja Também:

@...CLEAR, CLEAR

Comando DISPLAY:

Sintaxe:

DISPLAY [**OFF**] [<**escopo**>] <**lista de campos/expressões**>
[**FOR** <**condição**>] [**WHILE** <**condição**>]
[**TO PRINT/TO FILE** <**nome do arquivo**>]

Propósito:

Mostrar o conteúdo de um arquivo de dados, de acordo com o resultado de uma ou mais expressões para cada registro processado.

Utilização:

O comando DISPLAY permite a visualização dos registros dos arquivos de dados na tela, de acordo com várias alternativas apresentadas a seguir:

A <lista de campos/expressões> é uma lista obrigatória dos campos ou expressões os envolvendo, cujos resultados deverão ser apresentados. Os nomes dos campos ou as diferentes expressões devem vir separados por vírgulas. Os campos especificados devem existir no arquivo que estiver selecionado (ou em uso) no momento.
O <escopo> determina a quantidade de registros a ser processada, podendo ser: ALL para todos os registros, NEXT <n> os próximos n registros (inclusive o atual) ou RECORD <n> o registro de número n.

A cláusula OFF , se especificada, inibe a apresentação do número dos registros.
A cláusula FOR <condição> permite a especificação de uma condição para a seleção dos registros que deverão ser processados. Cada registro do arquivo, ou aqueles especificados pelo <escopo> serão testados de acordo com esta <condição>. Se o teste da <condição> resultar verdadeiro o registro será processado ou seja apresentado, caso contrário não, e o comando passará para o registro seguinte, até que todos os registros especificados tenham sido testados.
A cláusula WHILE <condição> permite a especificação de uma condição para a seleção dos registros que deverão ser processados enquanto a <condição> permanecer verdadeira. Os registros, a partir do registro atual, serão seqüencialmente testados de acordo com a <condição>. Enquanto o teste da <condição> resultar verdadeiro os registros serão processados, ou seja, apresentados. Se o teste resultar falso para algum registro o comando será imediatamente finalizado.
A cláusula [TO PRINT] redireciona a saída do comando para a impressora ao invés da tela (padrão normal).
A cláusula TO FILE <nome do arquivo> redireciona a saída do comando para um arquivo em disco, especificado pelo <nome do arquivo>, ao invés da tela.

Se nenhuma cláusula for utilizada apenas o registro corrente (registro no qual o apontador de registros está posicionado) será apresentado na tela.
O comando DISPLAY envia o resultado da <lista de campos/expressões> para a tela em um formato colunar, sendo cada coluna separada por um espaço. Contudo, ao contrário do dBASE III, no Clipper ele não faz uma pausa a cada tela ou inclui um cabeçalho para os dados que estão sendo apresentados. Na verdade seu resultado é praticamente idêntico ao do comando LIST. A única diferença é que o LIST sem cláusulas opcionais processa todos os registros (escopo = ALL), enquanto que o DISPLAY apenas o registro corrente (escopo = NEXT 1).

Biblioteca:

CLIPPER.LIB

Exemplos:

```
USE pessoal
DISPLAY ALL numero, nome, salario
```

* Irá apresentar na tela o conteúdo dos campos número, nome e salário de todos os registros do arquivo pessoal.

```
DISPLAY numero,nome,salario FOR salario > 10000.00
```

* Irá apresentar na tela o conteúdo dos campos número, nome e salário de todos os registros do arquivo pessoal que possuírem salário maior que 10.000,00.

```
USE Pessoal INDEX Indepto
SEEK "SISTEMAS"
DISPLAY numero,nome,salario WHILE depto="SISTEMAS"
```

* Irá apresentar o conteúdo dos campos número, nome salário enquanto o conteúdo do campo depto for SISTEMAS. Neste caso o arquivo deve estar classificado por departamento.

Para interromper o processamento do comando DISPLAY, pode ser utilizada a função INKEY( ) como parte da condição especificada.

```
USE Pessoal INDEX Indepto
SEEK "SISTEMAS"
DISPLAY ALL numero,nome,salario WHILE depto="SISTEMAS" .AND.
INKEY( ) < > 27
```

Neste caso o comando DISPLAY poderá ser interrompido teclando-se <Esc> cujo código é 27.

Veja Também:

LIST

Comando EJECT:

Sintaxe:

EJECT

Propósito:

O comando EJECT avança a cabeça de impressão da impressora para o topo de uma nova página do formulário, ou seja, ejeta uma página.

Utilização:

O comando EJECT deve ser utilizado quando se desejar, em um relatório, avançar a impressão para o topo da próxima página do formulário.

EJECT envia um código de alimentação de formulário para a impressora (o código ASCII 12). Para o posicionamento correto no topo do formulário, ajuste o papel antes de ligar a impressora. Refira-se ao manual de sua impressora para verificar estes detalhes.
Após a execução de um comando EJECT as funções PROW( ) (linha da impressora) e PCOL( ) (coluna da impressora) passam a valer zero (posição 00,00). Se for necessário mudar estes valores para outros, ou mesmo se for necessário zerá-los sem executar o comando EJECT, pode ser utilizada a função SETPRC( ).

Biblioteca:

CLIPPER.LIB

Veja Também:

CHR( ), SETPRC( ), PROW( ), PCOL( )

Comando LIST:

Sintaxe:

LIST **[OFF]** [<**escopo**>] <**lista de campos/expressões**>
[FOR <condição>]
[WHILE <condição>]
[TO PRINT]
[TO FILE <nome do arquivo>]

Propósito:

Listar o conteúdo de um ou mais campos dos registros de um arquivo de dados ou de expressões os envolvendo, de acordo com o escopo e condições especificadas.

Utilização:

O comando LIST permite que seja listado de forma tabular (em colunas) o conteúdo de um arquivo de dados, através dos campos de seus registros ou de expressões os envolvendo. LIST é idêntico ao comando DISPLAY, com a exceção de que seu <escopo> padrão é ALL, ou seja listar todos os registros, enquanto que o <escopo> padrão do DISPLAY é NEXT 1, ou seja, apenas o registro atual.

A <lista de campos/expressões> é uma lista dos campos do arquivo ou expressões válidas os envolvendo cujo conteúdo deverá ser listado. Os nomes dos arquivos ou as expressões devem ser separadas por vírgulas e devem ser válidas para o arquivo que estiver aberto na área de trabalho selecionada. Campos memo também podem ser listados.
O <escopo> define a quantidade de registros do arquivo de dados a ser processada pelo comando: ALL (todos), NEXT <n> (os próximos n registros a partir do atual) ou RECORD <n> (apenas o registro n). Se não for especificado serão assumidos todos (ALL).
A cláusula FOR especifica uma <condição>, dentro do <escopo> fornecido, para selecionar os registros que deverão ser processados pelo comando. Apenas os registros a satisfizerem (a tornarem verdadeira) terão seus dados listados.
A cláusula WHILE seleciona os registros a serem processados, a partir do registro atual, enquanto uma <condição> especificada for satisfeita (permanecer verdadeira). Para tanto o arquivo de dados deverá estar indexado.
A opção OFF inibe a apresentação do número dos registros listados.
A opção TO PRINT redireciona a saída do comando da tela (padrão normal) para a impressora.
A opção TO FILE <nome do arquivo> redireciona a saída do comando para um arquivo do disco cujo nome for especificado. Se uma extensão não for especificada para o arquivo será assumida a extensão (.txt).

Biblioteca:

CLIPPER.LIB

Exemplos:

```
USE Pessoal
LIST numero,nome,idade
```

* Irá apresentar na tela o conteúdo de todos os campos do arquivo pessoal em forma tabular (colunas).

```
LIST OFF numero, nome, idade TO PRINT
```

* Irá listar na impressora o conteúdo dos campos especificados para todos os registros, sem apresentar o respectivo número de registro.

```
LIST nome,salario FOR salario > 10000.00 TO PRINT
```

* Irá listar na impressora o conteúdo dos campos nome e salário de todos os registros do arquivo Pessoal.dbf que possuírem salário maior que 10.000,00.

LIST numero,nome,salario WHILE depto="SISTEMAS"

* Irá apresentar o conteúdo dos campos número, nome e salário enquanto o conteúdo do campo depto for SISTEMAS. Neste caso o arquivo deve estar indexado (ordenado) por departamento.

Um dos recursos mais interessantes do comando LIST no Clipper é permitir a movimentação pelos registros de um arquivo de acordo com o escopo e condições especificadas. Este recurso pode ser utilizado dentro de uma rotina que processe somente um registro (o registro atual) através de uma função-de-usuário, e faça a movimentação dos registros do arquivo através do LIST. A função Rel( ) a seguir exemplifica esta sensacional utilização do comando LIST, gerando um relatório formatado na impressora.

```
USE Pessoal Index Alfa
SET DEVICE TO PRINT
SET CONSOLE OFF
LIST Rel() FOR salario > 50000.00
SET CONSOLE ON
SET DEVICE TO SCREEN
RETURN

FUNCTION Rel
@ PROW()+1,       3 SAY numero
@ PROW(), PCOL()+2 SAY nome
@ PROW(), PCOL()+2 SAY salario PICTURE "@E 999,999.99"
RETURN ""
```

Para interromper o comando LIST utiliza-se a função INKEY( ) como parte da condição especificada. No exemplo a seguir, o LIST é finalizado ao ser pressionada a tecla <Esc>, cujo código ASCII é 27.

```
USE Pessoal
LIST numero, nome, depto, salario FOR INKEY( ) < > 27
```

Veja Também:

DISPLAY, FUNCTION

Comando RESTORE SCREEN:

Sintaxe:

RESTORE SCREEN [FROM <variável>]

Propósito:

Recuperar e reapresentar uma tela que tenha sido previamente salva na variável especificada.

Utilização:

Este comando deve ser utilizado em conjunto com o comando SAVE SCREEN (vide a seguir), para evitar que uma tela (em geral de trabalho) tenha que ser novamente "desenhada" (repetição de todos os comandos que a construiram) ao se apresentar temporariamente sobre ela outras informações, como por exemplo telas de "help" ao usuário, janelas de opções etc. Ao se executar o comando RESTORE SCREEN a tela salva é instantaneamente refeita e reapresentada.

A <variável> é o nome da variável que contém a tela a ser recuperada. A especificação do nome da variável é opcional pois o Clipper reserva uma área comum ("buffer") para salvar uma tela. Neste caso o comando RESTORE SCREEN recuperará uma tela que tenha sido salva pelo comando SAVE SCREEN, ambos sem fazer referência à uma variável.

A cláusula FROM deve ser utilizada apenas quando a tela a ser recuperada foi salva em uma variável especificada e não na área comum ("buffer").

As variáveis que contém telas são variáveis caractere como outras quaisquer. Nesse sentido podem ser salvas no disco e manipuladas normalmente, da forma que mais lhe convier.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
CLEAR
borda1=CHR(201)+CHR(205)+CHR(187)+CHR(186)+CHR(188)+;
       CHR(204)+CHR(200)+CHR(186)
borda2=CHR(201)+CHR(205)+CHR(187)+CHR(186)+CHR(188)+;
       CHR(204)+CHR(200)+CHR(186)+CHR(176)
@ 08,25,12,55 BOX borda1
@ 10,28 SAY "Teste SAVE/RESTORE SCREEN"
WAIT "Pressione uma tecla para continuar..."
SAVE SCREEN TO tela1
@ 01,00,23,79 BOX borda2
WAIT "Pressione <Enter> para retornar à tela anterior"
RESTORE SCREEN FROM tela1
```

Veja Também:

SAVE SCREEN, SAVE TO, RESTORE FROM, SAVESCREEN( ) RETSCREEN( ).

Comando SAVE SCREEN:

Sintaxe:

SAVE SCREEN [TO <variável>]

Propósito:

Salvar uma tela (inteira) na memória para após a apresentação de outras telas temporárias (por exemplo telas de "help" mostrada pelo HELP .prg), recuperá-la instantaneamente sem a necessidade de reconstruí-la (redesenhá-la).

Utilização:

Este comando deve ser utilizado em conjunto com o comando RESTORE SCREEN (veja acima), para salvar em uma área ("buffer") memória ou em variáveis, uma tela que novamente deverá ser apresentada.
A cláusula TO <variável> especifica o nome da variável de memória na qual a tela deverá ser salva. A variável criada é do tipo caractere, possuirá um comprimento de 4 Kbytes, podendo ser manipulada ou salva em disco como qualquer outra variável caractere.
Múltiplas telas podem ser salvas, cada uma em uma variável de memória específica. Se a variável não existir o comando SAVE SCREEN TO automaticamente a criará.
Além de variáveis comuns, pode-se salvar telas em elementos de vetores, facilitando a manipulação de múltiplas telas. Contudo não é possível salvar vetores ou elementos de vetores em arquivos no disco, mas apenas variáveis comuns.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
borda=CHR(201)+CHR(205)+CHR(187)+CHR(186)+CHR(188)+;
      CHR(204)+CHR(200)+CHR(186)
@ 08,25,12,55 BOX borda
@ 10,28 SAY "Teste SAVE/RESTORE SCREEN"
WAIT "Pressione uma tecla para continuar..."
SAVE SCREEN TO tela1
@ 01,00,23,79 BOX borda+CHR(176)
SAVE SCREEN TO tela2
WAIT "Pressione uma tecla para continuar..."
@ 01,00,23,79 BOX borda+"*"
```

```
SAVE SCREEN TO tela3
WAIT "Pressione (Enter) para retornar..."
RESTORE SCREEN FROM tela2
WAIT "Pressione (Enter) para retornar..."
RESTORE SCREEN FROM tela1
WAIT "Pressione (Enter) para retornar..."
RESTORE SCREEN FROM tela3
```

Veja Também:

RESTORE SCREEN, SAVESCREEN( ), RESCREEN( ), SAVE TO, RESTORE FROM.

5.4. COMANDOS PARA ENTRADA DE DADOS

Comando@...SAY...GET:

Veja Comandos Para Apresentação na Tela na seção 5.3.

Comando ACCEPT:

Sintaxe:

ACCEPT [<mensagem>] TO <variável>

Propósito:

Solicita a entrada de um dado tipo caractere através do teclado. Será criada uma variável de memória do tipo caractere onde é armazenada a entrada digitada. A digitação deve ser finalizada com <Enter>. Não é necessário delimitar-se com aspas o dado digitado.

Utilização:

O comando ACCEPT é utilizado para solicitar a entrada de um dado tipo caractere a partir do teclado. O dado digitado é armazenado na variável de memória especificada. Se esta variável já existir seu conteúdo será substituído pelo dado digitado, caso contrário será automaticamente criada.
Uma mensagem explicativa sobre o dado a ser entrado pode ser incluída e será apresentada na tela antes de sua solicitação. A entrada digitada não requer delimitadores (aspas).

A única tecla que finaliza uma entrada via ACCEPT é o <Enter> ou <Return>. Caso não seja digitado nenhum dado, um valor nulo ("") será atribuído à variável.
O tamanho máximo de uma string que pode ser entrada via ACCEPT para uma variável é de 254 caracteres.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
ACCEPT "Digite o seu Nome:" TO nome
CLEAR
@ 12,20 SAY "Seu Nome é:" +nome
```

* O nome deve ser digitado diretamente, sem a necessidade de estar delimitado por aspas. A variável criada sempre será do tipo caractere.

Veja Também:

INPUT, WAIT, @...SAY...GET, INKEY(0)

Comando CLEAR TYPEAHEAD:

Sintaxe:

CLEAR TYPEAHEAD

Propósito:

Limpar o "buffer" de teclado de qualquer caractere (respostas) que tiverem sido digitadas, baseadas em requisições de entradas anteriores.

Utilização:

Deve ser utilizado quando se deseja que caracteres digitados incorretamente, em geral por impaciência do usuário, sejam eliminados do buffer de teclado para não causarem problemas na entrada seguinte.
CLEAR TYPEAHEAD é últil para garantir que teclas armazenadas no "buffer" de teclado sejam as corretas para a operação seguinte e não teclas pendentes de operações anteriores. Este comando deve ser particularmente utilizado nas funções-de-usuário das funções ACHOISE( ) e DBEDIT( ).

Bibliotecca:

CLIPPER.LIB

Veja Também:

KEYBOARD, SET TYPEAHEAD, ACHOISE( ), DBEDIT( ), LASTKEY( ), NEXTKEY( ).

Comando INPUT:

Sintaxe:

INPUT [<mensagem>] TO <variável>

Propósito:

Solicita a entrada de um dado através do teclado, armazenando numa variável de memória o dado digitado.

Utilização:

O comando INPUT solicita a entrada de um dado ou de uma expressão através do teclado, apresentando uma mensagem na tela. O dado digitado ou o resultado da expressão é armazenado na variável de memória especificada. O tipo da <variável >criada corresponderá ao tipo da expressão ou dado digitado, isto é, será numérica se o dado for um valor, caractere se o dado for uma "string" delimitada por aspas e data se for uma data. A digitação precisa ser finalizada com <Enter>.

A <mensagem >será apresentada na tela antes da solicitação do dado e poderá se uma variável de memória tipo caractere ou uma "string" delimitada por aspas. A entrada digitada poderá ser numérica, alfanumérica (caractere delimitado por aspas), data (utilizando-se a função CTOD( )) ou lógica (.T. ou .F.).

O comando INPUT deve preferivalmente ser utilizado para a entrada de dados numéricos. Para a entrada de dados do tipo caractere (alfanuméricos) recomenda-se o comando ACCEPT e para a entrada de dados do tipo data recomenda-se o comando @...SAY...GET.

Se nenhum dado ou expressão forem digitados a variável de memória especificada no comando INPUT não será criada. A tecla <Esc> não finaliza o comando INPUT, apenas <Enter>.

A digitação de um dado ou expressão inválida causará erro.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
INPUT "Digite o seu Salário: " TO SAL
CLEAR
@ 12,30 SAY "O seu salário é Cz$"
@ 12,49 SAY SAL PICTURE "@E 999,999.99"
```

* Utilizando o conteúdo de uma variável como mensagem

```
MENS="Digite o valor:"
INPUT MENS TO VALOR
CLEAR
@ 12,30 SAY "O valor é igual a:"
@ 12,49 SAY VALOR PICTURE "@E 999,999.99"
```

Veja Também:

ACCEPT, WAIT, @...SAY...GET

Comando MENU TO:

Sintaxe:

MENU TO <variável>

Propósito:

Permite executar a escolha de opções em um menu criado pelo comando @...PROMPT, o qual cria um menu de barras.

Utilização:

O comando MENU TO deve ser sempre utilizado em conjunto com o comando @...PROMPT. Ele é o mecanismo de seleção do sistema de menu de barras luminosas definido pelo Clipper. Antes de executá-lo o menu e as respectivas mensagens associadas às opções devem ser construídas através do comando @...PROMPT...MESSAGE.
Podem ser produzidos menus semelhantes aos do Lotus 1-2-3, SideKick, Open Access etc.

Para selecionar uma opção basta movimentar a barra luminosa através das setas de direção e teclar <Enter> sobre a opção desejada. A cada opção será apresentada a correspondente mensagem associada. Outra forma de selecionar uma opção é através da digitação da primeira letra da opção desejada.
O número da opção escolhida será armazenado na variável numérica definida. Se a <variável> que receberá o resultado da escolha do menu não existir o comando MENU TO a criará como sendo do tipo numérico e posicionará a barra do menu na primeira opção. Se a variável existir, seu conteúdo será utilizado para determinar a opção @ ...PROMPT que deverá ficar ressaltada pela barra luminosa ao se iniciar o comando MENU TO.
Cada opção (PROMPT) é numerada de 1 a no máximo 32 (número máximo de opções por menu). O número correspondente à opção selecionada será armazenado na variável do comando MENU TO. Pressionando-se a tecla <Esc> o comando MENU TO é finalizado sem ter sido feita uma escolha, retornando o valor zero, que será armazenado na variável especificada.
As teclas de movimentação do menu são as seguintes:

Tecla	**Ação**
Seta para Cima	Opção anterior
Seta para Baixo	Próxima opção
Home	Primeira opção
End	Última opção
Seta Esquerda	Opção anterior
Seta Direita	Próxima opção
PgUp	Seleciona a opção atual
PgDw	Seleciona a opção atual
Enter/Return	Seleciona a opção atual
Esc	Aborta o menu, retornando zero
Primeira letra de uma opção	Seleciona a primeira opção que for iniciada pela mesma letra pressionada, retornando sua posição.

O "HELP" (vide Cap. 7) pode ser ativado e irá passar o nome da variável do MENU TO como parâmetro. Por exemplo, se a tecla <F1 >for pressionada para chamar o HELP.prg, o parâmetro passado como variável esperada é o nome da variável criada pelo comando MENU TO.
Menus podem ser encadeados dentro de rotinas do tipo SET KEY (por exemplo dentro de rotinas de help ou dentro de submenus), sem cancelar os GET's pendentes entre um e outro. Recomenda-se, entretanto, a utilização de nomes diferentes para as variáveis de espera de cada um dos menus.

Em uma rotina invocada pelo comando SET KEY, é possível verificar o nome da variável do menu que a chamou através da função READKEY( ). Isso é interessante como informação de "status" ou para determinar uma nova posição da barra luminosa do menu após a execução da rotina chamada.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
CLEAR
* Definição das Bordas para utilização no comando @...BOX
frame1=CHR(218)+CHR(196)+CHR(191)+CHR(179)+CHR(217)+CHR(196)+CHR(192);
       +CHR(179)
frame2=CHR(201)+CHR(205)+CHR(187)+CHR(186)+CHR(188)+CHR(205)+CHR(200);
       +CHR(186)
frame3=frame1+CHR(176)
frame4=frame2+CHR(176)
DO WHILE .T.
   @ 02,00,04,79 BOX frame1
   @ 06,00,23,79 BOX frame4
   mp=1
   SET MESSAGE TO 5
   @ 03,02 PROMPT "Inclusões"    Message "  Inclusões Nomes no Cadastro"
   @ 03,13 PROMPT "Alterações"   Message "  Alterações no Cadastro"
   @ 03,25 PROMPT "Exclusões"    Message "  Exclusões de Nomes"
   @ 03,36 PROMPT "Consultas"    Message "  Consultas na Tela"
   @ 03,47 PROMPT "Relatórios"   Message "  Menu de Relatórios"
   @ 03,59 PROMPT "Utilitários"  Message "  Reorganização e Back Up's"
   @ 03,73 PROMPT " Fim "        Message "  Fim das Operações"
   MENU TO mp
   @ 24,00 CLEAR
   DO CASE
      CASE mp=1
        DO Mala1
      CASE mp=2
        DO Mala2
      CASE mp=3
        DO Mala3
      CASE mp=4
         DO Mala4
      CASE mp=5
         DO Mala5
      CASE mp=6
         DO Mala6
      CASE mp=7
        @ 06,00 CLEAR
        @ 10,00,15,79 BOX frame2
        SET COLOR TO W+*
        @  12,36 SAY "Atenção !"
        SET COLOR TO W+
        @ 13,19 SAY "Não se esqueça de fazer CÓPIAS DE SEGURANÇA"
        SET COLOR TO W
        confirme=" "
```

```
         @ 24,60 SAY "Confirme (S/N)? GET confirme VALID(confirme$"SN")
         READ
         @ 24,00 CLEAR
         IF confirme<>"S"
            LOOP
         Endif
         CLEAR
         @ 8,0,14,79 BOX frame4
         @ 14,72 SAY " Vidal "
         SET COLOR TO W+*/N
         @ 10,30 SAY " Fim das Operações ! "
         SET COLOR TO W+
         @ 12,34 SAY " Obrigado ! "
         SET COLOR TO W
         EXIT
   ENDCASE
ENDDO
CLOSE DATABASE
CLEAR
RETURN
```

A rotina acima irá construir na tela um menu de opções semelhante aos do Sistema MLD do Cap. 10.

Veja Também:

@...PROMPT, SET MESSAGE, SET WRAP, ACHOICE( ), @...SAY...GET

Comando KEYBOARD:

Sintaxe:

KEYBOARD <expressão caractere>

Propósito:

Preenche o "buffer" de entrada do teclado com a expressão caractere (string) fornecida.

Utilização:

O comando KEYBOARD deve ser utilizado em conjunto com outros comandos e funções que solicitam entradas através do teclado, como dados ou teclas de controle. Por exemplo: ACCEPT, INPUT, READ, MENU TO, ACHOICE( ), DBEDIT( ) etc. Através dele pode-se simular o pressionamento das teclas correspondentes à cadeia de caracteres especificada.
O comando KEYBOARD pode ser utilizado, por exemplo, para preencher

o "buffer" do teclado com três respostas a três menus consecutivos, eliminando a necessidade de fornecer as respostas individualmente a cada um dos menus.
Cada execução do comando KEYBOARD limpa o "buffer" de entrada do teclado. Vários comandos KEYBOARD não poderão, entretanto, ser utilizados para criar uma fila de caracteres a serem digitados. Contudo, o comando KEYBOARD "" (cadeia nula) poderá ser utilizado para limpar o "buffer" do teclado.

Biblioteca:

CLIPPER.LIB

Exemplo:

Para retornar a um menu principal de um terceiro submenu (três níveis de profundidade) com o pressionamento de uma única tecla, sendo que a opção "F" e um <Enter> finaliza cada menu, utiliza-se:

```
KEYBOARD "F"+CHR(13)+"F"+CHR(13)+"F"+CHR(13)
```

onde CHR(13) é o código ASCII da tecla <Enter>.
O efeito é como se tivessem sido teclados um F e um <Enter> três vezes consecutivas.

Veja Também:

SET KEY, CHR( ), LASTKEY( ), NEXTKEY( ), CLEAR TYPEAHEAD

Comando READ:

Sintaxe:

READ [SAVE]

Propósito:

Permite a edição de dados em tela-cheia ("full-screen"). O comando READ lê os dados editados pelos comandos @...GET pendentes, gravando-os nas respectivas variáveis ou campos de arquivos. @...GET's pendentes são aqueles existentes após o último comando CLEAR, CLEAR ALL, CLEAR GETS ou READ. O comando READ deve ser sempre utilizado em conjunto com comandos @...GET para ler os dados editados.

Utilização:

O comando READ é usado para entrada ou edição (alteração) de dados em tela-cheia (full-screen), em conjunto com os comandos @...SAY...GET. Sua função é ler e gravar nas respectivas variáveis ou campos os dados digitados ou alterados em cada GET.
O READ posiciona o cursor no início de cada campo de edição (em vídeo reverso) determinado por cada GET, e aguarda até que um <Enter> ou outra tecla de finalização de edição seja digitada. Para controle da edição dos dados são utilizadas as seguintes teclas:

Tecla	Ação
Seta esquerda Ctrl-S	Move o cursor um caractere para esquerda. Não move o cursor para o GET anterior.
Seta direita Ctrl-D	Move o cursor um caractere para a direita. No final de um GET, move o cursor para o próximo GET.
Ctrl-Seta Esquerda Ctrl-A	Move o cursor uma palavra para a esquerda.
Ctrl-Seta Direita Ctrl-F	Move o cursor uma palavra para a direita.
Seta para Cima Ctrl-E	Move o cursor para o GET anterior.
Seta para Baixo Ctrl-X Return/Enter Ctrl-M	Movem o cursor para o próximo GET.
Home End	Move o cursor para o início do GET. Move o cursor para o final do GET.
Ctrl-Home Ctrl-End	Move o cursor para o primeiro GET. Move o cursor para o último GET.
Del Ctrl-G	Elimina o caractere sobre o cursor.

Tecla	Ação
BackSpace Ctrl-H	Elimina o caractere à esquerda do cursor.
Ctrl-T	Elimina a palavra à direita do cursor.
Ctrl-Y	Elimina todos os caracteres da posição do cursor ao final do GET.
Ctrl-U	Recupera o valor original do dado no GET sendo editado.
Ins Ctrl-V	Alterna entre Insere e Sobrescreve.
Ctrl-W Ctrl-C PgUp/PgDn	Finaliza o READ, gravando os GET's editados.
Return/Enter Ctrl-M	Finaliza um GET e finaliza um READ se for o último GET.
Esc	Finaliza um READ sem salvar o GET sendo editado.

Além das teclas de controle de edição os comandos CLEAR, CLEAR GETS ou CLEAR ALL executados a partir de uma rotina invocada pelo comando SET KEY ou uma função-de-usuário chamada por um VALID, também finalizam um comando READ.

Os comandos @...SAY...GET e READ são o único meio de se editar uma variável de memória.

Dentro de um READ é possível, portanto, editar o conteúdo dos comandos @...GET e "navegar" entre eles. Sempre que existir uma cláusula VALID (veja o comando @...SAY...GET) ao se sair de um GET o controle passa para ela. Se a cláusula VALID retornar verdadeiro (.T. — dado válido) o GET é finalizado e o comando READ passa para o seguinte ou é finalizado. Caso contrário, se a cláusula VALID retornar falso (.F. — dado inválido) o GET não é finalizado até que seja digitado um dado válido.

Se existir um arquivo de formato (.fmt) aberto, o comando READ passa o controle para este arquivo, antes de entrar em modo de edição. Isto significa que os dados serão editados de acordo com o formato de tela definido

neste arquivo. Para fechar um arquivo de formato utiliza-se o comando CLOSE FORMAT.
Comandos READ dentro de um arquivo de formato (.fmt) serão ignorados, pois o Clipper, ao contrário do dBASE III Plus, suporta apenas uma página de formatação de tela.
Um READ limpa todos os GET's pendentes após a finalização da entrada ou edição dos dados.

Opções:

A cláusula SAVE apenas está disponível nas versões Summer 87 e posteriores. Ela retém o conjunto atual de GET's apos um comando READ SAVE ser executado, permitindo editar os mesmos GET's apenas através de um outro READ. Se a cláusula SAVE não for especificada (padrão normal), os GET's são limpos.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
USE Mala INDEX Mala
CLEAR
borda1=CHR(218)+CHR(196)+CHR(191)+CHR(179)+CHR(217);
      +CHR(196)+CHR(192)+CHR(179)
@ 03,10,20,70 BOX borda1
DO WHILE .T.
     vnome=SPACE(40)
     vendereco=SPACE(40)
     vcidade=SPACE(20)
     vestado=SPACE(2)
     vcep=SPACE(5)
     @ 05,34 SAY "Mala-Direta"
     @ 07,27 SAY "Entrada/Alteração de Dados"
     @ 08,27 SAY "-------------------------"
     @ 11,15 SAY "Nome....:" GET vnome      PICTURE "@!"
     @ 13,15 SAY "Endereço:" GET vendereco PICTURE "@!"
     @ 15,15 SAY "Cidade..:" GET vcidade   PICTURE "@!"
     @ 17,15 SAY "Estado..:" GET vestado   PICTURE "!!"
     @ 17,30 SAY "Cep:"      GET vcep      PICTURE "99999"
     READ
     IF EMPTY(vnome)
        EXIT
     ENDIF
     APPEND BLANK
     REPLACE nome WITH vnome,endereco WITH vendereco
     REPLACE cidade WITH vcidade,estado WITH vestado
     REPLACE cep WITH vcep
ENDDO
CLEAR
USE
RETURN
```

Veja Também:

@...SAY...GET, CLEAR GETS, CLEAR, CLEAR ALL, SET FORMAT, LASTKEY( ), READKEY( ).

Comando WAIT:

Sintaxe:

WAIT [<mensagem>] [TO <variável>]

Propósito:

O comando WAIT interrompe o processamento até que uma tecla qualquer seja pressionada. A tecla pressionada pode ser armazenada numa variável especificada e uma mensagem pode ser apresentada para orientar o usuário sobre o estado de espera.

Utilização:

O comando WAIT é utilizado para causar uma interrupção no processamento e solicitar uma decisão do usuário através do pressionamento de uma única tecla. O processamento ficará suspenso até que uma tecla qualquer seja pressionada.
Se a cláusula TO <variável> for incluída, o caractere digitado será armazenado na variável indicada e será sempre do tipo caractere. Se um <Enter> ou outro caractere não imprimível for teclado, o conteúdo da variável será o caractere zero (0). Se a variável não existir será automaticamente criada e, se já existir, terá seu conteúdo sobreposto pelo caractere teclado.
A <mensagem> é um texto a ser especificado que será apresentado, ao ser interrompido o processamento, para orientar o usuário quanto ao procedimento a adotar. Poderá ser uma variável caractere ou uma "string" delimitada por aspas. Se não for especificada nenhuma mensagem e o comando SET CONSOLE estiver ligado (ON) será apresentada a mensagem padrão "Press any key to continue..."

Biblioteca:

CLIPPER.LIB

Exemplo:

Para interromper a execução do programa temporariamente pode ser utilizado o comando WAIT:

```
WAIT "Voce deseja continuar? (S/N)" TO SN
IF UPPER (SN)  "S"
   RETURN
ENDIF
```

Veja Também:

ACCEPT, INPUT, @...SAY...GET, INKEY( )

5.5. COMANDOS PARA MANUTENÇÃO DE ARQUIVOS DE DADOS

5.5.1. Comandos Para a Criação de Arquivos de Dados

Comando CREATE:

Sintaxe:

CREATE <nome do arquivo>

Propósito:

Cria um arquivo de definição de estrutura (estrutura estendida) vazio (sem registros), para posterior criação de um arquivo de dados através do comando CREATE FROM.

Utilização:

O comando CREATE é utilizado para criar um arquivo cujos registros descrevem os campos de um outro arquivo de dados a ser criado através do comando CREATE FROM. A este arquivo chamamos de arquivo de estrutura estendida ou arquivo de definição de estrutura.
O <nome do arquivo> é o nome do arquivo (.dbf) de definição de estrutura a ser criado.
O comando CREATE deve ser utilizado para criar ou modificar a estrutura de arquivos de dados sem a necessidade de se utilizar o processo interativo tradicional, através do utilitário CREATE ou DBU do Clipper ou dos comandos CREATE e MODIFY STRUCTURE do dBASE.

Um arquivo de estrutura estendida possui, por definição quatro campos em seus registros:

- Field_name: nome do campo, tipo caractere, até 10 posições.
- Field_type: tipo do campo, tipo caractere, 1 posição.
- Field_length: tamanho do campo, tipo numérico, até 3 dígitos.
- Field_decimals:casas decimais, tipo numérico, até 2 dígitos.

Ao contrário do comando COPY STRUCTURE EXTENDED, o comando CREATE produz um arquivo de definição de estrutura vazio, não requerendo a presença de outro arquivo de dados para criá-lo.
Cada registro do arquivo criado, portanto, deverá receber informações sobre os campos do novo arquivo a ser criado através do comando CREATE FROM. O número de campos do arquivo a ser criado através do CREATE FROM será igual ao número de registros do arquivo criado através do CREATE.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
CREATE estrut
DO WHILE .T.
   CLEAR
   vcampo=SPACE(10)
   vtipo =SPACE(1)
   vcomp =1
   vdec  =0
   @ 08,10 SAY "Nome do Campo.:" GET vcampo PICTURE "@!"
   @ 10,10 SAY "Tipo do Campo.:" GET vtipo  PICTURE "!";
   @ 12,10 SAY "Comprimento...:" GET vcomp  PICTURE "999"
   @ 14,10 SAY "Casas Decimais:" GET vdec   PICTURE "99"
   READ
   IF EMPTY(vcampo)
      USE
      EXIT
   ENDIF
   IF .NOT. vtipo$"CNDLM"
      LOOP
   ENDIF
   IF vcomp<=0 .OR. vcomp>255
      LOOP
   ENDIF
```

```
   IF vdec(0 .OR. vdec)19
      LOOP
   ENDIF
   APPEND BLANK
   REPLACE field_name WITH vcampo,field_type WITH vtipo
   REPLACE field_length WITH vcomp,field_decimals WITH vdec
ENDDO
CREATE novoarq FROM estrut
RETURN
```

* Cria o arquivo "novoarq" a partir da estrutura definida pelo arquivo "estrut" que contém em seus registros as características dos campos do novo arquivo.

Veja Também:

CREATE FROM, COPY STRUCTURE EXTENDED

Comando CREATE FROM:

Sintaxe:

CREATE <nome do arquivo> FROM <arquivo de estrutura>

Propósito:

Cria um arquivo de dados a partir de um arquivo de estrutura estendida ou de definição de estrutura definido pelo comando CREATE.

Utilização:

O comando CREATE FROM deve ser utilizado em conjunto com o comando CREATE, o qual cria um arquivo de estrutura estendida. Usando as definições de campos contidas neste arquivo, o comando CREATE FROM produz um novo arquivo de dados.
Para ser um arquivo de definição de estrutura ou de estrutura estendida, um arquivo de dados deve obrigatoriamente possuir a seguinte estrutura (inclusive com estes nomes de campos):

Campo	Nome	Tipo	Tamanho	Decimais
1	Field_name	Caractere	10	
2	Field_type	Caractere	1	
3	Field_len	Numérico	3	0
4	Field_dec	Numérico	3	0

Dicionário de Dados: para criar um dicionário de dados, o arquivo de definição de estrutura poderá possuir além dos campos acima, outros campos para descrever os dados a serem armazenados no arquivo. Por exemplo, podem ser incluídos campos para descrever atributos como campos-chave, título, descrição geral, formato de PICTURE utilizado, dados válidos, validação de dados através da cláusula VALID etc. Quando o comando CREATE FROM é utilizado para criar o arquivo, apenas os quatro campos de definição de estrutura são utilizados. Todos os outros campos são ignorados. Além disso os campos de definição de estrutura poderão aparecer em qualquer ordem.

Os comandos CREATE e CREATE FROM fornecem independência da estrutura de dados para as aplicações desenvolvidas em Clipper. Através deles um sistema poderá criar seus próprios arquivos de acordo com as necessidades do usuário, fornecendo assim grande flexibilidade e possibilidade de parametrização.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
CREATE estrut
DO WHILE .T.
   CLEAR
   vcampo=SPACE(10)
   vtipo =SPACE(1)
   vcomp =1
   vdec  =0
   @ 08,10 SAY "Nome do Campo.:" GET vcampo PICTURE "@!"
   @ 10,10 SAY "Tipo do Campo.:" GET vtipo  PICTURE "!";
   @ 12,10 SAY "Comprimento...:" GET vcomp  PICTURE "999"
   @ 14,10 SAY "Casas Decimais:" GET vdec   PICTURE "99"
   READ
   IF EMPTY(vcampo)
      USE
      EXIT
   ENDIF
   IF .NOT. vtipo$"CNDLM"
      LOOP
   ENDIF
   IF vcomp<=0 .OR. vcomp>255
      LOOP
   ENDIF
   IF vdec<0 .OR. vdec>19
      LOOP
   ENDIF
   APPEND BLANK
   REPLACE field_name WITH vcampo,field_type WITH vtipo
   REPLACE field_length WITH vcomp,field_decimals WITH vdec
ENDDO
CREATE novoarq FROM estrut
RETURN
```

* Cria o arquivo "novoarq" a partir da estrutura definida pelo arquivo "estrut" que contém em seus registros as características dos campos do novo arquivo.

Veja Também:

CREATE, COPY STRUCTURE EXTENDED, os utilitários CREATE.exe e DBU.exe.

Comando COPY STRUCTURE TO:

Sintaxe:

COPY STRUCTURE TO <nome do novo arquivo>
[FIELDS <lista de campos>]

Propósito:

Criar uma estrutura de arquivo de dados vazia, apenas com a definição dos campos dos registros, a partir da estrutura do arquivo de dados que estiver em uso.

Utilização:

Este comando é utilizado para criar, a partir de um arquivo de dados já existente, um outro arquivo de estrutura total ou parcialmente idêntica, sem contudo, incluir os dados do arquivo original.
O <nome do novo arquivo> especifica o nome do arquivo vazio a ser criado. A extensão (.dbf) será assumida como padrão, a menos que outra seja especificada.
Toda a estrutura do arquivo em uso é copiada, a menos que uma <lista de campos>, com os nomes dos campos que deverão ser copiados separados por vírgulas, seja fornecida através da cláusula FIELDS.
Este comando permite que o próprio sistema atualize seus arquivos, principalmente no caso de processamentos periódicos. Por exemplo, ao final de um período de processamento, por exemplo um mês, o arquivo do mês anterior deixa de ser utilizado e um novo arquivo, criado a partir da estrutura do anterior, passa a ser utilizado.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
USE movfev && arquivo do movimento de fevereiro
COPY STRUCTURE TO movmar && arq.do movimento de março
USE

USE arq1
COPY STRUCTURE TO arq2 FIELD numero,nome,salario
USE
* Apenas os campos número, nome e salário formarão a estrutura do novo
arquivo "arq2".
```

Veja Também:

COPY STRUCTURE EXTENDED, COPY TO, CREATE

Comando COPY STRUCTURE EXTENDED:

Sintaxe:

COPY TO <nome do arquivo> STRUCTURE EXTENDED
[FIELDS <lista de campos>]

Propósito:

Cria um arquivo de dados que contêm em seus registros, a descrição da estrutura do arquivo de dados em uso (arquivo de definição de estrutura ou de estrutura estendida).

Utilização:

O comando COPY STRUCTURE EXTENDED cria um novo arquivo de dados de definição de estrutura, com os seguintes quatro campos:

Campo	Nome	Tipo	Tamanho	Decimais
1	Field_name	Caractere	10	
2	Field_type	Caractere	1	
3	Field_len	Numérico	3	0
4	Field_dec	Numérico	3	0

O conteúdo do novo arquivo, especificado através do <nome do arquivo>, consistirá de um registro para cada campo do arquivo de origem, sendo que

cada registro conterá o nome de cada campo e suas respectivas características.
A cláusula FIELDS permite que se especifique um ou mais campos do arquivo em uso que deverão ter sua estrutura copiada como registros do arquivo de definição de estrutura a ser criado. Se não for especificada todos os campos terão sua estrutura copiada para o novo arquivo.
Este comando é usado para a criação ou modificação da estrutura de arquivos de dados dentro de um programa de aplicação escrito em Clipper, evitando-se a utilização dos utilitários DBU.exe, CREATE.exe ou do próprio dBASE para criar ou modificar a estrutura de arquivos.
O comando CREATE FROM deve ser utilizado em conjunto, para criar um novo arquivo de dados a partir do arquivo de estrutura estendida criado através do comando COPY STRUCTURE EXTENDED. Para criar um arquivo de definição de estrutura vazio, utiliza-se o comando CREATE.
Através dos comandos CREATE, CREATE FROM, COPY STRUCTURE EXTENDED e APPEND FROM, é possível desenvolver rotinas em Clipper para a criação e alteração da estrutura de arquivo de dados.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
USE Arquivo
COPY TO Arqestru STRUCTURE EXTENDED
USE Arqestru
LIST field_name, field_type, field_len, field_dec
```

Resultará:

```
1    NOME        C    30  0
2    ENDERECO    C    40  0
3    CIDADE      C    20  0
4    TELEF       C    8   0
5    CEP         C    5   0
6    ANIVER      D    8   0
7    VALOR       N    12  2
```

Veja Também:

CREATE, CREATE FROM, APPEND FORM, FIELD( ), TYPE( )

Comando COPY TO:

Sintaxe:

COPY TO **<nome do arquivo cópia>**
[<escopo>]
[FIELDS <lista de campos>]
[FOR <condição>]
[WHILE <condição>]
[SDF/DELIMITED [WITH <delimitador>]]

Propósito:

O comando COPY TO duplica todo ou parte do arquivo de dados ativo (em uso) para um novo arquivo. Este comando é também o principal comando para a geração de arquivos de dados de outros formatos que não os utilizados pelo Clipper.

Utilização:

O comando COPY TO copia para um novo arquivo, especificado em <nome do arquivo cópia>, todo ou parte do arquivo que estiver em uso.
Através do comando COPY TO, todos os registros, incluindo aqueles marcados para deleção, são copiados para um novo arquivo, a menos que seja especificado um escopo ou condições através das cláusulas FOR e WHILE, ou se o comando SET DELETED estiver ligado (ON). Se um filtro estiver ativo, através do comando SET FILTER, apenas os registros que passaram pelo filtro serão copiados para o novo arquivo.
Todos os campos do arquivo também são copiados, a menos que sejam selecionados apenas alguns através da cláusula FIELDS. A lista de campos deve conter o nome dos campos a serem copiados separados por vírgulas.
O <escopo> corresponde ao número de registro (ALL todos, NEXT <n> os próximos n, RECORD <n> apenas o registro n) que deverão ser copiados para o novo arquivo. Se não for especificado será assumido ALL, ou seja, todos os registros serão copiados.
A cláusula FOR especifica uma condição para selecionar os registros que deverão ser copiados dentro do escopo especificado. Apenas os registros que a satisfizerem (a tornarem verdadeira) serão copiados.
A cláusula WHILE especifica os registros que deverão ser copiados, a partir do registro onde estiver posicionado o apontador de registros, enquanto uma condição permanecer verdadeira.
Se o tipo de arquivo destino não for especificado, a extensão padrão (.dbf) é assumida para o novo arquivo. Se for especificado o tipo SDF ou DELIMI-

TED e não for especificada uma extensão para o novo arquivo, será assumida a extensão (.txt).
Campos MEMO não serão copiados se forem utilizadas as opções SDF ou DELIMITED.
Quando o comando COPY TO for utilizado em um ambiente multiusuário de rede, o Clipper irá abrir o arquivo (.dbf) destino como exclusivo (EXCLUSIVE), ou seja, de acesso restrito.

Opções:

- **DELIMITED [WITH <delimitador>]**: Copia os dados do arquivo em uso para um arquivo com "formato delimitado" tipo ASCII. Os dados são copiados caractere por caractere, iniciando da esquerda. Cada registro terá um tamanho variável e terminará com um "return" e um avanço de linha (carriage return/line feed), ou seja, será gravado um registro por linha. Cada campo terá um tamanho variável e será separado do seguinte por uma vírgula. A menos que seja especificado um delimitador, aspas ("") delimitarão os campos com dados do tipo caractere. Os espaços em branco à esquerda ou à direita dos campos serão eliminados. Campos tipo data serão convertidos para o formato "aaaammdd" ou seja "anomesdia" e os campos lógicos serão convertidos para T (verdadeiro) ou F (falso). DELIMITED WITH <delimitador>, permite que os campos caractere sejam delimitados por qualquer outro caractere desejado que não as aspas (por exemplo: apóstrofos ').
  A extensão padrão dos arquivos copiados com esta opção é (.txt) e um Ctrl-Z marcará o fim do novo arquivo criado.

- **SDF (System Data Format)**: Arquivo tipo ASCII. Os dados são copiados caractere por caractere, iniciando-se pela esquerda. Cada registro possui um comprimento fixo (que corresponde à soma do tamanho de todos os campos a serem copiados), sendo finalizados com um "carriage return/line feed", ou seja, será gravado um registro por linha. Os campos possuirão o mesmo tamanho que no arquivo (.dbf) original e não serão separados por nenhum caractere especial. Os brancos à esquerda e à direita dos campos serão mantidos. Os campos data serão convertidos para o formato "aaaammdd", ou seja "anomesdia", e os campos lógicos para T (verdadeiro) ou F (falso). A extensão padrão dos arquivos copiados com esta opção é (.txt) e seu final será marcado por um Ctrl-Z (marca padrão de fim-de-arquivo).

Biblioteca:

CLIPPER.LIB

Exemplos:

```
USE arq1
COPY TO arq2 FIELDS numero,nome,salario FOR salario > 10000
USE arq2
DISPLAY ALL

Registro  Número  Nome                  Salário
1         12323   Cecília Oliveira Silva 23000.00
2         23453   Ana Maria Sobral       14000.00
3         23432   Lígia Marcondes        19000.00
4         67567   Maria Teresa Vidal     36000.00
USE

USE Arq1
COPY TO Arq2 FIELDS numero,nome,salario,data DELIMITED
USE
TYPE Arq2.txt

"12323","Cecília Oliveira Silva","23000.00"
"23453","Ana Maria Sobral","14000.00"
"23432","Lígia Marcondes","19000.00"
"67567","Maria Teresa Vidal","36000.00"

USE Arq
COPY TO arqnovo FIELDS numero,nome,salario SDF
USE
TYPE Arq2.txt

12323Cecilia Oliveira Silva      23000.00
23453Ana Maria Sobral            14000.00
23432Lígia Marcondes             19000.00
67567Maria Teresa Vidal          36000.00
```

Veja Também:

APPEND FROM, COPY FILE, COPY STRUCTURE, SET DELETED, SET FILTER

Comando JOIN

Sintaxe:

JOIN WITH <alias> TO <novo arquivo de dados>
FOR <condição>
[FIELDS <lista de campos>]

Propósito:

Cria um novo arquivo de dados a partir de outros dois arquivos (.dbf) abertos em áreas de trabalho diferentes, fazendo um "merge" (união) de registros e campos através das condições especificadas.

Utilização:

O comando JOIN permite a união seletiva de dois arquivos de dados abertos em áreas de trabalho diferentes criando um terceiro, de acordo com a condição geral especificada pela cláusula FOR. O arquivo gerado possuirá registros e campos vindos dos dois arquivos originais envolvidos na operação.
Quando unindo o arquivo da área de trabalho atualmente selecionada com outro arquivo aberto em outra área de trabalho, o segundo arquivo deve ser identificado pelo seu <alias>. Este <alias> que identifica o arquivo a ser unido com o aberto na área de trabalho atual poderá ser o próprio nome do arquivo.
O <novo arquivo de dados> especifica o nome do arquivo a ser criado a partir da união do arquivo aberto na área de trabalho atual com o arquivo cujo <alias> foi especificado.
A <lista de campos> da cláusula FIELDS pode conter campos de ambos os arquivos de dados participantes do JOIN (união). Se não for especificada, todos os campos do arquivo aberto na área atualmente selecionada serão incluídos no arquivo gerado. Para especificar qualquer campo do arquivo de dados da outra área de trabalho que participa do JOIN deve ser usado seu alias através da seguinte sintaxe: alias→<nome do arquivo>. Obviamente não poderão aparecer campos duplicados no novo arquivo gerado.
A cláusula FOR especifica uma <condição> para selecionar os registros que deverão ser processados. Apenas os registros do arquivo secundário que a satisfizerem (a tornarem verdadeira) serão gravados no novo arquivo gerado.

Funcionamento:

O ponteiro de registros (registro atual) é posicionado no primeiro registro do arquivo selecionado na área de trabalho atual. Cada registro do segundo arquivo é avaliado de acordo com a condição especificada através da cláusula FOR <condição>. Se a condição especificada for verdadeira, um registro é

adicionado no novo arquivo. Quando todos os registros do segundo arquivo forem percorridos, o ponteiro de registros do arquivo selecionado avança para o próximo registro, e o processo é repetido. O ciclo continua até que todos os registros do arquivo selecionado na área de trabalho ativa tenham sido processados. Este processo é, em geral, demorado para grandes arquivos. O número de registros processados será igual ao número de registros do arquivo aberto na área de trabalho selecionada multiplicado pelo número de registros do arquivo aberto na outra área de trabalho participante do JOIN.

CUIDADOS: Devem ser tomados alguns cuidados na utilização do comando JOIN:

- Se dois arquivos de 1000 registros cada um forem unidos através do JOIN, o arquivo resultante poderá conter até um milhão de registros, se a condição especificada for verdadeira para todos os registros de ambos os arquivos. Isso além de consumir um tempo incrível poderá exceder o espaço disponível no disco do seu equipamento.
- Se os dois arquivos possuírem um nome de campo em comum, e é necessário especificar o campo do arquivo da área não selecionada, deve ser utilizado o alias→nome do campo (ex. arq2→nome) para diferenciá-los. Use somente o nome do campo para especificar o campo da área selecionada e o alias→nome do campo para especificar o campo do arquivo da outra área.

Biblioteca:

CLIPPER.LIB

Exemplos:

MATERIA.dbf

Número	Disciplina	Nota
123352	EAD-204	9.0
123352	EAD-213	7,5
145325	EAC-205	6,3
145325	EAD-204	5,2
223456	EAD-104	8,4
223456	EAD-102	4,2
123352	EAD-210	8,2

ALUNOS.dbf

Número	Nome	Endereço
123352	Hugo Oliveira Silva	Rua xxx, 111
223456	Ricardo Sobral	Rua zzz, 333
145325	Wilson Matar	Rua www, 123

```
CLEAR
SELECT 2
USE ALUNOS ALIAS Alu
SELECT 1
USE MATERIA ALIAS Nota
JOIN WITH Alu TO APROVADO FOR numero = alu->numero FIELDS;
numero, alu->nome, disciplina, nota
CLOSE DATABASE
USE APROVADO
DISPLAY ALL numero,nome,disciplina,nota
USE
```

APROVADO.dbf

Número	Nome	Disciplina	Nota
123352	Hugo de Oliveira Silva	EAD-204	9.0
123352	Hugo de Oliveira Silva	EAD-213	7,5
145325	Wilson Matar	EAC-205	6,3
145325	Wilson Matar	EAD-204	5,2
223456	Ricardo Sobral	EAD-104	8,4
223456	Ricardo Sobral	EAD-102	4,2
123352	Hugo de Oliveira Silva	EAD-210	8,2

Veja Também:

APPEND FROM, REPLACE, SET RELATION

Comando TOTAL TO:

Sintaxe:

TOTAL TO **<arquivo> ON <expressão chave>**
[<escopo>]
FIELDS <lista de campos>
[FOR <condição>]
[WHILE <condição>]

Propósito:

Cria um novo arquivo sumarizado a partir de um primeiro arquivo em uso, levando em consideração o agrupamento por uma determinada chave que se repete no arquivo de origem. O arquivo de origem deve estar ordenado por este campo chave.

O comando TOTAL soma os campos numéricos do arquivo de dados em uso para um segundo arquivo. Os campos numéricos deste arquivo conterão os totais de todos os registros que possuem a mesma chave no arquivo original.

Utilização:

Este comando é útil para a agregação de dados numéricos de um arquivo detalhado, gerando um arquivo consolidado.
O <arquivo> especifica o nome do arquivo de dados a ser criado para receber os registros totalizados. A menos que seja especificada outra extensão, será assumida a extensão padrão (.dbf).
A <expressão chave> define a chave do grupo de registros do arquivo em uso que será totalizado para produzir o novo arquivo consolidado. Para que o resultado seja correto, o arquivo de dados a ser totalizado deverá estar indexado ou classificado através da expressão chave especificada.
A cláusula FIELDS especifica uma lista de campos numéricos do arquivo a serem totalizados. Se não for especificada, a totalização não será executada pelo Clipper, sendo copiados para o novo arquivo o primeiro registro de cada valor diferente da expressão chave especificada. Com isto se tem uma alternativa interessante para eliminar registros com chaves duplicadas de um arquivo original, criando um novo onde estas chaves sejam únicas.
O <escopo> define a quantidade de registros a ser processada: ALL (todos) ou NEXT <n> (os próximo n registros a partir do atual). Se não for especificado serão totalizados todos os registros do arquivo em uso.
A cláusula FOR especifica uma condição para selecionar os registros que deverão ser processados. Apenas os registros a satisfazerem (a tornarem verdadeira) serão totalizados e copiados para o novo arquivo.
A cláusula WHILE seleciona os registros a serem processados, a partir do registro atual, enquanto uma condição especificada for satisfeita (permanecer verdadeira).
Todos os registros que possuírem o mesmo valor (conteúdo) da chave formarão um único registro no arquivo totalizado. Todos os seus campos numéricos que aparecerem na lista de campos determinada pela cláusula FIELDS conterão totais. Os outros campos conterão os dados do primeiro registro do arquivo original de cada grupo de mesma chave.
Se o tamanho dos campos numéricos do arquivo original não for suficiente para conter os totais no arquivo totalizado, serão colocados asteriscos no conteúdo do campo, indicando que ocorreu "numeric overflow" e o Clipper apresentará uma mensagem de erro no topo da tela. Para evitar isto deve-se aumentar o tamanho dos campos que receberão os totais, de forma que possam comportar o maior total esperado.
A estrutura do novo arquivo totalizado será idêntica à do arquivo original, exceto que se este arquivo possuir campos memo, estes não serão copiados para o novo arquivo.

Biblioteca:

CLIPPER.LIB

Exemplos:

DETALHE.dbf

Produto	Entrada	Saída
SORVETE	100	
CHOCOLATE	1000	
SORVETE		50
SORVETE	40	
AÇUCAR	300	
CHOCOLATE		700
AÇUCAR		200

```
CLEAR
USE DETALHE
INDEX ON produto TO INDPROD
TOTAL TO SALDOS ON produto
USE SALDOS
DISPLAY ALL
USE
```

SALDOS.dbf

Produto	Entrada	Saída
AÇUCAR	300	200
CHOCOLATE	1000	700
SORVETE	140	50

Comando UPDATE:

Sintaxe:

UPDATE ON **<expressão chave> FROM <alias>**
REPLACE <campo1> WITH <expressão 1>
[,<campo2> WITH <expressão 2>...]
[RANDOM]

Propósito:

Atualiza o arquivo de dados em uso a partir de um outro arquivo de dados, baseado no casamento um-a-um ou um-para-vários.
São usados dados de um arquivo existente (arquivo fonte) como base para alterações nos registros de um arquivo em uso. As mudanças são feitas através do emparelhamento (casamento) dos registros dos dois arquivos de dados que possuem uma mesma expressão chave.

Utilização:

O comando UPDATE permite a atualização de um arquivo de dados a partir dos dados (mais atuais) de outro arquivo (arquivo fonte), baseando-se numa expressão chave, comum a ambos, que determinará o "casamento" dos registros a serem atualizados.
O arquivo de dados que está sendo atualizado ("updated") deve estar aberto na área de trabalho atualmente selecionada; o arquivo de dados que servirá para a atualização deverá estar aberto em outra área de trabalho.
A <expressão chave> é a expressão a ser utilizada para definir o casamento dos registros dos dois arquivos na área de trabalho selecionada, a partir da área de trabalho onde está aberto o arquivo fonte.
O <alias> é o nome do alias do arquivo aberto na área de trabalho fonte, a ser utilizado para atualizar o arquivo aberto na área de trabalho corrente.
<campo 1>, <campo 2>, ... , <campo n> são os campos do arquivo a ser atualizado na área de trabalho selecionada, que receberão novos valores a partir do arquivo fonte.
<expressão 1>, <expressão 2>, ... , <expressão n> definem os valores a serem substituídos no arquivo a ser atualizado. Se uma das expressões envolver um campo do arquivo fonte de atualização (o que normalmente ocorre), o campo em questão deverá ser identificado pelo alias deste arquivo:

alias→nome do campo

A opção RANDOM permite que apenas o arquivo a ser atualizado esteja indexado ou classificado sobre o campo-chave da atualização; o arquivo fonte de atualização (FROM) pode estar em qualquer ordem.
Se a opção RANDOM não for utilizada, ambos os arquivos devem estar ou indexados ou classificados (sorteados) de acordo com a expressão chave especificada para a atualização.
O comando UPDATE suporta atualizações um-a-um ou um-para-vários entre os registros do arquivo a ser atualizado e do arquivo fonte respectivamente. Ele não suporta, entretanto, atualizações do tipo vários-para-um ou vários-para-vários. Todos os registros do arquivo fonte que coincidem com a chave especificada atualizam apenas o primeiro registro do arquivo a ser atualizado que possui aquela chave, se existirem outros, estes não serão atualizados.
Registros marcados para eliminação (deletados) pelo comando DELETE são normalmente processados se o comando SET DELETED estiver desligado (OFF). Se o SET DELETED estiver ON (ligado) todos os registros do arquivo a ser atualizado, incluindo os deletados, serão processados; contudo, os registros deletados do arquivo fonte serão ignorados.
Em ambiente multiusuário, o arquivo a ser atualizado deverá estar com acesso restrito através da função FLOCK( ) ou aberto com exclusividade através do comando USE...EXCLUSIVE. O arquivo fonte pode indiferentemente estar com acesso restrito ou compartilhado.

Biblioteca:

CLIPPER.LIB

Exemplo:

Atualização do arquivo ESTOQUE.dbf, a partir dos arquivos ENTRADAS.dbf e SAIDAS.dbf.

```
ENTRADAS.dbf

Produto         Entrada

SORVETE            1000
AÇUCAR               90
CHOCOLATE           300

SAÍDAS.dbf

Produto          Saída

SORVETE             400
AÇUCAR               30
CHOCOLATE           100

ESTOQUE.dbf (desatualizado)

Produto         Entrada         Saída

AÇUCAR              300           200
CHOCOLATE          1000           700
SORVETE             140            50
```

```
CLEAR
SELECT 3
USE SAIDAS ALIAS Sai
INDEX ON produto TO SAIPROD
SELECT 2
USE ENTRADAS ALIAS Entra
INDEX ON produto TO ENTPROD
SELECT 1
USE ESTOQUE Alias Est
INDEX ON produto TO ESTPROD
UPDATE ON produto FROM Entra REPLACE entrada WITH;
                        entrada + entra->entrada
UPDATE  ON  produto  FROM Sai REPLACE saida WITH;
                            saida + Sai->saida
DISPLAY ALL
CLOSE DATABASE
```

```
ESTOQUE.dbf (atualizado)

Produto         Entrada         Saída

AÇUCAR              390           230
CHOCOLATE          1300           800
SORVETE            1140           450
```

Veja Também:

APPEND, REPLACE, SET UNIQUE, TOTAL

5.5.2. Comandos Para Abertura ou Fechamento de Arquivos

Comando CLEAR ALL:

Sintaxe:

CLEAR ALL

Propósito:

O comando CLEAR ALL (limpa tudo) fecha todos os arquivos de dados (e seus respectivos arquivos memo, de índices e formatação) abertos, cancela todas as variáveis de memória e seleciona a área de trabalho número 1.

Utilização:

Utiliza-se o comando CLEAR ALL para reinicializar todo o ambiente de trabalho. Ao fechar os arquivos de dados, também serão fechados os arquivos associados de campos memo (.dbt) de índice (.ntx) e de formatação de telas (.fmt).

Veja Também:

CLOSE, CLEAR MEMORY, RELEASE

Comando CLOSE:

Sintaxe:

CLOSE ALL
ALTERNATE
DATABASES
FORMAT
INDEX

Propósito:

O comando CLOSE é utilizado para fechar (desativar) todos os arquivos que estiverem abertos ou arquivos específicos de diversos tipos: alternativos ("alternate files" .TXT), de dados ("database files" .DBF), de formato ("format files" .FMT) e de índices ("index files" .NTX).

Utilização:

CLOSE ALL fecha todos os arquivos de todos os tipos que estiverem abertos (ativos) em todas as áreas de trabalho, não afetando, contudo, as variáveis de memória definidas. Também são cancelados todos os filtros, estabelecidos pelo comando SET FILTER e todas as relações estabelecidas pelo comando SET RELATION.
Para fechar todos os arquivos de um determinado tipo especifica-se em seguida ao comando CLOSE o nome do tipo dos arquivos a serem fechados:

CLOSE ALTERNATE: fecha o arquivo alternativo ("alternate file" .TXT), definido pelo comando SET ALTERNATE TO, que estiver aberto. SET ALTERNATE TO sem nenhum argumento também fecha o arquivo alternativo aberto.
CLOSE DATABASES: fecha todos os arquivos de dados (.DBF) e os índices a eles associados (.NTX) que estiverem abertos em todas as áreas de trabalho selecionadas, cancelando todos os filtros estabelecidos pelo comando SET FILTER TO e todos os relacionamentos estabelecidos pelo comando SET RELATION TO.
CLOSE FORMAT: fecha o arquivo de formato ("format file" .FMT), aberto pelo comando SET FORMAT TO, na área de trabalho que estiver selecionada. SET FORMAT TO sem nenhum argumento também fecha o arquivo de formato aberto.
CLOSE INDEXES: fecha todos os arquivos de índice (.NTX) que estiverem abertos na área de trabalho atualmente selecionada.

Para fechar apenas o arquivo de dados e os índices a ele associados, abertos na área de trabalho atualmente selecionada, utilize o comando USE, ao invés do comando CLOSE.
Além do comando CLOSE, os seguintes outros comandos do CLIPPER também fecham arquivos:

QUIT
CANCEL
RETURN (no programa principal – primeiro nível)
CLEAR ALL
USE (sem argumentos)

Ao ocorrer algum erro durante a execução do programa, o Clipper também poderá fechar os arquivos, de qualquer tipo, que estiverem abertos. Neste caso são apresentadas mensagens no topo da tela e os arquivos serão ou não fechados, de acordo com a resposta fornecida pelo usuário:

- (Q/A/I)?: Q de Quit fecha os arquivos, retornando ao Sistema Operacional;
- Continue?: N de Não (ou No) fecha os arquivos, retornando ao Sistema Operacional.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
CLEAR
USE Arq1 INDEX Ind1,Ind2,Ind3
SET ALTERNATE TO Alter1
SET FORMAT TO Forma1
      .
      .
      .
CLOSE ALL  && fecha todos os arquivos abertos
CLEAR
RETURN
```

Veja Também:

CANCEL, CLEAR ALL, CLEAR MEMORY, QUIT, RETURN, SET ALTERNATE TO, SET FORMAT TO, SET INDEX TO, USE

Comando COMMIT:

Sintaxe:

COMMIT

Propósito:

Grava no disco todos os dados que estiverem nos "buffers" de memória de todas as áreas de trabalho selecionadas.
Este comando está disponível apenas a partir da versão Summer 87 do Clipper.

Utilização:

O comando COMMIT requer o DOS 3.3 ou versão superior. Deve ser utilizado para limpar todos os "buffers" de trabalho do Clipper gravando-os, através do DOS no disco.
É útil para evitar que os dados contidos nos "buffers" de trabalho sejam perdidos caso ocorra falta de energia elétrica ou outro acidente do tipo.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
CLEAR
USE Arquivo
FOR i=1 TO 30
     APPEND BLANK
     REPLACE numero WITH i
     COMMIT
     * direciona para o DOS e grava no disco
NEXT
RETURN
```

Veja Também:

SKIP, @...GET, READ, REPLACE

Comando SELECT:

Sintaxe:

SELECT <área de trabalho / <alias> / (<exp.numérica>)

Propósito:

Selecionar áreas de trabalho com arquivos de dados.

Utilização:

Uma <área de trabalho> é um número entre 1 (um) e 10 (dez) nas versões do Clipper anteriores à Summer 87, e entre 0 (zero) e 254 nela e posteriores.
A área de trabalho é um determinado espaço na memória do computador

onde o Clipper abre, mantém e manipula um arquivo de dados e os índices a ele associados. Em cada área de trabalho pode ser aberto um arquivo de dados e até 15 arquivos de índice (sete para as versões anteriores à Summer 87). Todas as informações referentes ao arquivo (como nome do alias, características da estrutura, chaves dos índices, filtros, relacionamentos etc.) são "trabalhadas" nessa área. O comando SELECT permite selecionar uma área de trabalho onde um determinado arquivo poderá ser aberto e "trabalhado". Se um novo arquivo for aberto em uma área de trabalho já selecionada, o arquivo anterior e seus índices serão automaticamente fechados. Para evitar a operação de abertura e fechamento de arquivos, que em geral é demorada, quando há necessidade de se utilizar vários arquivos simultaneamente deve-se abri-los um em cada área de trabalho que estiver disponível.

Ao se iniciar um programa com o Clipper, a área de trabalho ativa é a área de trabalho 1, também identificada por A. As dez primeiras áreas de trabalho (válidas para todas as versões do Clipper) são identificadas com os números de 1 a 10, ou com as letras de A a J. Se for especificada uma área de trabalho não existente ocorrerá erro.

O <alias> é o nome de uma área de trabalho já selecionada, idêntico ao nome do "alias" do arquivo nela aberto (veja o comando USE).

A (<expressão numérica>) (válida apenas para as versões Summer 87 e posteriores) é uma expressão que resulta um número entre 0 e 254. A expressão deve obrigatoriamente ser colocada entre parênteses para ser corretamente avaliada.

Ao usar uma variável para identificar a área de trabalho ou o nome do alias, a variável deve ser avaliada por "macro-substituição" (veja a função &). Por exemplo, se o nome do alias — Saída — for armazenado em uma variável de memória — var —, use o comando SELECT com a função macro:

```
var="Saida"
SELECT &var
```

Normalmente, comandos que alteram o conteúdo de campos de arquivos afetam apenas o arquivo aberto na área de trabalho atual. Para ler dados de um registro de um arquivo aberto em qualquer outra área de trabalho, que não a selecionada, deve ser utilizada a seguinte sintaxe:

```
<alias> → <nome do campo>
```

Por exemplo:

```
SELECT 1                    && Seleciona a área de trabalho 1
USE SAIDAS ALIAS Sai  && Abre nela o arquivo Saidas
SELECT 2                    && Seleciona a área de trabalho 2
USE ENTRADAS ALIAS Entra && Abre o arquivo Entradas
? produto,entrada,Sai→saida
```

```
* apresenta os campos produto e entrada do arquivo
* Entradas e o campo saida do arquivo Saidas
CLOSE DATABASE && Fecha ambos os arquivos
```

Um arquivo de formato ("format file" .fmt) pode ser aberto para cada área de trabalho selecionada, contudo, o mesmo arquivo não pode ser aberto para duas ou mais áreas simultaneamente.
Cada área de trabalho mantém um apontador de registros independente. Movê-los entre áreas de trabalho diferentes não afetará a posição do apontador de registros de cada área. Comandos que afetam a posição do apontador de registro (SKIP, GOTO, GO TOP, GO BOTTOM etc.) de um arquivo selecionado em uma determinada área de trabalho não afetam a posição do apontador de registros dos arquivos selecionados nas outras áreas, a menos que o comando SET RELATION TO esteja ativo.
Nas versões 3.2 e anteriores do DOS e nas versões do Clipper anteriores à Summer 87, o número máximo de arquivos de qualquer tipo (.dbf, .dbt, .ntx, .fmt etc.) que podem permanecer abertos é 15, mesmo que não tenham sido selecionadas todas as áreas de trabalho. Se todas as 10 áreas não puderem ser selecionadas e o número de arquivos abertos não for superior a 15, verifique o arquivo CONFIG.SYS, que deverá existir no diretório raiz do disco que carregou o Sistema Operacional (disco de boot). Este arquivo deverá conter as seguintes linhas:

```
FILES=20
BUFFERS=8
```

Refira-se ao manual do Sistema Operacional do seu equipamento para maiores esclarecimentos sobre o arquivo CONFIG.SYS e veja a seção 8.3.1 — Configurando o DOS.
Na versão Summer 87 e posteriores, utilizando-se o DOS 3.3 ou posterior, podem ser abertos simultaneamente até 255 arquivos de qualquer tipo. Para tanto o arquivo CONFIG.SYS deverá ser corretamente especificado. Veja a seção 8.3 — Executando os Programas Programas para mais detalhes.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
SELECT 1                              - Area 1
USE Pessoal INDEX Alfa ALIAS Pess     - Aberto arquivo PESSOAL

SELECT 2                              - Area 2
USE Estoque INDEX Codmat ALIAS Est    - Aberto arquivo ESTOQUE

SELECT 3                              - Area 3
USE Telefone INDEX Alfatel            - Aberto arquivo TELEFONE
```

Obs: Deve-se notar que ao abrirmos áreas de trabalho diferentes com arquivos diferentes, para processarmos os dados de um determinado arquivo devemos selecionar a área de trabalho na qual o mesmo foi aberto através do comando SELECT <área/alias>.

```
SELECT Pess     && seleciona-se a área pelo ALIAS do arquivo.
DISPLAY ALL

SELECT Est
LIST TO PRINT

SELECT 3
LIST nome,telefone TO PRINT
```

Veja Também:

USE, SET INDEX, SET RELATION, ALIAS( ), SELECT( )

Comando UNLOCK:

Sintaxe

UNLOCK [ALL]

Propósito:

Cancelar a restrição de acesso a um arquivo ou a um registro determinada pelo usuário atual.

Utilização:

O comando UNLOCK é específico para ambiente multiusuário em rede. Ele cancela as restrições de acesso a arquivos ou registros feitas pelo usuário atual.
A cláusula ALL cancela todas as restrições de acesso em todas as áreas de trabalho selecionadas. Se não for especificada apenas as restrições de acesso definidas na área atualmente selecionada, serão canceladas.

Biblioteca:

CLIPPER.LIB

Veja Também:

SET EXCLUSIVE, USE...EXCLUSIVE, FLOCK( ), RLOCK( )

Comando USE:

Sintaxe:

USE [<**arquivo**>]
[**INDEX** <**lista de arquivos**>]
[**EXCLUSIVE**]
[**ALIAS** <**alias**>]

Propósito:

O comando USE abre (ativa) um arquivo de dados existente e, opcionalmente até sete índices a ele associados (15 a partir da versão Summer 87), na área de trabalho que estiver selecionada. Se o arquivo de dados contém campos memo, o arquivo Memo (.dbt) associado é aberto automaticamente.
USE sem nenhum parâmetro especificado fecha o arquivo de dados e os índices abertos na área de trabalho selecionada.

Utilização:

Para ter acesso aos dados de um arquivo de dados (.dbf) é necessário que o mesmo seja aberto através do comando USE. Este comando tem a função de abrir e fechar arquivos de dados e opcionalmente os índices a ele associados. No Clipper, "**abrir**" um arquivo significa ler parte do arquivo do disco e carregá-la numa área da memória do computador ("buffer" de trabalho), criando-se assim um acesso a seus dados. "**Fechar**" um arquivo significa descarregar da memória o arquivo (limpar o "buffer" de trabalho), atualizando-o no disco, liberando aquela área de trabalho da memória para a abertura de outro arquivo.
O comando USE <nome do arquivo> abre o arquivo de dados especificado para que se possa acessá-lo. O comando USE desacompanhando do nome de um arquivo fecha o arquivo que estiver aberto na área de trabalho atualmente selecionada.
A cláusula INDEX <lista de arquivos> permite que sejam abertos simultaneamente até sete arquivos de índice (quinze na versão Summer 87) associados ao arquivo de dados aberto na área de trabalho selecionada. Os nomes dos arquivos de índice devem ser especificados separados por vírgulas.
A menos que especificado, o Clipper assume a extensão **.dbf** para os arquivos de dados e a extensão **.ntx** para os arquivos de índice.
A cláusula ALIAS permite a definição de um nome a ser associado à área de trabalho onde foi aberto o arquivo. Se esta cláusula não for especificada o alias será assumido como sendo o próprio nome do arquivo aberto.
Em ambiente multiusuário a cláusula EXCLUSIVE abre o arquivo de dados

de forma não compartilhada (restrita). Todos os outros usuários, neste caso, não poderão acessar o arquivo, até que ele seja fechado.
A abertura de um novo arquivo na mesma área de trabalho causa automaticamente o fechamento do anterior.
Quando um arquivo de dados é aberto sem um índice a ele associado, o apontador dos registros é posicionado no registro número um (1). Por outro lado, quando um arquivo de dados é aberto simultaneamente com um ou mais arquivos de índice, o apontador dos registros é posicionado no primeiro registro lógico de acordo com o primeiro índice da lista. Este índice é chamado "índice mestre" (ou índice de controle) e será ele que determinará a ordem de acesso aos registros do arquivo. Ele e todos os demais serão automaticamente atualizados caso ocorram modificações no arquivo de dados (inclusões, alterações ou exclusões de registros). Dessa forma, todos os arquivos de índice associados ao arquivo de dados são mantidos atualizados. A atualização dos índices, contudo, consome um certo tempo adicional, necessário justamente para refletir as mudanças realizadas no arquivo de dados nos arquivos de índice.
Para alterar a ordem de acesso ao arquivo aberto sem a necessidade de reabrí-lo, utiliza-se o comando SET ORDER TO, que indica o índice que deverá ser utilizado como índice de controle.
Os arquivos abertos através do comando USE, permanecerão abertos até que sejam fechados pelos comandos USE (desacompanhado de parâmetros), CLOSE DATABASE, CLOSE ALL, CLEAR ALL, CANCEL ou QUIT.
As versões do Clipper anteriores à versão Summer 87 suportam um máximo de 15 arquivos de qualquer tipo (.dbf, .ntx, .dbt, .fmt, .frm ou .lbl) abertos simultaneamente. Caso este limite seja atingido, será apresentada uma mensagem de erro: DOS Error 2, indicando que o Sistema Operacional não consegue gerenciar mais arquivos. Caso esta mensagem seja apresentada mesmo não existindo 15 arquivos abertos, verifique o arquivo CONFIG.sys, que deverá estar presente no diretório de "boot" do seu equipamento. Ele deverá apresentar o seguinte conteúdo:

```
files=20
buffers=8
```

Este arquivo configura corretamente o DOS para a execução dos programas escritos em Clipper, pois o padrão de arquivos que podem ser gerenciados simultaneamente no DOS é de apenas 8.
Na versão Summer 87 e posteriores, se for utilizado o DOS 3.3 ou versões maiores, podem ser abertos simultaneamente até 255 arquivos de qualquer tipo. Neste caso o arquivo CONFIG.sys e o comando SET CLIPPER deverão ser alterados para informar ao DOS o número de arquivos que poderão ser utilizados. Por exemplo:

CONFIG.sys:

files=120
buffers=8

SET CLIPPER=F120

Neste caso até 120 arquivos poderão ser gerenciados pelo DOS 3.3 ou maior.

Biblioteca:

CLIPPER.LIB

Exemplos:

* Para abrir o arquivo Estoque.dbf sem nenhum índice com o Alias padrão Estoque:

```
SELECT 1
USE Estoque
```

* Para abrir agora o mesmo arquivo com o índice IndProd e com o Alias Est:

```
SELECT 1
USE Estoque INDEX IndProd ALIAS Est
```

* Para abrir o arquivo Mala, com os índices IndCod, IndAlfa, IndCid, IndUF e IndCep, com o Alias MDIR:

```
SELECT 2
USE Mala INDEX IndCod, IndAlfa, IndCid, IndUF, IndCep ALIAS MDIR
```

Veja Também:

CLOSE DATABASE, CLOSE ALL, CLEAR ALL, QUIT, INDEX, CREATE, SELECT, SET INDEX, SET ORDER, SET EXCLUSIVE

5.5.3. Comandos Para a Inclusão de Novos Registros

Comando APPEND BLANK:

Sintaxe:

APPEND BLANK

Propósito:

Adiciona um novo registro vazio (em branco) no final do arquivo atualmente em uso na área de trabalho selecionada.

Utilização:

O comando APPEND BLANK adiciona um novo registro em branco (vazio) no final do arquivo em uso. Este registro passa a ser o registro corrente, ou seja, o apontador de registros nele ficará posicionado.
APPEND BLANK é em geral utilizado em conjunto com o comando @...SAY...GET e READ,para proceder à entrada de novos dados em arquivos. APPEND BLANK adiciona um novo registro com os campos em branco ao final do arquivo; @...SAY...GET's editam os campos do novo registro, permitindo a inclusão de dados; finalmente o comando READ os "lê" para o registro, finalizando a entrada.
Quando em processamento multiusuário, através de uma rede local de microcomputadores, estando o arquivo de dados compartilhado, o novo registro adicionado ficará inacessível para os outros usuários. Ele permanecerá restrito até que o usuário que o adicionou o liberar através do comando UNLOCK ou o acesso a um outro registro for restringido. Se um outro usuário tiver restringido (bloqueado) o acesso ao arquivo de dados ou se estiver tentando executar um APPEND BLANK ao mesmo tempo, a função NETERR( ) (erro de rede) se tornará verdadeira (.T.) e a operação não será realizada. O comando APPEND BLANK não cancela a restrição de acesso ao arquivo (FLOCK( )) executada pelo usuário atual.

Biblioteca:

CLIPPER.LIB

Exemplo:

O exemplo a seguir é uma rotina básica para a inclusão de novos dados em um arquivo de dados, no caso um arquivo de Mala-Direta:

```
USE MALA INDEX MALA
* Abre o arquivo MALA.dbf e o índice associado MALA.ntx
CLEAR
* Limpa a Tela
borda1=CHR(218)+CHR(196)+CHR(191)+CHR(179)+CHR(217);
       +CHR(196)+CHR(192)+CHR(179)
* Define os caracteres a serem utilizado para criar a
* moldura no comando @...BOX
@ 03,10,20,70 BOX borda1
* Desenha a moldura
DO WHILE .T.
```

```
      APPEND BLANK
      * Adiciona um registro em branco
      @ 05,34 SAY "Mala-Direta"
      @ 07,27 SAY "Entrada/Alteração de Dados"
      @ 08,27 SAY "--------------------------"
      @ 11,15 SAY "Nome....:" GET nome     PICTURE "@!"
      @ 13,15 SAY "Endereço:" GET endereco PICTURE "@!"
      @ 15,15 SAY "Cidade..:" GET cidade   PICTURE "@!"
      @ 17,15 SAY "Estado..:" GET estado   PICTURE "!!"
      @ 17,30 SAY "Cep:"      GET cep      PICTURE "99999"
      * Edição dos campos do novo registro, permitindo a
      * inclusão de dados
      READ
      IF EMPTY(NOME)
          *  Se o nome for deixado em branco o registro é
         * eliminado através dos comandos DELETE e PACK
         DELETE
         PACK
         EXIT
      ENDIF
ENDDO
CLEAR
USE
* Fecha o arquivo
RETURN
```

Veja Também:

APPEND FROM, SET CARRY, SET CONFIRM, SET FORMAT

Comando APPEND FROM:

Sintaxe:

APPEND [<**escopo**>] [<**lista de campos**>]
FROM <nome do arquivo>
[**FOR/WHILE <condição>**]
[**SDF/DELIMITED WITH <delimitador>**]

Propósito:

Copia (ou adiciona) registros de um arquivo de dados existente, ou de um arquivo padrão ASCII para o final do arquivo que estiver em uso. O arquivo origem não precisa ser obrigatoriamente um arquivo de dados do Clipper (.dbf), poderá ser um arquivo tipo texto.

Utilização:

O comando APPEND FROM é utilizado para copiar (ou adicionar) registros de um arquivo de dados (.dbf) ou de um arquivo tipo texto, para o arquivo que estiver em uso.

O <nome do arquivo> é nome do arquivo a ser copiado ou arquivo de origem dos dados. Se não forem especificadas as cláusulas SDF ou DELIMITED, será assumido como sendo um arquivo de dados do Clipper, extensão (.dbf). Caso seja especificada uma das cláusulas, será assumido o tipo de arquivo-texto correspondente, com a extensão (.txt), a menos que outra seja informada.
Se o arquivo origem (FROM) for um arquivo de dados do Clipper ou do dBASE III (arquivo .dbf) o processamento se efetuará da seguinte forma:

- Somente os campos encontrados em ambos os arquivos (origem e destino) serão copiados. Eles não precisam aparecer na mesma ordem.
- Os campos dos arquivos origem e destino deverão ser do mesmo tipo de dado, caso contrário será apresentada a seguinte mensagem de erro: "Type conflict in REPLACE (Q/A/I)?" ou seja, tipo de dado em conflito na troca (finaliza, aborta ou ignora)?
- Não é necessário informar a extensão do arquivo origem; .dbf é assumido.
- Se um campo do arquivo de origem for maior que o do arquivo destino, dados tipo caractere serão truncados e dados numéricos serão substituídos por asteriscos.
- As condições das cláusulas FOR e WHILE referem-se apenas a nomes de campos comuns entre os dois arquivos.

Se o arquivo origem (FROM) não for um arquivo de dados do Clipper ou do dBASE III (arquivo .dbf), há três tipos de arquivos que o Clipper pode "importar":

- **SDF** – ("System Data Format") – Arquivo ASCII: Os dados são copiados caractere por caractere, começando pela esquerda. Cada registro deve possuir um tamanho fixo e ser finalizado por um "return" e um avanço de linha ("line feed").
- **DELIMITED WITH <delimitador>** – Formato com delimitador: Também é um arquivo ASCII; os dados são copiados caractere por caractere, começando pela esquerda. Cada registro deve ser finalizado por um "return" e um avanço de linha (line feed). A menos que um delimitador seja especificado, uma vírgula (,) deve separar os campos e aspas devem delimitar os dados do tipo caractere. DELIMITED WITH <delimitador> deve ser utilizado para arquivos contendo campos separados por vírgulas e com os dados tipo caractere opcionalmente delimitados pelo delimitador especificado. O delimitador padrão são as aspas.
- **DELIMITED WITH BLANK** – Delimitado por brancos: deve ser utilizado para arquivos do tipo anterior, contendo campos separados por espaços em branco.

A opção <escopo> pode ser utilizada para determinar a quantidade de registros do arquivo fonte a ser processada: ALL adiciona todos os registros do arquivo fonte no arquivo destino, NEXT <n> adiciona os próximos n registros e RECORD <n> adiciona o registro número n do arquivo fonte no arquivo destino. Se nenhum escopo for especificado ALL (todos) será assumido.
A <lista de campos> é utilizada para determinar os campos dos registros do arquivo fonte (.dbf) que deverão ser copiados para o arquivo destino. Se não for especificada uma lista de campos, todos serão copiados.
A <condição> das cláusulas FOR e WHILE é utilizada para selecionar os registros que deverão ser copiados do arquivo origem para o arquivo destino, desde que os campos referidos na condição sejam comuns a ambos. FOR <condição> varre todo o arquivo origem copiando os registros para os quais a condição se tornar verdadeira. WHILE <condição> copia os registros, a partir do atual, enquanto a condição permanecer verdadeira.
Se no arquivo origem existirem registros marcados para deleção, estes serão normalmente copiados para o arquivo destino perdendo a marcação para deleção. Se, contudo, SET DELETED estiver ON, os registros do arquivo origem que estiverem marcados para deleção não serão copiados.
Em processamento multiusuário de rede, o comando APPEND FROM não requer que o arquivo destino seja bloqueado através da função FLOCK( ) ou do comando USE EXCLUSIVE. O Clipper automaticamente impedirá o acesso ao arquivo destino durante a operação do APPEND FROM.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
USE DESTINO
APPEND FROM ORIGEM
```

* Neste caso o arquivo origem é um arquivo .dbf

```
USE DESTINO
APPEND FROM ORIGEM DELIMITED
```

* Neste caso o arquivo origem é um arquivo texto. Cada linha deste arquivo será considerada como um registro e cada campo deverá estar separado por vírgulas, como abaixo:

```
"JOSE DA SILVA","RUA DA PAZ, 111","SAO PAULO","SP"
"MARIA DOS SANTOS","RUA DA CRUZ, 34","ITU","SP"
"CARLOS DE LIMA","AV. BRASIL, 1234","SAO PAULO","SP"
```

Veja Também:

APPEND BLANK, COPY TO, FREAD( ), COPY FILE

5.5.4. Comandos Para Alteração de Registros dos Arquivos

Comando REPLACE:

Sintaxe:

REPLACE [<escopo>]
[<alias→>]<campo1> WITH <expressão>
, [[<alias→>] <campo2> WITH <expressão>
, ...]
[FOR <condição>]
[WHILE <condição>]

Propósito:

O comando REPLACE permite que os registros de um arquivo de dados sejam alterados pela substituição do conteúdo de campos especificados de seus registros pelo resultado de uma expressão. Esta troca pode ser individual (de um único registro), global (de todos os registros) ou seletiva (de alguns registros selecionados).

Utilização:

O comando REPLACE é muito importante. Através dele o conteúdo dos campos dos registros de um arquivo de dados pode ser alterado em modo "batch", tendo seu valor substituído por um dado, o conteúdo de uma variável ou pelo resultado de uma expressão especificada.
O <campo n> é o nome de um campo do arquivo de dados a ter seu conteúdo substituído. Este campo pode ser de qualquer tipo, incluíndo campos memo. Deverá haver um campo para cada cláusula WITH.
O <escopo> define a quantidade de registros do arquivo de dados a ser processada pelo comando: ALL (todos), NEXT <n> (os próximos n registros a partir do atual) ou RECORD <n> (apenas o registro n). Se não for especificado, apenas o registro corrente será processado.
A cláusula FOR especifica uma <condição>, dentro do <escopo> fornecido, para selecionar os registros que deverão ser processados pelo comando. Apenas para os registros que a satisfizerem (a tornarem verdadeira) terão seus dados alterados.

A cláusula WHILE seleciona os registros a serem processados, a partir do registro corrente, enquanto uma <condição> especificada por satisfeita (permanecer verdadeira). Para tanto, o arquivo de dados deverá estar indexado de acordo com a condição especificada.
A menos que seja especificado um escopo (ALL, NEXT <n> ou RECORD <n>) ou condições através das cláusulas FOR e WHILE, apenas o registro no qual o apontador de registros estiver posicionado (registro corrente) terá os conteúdos de seus campos substituídos.
A <expressão> é qualquer expressão válida do Clipper que, sendo avaliada, substituirá com seu resultado o conteúdo do campo especificado.
O campo do registro e a expressão a substituí-lo (que segue o WITH) devem ser do mesmo tipo de dado, isto é, ambas devem ser caractere, numéricas, datas ou lógicas.
No caso de campos numéricos, se o novo valor substituído pelo comando REPLACE for superior ao valor máximo comportado pelo campo, será apresentada a mensagem "Numeric Overflow", indicando que houve erro, sendo colocados asteriscos para indicar o "estouro" da capacidade numérica do campo.
Substituições através do comando REPLACE em campos que são chaves para índices, automaticamente atualizam o índice. Quando a troca de conteúdo é feita, o registro move-se para uma nova posição no arquivo de índice e conseqüentemente a posição do apontador de registros também é alterada. Por esse motivo não utilize um escopo ou condição FOR ou WHILE, quando estiver fazendo substituições em um campo que seja chave para o índice de controle enquanto este índice estiver aberto.
Para executar um REPLACE em vários registros em um campo que seja chave para um índice, feche este índice (através dos comandos SET INDEX TO ou CLOSE INDEX), execute o REPLACE, abra novamente o índice (SET INDEX TO) e reindexe o arquivo (através do comando REINDEX).
No Clipper os campos memo podem ser manipulados como se fossem cadeias de caracteres; por esse motivo é possível executar um REPLACE em campos memo por uma cadeia de caracteres. Por exemplo:

```
REPLACE c_memo WITH "Clipper Compilador do dBASE III"
```

O comando REPLACE não verifica se o registro a ter seus campos substituídos é o registro correto; por este motivo esteja certo de estar posicionado no registro desejado.
Em ambiente multiusuário de rede, o comando REPLACE requer que o registro a ser processado tenha acesso restrito através da função RLOCK( ). Caso sejam processados vários registros com o comando REPLACE é necessário que o arquivo tenha acesso restrito através do comando USE...EXCLUSIVE ou da função FLOCK( ). Se um registro estiver sendo processado em uma outra área de trabalho que não a selecionada, também deverá ter seu acesso bloqueado através da função RLOCK( ).

Opções:

REPLACE <alias>→<nome do campo> WITH <expressão>

O comando REPLACE também substitui campos de registros de outras áreas de trabalho que não a atualmente selecionada. Para se conseguir isso basta utilizar o ALIAS do arquivo aberto em outra área, que se deseja alterar.

Biblioteca:

CLIPPER.LIB

Exemplos:

REPLACE ALL salario WITH salario*1.63
* troca o valor do campo salário de todos os registros pelo salário
* multiplicado por 1.63.

REPLACE depto WITH "MARKETING" FOR depto="VENDAS"
* troca o conteúdo do campo DEPTO por MARKETING quando seu
* conteúdo for igual a VENDAS.

A rotina a seguir é um exemplo de entrada de dados em um arquivo através dos comandos @...GET, READ, APPEND BLANK e REPLACE:

```
SELECT 1
USE Entradas Alias Entra
SELECT 2
USE Estoque INDEX IndProd Alias Est
CLEAR
borda1=CHR(218)+CHR(196)+CHR(191)+CHR(179)+CHR(217);
       +CHR(196)+CHR(192)+CHR(179)
@ 03,10,20,70 BOX borda1
DO WHILE .T.
     CLEAR
     vproduto=SPACE(20)
     ventrada=0
     vpreco=0.00
     vdata=CTOD(SPACE(8))
     @ 05,34 SAY "E s t o q u e"
     @ 07,31 SAY "Entrada de Produtos"
     @ 08,31 SAY "----------------------"
     @ 11,15 SAY "Produto...:" GET vproduto  PICTURE "@!"
     @ 13,15 SAY "Quantidade:" GET ventrada
     @ 15,15 SAY "Preço.....:" GET vpreco
     @ 17,15 SAY "Data......:" GET vdata
     READ
     IF EMPTY(vproduto)
        EXIT
     ENDIF
     SELECT Est
```

```
      SEEK vproduto
      IF .NOT.FOUND()
         WAIT "        Produto Não Encontrado, Tecle <Enter>"
         LOOP
      ELSE
         SELECT Entra
         APPEND BLANK
         REPLACE  entrada  WITH entrada+ventrada, preco WITH
            vpreco/ventrada, data WITH vdata, entra->produto
            vproduto WITH vproduto,  entra->quantidade  WITH
            ventrada,  entra->preco WITH vpreco, entra->data
            WITH vdata
      ENDIF
ENDDO
CLEAR
USE
RETURN
```

Veja Também:

APPEND, JOIN, UPDATE, STRTRAN( )

5.5.5. Comandos Para a Exclusão de Registros dos Arquivos

Comando DELETE:

Sintaxe:

DELETE [<escopo>]
[**FOR** <condição>]
[**WHILE** <condição>]

Propósito:

O comando DELETE marca registros do arquivo de dados em uso, na área de trabalho selecionada, para eliminação.

Utilização:

A menos que seja especificado no <escopo> ou através de uma condição com as cláusulas FOR ou WHILE, apenas o registro corrente (registro no qual está posicionado o apontador de registros) é marcado para eliminação. Portanto, para "deletar" um determinado registro é necessário previamente posicionar o apontador de registros sobre ele; (veja os comandos SEEK, GO-TO, SKIP etc.).

O <escopo> define os registros a serem "deletados": ALL (todos), NEXT <n> (os próximos n registros a partir do atual) ou RECORD <n> (apenas o registro n).
A cláusula FOR especifica uma condição para selecionar os registros que deverão ser "deletados". Apenas os registros satisfizerem a condição (a tornaram verdadeira) serão "deletados".
A cláusula WHILE seleciona registros a serem "deletados", a partir do registro atual, enquanto uma condição especificada for satisfeita (permanecer verdadeira).
Usando os comandos DISPLAY e LIST, os registros marcados para apagamento ou deletados serão identificados por um asterisco (*), na primeira posição do registro.
Outra forma de se verificar se um registro está "deletado" (marcado para eliminação) é através da função DELETED( ), que resultará verdadeira (.T.) se o registro corrente estiver marcado, e falsa (.F.) caso contrário.
O comando DELETE apenas marca os registros para futura eliminação do arquivo de dados. Por segurança, para excluí-los definitivamente do arquivo deve ser utilizado o comando PACK.
Caso seja necessário desmarcar registros deletados pelo comando DELETE, pode ser utilizado o comando RECALL, que é o seu inverso.
Para eliminar simultaneamente todos os registros de um arquivo é muito mais rápido utilizar o comando ZAP ao invés de DELETE ALL e PACK.
O comando SET DELETED on/OFF filtra os registros marcados para eliminação. Se SET DELETED for definido como ON (ligado) os registros "deletados" serão ignorados em qualquer processamento, como se não fizessem mais parte do arquivo. Se SET DELETED for definido como OFF (desligado — padrão de operação do Clipper) os registros deletados serão processados exatamente como os demais.
Se utilizado em ambiente multiusuário de rede, o comando DELETE requer que o registro a ser deletado seja tornado de acesso restrito através da função RLOCK( ). Se forem deletados vários registros de um arquivo, este precisa ser colocado em acesso restrito através da função FLOCK( ) ou aberto com exclusividade através do comando USE EXCLUSIVE.

Biblioteca:

CLIPPER.LIB

Exemplos:

DELETE ALL — marca todos os registros do arquivo para apagamento.
DELETE — marca apenas o registro corrente.
DELETE NEXT 4 — marca os próximos 4 registros.
DELETE ALL FOR DEPTO="DIRETORIA" — marca todos os registros do arquivo cujo conteúdo do campo DEPTO seja DIRETORIA.

Veja Também:

PACK, RECALL, SET DELETED, ZAP, DELETED( ), RLOCK( ), FLOCK( )

Comando PACK:

Sintaxe:

PACK

Propósito:

O comando PACK remove fisicamente do arquivo de dados em uso os registros previamente marcados para eliminação (deleção) através do comando DELETE.

Utilização:

A exclusão efetiva dos registros marcados para eliminação através do comando DELETE é efetuada pelo comando PACK, que os elimina fisicamente, regravando apenas os registros não marcados. Todos os índices associados ao arquivo que estiverem abertos, são também, automaticamente reconstruídos (o arquivo é reindexado). O espaço físico ocupado pelos registros eliminados é recuperado e todos os registros do arquivo são renumerados.
Ao utilizar este comando procure certificar-se de que deseja realmente excluir do arquivo os registros que estão marcados, pois uma vez executado não será mais possível recuperá-los, a não ser pela sua reinclusão.
O comando PACK não cria um "back-up" (cópia de segurança) ou utiliza arquivos temporários. Todas as operações são realizadas no próprio arquivo de dados em uso.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
USE Pessoal
? LASTREC()                 && retorna 230
DISPLAY RECORD 10 nome      && retorna "MARIA"
DELETE RECORD 10
PACK
? LASTREC()                 && retorna 229
DISPLAY RECORD 10 nome      && retorna "JOSE"
* o registro 10 foi fisicamente excluído
* após o comando PACK os registros são renumerados
```

O exemplo a seguir ilustra um programa para a exclusão de registros de um arquivo de Mala-Direta.

```
**********************************
* MALA3.prg - Exclusao de Registros
**********************************
reorg=.F.
USE Mala INDEX Malacod ALIAS Mala
ex=" "
DO WHILE .T.
   CLEAR
   vcod=SPACE(5)
   @  12,25 "Código do Registro a Excluir: " GET vcod
   READ
   IF EMPTY(vcod)
      EXIT
   ENDIF
   SEEK vcod
   IF .NOT. FOUND()
      LOOP
   ENDIF
   CLEAR
   @ 06,00 TO 23,79 DOUBLE
   @  8, 15  SAY "Código:"
   @  8, 32  SAY "Data:"
   @  8, 48  SAY "Registro Nº:"
   @ 10, 15  SAY "Nome.....:"
   @ 12, 15  SAY "Cargo.....:"
   @ 14, 15  SAY "Razão Social:"
   @ 16, 15  SAY "Endereço....:"
   @ 18, 15  SAY "CEP:"
   @ 18, 33  SAY "Municipio:"
   @ 20, 15  SAY "Estado....:"
   @ 20, 33  SAY "Telefone.....:"
   SET COLOR TO W/R+
   @  8, 23  SAY  Mala->codigo
   @  8, 38  SAY  Mala->data
   @  8, 60  SAY  STR(RECNO(),4)
   @ 10, 29  SAY  Mala->nome
   @ 12, 29  SAY  Mala->cargo
   @ 14, 29  SAY  Mala->empresa
   @ 16, 29  SAY  Mala->endereco
   @ 18, 20  SAY  Mala->cep
   @ 18, 44  SAY  Mala->cidade
   @ 20, 27  SAY  Mala->estado
   @ 20, 50  SAY  Mala->telefone
   SET COLOR TO W/N
   @ 24, 60 SAY "Exclui (S/N) ? " GET ex PICTURE "!"
   READ
   @ 24, 00 CLEAR
   IF ex="S"
      DELETE
      reorg=.T.
   ENDIF
ENDDO
CLEAR
@ 06,00 TO 23,79 DOUBLE
IF reorg
   @ 10, 25 SAY "Deseja Reorganizar o Arquivo ?"
```

```
   @ 12, 23 SAY "Esta operação é um pouco demorada !"
   confirme=SPACE(1)
   @ 24, 60 SAY "Confirme (S/N) ? " GET confirme PICTURE "!"
   READ
   CLEAR
   @ 06,00 TO 23,79 DOUBLE
   IF confirme="S"
      SET COLOR TO W*
      @ 10, 35 SAY "Aguarde..."
      SET COLOR TO W
      @ 12, 29 SAY "Reorganizando o Arquivo"
      PACK
   ENDIF
ENDIF
USE
RETURN
```

Veja Também:

DELETE, RECALL, REINDEX, ZAP

Comando RECALL:

Sintaxe:

RECALL [<**escopo**>]
[**FOR** <**condição**>]
[**WHILE** <**condição**>]

Propósito:

O comando RECALL retira a marca para eliminação (deleção) do registro, colocada pelo comando DELETE, reincorporando normalmente o registro ao arquivo.

Utilização:

O comando RECALL serve para "chamar de volta" registros que haviam sido marcados para eliminação através do comando DELETE. É útil para recuperar um registro quando houve deleção incorreta.
A menos que especificado pelo escopo ou pela condição das cláusulas FOR ou WHILE, apenas o registro corrente (onde está posicionado o apontador de registros) é desmarcado ("RECALLed").
O RECALL não pode trazer de volta os registros excluídos fisicamente pelos comandos PACK ou ZAP.
Se o comando SET DELETED estiver ligado (ON), o comando RECALL

desmarca apenas o registro atual ou um registro específico, determinado pelo escopo RECORD <n>.
Para desmarcar registros é necessário que os mesmos sejam especificados ou o apontador de registros seja neles posicionado.
O <escopo> define a quantidade de registros do arquivo de dados a ser processada pelo comando: ALL (todos), NEXT <n> (os próximos n registros a partir do atual) ou RECORD <n> (apenas o registro n). Se não for especificado será assumido apenas o registro atual (NEXT 1).
A cláusula FOR especifica uma <condição>, dentro do <escopo> fornecido, para selecionar os registros que deverão ser processados pelo comando. Apenas os registros que a satisfizerem (a tornarem verdadeira) serão desmarcados.
A cláusula WHILE seleciona os registros a serem desmarcados, a partir do registro atual, enquanto uma <condição> especificada for satisfeita (permanecer verdadeira). Para tanto o arquivo de dados deverá estar indexado.
Em ambiente multiusuário de rede, para executar o comando RECALL é necessário que o registro atual esteja com acesso restrito através da função RLOCK( ), se for desmarcado apenas este registro. Se forem desmarcados vários registros é necessário que todo o arquivo esteja com acesso restrito através da função FLOCK( ) ou do comando USE...EXCLUSIVE.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
RECALL ALL      && desmarca todos os registros que
                   estiverem marcados para deleção.

RECALL RECORD 10 && desmarca o registro número 10.

RECALL && desmarca apenas o registro corrente.

RECALL FOR salario < 30000
* desmarca todos os registros deletados cujo campo
* salário contiver um valor inferior a 30.000
```

Veja Também:

DELETE, DELETED( ), PACK, SET DELETED, ZAP, RLOCK( ), RLOCK( ).

Comando ZAP:

Sintaxe:

ZAP

Propósito:

O comando ZAP exclui diretamente todos os registros de um arquivo de dados, não sendo necessário marcá-los previamente para eliminação através do comando DELETE.

Utilização:

O comando ZAP é equivalente a um DELETE ALL seguido de um PACK, mas é muito mais rápido. Deve ser utilizado quando se deseja "limpar" um arquivo, ou seja, eliminar todos os seus registros.
É necessário, portanto, muito cuidado na utilização deste comando, pois uma vez executado não será mais possível recuperar os registros do arquivo.
Os índices associados ao arquivo de dados que estiverem abertos e arquivos de campos memo serão também automaticamente limpos ("zapados").
Em um ambiente multiusuário, para executar um comando ZAP sobre um arquivo exige-se que o arquivo tenha sido aberto com exclusividade (acesso restrito) através do comando USE...EXCLUSIVE.

Biblioteca:

CLIPPER.LIB

Exemplo:

* Para excluir todos os registros do arquivo da Mala-Direta:

```
USE Mala INDEX MalaCod
ZAP
USE
```

Veja Também:

DELETE, PACK, RECALL, SET DELETED, DELETED( )

5.5.6. Comandos Para Posicionamento nos Registros

Comando GO/GOTO:

Sintaxe:

GO/GOTO <expressão numérica>

Propósito:

Posiciona o apontador de registros (registro corrente) em um registro especificado do arquivo de dados aberto (ativo) na área de trabalho selecionada.

Utilização:

Através do comando GO o apontador de registros pode ser posicionado em qualquer registro do arquivo de dados em uso na área de trabalho selecionada. A <expressão numérica> determina o número do registro para onde o apontador deve ser movido. GOTO move o apontador de registros para o registro especificado mesmo se este estiver "deletado" (marcado pelo comando DELETE) e o SET DELETED estiver em ON, ou mesmo se o comando SET FILTER estiver ativo e o registro indicado não atender ao filtro estabelecido. Se o arquivo em uso estiver indexado (for aberto com um índice a ele associado) o comando GO <n> refere-se ao número do registro, e não à <n-ésima> posição do índice.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
USE Mala INDEX IndCod
GO 10 — posiciona o apontador de registros no registro número 10.

GOTO 10 — idêntico ao anterior.

a=10
b=5
GO (a+b) — posiciona o apontador de registros no resultado da expressão,
no caso no registro número 15.

USE
```

Veja Também:

GO TOP, GO BOTTOM, SKIP, LOCATE, INDEX, SEEK, SET INDEX, SET ORDER, SET DELETED, EOF( ), BOF( ), FOUND( )

Comando GO BOTTOM:

Sintaxe:

GO BOTTOM

Propósito:

Posiciona o apontador de registros (registro corrente) no último registro do arquivo em uso na área de trabalho selecionada.

Utilização:

GO BOTTOM permite o posicionamento direto no último registro de um arquivo de dados, mesmo que não se conheça o seu número.
Se o arquivo em uso estiver indexado (estiver aberto juntamente com um índice a ele associado) GO BOTTOM se refere ao último registro (lógico) do arquivo de dados, de acordo com a ordem estabelecida pelo índice.
Se o comando SET DELETED estiver ligado (ON) ou se existir um filtro ativo através do comando SET FILTER, o comando GO BOTTOM posicionará o apontador de registros no último registro dentro do escopo especificado.

Biblioteca:

CLIPPER.LIB

Exemplos:

A rotina a seguir mostra na tela o último registro (lógico, ou seja, de acordo com a ordem do índice) do arquivo Mala.dbf.

```
USE Mala INDEX IndCod
GO BOTTOM
DISPLAY
USE
```

A rotina a seguir verifica se o comando GO BOTTOM e a função LASTREC( ) coincidem ao fornecer o número do último registro:

```
USE Mala
? LASTREC()
* retorna o número do último registro: 1230
GO BOTTOM
* posiciona o apontador no último registro
? RECNO()
* retorna o número do registro atual: 1230
```

Veja Também:

GO, GO TOP, SKIP, LOCATE, INDEX, SEEK, SET INDEX, SET ORDER, SET DELETED, EOF( ), BOF( ), FOUND( ), RECNO( ), LASTREC( )

Comando GO TOP:

Sintaxe:

GO TOP

Propósito:

Posiciona o apontador de registros (registro corrente) no primeiro registro do arquivo em uso na área de trabalho selecionada.

Utilização:

GO TOP permite o posicionamento direto no primeiro registro lógico de um arquivo de dados, mesmo que não se conheça o seu número.
Se o arquivo em uso estiver indexado (estiver aberto juntamente com um índice a ele associado) GO TOP se refere ao primeiro registro (lógico) do arquivo de índice (não necessariamente ao registro número 1).
Se o arquivo não estiver indexado, o comando GO TOP moverá o apontador de registros para o registro número um (1).
Se o comando SET DELETED estiver ligado (ON) ou se existir um filtro ativo através do comando SET FILTER, o comando GO TOP posicionará o apontador de registros no primeiro registro dentro do escopo especificado.

Biblioteca:

CLIPPER.LIB

Exemplos:

A rotina a seguir mostra na tela o primeiro registro (lógico, ou seja, de acordo com a ordem do índice) do arquivo Mala.dbf.

```
USE Mala INDEX IndCod
GO TOP
DISPLAY
USE
```

Veja Também:

GC, GO BOTTOM, SKIP, LOCATE, INDEX, SEEK, SET INDEX, SET ORDER, SET DELETED, EOF( ), FOUND( ), RECNO( )

Comando SKIP:

Sintaxe:

SKIP <exp. numérica> [ALIAS <nome do alias>]

Propósito:

O comando SKIP move o apontador de registros para frente ou para trás sobre o arquivo de dados em uso na área de trabalho selecionada, ou no arquivo especificado pelo seus alias, aberto em outra área de trabalho. Se o resultado da expressão numérica for positivo o apontador é deslocado <n> posições para frente; se for negativo, <n> posições para trás.

Utilização:

O comando SKIP é utilizado para fazer "pular" de registro em registro o apontador de registros. É muito útil para se proceder uma "varredura" do arquivo, ou seja, para percorrer todos os seus registros, um a um.
A <expressão numérica> determina o número de registros pelo qual será deslocado o apontador de registros. Um valor positivo desloca o apontador de registros para a frente, enquanto que um valor negativo o desloca para trás. Se o resultado da expressão for igual a zero (0), o "buffer" de memória do Clipper será transferido para o "buffer" do DOS (veja o comando COMMIT).
A cláusula ALIAS <nome do alias> desloca o apontador de registros do arquivo aberto na área de trabalho designada pelo nome do alias. Se não for

especificada, o deslocamento será efetuado no arquivo de dados aberto na área de trabalho selecionada.

Se um arquivo de índice estiver aberto juntamente com o arquivo de dados, o comando SKIP segue a ordem lógica especificada pelo índice; caso contrário, o comando SKIP segue seqüencialmente a ordem do número dos registros.

Quando não for incluída uma <expressão numérica>, o comando SKIP, por definição move o apontador para o próximo registro (um registro para frente, equivalente a SKIP 1).

Se o comando SKIP for acionado quando o apontador de registros estiver no último registro, o número do registro (RECNO( )) passa a ser um maior que o último registro (LASTREC( )+1), e a função EOF( ) (end-of-file — fim de arquivo) torna-se verdadeira (.T.).

Por outro lado, se o SKIP for acionado com valor negativo quando o apontador de registros estiver no primeiro registro, o número do registro (RECNO( )) permanece em 1, e a função BOF( ) (beginning-of-file — início do arquivo) passa a ser verdadeira (.T.).

Em qualquer dos casos não será registrada uma condição de erro.

Exemplos:

```
CLEAR
USE Mala INDEX IndCod
DISPLAY  && mostra na tela o primeiro registro
SKIP     && pula para o registro seguinte
DISPLAY  && mostra o registro seguinte
SKIP 10  && pula 10 registros para frente
DISPLAY  && mostra o registro
SKIP -5  && pula 5 registros para trás
DISPLAY  && mostra o registro
a=2
b=10
SKIP a*b && pula para frente 20 registros
DISPLAY  && mostra o registro
a=-1
SKIP a*b && pula para trás 10 registros
DISPLAY  && mostra o registro
USE
RETURN
```

Exemplo do uso da cláusula ALIAS:

```
SKIP ALIAS Mala
```

é o mesmo que:

```
SELECT Mala
SKIP
SELECT Estoque
```

Exemplo de impressão de Relatório de Mala Direta; o comando **SKIP** é utilizado para "percorrer" todo o arquivo, registro por registro.

```
************************************
*  MALA51.PRG - RELATORIO POR CODIGO
************************************
USE Mala INDEX IndCod
pg=0
i=0
SET DEVICE TO PRINT
DO WHILE .NOT. EOF()
   empre="C L I P P E R  -  V I D A L  -"
   siste="SISTEMA DE MALA-DIRETA"
   titulo1="Relatorio Geral"
   titulo2="Por Ordem de Codigos"
   emp     = Space( ( 66 - Len(EMPRE) ) / 2 ) + EMPRE
   sist    = Space( ( 66 - Len(SISTE) ) / 2 ) + SISTE
   titulo1 = Space( ( 66   Len(TITUL01) ) / 2 ) + TITUL01
   titulo2 = Space( ( 66 - Len(TITUL02) ) / 2 ) + TITUL02
   pg      = pg + 1
   @ 01, 03 SAY "Emissao:"
   @ 01, 12 SAY Date()
   @ 01,118 SAY "Pagina No." + STR(pg,3)
   @ 02, 0  SAY REPLICATE("=",132)
   @ 03, 01 SAY CHR(14) + emp
   @ 04, 01 SAY CHR(14) + sist
   @ 05, 01 SAY CHR(14) + titulo1
   @ 06, 01 SAY CHR(14) + titulo2
   @ 07, 0  SAY REPLICATE("=",132)
   DO WHILE PROW()<56 .AND. .NOT. EOF()
      @ PROW()+2,07 SAY "Codigo: "+codigo
      @ PROW(),27   SAY "Nome.....: "+nome
      @ PROW(),79   SAY "Funcao..: "+cargo
      @ PROW()+1,27 SAY "Razao Social: "+empresa
      @ PROW(),79   SAY "Endereco: "+endereco
      @ PROW()+1,07 SAY "CEP...: "+cep
      @ PROW(),27   SAY "Municipio...: "+cidade
      @ PROW(),66   SAY "Estado: "+estado
      @ PROW(),79   SAY "Telefone: "+telefone
      @ PROW(),108  SAY "Data..: "+DTOC(data)
      i=i+1
      SKIP
   ENDDO        && Quebra de página
   ENDDO           && Até o final do arquivo
   @ PROW()+2,0 SAY REPLICATE("=",132)
   @ PROW()+1,50 SAY "TOTAL DE REGISTROS LISTADOS:"+STR(i,4)
   @ PROW()+1,0 SAY REPLICATE("=",132)
   EJECT
   SET DEVICE TO SCREEN
   RETURN
```

Veja Também:

COMMIT, LOCATE, CONTINUE, GOTO, GO TOP, GO BOTTOM, FIND, SEEK, BOF( ), EOF( ), RECNO( )

5.5.7. Comandos Para Organização de Arquivos de Dados (Classificação de Arquivos)

No Clipper existem duas maneiras de se organizar ou classificar arquivos de dados:

1 – Organização física dos registros, através de uma cópia a partir do arquivo de dados (.dbf) que se quer organizar. Para isso utilizamos o comando SORT.
2 – Organização lógica dos registros, através de um arquivo auxiliar de índices do tipo (.ntx). Para isso utilizamos o comando INDEX.

Comando INDEX:

Sintaxe:

INDEX ON <expressão chave> TO <arquivo de índice>

Propósito:

Cria um arquivo de índice associado ao arquivo de dados em uso na área de trabalho selecionada, de acordo com uma expressão chave, através da qual o arquivo de dados é ordenado e passa a ser acessado alfabeticamente, cronologicamente ou numericamente.

Utilização:

A criação automática de arquivos de índice é um dos pontos fortes do Clipper, pois qualquer trabalho com arquivos ou Bancos de Dados necessita ordená-los e acessá-los de acordo com a visão necessária para a geração de informações.
A criação de arquivos de índice permite a obtenção de vários recursos fundamentais para o processamento de arquivos de dados:

1 – Classificação do arquivo de dados para acesso direto através de qualquer ordem;
2 – Facilidade de executar pesquisas instantâneas sobre a chave de criação dos índices, permitindo a consulta imediata às informações contidas nos registros;
3 – Relacionamento de diversos arquivos de dados através da chave dos índices, para a obtenção de informações contidas em vários arquivos simultaneamente.

4 — Atualização automática dos índices quando da inclusão, alteração ou exclusão dos registros do arquivo de dados que estiver indexado.

Um arquivo de índice, criado pelo comando INDEX, recebe no Clipper a extensão "**.ntx**" e contém apenas o valor da chave da ordenação de cada registro e uma referência à sua localização física (o número do registro) no arquivo de dados. Um arquivo de índices é um arquivo interno auxiliar, não sendo, portanto, acessível ao usuário.
Quando um índice é utilizado, os registros do arquivo de dados aparecem na ordem da expressão chave, embora não sejam fisicamente rearranjados pela operação de indexação. Isto permite que sejam criadas e automaticamente mantidas diversas ordens lógicas para acesso aos registros dos arquivos.
O Clipper utiliza uma variação da técnica de árvore-B ("B-tree") para a construção de índices, com páginas de 1.024 bytes. O arquivo (.ntx) possui um cabeçalho para o índice que informa qual é a chave do índice, onde é o "topo" da árvore etc. Quando uma chave é modificada no índice, a chave original é removida de sua localização e a nova chave é inserida no local apropriado.
Os índices normalmente criados pelo Clipper não são compatíveis com os índices do dBASE III que recebem a extensão (.ndx). Contudo, nas versões Summer 87 e posteriores é possível criar índices (.ndx), compatíveis com o dBASE III, desde que o módulo-objeto NDX.obj que as acompanha seja link-editado juntamente com os outros módulos-objeto de sua aplicação. De qualquer forma recomenda-se utilizar sempre os índices normais do Clipper (.ntx) e apenas utilizar os (.ndx) durante a fase de testes do sistema.
A chave de indexação de um arquivo de índice pode ser um campo ou uma expressão que envolve um ou mais campos do arquivo de dados; contudo, campos lógicos ou campos memo não podem ser utilizados como chave de índices. A <expressão chave> é uma expressão que retorna para cada registro do arquivo de dados um valor-chave a ser posicionado, em ordem ascendente, dentro de um arquivo de índice. O número máximo de posições que uma <expressão chave> pode ter é 250 caracteres.
Todos os registros do arquivo de dados, quer estejam marcados para deleção (comando DELETE) ou filtrados (comando SET FILTER) são incluídos no índice.
A indexação é feita sempre em ordem ascendente. Caso haja necessidade de uma indexação em ordem descendente em campos numéricos, pode ser utilizado o sinal de subtração (—) imediatamente antes da chave utilizada.
Nas versões Summer 87 e posteriores, para criar um índice em ordem decrescente, pode ser utilizada a função DESCEND( ). Esta função aceita qualquer tipo de dado como argumento e retorna seu inverso. Por exemplo, as instruções a seguir criam um índice em ordem cronológica descendente de acordo com a data de aniversário:

```
USE Pessoal
INDEX ON DESCEND(dataniv) TO Invainv
```

Ao se realizarem pesquisas (através do comando SEEK) sobre um índice criado em ordem descendente, a função DESCEND( ) deve ser utilizada para inverter também a chave da pesquisa (SEEK DESCEND(data)).
Quando a chave de um índice for constituída de vários itens, o item de maior importância para a determinação da ordenação deve ser colocado em primeiro lugar. Os demais em seguida, de acordo com sua importância na ordenação. Quando alguns registros possuírem para o primeiro item da chave o mesmo valor, o desempate, para determinar a ordem entre eles, será feito pelos itens subseqüentes.
ATENÇÃO a chave de um índice não pode misturar dados de diferentes tipos. Para combinar em uma chave de indexação tipos diferentes de dados, caractere, numérico e data, utilize respectivamente as funções STR( ), VAL( ), CTOD( ) e DTOS( ). Por exemplo, uma expressão chave que inclua uma data como parte da chave, deve ser criada como tipo caractere e utilizar a função DTOS( ) para converter a data para caractere, no formato "AAAAMMDD":

```
USE Pessoal
INDEX ON depto+DTOS(datadm) TO Depdata
```

A chave de indexação pode ser apenas um dos campos do registro, parte de um campo (veja a função SUBSTR( )), a concatenação de dois ou mais campos ou partes de campos. O comprimento total da chave de indexação não poderá exceder a 250 caracteres.
O Clipper permite a construção de índices com chaves únicas através do comando SET UNIQUE. Quando SET UNIQUE estiver ligado (ON), serão colocados no índice apenas os registros que possuírem chaves únicas (não duplicadas). Entretanto o comando SET UNIQUE é uma definição global e não uma atribuição dos arquivos de índice. Isso significa que quando SET UNIQUE estiver desligado (OFF), chaves duplicadas serão colocadas no índice, mesmo que ele tenha sido criado quando o SET UNIQUE estava ligado (ON).
A utilização da função **TRIM( )** para economizar espaço nos arquivos de índice **não funciona**, pois o Clipper aloca um espaço fixo para a chave dos índices (igual à soma dos comprimentos dos campos). A utilização da função TRIM( ) apenas irá confundir a indexação. Para criar um arquivo de índice que ocupe menos espaço utilize a função SUBSTR( ), definindo um tamanho fixo para a chave, de acordo com o comprimento da subcadeia especificada.
Pode-se construir quantos arquivos de índice forem necessários. Dessa forma pode-se acessar os registros dos arquivos de dados através de diversas ordens diferentes.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
USE Pessoal
INDEX ON nome TO Alfa
* o arquivo estará organizado alfabeticamente pelo nome através do índice
ALFA.ntx.

USE Pessoal
INDEX ON depto+nome TO Depalfa
* o arquivo estará organizado alfabeticamente pelo departamento e em seguida pelo nome, através do índice Depalfa.ntx.
```

Para indexar o arquivo Pessoal.dbf por ordem cronológica de admissão (campo datadm — data da admissão) e ordem alfabética utilizamos a função DTOS( ), que transforma uma data (dd/mm/aa) em caractere (aaaammdd):

```
INDEX ON DTOS(datadm)+nome TO Datnome
```

Arquivos de índice podem ser criados apenas uma vez, devendo posteriormente ser abertos juntamente com os arquivos de dados a que estiverem associados. Dessa forma toda atualização feita no arquivo de dados é automaticamente refletida nos arquivos de índice pelo Clipper. Por exemplo:

USE Pessoal INDEX Alfa

Abre o arquivo de dados PESSOAL.dbf e ao mesmo tempo o arquivo de índice ALFA.ntx. A partir disso os registros do arquivo PESSOAL serão acessados automaticamente em ordem alfabética, de acordo com o arquivo de índice ALFA.ntx. Qualquer alteração realizada no arquivo de dados (inclusão de novos registros, alteração ou exclusão de registros etc.) será automaticamente refletida no arquivo de índice.
O número máximo de arquivos de índice que podem ser abertos ao mesmo tempo para um arquivo de dados é 15 (quinze) na versão Summer 87 e 7 (sete) para as anteriores, não há porém limite para o número de índices que podem ser criados.

USE Pessoal INDEX Alfa, Depalfa

Através do comando acima é aberto o arquivo de dados PESSOAL, tendo seus registros organizados de conformidade com o primeiro índice aberto (ALFA — índice master), e os demais índices (no caso DEPALFA) são aber-

tos apenas para fins de atualização, não interferindo no arquivo de dados, mas afetando o tempo de processamento (todos os índices abertos estarão sendo atualizados).

Veja Também:

CLOSE, FIND, SEEK, REINDEX, SET INDEX TO, SET UNIQUE, SET ORDER, USE, DTOS( ), INDEXEXT( ), INDEXKEY( ), INDEXORD( ), DESCEND( ), SUBSTR( ), TRIM( )

Comando REINDEX:

Sintaxe:

REINDEX

Propósito:

O comando REINDEX reconstrói todos os arquivos de índice que estiverem abertos na área de trabalho selecionada.

Utilização:

O comando REINDEX reconstrói todos os índices associados ao arquivo de dados que estiverem abertos na área de trabalho selecionada. É útil para a reconstrução de índices que não estavam abertos quando foram efetuadas manutenções no arquivo de dados.

Ao reindexar um arquivo, a chave do índice é mantida exatamente igual à chave do índice original. Deve-se tomar cuidado, entretanto, com o comando SET UNIQUE. Se SET UNIQUE estiver ligado (ON), todos os índices serão construídos com chave única, mesmo que originalmente não o tenham sido. No Clipper, o SET UNIQUE é uma definição global e não um atributo específico dos arquivos de índice.

Pode ocorrer, durante a utilização de programas criados através do Clipper, que os índices de acesso aos arquivos de dado se "desestruturem". Para reconstruí-los pode-se, periodicamente, utilizar o comando REINDEX.

Em ambiente multiusuário de rede, o comando REINDEX requer que o arquivo de dados a ser reindexado (ter seus índices reconstruídos) esteja aberto com acesso restrito através do comando USE...EXCLUSIVE ou da função FLOCK( ).

Biblioteca:

CLIPPER.LIB

Exemplo:

O arquivo da Mala-Direta (Mala.dbf) possui os índices IndCod (por código), IndNome (por nome) e DatNome (data de admissão e nome). Para reindexá-los:

```
USE Mala
SET INDEX TO IndCod, IndNome, DataNome
REINDEX
USE
```

Veja Também:

INDEX, PACK, SET INDEX, SET UNIQUE, USE

Comando SORT:

Sintaxe:

SORT [<escopo> **TO** <novo arquivo>
ON <campo1> [/A] [/C] [/D]
[<campo2> [/A] [/C] [/D]...]
[**FOR** <condição>]
[**WHILE** <condição>]

Propósito:

O comando SORT cria um novo arquivo de dados, a partir do arquivo de dados em uso (aberto na área de trabalho selecionada), no qual os registros do arquivo estão ordenados alfabeticamente, cronologicamente ou numericamente, através dos campos especificados.

Utilização:

O comando SORT cria um novo arquivo de dados a partir do arquivo em uso, que contém seus mesmos registros, só que na ordem especificada pela chave da classificação.
<campo 1>, <campo 2>... , são os campos do arquivo a serem utilizados como chave para realizar a classificação. O comando SORT não trabalha com subcadeias ou expressões para serem utilizadas como chave; apenas com campos. Além disso, a classificação não pode ser feita tendo como chave campos Lógicos ou tipo Memo.

O <novo arquivo> é o nome do novo arquivo de dados a ser criado com os registros do arquivo original classificados na ordem determinada. A menos que outra extensão seja especificada, a extensão (.dbf) será assumida pelo Clipper.
O <escopo> define a quantidade de registros a ser processada: ALL (todos), NEXT <n> (os próximos n registros a partir do atual). Caso não seja especificado um escopo, todos os registros do arquivo serão "sorteados".
A cláusula FOR especifica uma condição para selecionar os registros que deverão ser processados. Apenas os registros a satisfizerem (a tornarem verdadeira) serão classificados e copiados para o novo arquivo.
A cláusula WHILE seleciona os registros a serem processados, a partir do registro atual, enquanto uma condição especificada for satisfeita (permanecer verdadeira). Somente enquanto a condição for satisfeita os registros serão classificados e copiados para o novo arquivo.
A classificação do arquivo através do comando SORT, para um novo arquivo, é realizada normalmente em ordem ascendente (/**A**), a menos que outra ordem seja especificada (veja as opções abaixo).
O comando SORT também classifica e copia registros deletados para o novo arquivo, onde estes registros permanecerão com a marca para eliminação.
Quando for necessário classificar-se um arquivo por múltiplos campos, inicie pelo campo que represente a chave mais importante. Em seguida, separados por vírgulas, indique os outros campos em sua ordem de importância (do mais importante para o menos importante).
Um arquivo não pode ser "sorteado" (classificado) sobre ele mesmo ou sobre outro arquivo que estiver aberto em outra área de trabalho.
A classificação através do comando SORT é uma classificação física, de acordo com o valor ASCII das chaves especificadas, pois o arquivo de dados é recopiado para outro arquivo de acordo com a ordem estabelecida pela chave. Caso este novo arquivo sofra alguma modificação em seus registros (inclusões, alterações ou exclusões), a classificação precisa ser refeita.
Em um ambiente multiusuário de rede o arquivo de dados a ser classificado através do comando SORT deve ter seu acesso restrito, através da função FLOCK( ) ou ser aberto pelo comando USE...EXCLUSIVE.

Opções:

As palavras chave ASCENDING (ascendente) e DESCENDING (descendente) são alternativas para /A e /D, e determinam a ordem de classificação desejada.
A cláusula /C faz com que o comando SORT, durante a classificação, não diferencie entre letras maiúsculas e minúsculas. /C pode ser combinada tanto com /A como com /D. Quando isso acontecer utiliza-se apenas uma barra, por exemplo: /DC ou /DA.

SORT x INDEX:

Sempre que possível é preferível utilizar o comando INDEX para ordenar arquivos de dados, pois o arquivo de índice é bem menor que o de dados e é automaticamente atualizado sempre que o arquivo de dados for modificado. Além disso, através de índices podem ser realizadas pesquisas instantâneas com os comandos SEEK e FIND, podendo ser criados quantos índices forem necessários para visualizar o arquivo de dados por diversas ordens de classificação.
Outro problema do comando SORT é que o Clipper utiliza um arquivo temporário (Clipsort.tmp) para elaborar a classificação. Este arquivo pode ser tão grande quanto o arquivo de dados sendo classificado, exigindo, portanto, muito espaço disponível no disco para realizar a operação do SORT. Na verdade o comando SORT utiliza três arquivos para realizar a classificação: o arquivo de dados original, o arquivo de dados destino e o arquivo de trabalho temporário.
Finalmente, a operação de classificação, através do comando SORT, é muito mais lenta que a operação de criação de arquivos de índice, através do comando INDEX, contudo o resultado prático dos mesmos é, em geral, idêntico.

Biblioteca:

CLIPPER.LIB

Exemplos:

O arquivo da Mala-Direta (Mala.dbf) pode ser classificado para o arquivo Malalfa.dbf, onde os registros estarão ordenados alfabeticamente:

```
USE Mala
SORT ON nome TO Malafa
USE Malalfa
DISPLAY ALL nome
USE
```

```
Caso queiramos classificá-lo por data de admissão e por
ordem alfabética, teríamos:

USE Mala
SORT ON data,nome TO Maldat
USE Maldat
DISPLAY ALL data,nome
USE
```

Veja Também:

INDEX, ASORT( )

5.6. COMANDOS PARA PESQUISA EM ARQUIVOS

Comando LOCATE:

Sintaxe

LOCATE [<escopo>] **FOR** <condição>
[**WHILE** <condição>]

Propósito:

Localiza o primeiro registro que satisfaz uma condição especificada, pela busca seqüencial através dos registros do arquivo de dados em uso.

Utilização:

O comando LOCATE destina-se à realização de pesquisas sobre o arquivo de dados em uso, localizando o primeiro registro para o qual uma condição especificada se tornar verdadeira.
O <escopo> define a quantidade de registros do arquivo de dados a serem processados pelo comando: ALL (todos), NEXT <n> (os próximos n registros a partir do atual). Se não for especificado serão assumidos todos (ALL).
O <escopo> NEXT <n> limita a pesquisa ao número de registros especificados e inicia a pesquisa a partir do registro corrente e não do primeiro, como é o normal.
A cláusula FOR especifica a <condição>, dentro do <escopo> fornecido, que será utilizada para pesquisar os registros. A condição é uma expressão lógica do tipo nome="JOSE", codigo="1234" ou salario<100000.
A partir do primeiro registro definido pelo <escopo> e um após o outro, a condição especificada é testada para todos os registros. A pesquisa termina se para algum registro a <condição> se tornar verdadeira, ou se for encontrado o fim do <escopo> do LOCATE. Se a pesquisa for bem sucedida, o registro encontrado passa a ser o registro atual (registro sobre o qual está posicionado o apontador de registros) e a função FOUND( ) torna-se verdadeira (.T.). Se a pesquisa não for bem sucedida, a função FOUND( ) torna-se falsa (.F.) e o posicionamento do apontador de registros dependerá do escopo especificado.

Se a função EOF( ) tornar-se verdadeira (.T.) e a função FOUND( ) tornar-se falsa (.F.), significa que todos os registros do arquivo foram pesquisados e nenhum deles satisfez a condição solicitada.

A cláusula WHILE seleciona os registros a serem localizados, a partir do registro atual, enquanto uma condição especificada for satisfeita (permanecer verdadeira). Para tanto o arquivo de dados deverá estar indexado. WHILE limita a pesquisa aos registros que tornarem a condição verdadeira e inicia a pesquisa a partir do registro corrente e não do primeiro, como é o normal.

Para achar as ocorrências subseqüentes da condição pesquisada, deve ser utilizado o comando CONTINUE (veja a seguir). O comando CONTINUE trabalha em conjunto com o comando LOCATE, permitindo que, após o primeiro registro ser encontrado, a pesquisa do comando LOCATE possa ser continuada a partir da posição atual do apontador de registros.

Os comandos LOCATE e CONTINUE são específicos à área de trabalho na qual foram executados. Pode-se ter diferentes LOCATE's e CONTINUE's para cada uma das áreas de trabalho selecionadas. Mesmo que se mude de uma área de trabalho para outra, através do comando SELECT, o LOCATE de cada área ainda permanecerá ativo, caso se retorne a ela.

Biblioteca:

CLIPPER.LIB

Exemplos:

Por exemplo, usando o arquivo Pessoal:

```
USE Pessoal
LOCATE FOR nome="JOSE" && localiza o primeiro JOSE
DISPLAY
LOCATE FOR nome="JOSE" .AND. depto="VENDAS"
DISPLAY
? EOF()     && se EOF() = .T. e
? FOUND()   && se FOUND() = .F. não há mais registros.
USE
```

Para encontrar todos os registros de um arquivo que satisfaçam uma condição utilize a seguinte construção:

```
USE Arquivo
LOCATE FOR <condição>
DO WHILE FOUND()
   DISPLAY
   CONTINUE
ENDDO
USE
RETURN
```

Veja Também:

CONTINUE, FIND, SEEK, FOUND( ), EOF( ), STRTRAN( ), RECNO( )

Comando CONTINUE:

Sintaxe:

CONTINUE

Propósito:

O comando CONTINUE retoma a pesquisa do comando LOCATE após ter sido encontrado o primeiro registro que satisfez a condição especificada. CONTINUE reinicia a pesquisa para encontrar o próximo registro do arquivo de dados selecionado que satisfaz a condição especificada pelo último comando LOCATE executado.

Utilização:

Se houver mais de um registro que satisfaça a condição especificada num comando LOCATE, a pesquisa pode ser continuada através do comando CONTINUE. Após o primeiro registro ter sido localizado, o Clipper continua a pesquisar, a partir do registro atual, até que um outro registro satisfaça novamente a condição especificada no último LOCATE executado.

O comando CONTINUE deve sempre ser utilizado em conjunto com o comando LOCATE. Ele pesquisa, a partir do registro onde estiver posicionado o apontador de registros, o próximo registro que satisfaz a condição especificada no último comando LOCATE executado na área de trabalho selecionada.

O comando CONTINUE termina após um novo registro ser encontrado ou quando o fim do arquivo ou do escopo do comando LOCATE for atingido.

Os comandos CONTINUE e LOCATE são específicos à área de trabalho na qual foram executados. Pode-se ter diferentes LOCATE's e CONTINUE's para cada área de trabalho selecionada. Mesmo que se mude de uma área de trabalho para outra, através do comando SELECT, os LOCATE's executados em cada área permanecerão ativos, até que um novo LOCATE seja executado.

Quando um "CONTINUE" é bem sucedido, a função FOUND( ) torna-se verdadeira (.T.), e o apontador de registros se posicionará no registro que satisfez a condição. Caso contrário, a função FOUND( ) será falsa (.F.), e o apontador de registros se posicionará no fim do escopo do comando LOCATE ou no fim do arquivo, tornado a função EOF( ) verdadeira (.T.).

Biblioteca:

CLIPPER.LIB

Exemplo:

```
USE Pessoal
LOCATE FOR nome="JOSE" && pesquisa o primeiro JOSE
DISPLAY                && mostra o registro
CONTINUE               && pesquisa o segundo JOSE
? EOF(), FOUND(), RECNO()
* retorna, por exemplo, .F., .T. e 37.
DISPLAY                 && mostra o registro
LOCATE FOR nome="JOSE" .AND. depto="VENDAS"
* um novo locate com escopo diferente do primeiro
DISPLAY                 && mostra o registro
CONTINUE                && pesquisa o próximo
DISPLAY                 && mostra o registro
? EOF()     && se EOF() = .T. e se FOUND() = .F.
? FOUND()   && não há mais registros a encontrar
USE
RETURN
```

Veja Também:

LOCATE, FOUND( ), EOF( ), RECNO( )

Comando FIND:

Sintaxe:

FIND <cadeia de caracteres>/<número>

Propósito:

O comando FIND pesquisa sobre um arquivo de dados indexado pelo primeiro registro que possua uma chave (expressão chave do índice) coincidente com a cadeia de caracteres ou número especificado, posicionando o apontador de registros sobre o registro correspondente. Por atuar sobre a chave dos índices, o comando FIND constitui uma pesquisa extremamente rápida.

Utilização:

O comando FIND somente pode ser utilizado se o arquivo de dados estiver indexado. A pesquisa só pode ser feita de acordo com a chave do índice,

isto é, o comando FIND somente "acha" registros cuja chave do índice coincidam com a cadeia de caracteres ou número especificado.
A técnica de pesquisa empregada pelo comando FIND (árvore binária) é extremamente rápida e eficiente, praticamente não sendo afetada pelo tamanho (número de registros) do arquivo de dados.
A <cadeia de caracteres> pode ser toda ou apenas parte da chave do índice para se encontrar o registro que está sendo procurado.
O comando SEEK (veja a seguir)) possui função idêntica, mas, ao contrário do FIND, permite a utilização de expressões para a pesquisa de registros. É, em geral, o mais utilizado em programação.
O comando LOCATE (veja acima) também possui uma função similar, não requerendo, entretanto, que o arquivo de dados esteja indexado. Sua pesquisa é, porém, extremamente mais lenta, pois são "varridos" seqüencialmente todos os registros do arquivo de dados na procura pelo registro que atenda a condição (ou chave) especificada.

Funcionamento:

Se for encontrado um registro que possuir como chave do índice um valor idêntico à cadeia de caracteres pesquisada, o apontador de registros se posicionará automaticamente sobre ele e a função FOUND( ) passará a ter o valor verdadeiro (.T.). Caso contrário, o apontador de registros se posicionará no fim do arquivo, tornando a função EOF( ) verdadeira (.T.) e a função FOUND( ) falsa (.F.).
A <cadeia de caracteres> a ser pesquisada precisa ser delimitada apenas quando contiver brancos. Neste caso ela deve conter exatamente o mesmo número de brancos que existe na chave do índice a ser encontrada, devendo ser delimitada por aspas ou apóstrofos.
Quando a pesquisa for efetuada através do conteúdo de uma variável de memória, a variável deve ser usada com a função macro-substituição (&). Se a variável contiver brancos, também é necessário que seja delimitada: "&<variável>". Neste caso, entretanto, é preferível utilizar-se o comando SEEK, que, por pesquisar expressões, é muito mais adequado.
Quando efetuando uma pesquisa, se o comando SET SOFTSEEK (disponível apenas da versão Summer 87 em diante) estiver em OFF (desligado), como é o padrão (default), o comando FIND funcionará exatamente como descrito acima.
Porém, se o comando SET SOFTSEEK estiver em ON (ligado), o apontador de registros se posicionará no primeiro registro que possuir um valor para a chave do índice **igual ou superior** ao da <cadeia de caracteres> ou número pesquisado. Neste caso, tanto a função FOUND( ) como a função EOF( ) serão falsas (.F.). A função EOF( ) apenas será verdadeira se não existir no índice uma chave igual ou superior à pesquisada. Neste caso o apontador de registros será posicionado no final do arquivo.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
USE Mala
INDEX ON codigo TO IndCod
FIND 12123
IF FOUND()
   DISPLAY
ELSE
   ? "Chave nao Encontrada !"
ENDIF
var="23457"
FIND &var
IF FOUND()
   DISPLAY
ELSE
   ? "Chave nao Encontrada !"
ENDIF
INDEX ON nome TO IndNome
FIND "JOSE DOS SANTOS"
DISPLAY
FIND JOSE
DISPLAY
nom="MARIA APARECIDA"
FIND "&nom"
DISPLAY
nom="WILSON"
FIND &nom
DISPLAY
USE
RETURN
```

Veja Também:

INDEX, SEEK, SET INDEX, SET ORDER, LOCATE, EOF( ), FOUND( ), SET EXACT, SET SOFTSEEK

Comando SEEK:

Sintaxe:

SEEK <expressão chave>

Propósito:

O comando SEEK pesquisa sobre um arquivo de dados indexado pelo primeiro registro que possua uma chave (expressão chave do índice) coincidente

com o conteúdo da expressão especificada. SEEK constitui uma pesquisa de registros extremamente rápida.

Utilização:

O comando SEEK é um poderoso meio de acessar diretamente os registros de um arquivo de dados, pois encontra o primeiro registro cuja chave de indexação coincida, total ou parcialmente, com a expressão especificada.
O comando SEEK somente pode ser utilizado se o arquivo de dados estiver indexado. A pesquisa é feita sobre a chave do índice de controle, ou seja, índice que estabelece a ordem de acesso aos dados: o primeiro índice da lista de índices do comando USE ou o índice definido pelo comando SET ORDER.
A técnica de pesquisa empregada é uma variação da "pesquisa binária", que é extremamente rápida e eficiente, praticamente não sendo afetada pelo tamanho (número de registros) do arquivo.
O comando FIND (veja a acima) possui uma função similar, mas, ao contrário do SEEK, permite a utilização de cadeias de caracteres ou valores numéricos para a pesquisa. Não é muito utilizado em programação.
O comando LOCATE também possui uma função similar, não requerendo que o arquivo esteja indexado. Contudo é extremamente mais lento, pois "varre" seqüencialmente todos os registros do arquivo na procura da condição (chave) especificada.
Ao se efetuar uma pesquisa com o comando SEEK, se a expressão especificada for encontrada, o apontador de registros se posicionará no registro encontrado, tornando-o o registro atual e a função FOUND( ) verdadeira (.T.). Se, por outro lado, a expressão especificada não for encontrada, o apontador de registros se posicionará no fim do arquivo, tornando a função EOF( ) verdadeira (.T.) e a função FOUND( ) falsa (.F.), indicando que nenhum registro foi encontrado.
Ao pesquisar-se através do SEEK, se o comando SET SOFTSEEK (disponível apenas da versão Summer 87 em diante) estiver em OFF (desligado), como é o padrão (default), a pesquisa do comando SEEK funcionará exatamente como descrito acima. Porém, se o comando SET SOFTSEEK estiver em ON (ligado), o apontador de registros se posicionará no primeiro registro que possuir um valor para a chave do índice **igual ou superior** ao da expressão especificada. Neste caso, tanto a função FOUND( ) como a função EOF( ) serão falsas (.F.). A função EOF( ) apenas será verdadeira se não existir no índice uma chave igual ou superior à pesquisada, quando o apontador de registros se posicionará no final do arquivo.
Se a expressão a ser pesquisada for uma cadeia de caracteres, deve ser delimitada por aspas ou apóstrofos. Se a expressão for uma variável de memória ou uma expressão numérica, não devem ser colocados delimitadores.
Podem ser utilizadas chaves parciais (expressões que apenas formem parte da chave contida no índice) desde que a parte da chave pesquisada inicie-se

da posição mais à esquerda da chave do índice. Por exemplo, se a expressão chave para o índice é SILVAJOSE (formada por INDEX ON sobrenome+ nome TO IndSobre), uma expressão parcial válida para pesquisa é "SILVA" e não "JOSE".

Biblioteca:

CLIPPER.LIB

Exemplos:

```
USE Mala
INDEX ON codigo TO IndCod
? INDEXKEY(0)
* Resulta CODIGO
vpesq="12123"
SEEK vpesq
IF FOUND()
   DISPLAY
   * Apresenta os dados do registro encontrado
ELSE
   ? "Chave nao Encontrada !"
ENDIF
var="23457"
SEEK var
IF FOUND()
   DISPLAY
ELSE
   ? "Chave nao Encontrada !"
ENDIF
INDEX ON nome TO IndNome
? INDEXKEY(0)
* Resulta NOME
SEEK "JOSE DOS SANTOS"
DISPLAY
* Apresenta os dados do registro encontrado
SEEK "JOSE"
DISPLAY
nom="MARIA APARECIDA"
SEEK nom
DISPLAY
* Apresenta os dados do registro encontrado
nom="WILSON"
SEEK nom
DISPLAY
* Apresenta os dados do registro encontrado
USE
RETURN
```

Veja Também:

FIND, LOCATE, INDEX, SET INDEX, SET ORDER, SET DELETED, SET EXACT, SET SOFTSEEK, USE, EOF( ), FOUND( )

5.7. COMANDOS PARA MANIPULAÇÃO DE ARQUIVOS

Comando COPY FILE:

Sintaxe:

```
COPY FILE   <nome do arquivo origem>.<extensão>
     TO     <nome do arquivo  cópia>.<extensão>
```

Propósito:

O comando COPY FILE cria uma cópia duplicata de qualquer tipo de arquivo.

Utiiização:

O comando COPY FILE pode ser utilizado para copiar qualquer tipo de arquivo, mesmo arquivos não pertencentes ao Clipper.
Tanto o nome do arquivo origem (arquivo que deve ser copiado) quanto o nome do arquivo cópia (arquivo destino da cópia) devem incluir sua extensão, a designação do acionador de discos (drive) e do diretório onde estão localizados, caso não seja utilizado o acionador e o diretório padrão (default).
Se o comando COPY FILE for utilizado para copiar um arquivo de dados (.dbf) que contenha campos memo, o arquivo memo (.dbt) associado também deverá ser copiado.
Um arquivo de dados que estiver em uso (aberto) não pode ser copiado através do COPY FILE; veja o comando COPY TO.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
COPY FILE Mala.dbf TO Mala1.dbf
* cria uma duplicata do arquivo de dados Mala com o nome de Mala1

COPY FILE C:\mala\IndCod.ntx TO B:IndCod1.ntx
* cria uma duplicata do arquivo de índice IndCod do drive C: e subdiretório
  mala com o nome de IndCod1 no drive B:.
```

Veja Também:

CLOSE, COPY TO, USE, RUN e o comando COPY do DOS.

Comando DIR:

Sintaxe:

DIR [<**drive**> :] [<**diretório**>\] [<**seleção**>]

Propósito:

Mostra a lista de arquivos existentes no disco (diretório), de acordo com a seleção especificada.

Utilização:

O comando DIR do Clipper é semelhante ao comando ao comando DIR do DOS, contudo possui duas formas de apresentação, dependendo se uma seleção de arquivos for ou não especificada.
Se nenhuma seleção for especificada, o comando DIR atua somente sobre os arquivos de dados existentes no drive e diretórios especificados ou no atual. São apresentados o nome, o número de registros, a data da última atualização e o número de bytes ocupados por todos os arquivos de dados (.dbf) existentes no disco.
Se uma seleção for especificada, são apresentados os nomes dos arquivos, a extensão, o número de bytes ocupados e a data da última atualização dos arquivos que atendem a essa seleção.
As regras para especificar o drive e o diretório desejados, bem como a seleção dos arquivos a serem apresentados, são as mesmas observadas no DOS.
O comando DIR pode não refletir as últimas mudanças nos arquivos de dados, principalmente após inclusões ou eliminações de registros, até que os arquivos de dados sejam fechados e suas informações atualizadas no disco.
Para criar aplicações onde são apresentados o diretório do disco ao usuário é preferível utilizar a função ADIR( ). Esta função permite a construção de vetores cujos elementos conterão os atributos dos arquivos selecionados. Você poderá, a seguir, utilizar estes vetores para apresentar a informação do diretório ao usuário da forma que desejar.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
DIR     && mostra todos os arquivos de dados existentes no drive padrão
           (default).
```

DIR *.* && mostra todos os arquivos (de qualquer tipo) existentes no drive default.

DIR *.ntx && mostra apenas os arquivos de índice existentes no drive default.

DIR C:\mala\Ind*.* && mostra todos os arquivos existentes no drive C, no subdiretório mala e que se iniciem por Ind.

Veja Também:

ADIR( ), RUN e o comando DIR do DOS.

Comando ERASE/DELETE FILE:

Sintaxe:

ERASE/DELETE FILE <nome do arquivo>.<extensão>

Propósito:

O comando ERASE ou DELETE FILE elimina (remove) um arquivo do disco.

Utilização:

Este comando é utilizado para remover arquivos, não mais necessários, do disco. Deve ser utilizado com cuidado, pois após sua execução será quase impossível a recuperação do arquivo eliminado.
O nome do arquivo a ser removido deve incluir sua extensão e, caso o arquivo não esteja no drive ou diretório atual (default), suas designações.
Um arquivo aberto (ativo em qualquer área de trabalho selecionada) não pode ser eliminado; é necessário antes fechá-lo através dos comandos USE ou CLOSE.
Se um arquivo de dados (.dbf) que contenha campos memo for removido, não se esqueça de remover também o arquivo memo (.dbt) a ele associado.
Ao contrário do comando ERASE do DOS, o ERASE do Clipper não permite que sejam utilizadas máscaras para a eliminação de vários arquivos simultaneamente. Somente é possível a eliminação de um arquivo por vez.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
? FILE("Mala.dbf")      && resulta .T. (verdadeiro)

ERASE Mala.dbf    && elimina o arquivo Mala.dbf do disco

? FILE("Mala.dbf")      && resultará .F. (falso)
```

Veja Também:

CLOSE, USE, RUN e os comandos ERASE e DEL do DOS

Comando RENAME:

Sintaxe:

RENAME <nome atual>.<ext.> TO <novo nome>.<ext.>

Propósito:

O comando RENAME permite a mudança do nome de um arquivo do disco.

Utilização:

O comando RENAME deve ser utilizado para trocar o nome de um arquivo existente no disco.
Ambos os nomes do antigo e do novo arquivo devem incluir suas extensões e, se o arquivo não estiver no drive e diretório padrão, suas designações.
Não pode haver nomes duplicados no disco, neste caso a troca de nomes não será feita.
Se um arquivo de dados (.dbf) que contenha campos memo for renomeado, o arquivo memo associado (.dbt) deve ser necessariamente também renomeado. Esta operação não é automaticamente feita pelo Clipper.
Um arquivo que estiver aberto em qualquer área de trabalho selecionada não pode ser renomeado. Utilize o comando USE ou CLOSE DATABASE para fechá-lo e em seguida mudar seu nome.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
RENAME Mala.dbf TO Mdireta.dbf
* troca o nome do arquivo Mala.dbf para Mdireta.dbf

RENAME A:Estoque.dbf TO A:Produtos.dbf
```

Veja Também:

COPY FILE, ERASE, FILE( ), CLOSE, USE, RUN e o comando RENAME do Sistema Operacional.

Comando TYPE:

Sintaxe:

TYPE **<nome do arquivo>.<extensão>**
[TO PRINT]
[TO FILE <nome do arquivo destino>]

Propósito:

Apresentar o conteúdo de um arquivo tipo "texto" (padrão ASCII) na tela, opcionalmente imprimi-lo na impressora ou gravá-lo em um outro arquivo no disco.

Utilização:

Este comando é semelhante ao comando TYPE do DOS, cuja função é exibir o conteúdo de um arquivo-texto padrão ASCII.
O <nome do arquivo> a ser apresentado deve incluir sua extensão e, caso o arquivo não se encontre no drive ou diretório atual, sua designação.
Como são apenas apresentados arquivos-texto, arquivos de outros formatos (arquivo de dados (.dbf), de índice (.ntx), memo (.dbt), objeto (.obj) ou executável (.exe)) não podem ser processados pelo comando TYPE; apenas os arquivos de comandos (os programas (.prg)).
A cláusula TO PRINT redireciona a saída do comando TYPE para a impressora conectada ao equipamento.
A cláusula TO FILE grava a saída do comando TYPE no arquivo cujo nome for especificado. Se não for definida uma extensão, será assumida a extensão padrão (.txt).

As teclas <Ctrl-S> pressionadas simultaneamente permitem interromper momentaneamente a apresentação do arquivo. Para continuar basta pressioná-las novamente.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
TYPE Prog1.prg      && mostra o Prog1.prg na tela

TYPE Prog1.prg TO PRINT      && imprime o Prog1.prg na impressora

TYPE Prog1.prg TO FILE Progx.prg
* grava uma cópia do arquivo Prog1.prg no disco, com o nome de Progx.prg
```

Veja Também:

COPY FILE, RUN e o comando TYPE do Sistema Operacional.

5.8. COMANDOS PARA CÁLCULO SOBRE OS REGISTROS DOS ARQUIVOS DE DADOS

Comando AVERAGE:

Sintaxe:

AVERAGE [<escopo>] <lista de campos / expressões>
TO <lista de variáveis>
[FOR <condição>] [WHILE <condição>]

Propósito:

O comando AVERAGE computa a média aritmética de campos numéricos de um arquivo, sendo o resultado armazenado em variáveis de memória.

Utilização:

Utiliza-se o comando AVERAGE para calcular a média aritmética simples de campos numéricos de registros de arquivos de dados.

Todos os registros (ALL) são incluídos no cálculo da média, a menos que sejam selecionados pelo escopo (NEXT <n> ou RECORD <n>) ou pelas condições das cláusulas FOR e WHILE.
O <escopo> define a quantidade de registros a ser processada: ALL (todos), NEXT <n> (os próximos n registros a partir do atual) ou RECORD <n> (apenas o registro n).
A cláusula FOR especifica uma condição para selecionar os registros que deverão ser processados dentro do escopo determinado. Apenas os registros que a satisfizerem (a tornarem verdadeira) entrarão no cálculo da média.
A cláusula WHILE seleciona os registros a serem processados, a partir do registro atual, enquanto uma condição especificada for satisfeita (permanecer verdadeira). Assim que a condição se tornar falsa (.F.), o cálculo da média é interrompido e o resultado armazenado nas variáveis especificadas.
Deve obrigatoriamente ser especificada a <lista de campos ou expressões> envolvendo os campos do arquivo, que deverão ter suas médias calculadas.
Deve haver uma variável disponível para o armazenamento dos resultados, respectivamente para cada média calculada. Caso as variáveis ainda não existam, serão automaticamente criadas pelo comando AVERAGE.
A lista de campos ou de expressões a serem calculados deverá ter o mesmo número de elementos que a lista de variáveis receptoras do resultado.

Biblioteca:

CLIPPER.LIB

Exemplos:

* Considerando um arquivo de estoque, para calcular o número médio de unidades existentes por produto do estoque e o preço unitário médio, poderíamos utilizar a seguinte rotina:

```
CLEAR
USE Estoque
AVERAGE saldo,custo TO sdmedio,czmedio
@ 10,25 SAY "Saldo Médio ="
@ 10.39 SAY sdmedio PICTURE "@E 999,999"
@ 12,25 SAY "Custo Médio ="
@ 12,39 SAY czmedio PICTURE "@E 999,999.99"
USE
```

* Supondo agora o mesmo cálculo, mas apenas para os produtos que possuírem saldo em estoque maior que 50 unidades:

```
CLEAR
USE Estoque
AVERAGE saldo,custo TO sdmedio,czmedio FOR saldo>50
@ 10,25 SAY "Saldo Médio ="
@ 10,39 SAY sdmedio PICTURE "@E 999,999"
@ 12,25 SAY "Custo Médio ="
@ 12,39 SAY czmedio PICTURE "@E 999,999.99"
USE
```

Veja Também:

SUM, COUNT, TOTAL

Comando COUNT:

Sintaxe:

COUNT [<**escopo**>]
[**FOR** <**condição**>]
[**WHILE** <**condição**>]
TO <**variável**>

Propósito:

O comando COUNT calcula o número de registros de um arquivo, de acordo com um escopo especificado, que satisfaz a uma determinada condição (ou todos). O resultado é armazenado em uma variável de memória.

Utilização:

Utiliza-se o comando COUNT principalmente para se obter contagens "estatísticas" sobre os dados armazenados nos registros de um arquivo de dados.
O nome de uma <variável> deve ser obrigatoriamente especificado. Nesta variável será armazenado o resultado da contagem efetuada pelo comando COUNT.
A menos que não seja especificado pelo escopo ou pela condição das cláusulas FOR ou WHILE, todos os registros serão contados.
O <escopo> define a quantidade de registros a ser processada: ALL (todos), NEXT <n> (os próximos n registros a partir do atual) ou RECORD <n> (apenas o registro n).
A cláusula FOR especifica uma condição para selecionar os registros que deverão ser processados. Apenas os registros que a satisfizerem (a tornarem verdadeira) serão contados.
A cláusula WHILE seleciona os registros a serem processados, a partir do registro atual, enquanto uma condição especificada for satisfeita (permanecer verdadeira). Assim que a condição se tornar falsa (.F.), a contagem é interrompida e o resultado armazenado na variável especificada.
Se a variável que irá armazenar o resultado do comando COUNT ainda não existir, será automaticamente criada.

Biblioteca:

CLIPPER.LIB

Exemplos:

* Para contar o número de produtos do estoque que estão com o saldo abaixo de um certo limite mínimo:

```
CLEAR
USE Estoque
COUNT TO emfalta FOR saldo<50
@ 10,25 SAY "Em Falta... ="
@ 10,39 SAY emfalta PICTURE "@E 999,999"
USE

* Para contar o número de produtos do estoque que estão
com o saldo exagerado, ou seja, acima de um certo
limite ideal:

CLEAR
USE Estoque
COUNT TO excesso FOR saldo>500
@ 10,25 SAY "Em Excesso. ="
@ 10,39 SAY excesso PICTURE "@E 999,999"
USE

* Finalmente, para contar o número de produtos que
estão com o saldo dentro dos limites normais:

CLEAR
USE Estoque
COUNT TO normais FOR saldo>50 .AND. saldo<500
@ 10,25 SAY "Normais.... ="
@ 10,39 SAY normais PICTURE "@E 999,999"
USE
```

Veja Também:

AVERAGE, SUM, TOTAL, LASTREC( ), RECCOUNT( )

Comando SUM:

Sintaxe:

SUM [<escopo>]<lista de campos/expressões>
TO <lista de variáveis>
[FOR <condição>]
[WHILE <condição>]

Propósito:

O comando SUM computa a somatória de expressões numéricas para um determinado conjunto de registros de um arquivo de dados aberto na área de trabalho selecionada, armazenando os resultados em variáveis.

Utilização:

O comando SUM permite calcular somatórias de campos numéricos ou expressões envolvendo campos numéricos de um arquivo de dados e armazenar os resultados em variáveis de memória.
A <lista de campos/expressões> é uma lista de campos numéricos ou de expressões numéricas envolvendo-os. Cada elemento desta lista será somado para cada registro a ser processado do arquivo em uso.
A <lista de variáveis> identifica o nome das variáveis que receberão os resultados da somatória computada. Se não existirem serão automaticamente criadas. Se existirem o conteúdo anterior será sobreposto. Esta lista deve conter o mesmo número de elementos que a lista de expressões ou campos a serem somados.
O <escopo> define a quantidade de registros a ser processada: ALL (todos), NEXT <n> (os próximos n registros a partir do atual) ou RECORD <n> (apenas o registro n). Se o escopo não for especificado todos os registros do arquivo serão somados.
A cláusula FOR especifica uma condição para selecionar os registros que deverão ser processados. Apenas os registros que a satisfizerem (a tornarem verdadeira) serão somados.
A cláusula WHILE seleciona os registros a serem processados, a partir do registro atual, enquanto uma condição especificada for satisfeita (permanecer verdadeira). Os registros serão somados enquanto satisfizerem esta condição.

Biblioteca:

CLIPPER.LIB

Exemplos:

* Para se obter o saldo total de produtos no estoque:

```
CLEAR
USE Estoque
SUM saldo TO numtot
@ 10,25 SAY "Total..... ="
@ 10,39 SAY numtot PICTURE "@E 999,999"
USE
```

* Para se obter o saldo total de produtos da marca "XYZ":

```
CLEAR
USE Estoque
SUM saldo TO numtot FOR marca="XYZ"
@ 10,25 SAY "Total XYZ..."
@ 10,39 SAY numtot PICTURE "@E 999,999"
USE
```

* Para se obter o valor total do estoque:

```
CLEAR
USE Estoque
SUM saldo*preco TO valtot
@ 10,25 SAY "Total..... "
@ 10,39 SAY valtot PICTURE "@E 999,999,999.99"
USE
```

Veja Também:

AVERAGE, COUNT, TOTAL

5.9. COMANDOS DE FLUXO DE EXECUÇÃO (ESTRUTURAS DE PROGRAMAÇÃO)

5.9.1. Estruturas Básicas de Programação Estruturada

O Clipper possui uma linguagem de programação bastante poderosa, contudo, para que se possam criar estruturas adequadas de programação, esta linguagem requer que os comandos sejam dispostos de forma lógica, ordenada e otimizada, com o intuito de obter-se o máximo de sua eficiência.

Esta disposição lógica e ordenada está baseada em estruturas de programação muito simples e eficientes, conhecidas como linguagem estruturada. Nesta classe de linguagem o fluxo de execução é composto por blocos de instruções bem definidos, não existindo comandos que executem desvios incondicionais.

As estruturas básicas estão dispostas em quatro grupos de comandos:

1 — *SEQÜÊNCIA*: constituída pelo fluxo normal de execução dos comandos, isto é, um após o outro, representada pela maioria dos comandos e explicitamente pelo comando BEGIN SEQUENCE...END.

2 — *DESVIO CONDICIONAL*: constituída pelo desvio do fluxo de execução entre dois caminhos alternativos, de acordo com uma condição, representada pelo comando IF...ELSE...ENDIF.

3 — *REPETIÇÃO*: constituída pela repetição controlada de um bloco de instruções dentro do fluxo de execução, representada pelos comandos DO WHILE...ENDDO e FOR... NEXT.

4 — *DESVIO MÚLTIPLO*: constituída pelo desvio do fluxo de execução de comandos de acordo com vários caminhos alternativos, representada pelos comandos DO CASE...CASE...ENDCASE e IF...ELSEIF... ENDIF.

Todas as estruturas de programação do Clipper têm como princípio básico um único ponto de **ENTRADA** e um único ponto de **SAÍDA**, entre os quais é inserido o bloco de instruções. Esta característica as torna muito eficientes e simples do ponto de vista de **programação, manutenção e depuração de erros**. Com estas estruturas pode-se elaborar qualquer tipo de programa.

Normalmente os comandos que formam um bloco de instruções, ou seja, estão contidos entre a entrada e a saída de uma das estruturas de programação sofrem uma **indentação**. São escritos alguns espaços mais para dentro para exatamente caracterizar o bloco de instruções contido dentro de uma das estruturas básicas do Clipper. Esta estética, que poderá ser observada em todos os exemplos do livro, além de tornar os programas mais claros e fáceis de serem lidos, ajuda na manutenção e depuração de erros. **Recomendamos enfaticamente que esta regra seja seguida.**

Exemplo de Indentação

```
DO WHILE <condição>
    <comandos>
    <comandos>
    <comandos>        Bloco de Instruções
    <comandos>
    <comandos>
    <comandos>
ENDDO
```

5.9.1.1. Seqüência de Comandos

A **seqüência simples** de comandos é a estrutura de programação mais natural, pois consiste apenas na colocação dos comandos numa seqüência lógica e precisa. Trabalhando isoladamente os comandos são executados apenas uma única vez, não havendo mudança no fluxo de execução, como ilustrado na Fig. 5.9.1.

Sequencia Simples

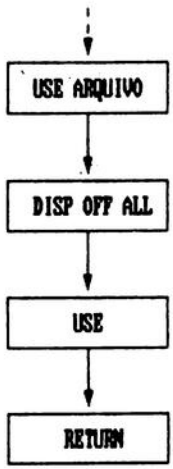


Fig. 5.9.1.

Por exemplo:

```
*** Seqüência.prg
*** Inicio de uma Seqüência
*
*
CLEAR
ACCEPT "Qual e o seu Nome ? " TO vnome
INPUT "Qual e o seu Salario ? " TO vsal
WAIT "Aperte qualquer tecla para continuar..."
CLEAR
@ 8,5 SAY vnome+" "+Cz$ "+STR(vsal)+" é um ótimo salario."
RETURN
*
*** Fim da Seqüência ***
```

Nota: Qualquer linha de um programa que se inicie por um asterisco é ignorada durante a execução.

Uma forma um pouco mais sofisticada é a **seqüência com desvio.** Neste caso os comandos que formam a seqüência são colocados em um bloco delimitado pelo comando BEGIN SEQUENCE...END (Início da Seqüência....Fim). Trabalhando isoladamente os comandos são executados apenas uma única vez, contudo pode haver mudança no fluxo de execução do programa. Se for encontrado o comando **BREAK** a seqüência é interrompida e o fluxo de execução passa para o primeiro comando após o final da seqüência, como esquematizado na Fig. 5.9.2 a seguir.

Sequencia com Desvio

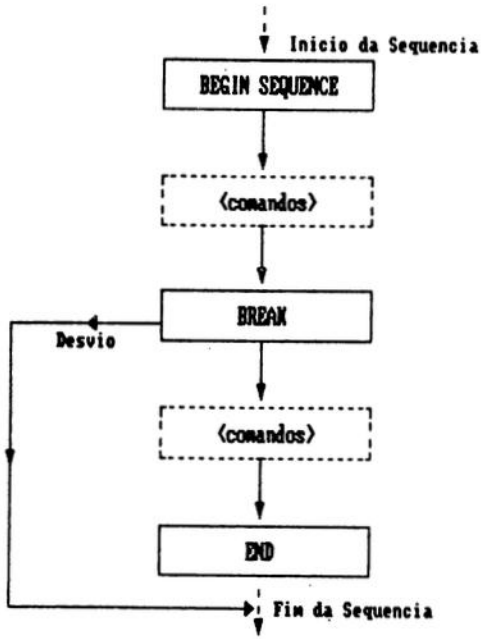


Fig. 5.9.2.

Por exemplo:

```
*** SeqDesvio.prg
*** Inicio de uma Seqüência
BEGIN SEQUENCE
CLEAR
ACCEPT "Qual e o seu Nome ? " TO vnome
IF vnome=""
   BREAK       && o desvio é efetuado caso o nome não seja
ENDIF          && digitado
INPUT "Qual e o seu Salario ? " TO vsal
WAIT "Aperte qualquer tecla para continuar..."
CLEAR
@ 8,5 SAY vnome+" "+Cz$ "+STR(vsal)+" é um ótimo salario."
END
*** Fim da Seqüência ***
RETURN
```

Atenção: Nas versões do Clipper anteriores à Summer 87 o comando BEGIN SEQUENCE...END não está disponível.

5.9.1.2. Desvio Condicional

O comando **IF...ELSE...ENDIF** (SE...SENÃO...FIM DO SE) permite condicionar a execução de uma seqüência de comandos. Esta estrutura de programação é uma decisão entre dois caminhos alternativos, dependendo de uma certa condição especificada ser **verdadeira** ou **falsa**. Após a execução do caminho selecionado, os dois caminhos unem-se novamente, como esquematizado na Fig. 5.9.3.

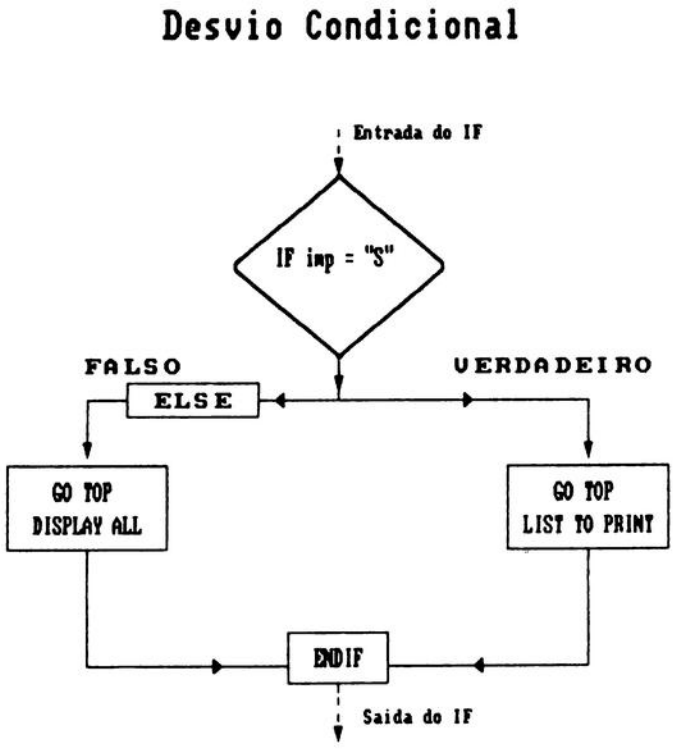


Fig. 5.9.3.

Por exemplo:

```
**** Decisao.prg, Decisão entre Alternativas
*
CLEAR
ACCEPT "Qual e o seu Nome ? " TO vnome
INPUT "Qual e o seu Salário ? " TO vsal
IF vsal>5000.00                  &&   SE vsal maior que 5000
   desconto = vsal * 0.15        &&   Se for verdadeiro
ELSE                             &&   SENÃO
   desconto = vsal * 0.08        &&   Se for falso
ENDIF                            &&   FIM do SE
```

```
WAIT "Digite uma tecla para ver os resultados ..."
CLEAR
@ 5,10 SAY "O seu Salário Bruto é.....Cz$ "
@ 5,43 SAY vsal PICTURE "@E 999,999.99"
@ 7,10 SAY "O valor do seu Desconto é Cz$ "
@ 7,43 SAY desconto PICTURE "@E 999,999.99"
@ 9,10 SAY "O seu Salário Liquido é...Cz$ "
@ 9,43 SAY vsal-desconto PICTURE "@E 999,999.99"
RETURN
```

5.9.1.3. Repetição

Dentro de um programa, muitas vezes temos a necessidade de repetição de uma sequência de comandos por uma quantidade de vezes muitas vezes desconhecida. Esta necessidade requer um comando que execute as instruções até que ocorra uma condição no processamento que o interrompa. Este comando é o **DO WHILE** (Execute Enquanto). Ele executa repetidamente um determinado número de instruções que vierem após o seu início até o seu fim, enquanto uma determinada condição permanecer **verdadeira**. A Fig. 5.9.4 ilustra o funcionamento do comando do DO WHILE...ENDDO.

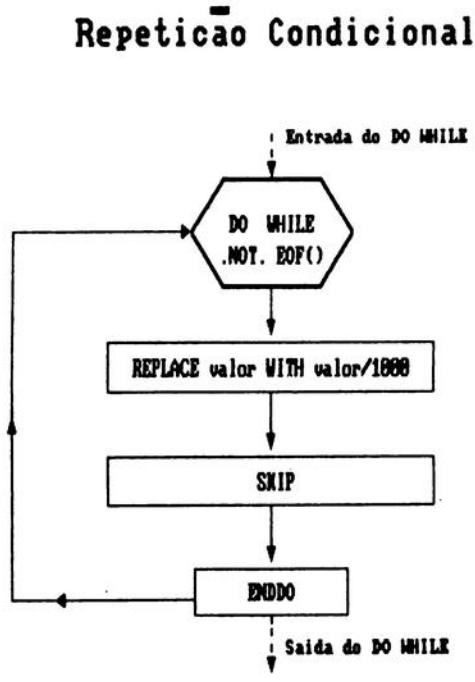


Fig. 5.9.4.

Por exemplo:

```
   *** Repeticao.prg ***
   *** Início do "DO WHILE" ***
   *
   opcao = "S"
   data = CTOD("  /  /  ")
   vnome = SPACE(30)
+->DO WHILE opcao = "S"    -----> Faça enquanto opçao="S"
|     CLEAR
|     @ 5,10 SAY "Qual é o seu Nome ? " GET vnome
|     @ 7,10 SAY "Qual a data do seu nascimento ? " GET data
|     READ  (*)
^     @ 12,15 SAY vnome+" você nasceu numa "+CDOW(data)
|     @ 14,15 SAY "Sua idade é "
|     @ 14,28 SAY INT(DATE()-data)/365
|     @ 14,31 SAY "anos."
|     @ 20,15 SAY "Outra vez (S/N) ?" GET opcao PICTURE "!"
|   READ
+<-ENDDO       ------> Fim da repetição do "DO WHILE"
   CLEAR
   RETURN
```

Em algumas ocasiões poderemos ter a necessidade de interrompera execução dos comandos contidos "dentro" de um DO WHILE (ciclo de repetição) sem que este chegue ao seu final. Por exemplo, no programa acima poderíamos acrescentar os seguintes comandos após o primeiro comando READ (marcado com um asterisco *).

```
IF EMPTY(vnome) .OR. data>=DATE( )
   LOOP
ENDIF
```

Ou seja, SE não for digitado o nome (nome vazio) ou se a DATA do nascimento for maior ou igual à função DATE( ) (data atual), o programa volta ao início do DO WHILE e fica aguardando até que novos dados sejam digitados, pois os dados digitados certamente estão incorretos.

O comando **LOOP** permite a interrupção lógica do comando "DO WHILE", pela volta ao início do mesmo. A estrutura de programação seria, portanto, a esquematizada na Fig. 5.9.5.

Uma outra, forma de interrompermos a repetição do DO WHILE é através do comando EXIT. Este comando funciona de maneira semelhante ao LOOP, só que ao invés de retornar ao início do DO WHILE, a execução dos comandos continua depois do ENDDO, ou seja, o EXIT (saída) faz com que a execução dos comandos saia do elo de repetição do DO WHILE, finalizando-o.

A estrutura de programação seria, portanto, a esquematizada pela Fig. 5.9.6.

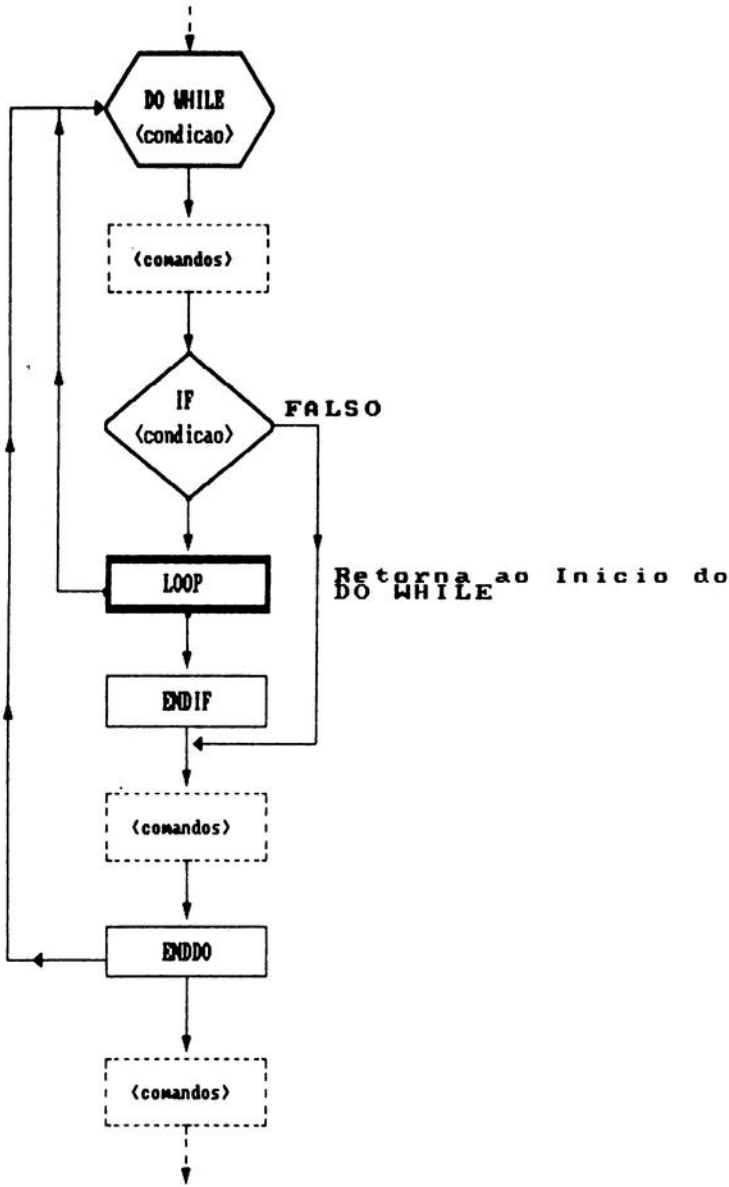

DO WHILE
(condicao)
(comandos)
IF
(condicao)
FALSO
LOOP
Retorna ao Inicio do
DO WHILE
ENDIF
(comandos)
ENDDO
(comandos)

Fig. 5.9.5.

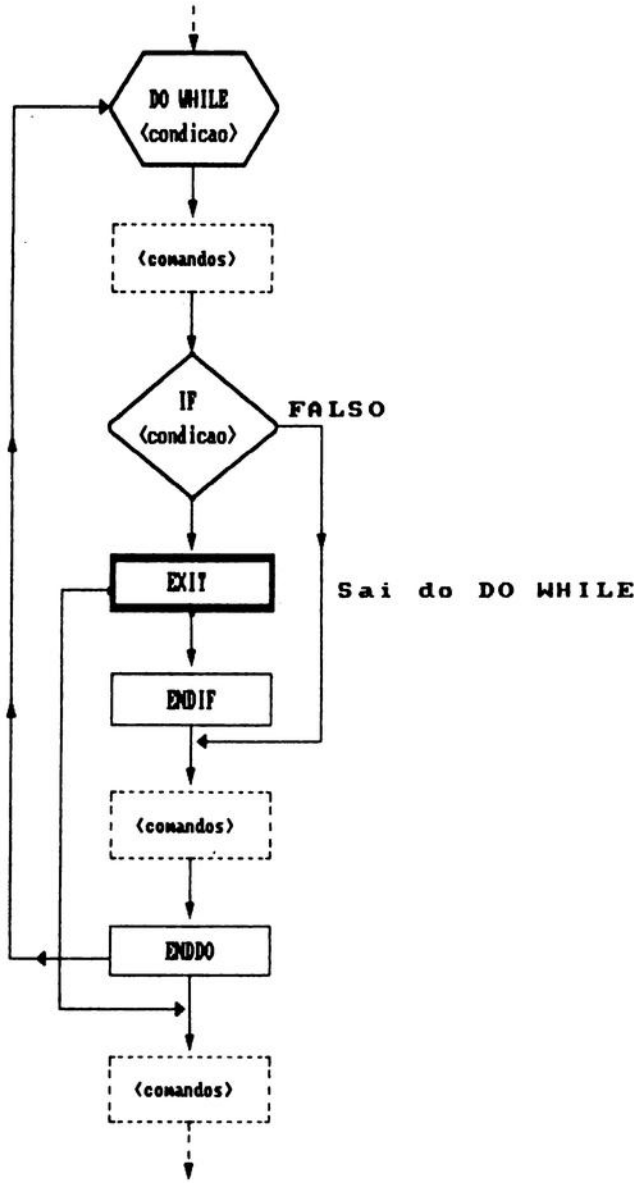

DO WHILE
<condicao>
<comandos>
IF
<condicao>
FALSO
EXIT
Sai do DO WHILE
ENDIF
<comandos>
ENDDO
<comandos>

Fig. 5.9.6.

Em um programa muitas vezes temos também a necessidade de repetição de uma seqüência de comandos por uma determinada quantidade de vezes. Esta necessidade requer um comando que execute um bloco de instruções de acordo com um contador, que controle o número de vezes que o mesmo já foi repetido e finalize a execução-ao atingir o limite previsto. Este comando é o **FOR...NEXT.** Ele executa repetidamente um bloco de instruções que vierem após o seu início até o seu final, enquanto um contador não atingir um limite especificado de acordo com um determinado incremento. A estrutura do comando FOR...NEXT está esquematizada na Fig. 5.9.7 a seguir.

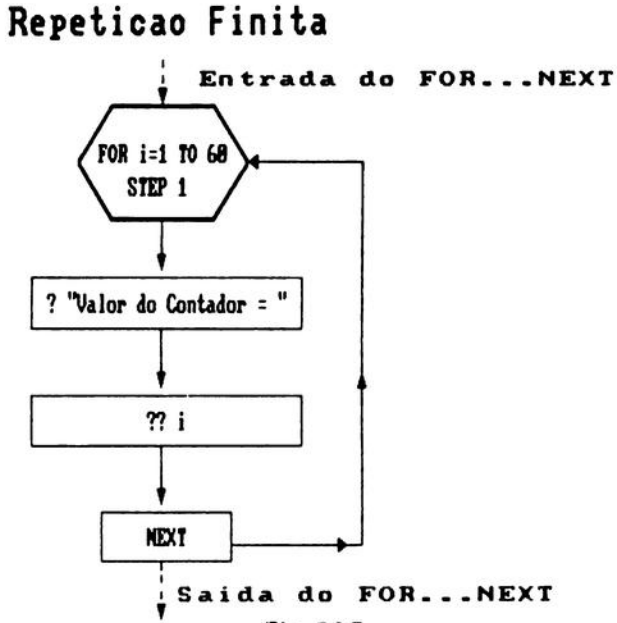


Fig. 5.9.7.

Por exemplo:

```
* Preenche.PRG
* Rotina para preecher os elementos de dois vetores
CLEAR
DECLARE vnum[60],vcar[60]
FOR i=1 TO 60
    vnum[i] = 0
    vcar[i] = SPACE(40)
NEXT
? "Fim"
RETURN
```

O comando FOR...NEXT, assim como o DO WHILE, também aceita o comando EXIT, que interrompe as repetições mesmo que o contador ainda não tenha chegado ao limite especificado.

A estrutura de programação seria a ilustrada pela Fig. 5.9.8.

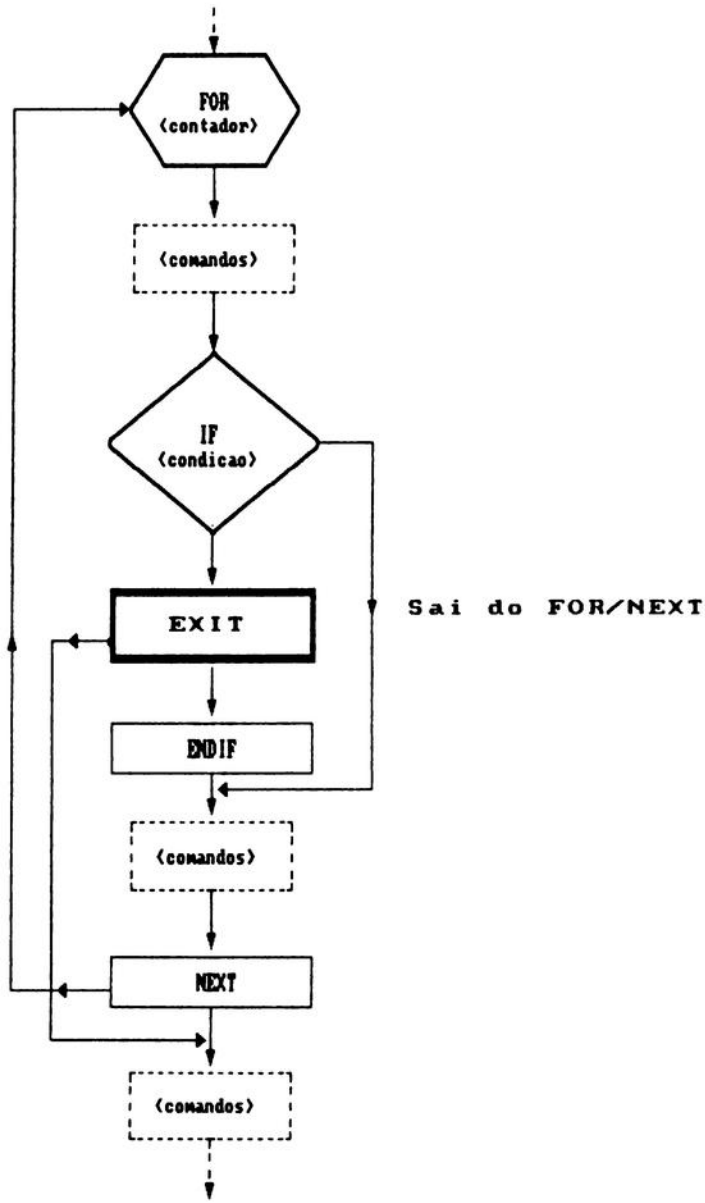

FOR
(contador)
(comandos)
IF
(condicao)
EXIT
Sai do FOR/NEXT
ENDIF
(comandos)
NEXT
(comandos)

Fig. 5.9.8.

5.9.1.4. Desvio Múltiplo

Em alguns programas há a necessidade de se escolher (tomar uma decisão) entre diversos caminhos alternativos para o fluxo de execução, dependendo de determinadas condições. Os comandos **DO CASE...ENDCASE** e **IF...ELSEIF... ENDIF** permitem a tomada de decisão entre três ou mais alternativas possíveis.

Quando a escolha do caminho a ser seguido pelo fluxo de controle envolve mais de duas alternativas, é conveniente o uso da estrutura de "Desvio Múltiplo" ou "Caso". A escolha é feita dependendo da condição de cada alternativa e apenas uma das alternativas (a que primeiro atender sua condição) é executada.
A estrutura do comando "Caso" é a ilustrada pela Fig. 5.9.9, a seguir.

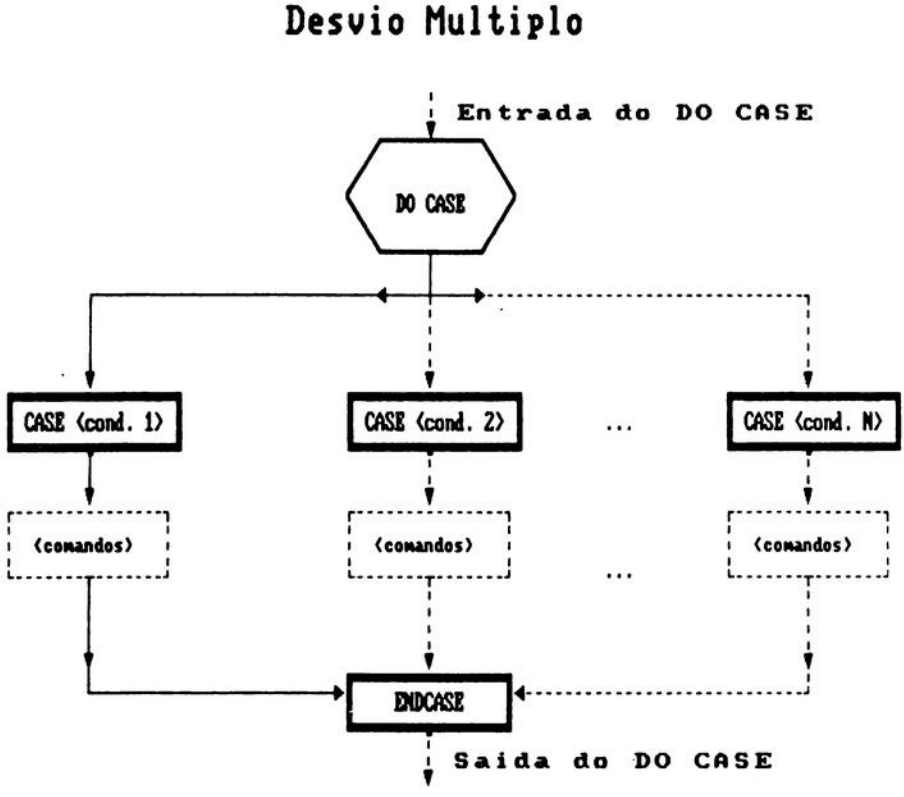


Fig. 5.9.9.

Por exemplo, o programa a seguir representa um simples "Menu de Opções" para a escolha de diferentes tarefas, e é o exemplo mais típico da utilização do comando DO CASE...ENDCASE:

```
*** Caso.prg --> Menu de Opções ***.
    STORE " " TO opcao
+-> DO WHILE opcao <> "5"
¦       CLEAR
¦       STORE " " TO opcao
¦       @  8,20 SAY "MENU PRINCIPAL"
¦       @  9,20 SAY "--------------"
¦       @ 12,15 SAY "1 - Inclusão de registros"
¦       @ 14,15 SAY "2 - Alteração de registros"
¦   ¦¦  @ 16,15 SAY "3 - Exclusão de registros"
¦   ¦¦  @ 18,15 SAY "4 - Emissão de Relatorios"
¦   ¦¦  @ 20,15 SAY "5 - Fim das operações"
^   \/  @ 23,15 SAY "Qual e a sua opção ? " GET opcao
¦       READ
¦       DO CASE
¦          CASE opcao = "1"
¦               DO Inclusao
¦          CASE opcao = "2"
¦               DO Alteracao
¦          CASE opcao = "3"
¦               DO Exclusao
¦          CASE opcao = "4"
¦               DO Relatorio
¦          CASE opcao = "5"
¦               EXIT--------+
¦          OTHERWISE        ¦
¦<--------------LOOP         ¦
¦      ENDCASE               v
+-<-ENDDO                    ¦
    CLEAR <------------------+
    RETURN
```

A cláusula **OTHERWISE** (de outra forma) é opcional e significa que caso nenhuma das condições for satisfeita, devem ser executados os comandos que vierem entre ela e o ENDCASE (Fim do CASE).

O Desvio Múltiplo **IF...ELSEIF...ENDIF** é uma forma alternativa, pois possui uma estrutura e resultados exatamente equivalentes à estrutura e aos resultados do comando DO CASE...CASE... ENDCASE. A Fig. 5.9.10 ilustra essa equivalência.

O mesmo exemplo anterior pode ser construído utilizando-se a estrutura do Desvio Múltiplo:

```
*** Elseif.PRG --> Menu de Opções ***
    STORE " " TO opcao
+-> DO WHILE opcao <> "5"
¦       CLEAR
¦       STORE " " TO opcao
¦       @  8,20 SAY "MENU PRINCIPAL"
¦       @  9,20 SAY "--------------"
¦       @ 12,15 SAY "1 - Inclusão de registros"
¦       @ 14,15 SAY "2 - Alteração de registros"
¦   ¦¦  @ 16,15 SAY "3 - Exclusão de registros"
¦   ¦¦  @ 18,15 SAY "4 - Emissão de Relatórios"
¦   ¦¦  @ 20,15 SAY "5 - Fim das operações"
^   \/  @ 23,15 SAY "Qual e a sua opção ? " GET opcao
```

```
        READ
        IF opcao = "1"
                DO Inclusao
        ELSEIF opcao = "2"
                DO Alteracao
        ELSEIF opcao = "3"
                DO Exclusao
        ELSEIF opcao = "4"
                DO Relatorio
        ELSEIF opcao = "5"
                EXIT
        ELSE
        LOOP
        ENDIF
    ENDDO
    CLEAR
    RETURN
```

Desvio Multiplo

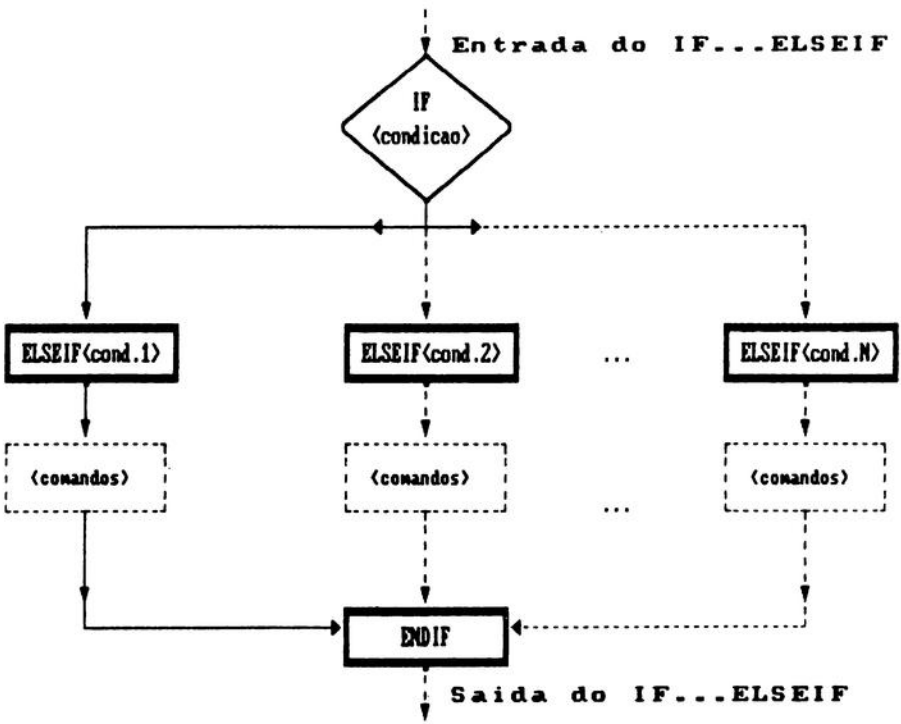


Fig. 5.9.10.

A cláusula **ELSE** (senão) é opcional e significa que se caso nenhuma das condições for satisfeita devem ser executados os comandos que vierem entre ela e o ENDIF (Fim do Desvio Múltiplo).

ATENÇÃO: A estrutura de Desvio Múltiplo foi implementada no Clipper apenas a partir da versão Summer 87.

5.9.2. Comandos Estruturados

Comando &&:

Sintaxe:

&&

Propósito:

O comando && permite que sejam colocados comentários em um programa, na mesma linha que os comandos.

Utilização:

É sempre recomendável que os programas contenham vários comentários, para torná-los mais claros e facilitar eventuais alterações e manutenções. Comentários são explicações sobre os comandos, colocadas pelo programador, que não devem ser consideradas como instruções pelo computador, mas apenas como uma espécie de documentação interna ao próprio programa.
O comando && permite que comentários sejam colocados na própria linha dos comandos. Para isso basta que os comentários sejam incluídos após os comandos e precedidos por &&.

Biblioteca:

CLIPPER.LIB

Exemplos:

Comandos	*Comentários*
USE Estoque INDEX IndCod	&& Abre o arquivo do Estoque
CLEAR	&& Limpa a Tela
DISPLAY ALL	&& Mostra todos os registros
USE	&& Fecha o Arquivo do Estoque
RETURN	&& Retorna ao DOS

Veja Também:

NOTE ou *.

Comando BEGIN SEQUENCE...END:

Sintaxe:

```
BEGIN SEQUENCE
        .
        .
   <comandos>
        .
        .
     BREAK
        .
        .
   <comandos>
        .
        .
END
```

Propósito:

Definir uma estrutura seqüencial de controle dentro do fluxo de um programa.
Disponível a partir da versão Summer 87.

Utilização:

O comando BEGIN SEQUENCE...END é uma estrutura de controle que facilita a construção de sistemas de tratamento de exceções. Quando uma exceção ocorre, o comando BREAK desvia o fluxo de execução do programa para o comando imediatamente seguinte ao END, que termina a seqüência definida. O comando BREAK pode ocorrer no nível da própria seqüência ou em qualquer nível de comandos ou rotinas intercaladas.
Não há limitação de rotinas que podem ser intercaladas abaixo de uma estrutura BEGIN SEQUENCE...END. Se em uma delas for encontrado o comando BREAK a seqüência definida é finalizada.
A estrutura BEGIN SEQUENCE...END é especialmente útil para o desenvolvimento de rotinas para o tratamento de erros. O comando BREAK deve ser utilizado toda vez que ocorrer uma exceção (erro) para finalizar a seqüência.

Biblioteca:

CLIPPER.LIB

Exemplo:

A estrutura típica do comando BEGIN SEQUENCE é a seguinte:

```
BEGIN SEQUENCE
        .
        .
   <comandos>
      .
      .
IF <condição de exceção>
    BREAK
ENDIF
        .
        .
   <comandos>
        .
        .
END
     .
<comandos para recuperação da exceção>
```

Comando CALL:

Sintaxe:

CALL <rotina>
WITH <lista de parâmetros>
[WITH WORD (<exp.numérica>)]

Propósito:

O comando CALL permite que sejam "chamadas" rotinas compiladas separadamente em outras linguagens ou rotinas e programas em Assembler a serem executadas e utilizadas "dentro" do Clipper.

Utilização:

As rotinas chamadas pelo comando CALL, normalmente realizam operações não disponíveis no Clipper. Com o uso deste comando ganha-se muita flexibilidade, praticamente eliminando-se qualquer limitação que possa ser encontrada no Clipper.

As rotinas a serem chamadas, que deverão estar disponìveis ao link-editor durante a operação de encadeamento, podem ser escritas em qualquer linguagem compilada, como por exemplo "C" ou Assembler.
Recomenda-se, entretanto, que a utilização desse comando seja feita apenas em caso de real necessidade, sendo preferível encontrar soluções dentro do próprio Clipper. Mesmo assim, não utilize este comando a menos que você seja um programador experimentado em "C", Assembler ou outra linguagem qualquer.
As rotinas chamadas pelo comando CALL devem seguir estritamente as seguintes regras:

- Devem estar em arquivos de formato objeto relocável para o processador INTEL 8086, com a extensão (.obj).
- Devem seguir as regras e convenções de passagem de parâmetros e chamadas de funções da linguagem C (o Clipper é escrito em C).
- Devem estar disponíveis ao link-editor durante a operação de encadeamento – link-edição.

Quando for chamada uma rotina escrita em outra linguagem de alto-nível (qualquer linguagem que não o Assembler), a biblioteca do compilador desta linguagem deverá estar disponível ao link-editor. Veja o manual do compilador da linguagem utilizada e o Cap. 8, para maiores detalhes.
A lista de parâmetros do comando CALL pode conter até sete parâmetros e sua passagem segue as mesmas regras da passagem de parâmetros da linguagem "C".
Vetores inteiros não podem ser passados como parâmetros para rotinas em "C" ou Assembler. O vetor, contudo, pode ser passado por valor, um elemento de cada vez.
Em geral, a menos que você seja um "expert", devem ser utilizadas rotinas já prontas, como as disponíveis no dBASE Tools (veja a seção 9.3 – Outros Utilitários), pois as regras de utilização deste comando são extremamente complexas e estão além do escopo deste texto, pois exigem conhecimento aprofundado da linguagem "C" e de Assembler.

Biblioteca:

CLIPPER.LIB

Veja Também:

DO WITH, WORD( )

Comando CANCEL:

Sintaxe:

CANCEL

Propósito:

O comando CANCEL encerra a execução do programa e retorna o controle ao Sistema Operacional (DOS), fechando todos os arquivos que estiverem abertos.

Utilização:

Utiliza-se este comando em uma rotina ou programa intermediário quando ocorre alguma falha grave de processamento para cancelar a execução do sistema.
Após ser encontrado o comando CANCEL, o Clipper ignorará qualquer outro comando e finalizará a execução do programa imediatamente fechando todos os arquivos que estiverem abertos.
Um comando RETURN encontrado na rotina ou programa de maior nível (o programa principal) causa o mesmo efeito que o comando CANCEL.

Biblioteca:

CLIPPER.LIB

Exemplo:

Para cancelar a execução de um programa em caso de erro, inclua o comando CANCEL dentro de uma estrutura IF...ENDIF ou DO CASE...ENDCASE. Por exemplo:

```
CLEAR
IF FILE ("Estoque.dbf") && a função FILE( ) retorna
   USE Estoque          && verdadeira se o arquivo
ELSE                    && existir.
?
? "Erro – Arquivo Não Encontrado!"
CANCEL
ENDIF
```

Veja Também:

RETURN, QUIT

Comando DO:

Sintaxe:

DO <nome do programa / rotina>
[WITH <(lista de parâmetros)>]

Propósito:

o comando DO (execute) executa uma rotina ou programa e indica para o compilador do Clipper, quais as rotinas ou programas devem ser compiladas. O comando DO também permite a passagem de parâmetros (dados) necessários à execução da rotina invocada.

Utilização:

O comando DO invoca um programa ou rotina a partir do programa em execução, permitindo também a passagem de parâmetros esperados pela rotina ou programa chamado.
O <nome do programa / rotina> é o nome do programa ou rotina a ser executada. Estes poderão ser escritos em Clipper, linguagem "C" ou Assembler. Quando um programa ou rotina chamados pelo comando DO são finalizados, o controle do fluxo de execução retorna ao programa que os chamou, exatamente na próxima linha do programa a ser executada, após o comando DO.
A opção WITH permite a passagem de até 128 parâmetros ao programa ou rotina chamados. A lista de parâmetros pode conter qualquer expressão válida do Clipper. Estas expressões serão avaliadas e apenas o seu resultado será passado. Variáveis poderão ser passadas por valor ou por referência.
As seguintes regras aplicam-se quando são passados parâmetros às rotinas ou programas chamados:

1 – Para receber os parâmetros passados, a rotina ou programa chamado, através do comando DO, deverá possuir em sua primeira linha o comando PARAMETERS e a respectiva lista variáveis para armazenar os parâmetros que estão sendo passados.
2 – Parâmetros são passados por referência às rotinas e programas apenas se for diretamente especificado o nome de uma variavel (identificador do parâmetro). Uma expressão, campo de registro

ou variável de mamória cercada por parêntesis são passados apenas por valor.

3 — Um parâmetro que seja passado por referência pode ser alterado (ter seu conteúdo modificado) pela rotina chamada. Um parâmetro passado por valor é uma duplicata do parâmetro original. Mudanças nesta duplicata não afetarão o valor do original.

Veja o comando PARAMETERS para mais detalhes sobre a passagem de parâmetros a rotinas e programas.

Durante a compilação, quando o Clipper encontra o comando DO e a rotina ou programa a ser executado ainda não foi compilado, ele irá procurar no diretório atual por um arquivo com a extensão (.prg) com o mesmo nome da rotina ou programa chamado e em seguida compilá-lo se este for encontrado. Caso este arquivo (.prg) não seja encontrado no disco será assumido como externo. Quando for feita a operação de "link-edição" todos os programas e rotinas deverão estar disponíveis (em código-objeto) ao link-editor. Para maiores detalhes sobre a compilação e a "link-edição", veja o Cap. 8.

Biblioteca:

CLIPPER.LIB

Exemplos:

O programa exemplificado a seguir é um Menu de Opções, que dependendo da escolha do usuário executa uma determinada rotina:

```
*** MENU.PRG --> Menu de Opções ***
    STORE " " TO opcao
    DO WHILE opcao <> "5"
        CLEAR
        STORE " " TO opcao
        @  8,20 SAY "MENU PRINCIPAL"
        @ .9,20 SAY "--------------------"
        @ 12,15 SAY "1 - Inclusão de registros"
        @ 14,15 SAY "2 - Alteração de registros"
        @ 16,15 SAY "3   Exclusão de registros"
        @ 18,15 SAY "4 - Emissão de Relatórios"
        @ 20,15 SAY "5 - Fim das operações"
        @ 23,15 SAY "Qual e a sua opção ? " GET opcao
        READ
        DO CASE
           CASE opcao = "1"
              DO Inclusao  && Executa a rotina de Inclusão
           CASE opcao = "2"
              DO Alteracao && Executa a rotina de Alteração
           CASE opcao = "3"
              DO Exclusao  && Executa a rotina de Exclusão
```

```
        CASE opcao = "4"
           DO Relatorio && Executa a rotina de Relatórios
        CASE opcao = "5"
           EXIT
        OTHERWISE
             DO Mensagem WITH "Opção Inválida !",5
             * Executa a rotina Mensagem  passando como
             * parâmetros o texto da mensagem e o tempo
             * que ela deve ser apresentada.
             LOOP
   ENDCASE
ENDDO
CLEAR
RETURN
```

```
MENSAGEM.prg
* Programa Mensagem.prg
* Mostra uma mensagem  centralizada na linha 24 da tela
* por um determinado tempo, de acordo com os parâmetros
* TEXTO e TEMPO passados.
*
PARAMETERS texto,tempo
SET COLOR TO W+
@ 24,40-(LEN(texto)/2) SAY texto
SET COLOR TO W
* Para "dar um tempo"
FOR i=1 TO tempo*100
NEXT
@ 24,00 CLEAR
RETURN
```

O exemplo a seguir mostra como passar a variável "var 1" para a rotina ROT1 por referência. Qualquer mudança na rotina chamada muda o valor original da variável var1, pois este parâmetro é um identificador.

```
DO ROT1 WITH var1
```

o exemplo a seguir passa o valor (resultado) da expressão "var1 + 1" e o valor do elemento alfa [1] para a rotina ROT2.

```
DO ROT2 WITH var1+1 , alfa[1]
```

Para se passar apenas o valor de uma variável (não passando-a por referência) deve-se colocá-la entre parêntesis. Dessa forma uma alteração no parâmetro passado não alterará o seu valor original.

```
DO ROT3 WITH (var1)
```

Se for passado apenas o nome de um vetor, a rotina chamada terá acesso a todos os elementos do vetor, e qualquer alteração no vetor passado altera o vetor original. A passagem será por referência.

```
DO ROT4 WITH vetor
```

Um vetor inteiro não pode ser passado como parâmetro para rotinas em C ou Assembler, através do comando CALL.

Veja Também:

PARAMETERS, PRIVATE, PROCEDURE, PUBLIC, RETURN, SET PROCEDURE

Comando DO CASE:

Sintaxe:

```
DO CASE
   CASE <condição>
            .
        <comandos>
            .
   CASE <condição>
            .
        <comandos>
            .
            .
            .
    OTHERWISE
ENDCASE
```

Propósito:

O comando DO CASE...ENDCASE é um comando de programação estruturada que seleciona apenas um caminho entre um conjunto de caminhos alternativos para o curso do fluxo de execução. O caminho selecionado é o primeiro para o qual uma condição especificada for avaliada como verdadeira.

Utilização:

O comando DO CASE deve ser utilizado dentro do fluxo de execução dos programas para determinar a escolha de um, entre vários caminhos alternativos a serem executados. O caminho executado será aquele que primeiro for avaliado como verdadeiro dentro das condições CASE especificadas.
É obrigatório a existência do ENDCASE (fim do DO CASE) ao final da estrutura do comando DO CASE.

A cláusula OTHERWISE é um caminho alternativo caso nenhum CASE tenha sido avaliado como verdadeiro, isto é, se todos os CASES forem falsos e existir o OTHERWISE, serão executados os comandos que estiverem após OTHERWISE e antes do ENDCASE.
Comandos DO CASE intercalados, isto é, uns dentro dos outros, são permitidos, desde que intercalados corretamente.
Ao se incluir pares de outras estruturas de comandos como DO CASE...ENDCASE, IF...ENDIF, DO WHILE...ENDDO e FOR...NEXT dentro de um DO CASE principal, estes devem estar corretamente intercalados, isto é, devem iniciar-se e finalizar-se dentro da estrutura DO CASE principal. Não há limitações para o número de estruturas de comandos que podem ser intercaladas.
A cláusula CASE <condição> define uma declaração condicional a ser avaliada como falsa ou verdadeira. Esta condição deve ser uma expressão lógica como A=B ou var1<∅ etc. Não há limite para o número de CASE's que podem existir dentro de uma estrutura DO CASE...ENDCASE.
Quando a condição de um CASE é avaliada como verdadeira, todos os comandos que a seguem, até o próximo CASE, OTHERWISE ou ENDCASE são executados. Neste caso, mais nenhum outro CASE será avaliado. A seguir o fluxo de execução passará diretamente para o primeiro comando após o ENDCASE.
Se nenhum CASE for avaliado como verdadeiro e não existir a cláusula OTHERWISE, o fluxo de execução do programa passa diretamente para a primeira instrução após o ENDCASE.
Em situações onde pode haver apenas uma alternativa (ou um caminho) correto (condição avaliada como verdadeira), ou onde apenas a primeira condição verdadeira deva ser processada, o uso do DO CASE é preferível ao uso do comando IF...ENDIF.
Na versão Summer 87, o novo comando IF...ELSEIF...ENDIF pode ser utilizado como substituto do comando DO CASE...ENDCASE; seu funcionamento é equivalente.
A estrutura do DO CASE...ENDCASE deve ser sempre utilizada onde houver um certo número de exceções à uma regra de execução a ser seguida. Os CASE <condição> representarão estas exceções, e o OTHERWISE a situação mais comum.

Biblioteca:

CLIPPER.LIB

Exemplos:

* A rotina a seguir decide sobre o desconto a ser efetuado sobre o salário dos funcionários, de acordo com sua faixa salarial:

```
CLEAR
USE Folhapg INDEX IndNum
DO WHILE .NOT. EOF()
   DO CASE
      CASE salario > 12000 .AND. salario < 20000
           desconto = salario * 0.10
      CASE salario >= 20000 .AND. salario < 40000
           desconto = salario * 0.15
      CASE salario >= 40000 .AND. salario < 60000
           desconto = salario * 0.20
      CASE salario >= 60000 .AND. salario < 90000
           desconto = salario * 0.25
      CASE salario >= 90000 .AND. salario <15000
           desconto = salario * 0.30
      CASE salario >= 150000 .AND. salario < 200000
           desconto = salario * 0.40
      OTHERWISE
           desconto = salario * 0.50
   ENDCASE
   REPLACE desconto WITH desconto - salario/100
   SKIP
ENDDO
USE
RETURN
```

Veja Também:

DO, DO WHILE, IF..ELSE..ENDIF, IF...ELSEIF...ENDIF, FOR...NEXT, IF( )/IIF( ) e a Seção 5.9.1– Estruturas Básicas de Programação Estruturada.

Comando DO WHILE:

Sintaxe:

DO WHILE <condição>

.

<comandos>

.

LOOP

.

<comandos>

.

EXIT

.

<comandos>

.

ENNDO

Propósito:

O DO WHILE é um comando de programação estruturada que permite que instruções contidas entre o DO WHILE e o associado ENDDO sejam repetidas enquanto uma determinada condição permanecer verdadeira. Normalmente este tipo de estrutura de programação é chamado de "loop" ou repetição.

Utilização:

O comando DO WHILE deve ser utilizado sempre que for necessária, dentro do fluxo de execução dos programas, a repetição de um bloco de instruções por um determinado número de vezes (em geral desconhecido), enquanto uma condição especificada permanecer verdadeira.
A estrutura de repetição DO WHILE deve obrigatoriamente ser finalizada por um ENDDO.
A <condição> é uma "expressão lógica" para o controle do número de vezes que o bloco de comandos deverá ser repetido. Ela será avaliada cada vez que o DO WHILE for repetido. Se for verdadeira os comandos entre o DO WHILE e o ENDDO serão executados, caso contrário o fluxo de execução passará para o primeiro comando após o ENDDO.
Qualquer estrutura de comandos dentro de um DO WHILE, como outro DO WHILE...ENDDO, um IF...ENDIF, um DO CASE...ENDCASE ou um FOR...NEXT, deve ser corretamente intercalada, isto é, deve iniciar-se e finalizar-se dentro do DO WHILE...ENDDO principal. Não há limites para o número de comandos ou estruturas que podem existir dentro de um DO WHILE.
A intercalação de vários DO WHILE's estará correta desde que cada um tenha seu correspondente ENDDO e esteja sempre contido dentro do DO WHILE de maior nível. Por exemplo:

```
DO WHILE (1)
.    DO WHILE (2)
.    .
.    .
.    .    DO WHILE (3)
.    .    .
.    .    .
.    .    ENDDO (3)
.    ENDDO (2)
ENDDO (1)
```

O uso da função macro-substituição & é permitido na substituição de condições do DO WHILE, mesmo de forma recursiva (mudando o nome da variável

a ser substituída) ou em qualquer outro lugar dentro do bloco de comandos entre o DO WHILE e o ENDDO.
Qualquer comentário após o ENDOO, na mesma linha, pode ser inserido para documentá-lo, pois será ignorado pelo Clipper.

Opções:

O comando EXIT, se colocado dentro de um DO WHILE...ENDDO imediatamente finaliza o DO WHILE e transfere o fluxo de execução do programa para a primeira linha de instruções após o ENDDO.
O comando LOOP, se colocado dentro de um DO WHILE...ENDDO imediatamente interrompe a execução dos comandos e retorna o fluxo de execução novamente para o início do DO WHILE, ou seja, reinicia uma nova repetição.

Funcionamento:

A declaração DO WHILE <condição> abre a estrutura de repetição, que processa os comandos subseqüentes apenas **enquanto** a condição permanecer verdadeira. Essa condição é uma expressão lógica como por exemplo: .NOT. EOF( ), ou var1>100 .OR. var2>50, ou var3<=50 .AND. var3>=10.
Se a condição for avaliada como sendo verdadeira, todos os comandos subseqüentes são executados até ser atingido o ENDOO ou um LOOP. O controle, então, retorna ao início do comando DO WHILE para novamente avaliar a condição e proceder às repetições. Se for encontrado um EXIT, o DO WHILE é finalizado e o fluxo de execução passa para a primeira linha de instruções após o ENDDO.
Se a condição for avaliada como falsa (em geral após o bloco de instruções contido entre o DO WHILE e o ENDDO ter sido repetido algumas vezes), o fluxo de execução do programa passa diretamente para a primeira instrução após o ENDDO, finalizando a repetição.

Biblioteca:

CLIPPER.LIB

Exemplo:

* Exemplo de um relatório de Produtos do Estoque:

```
*************************
* Relatorio do Estoque
*************************
USE Estoque INDEX Indcod
pg=0
```

```
SET DEVICE TO PRINT
* Repetição enquanto não for atingido o final do arquivo
DO WHILE .NOT. EOF()
.  @       01,01  SAY "Emissao:"+DTOC(DATE())
.  @       01,118 SAY "Pagina:"+STR(pg,3,0)
.  @ PROW()+1,00  SAY REPLICATE("-",132)
.  @ PROW()+1,10  SAY "CODIGO"
.  @ PROW(),  30  SAY "NOME PRODUTO"
.  @ PROW(),  67  SAY "ESTOQUE"
.  @ PROW(),  79  SAY "UNIDADE DE"
.  @ PROW(),  94  SAY "UNIDADE DE"
.  @ PROW(), 104  SAY "FATOR  DE"
.  @ PROW()+1,68  SAY "MINIMO"
.  @ PROW(),  79  SAY "MOVIMENTACAO"
.  @ PROW(),  94  SAY "COMPRA"
.  @ PROW(), 104  SAY "CONVERSAO"
.  @ PROW()+1,00  SAY REPLICATE("-",132)
.  * Repetição enquanto não for atingido o final da página
.  * do formulário ou o final do arquivo.
.  DO WHILE PROW()<58 .AND. .NOT. EOF()
.  .  @ PROW()+1,10  SAY codigo PICTURE "@R 11-9999"
.  .  @ PROW(),  22  SAY nome
.  .  @ PROW(),  67  SAY estmi PICTURE "@E 999,999"
.  .  @ PROW(),  79  SAY unidm
.  .  @ PROW(),  94  SAY unidr
.  .  @ PROW(), 109  SAY fator PICTURE "@E 9,999.9999"
.  .  SKIP
.  ENDDO    Fim do segundo DO WHILE
ENDDO       Fim do primeiro DO WHILE
@ PROW()+2,0 SAY REPLICATE("=",132)
EJECT
USE
SET DEVICE TO SCREEN
CLEAR
RETURN
```

* Utilização do comando LOOP:

```
DO WHILE <Condição>
          .
          .
  <comandos iniciais>
          .
          .
  IF <condição específica>
     LOOP
  ENDIF
          .
          .
  <comandos finais>
          .
          .
ENDDO
```

* Utilização do comando EXIT

```
DO WHILE <condição>
          .
          .
  <comandos iniciais>
          .
          .
  IF <condição específica>
    EXIT
ENDIF
          .
          .
  <comandos finais>
          .
          .
ENDDO
```

* Exemplo para busca seqüencial em um arquivo de dados que possua valores repetidos:

```
LOCATE FOR <condição>
DO WHILE FOUND( )
      .
      .
<comandos>
      .
      .
  CONTINE
ENDDO
```

Veja Também:

LOOP, EXIT, RETURN e a seção 5.9.1.– Estruturas Básicas de Programação Estruturada.

Comando EXIT:

Sintaxe:

```
EXIT
```

Propósito:

O comando **EXIT** causa a saída de um ciclo de repetição DO WHILE...ENDDO, transferindo o fluxo de execução do programa para o primeiro comando imediatamente seguinte ao ENDDO.

Utilização:

Utiliza-se o comando **EXIT** para forçar a finalização de um DO WHILE, mesmo que a condição de repetição ainda permaneça verdadeira.

Exemplo:

* A rotina a seguir inclui repetidamente registros no arquivo Mala.dbf. Para finalizar a inclusão basta deixar o último nome em branco. A finalização do DO WHILE é feita pelo comando EXIT:

```
   USE Mala INDEX Mala
   CLEAR
   @ 03,10 TO 20,70 DOUBLE
   DO WHILE .T. && a condição sempre será verdadeira
      vnome=SPACE(40)
      vendereco=SPACE(40)
      vcidade=SPACE(20)
      vestado=SPACE(2)
      vcep=SPACE(5)
      @ 05,34 SAY "Mala-Direta"
      @ 07,27 SAY "Entrada/Alteração de Dados"
      @ 08,27 SAY "--------------------------"
      @ 11,15 SAY "Nome....:" GET vnome      PICTURE "@!"
      @ 13,15 SAY "Endereço:" GET vendereco PICTURE "@!"
      @ 15,15 SAY "Cidade..:" GET vcidade    PICTURE "@!"
      @ 17,15 SAY "Estado..:" GET vestado    PICTURE "!!"
      @ 17,30 SAY "Cep:"      GET vcep       PICTURE "99999"
      READ
      IF EMPTY(vnome)  && se o nome estiver vazio
+--(---- EXIT           && finaliza-se o DO WHILE
|     ENDIF
|     APPEND BLANK
|     REPLACE nome    WITH vnome,endereco WITH vendereco
|     REPLACE cidade  WITH vcidade,estado WITH vestado
|     REPLACE cep     WITH vcep
|  ENDDO
+-)CLEAR
   USE
   RETURN
```

Comando EXTERNAL:

Sintaxe:

EXTERNAL <lista de rotinas>

Propósito

Este comando deve ser utilizado para declarar um símbolo, ou seja, uma rotina externa, para o link-editor. Rotinas que tenham sido declaradas externas a um programa podem ser colocadas em "overlays" e ainda serem chamadas através da função macro-substituição (&).

Utilização:

Utiliza-se o comando EXTERNAL para declarar rotinas externas ao Clipper para serem encadeadas pelo link-editor.
A <lista de rotinas> é uma lista do nome das rotinas, funções-de-usuário ou arquivos de formatação separados por vírgulas, a serem incluídos na tabela de símbolos do Compilador.
Rotinas (contidas em um arquivo de procedures), funções-de-usuário e rotinas invocadas pelo comando SET KEY devem ser declaradas como externas caso sejam chamadas através de macro-substituição ou colocadas dentro de overlays.
Outros exemplos de rotinas a serem declaradas são as da biblioteca adicional de rotinas em "C" ou Assembler que acompanha o Clipper na versão Autunm 86 e anteriores: as das funções ISPRINTER( ), HEADER( ), DISKSPACE( ), RECSIZE( ) etc. Na versão Summer 87, todas estas rotinas já fazem parte da biblioteca EXTEND.LIB do Clipper, não sendo mais necessário declará-las como externas.
Quando for feita a link-edição, os arquivos-objeto que contenham as rotinas declaradas externas devem estar disponíveis no link-editor.

Biblioteca:

CLIPPER.LIB

Exemplo:

* Declaração das Rotinas Externas:

```
CLEAR
USE Mala.dbf INDEX IndCod
EXTERNAL ISPRINTER(),HEADER(),DISKSPACE()
```

* A partir da declaração estas funções já podem ser utilizadas normalmente, desde que o módulo-objeto que as contêm esteja disponível ao link-editor durante a operação de link-edição.

```
EXTERNAL Rot1,Rot2,Rot3
FOR i=1 TO 3
    j=STR(i,1,0)
    DO Rot&j
NEXT
```

Veja Também:

ISPRINTER( ), HEADER( ), DISKSPACE( ), GETE( ), RECSIZE( ), LUPDATE( ), DTOS( ), CALL e o Cap. 8 – Compilando e Executando os Programas.

Comando FOR...NEXT:

Sintaxe:

FOR <variável> = <exp.numérica1> TO <exp.numérica2>
[STEP <exp.numérica3>]
<comandos>
.
[EXIT]
.
<comandos>
.
NEXT

Propósito:

O comando FOR...NEXT permite a repetição de um bloco de instruções, localizado entre o FOR e o NEXT por um determinado número de vezes, avaliado por um contador, através de uma expressão numérica. O contador é automaticamente incrementado de um em um ou pode ser incrementado ou decrementado de acordo com um outro valor especificado.

Utilização:

Utiliza-se o comando estruturado FOR...NEXT quando, no fluxo de execução do programa, há a necessidade de repetição de um determinado conjunto de instruções (bloco) por um número expecífico de vezes, de acordo com um contador.

Cada vez que o bloco de instrucões é executado o contador é incrementado de uma unidade ou de acordo com o incremento especificado na cláusula STEP. Quando o contador atinge o limite máximo de repetições, o "loop" é finalizado e o fluxo de execução passa para a instrução imediatamente seguinte ao NEXT.

A <variável> define o nome da variável de memória que servirá de contador para controle do número de repetições.

A <exp.numérica 1> define o valor inicial a ser assumido pela variável de controle (contador).

A <exp.numérica 2> define o valor final máximo que poderá ser assumido pela variável de controle ou contador.

Cada vez que uma repetição é iniciada o Clipper reavalia todas as expressões que definem o FOR...NEXT, o que o caracteriza como dinâmico. Isto significa que é possível, mesmo durante a execução do FOR...NEXT, alterar seus limites e parâmetros, modificando os valores envolvidos nas expressões.

Opções:

A cláusula STEP redefine o incremento da variável de controle através da <exp.numérica 3>. Caso não seja especificada será assumido automaticamente um incremento igual a 1.

O comando EXIT pode ser utilizado, apenas na versão Summer 87 e posteriores. Ele finaliza um FOR...NEXT, mesmo não tendo sido atingido o limite máximo do contador. Ao ser encontrado um EXIT dentro de um FOR...NEXT o fluxo de execução passará imediatamente para a primeira linha de instrução após o NEXT.

Biblioteca:

CLIPPER.LIB

Exemplo:

* O exemplo abaixo irá apresentar os valores do contador i, incrementado de −3 em −3:

```
FOR i = 15 TO 0 STEP -3
    ? i
NEXT
```

* Mostrará: 15, 12, 9, 6, e 0

* O exemplo a seguir permite a visualização dos valores contidos nos elementos de um vetor:

```
numele=LEN(vetor)
FOR i=1 TO numele
    ? vetor[i]
NEXT
```

Veja Também:

DO WHILE, DO CASE, IF e a seção 5.9.1.– Comandos Básicos de Programação Estruturada.

Comando FUNCTION – Função-de-usuário:

Sintaxe:

FUNCTION <nome da função>
PARAMETERS
.
.
.
<comandos>
.
.
.
RETURN <expressão>

Propósito:

Declarar uma função definida pelo usuário (função-de-usuário) escrita em Clipper.

Utilização:

O comando FUNCTION permite a declaração e a construção de funções definidas pelo próprio usuário, utilizando os recursos da linguagem Clipper. Este comando praticamente abre a arquitetura da linguagem Clipper para você, pois através dele você poderá expandi-la, criando suas próprias funções, de acordo com suas necessidades.
Além de permitir a definição de funções para atender qualquer necessidade de processamento específico que você possua, o comando FUNCTION permite a criação de sua própria biblioteca de funções, que poderá ser utilizada em todas as suas aplicações desenvolvidas em Clipper.
Funções definidas pelo usuário ou funções-de-usuário são parecidas com programas ou rotinas normais, com duas exceções:

- se iniciam obrigatóriamente pelo comando FUNCTION, que as declara;
- contém um comando RETURN com um argumento, que será o resultado (ou retorno) da função.

O <nome da função> é o nome pelo qual a função será chamada, podendo possuir até 10 caracteres.
A <expressão> é o valor a ser retornado pela função. Todas as funções-de-usuário devem obrigatóriamente retornar um valor.
Para chamar uma função-de-usuário, utiliza-se a mesma notação que as funções normais do Clipper, ou seja:

Função (<lista de parâmetros/argumentos>)

Se em uma aplicação específica o valor de retorno da função não o interessar, deve ser utilizada a mesma notação para colocar a função-de-usuário sózinha em uma linha do programa. Neste caso o valor de retorno da função será automaticamente ignorado.
Os parâmetros (ou argumentos) são passados para as funções-de-usuário sempre por valor, com apenas duas exceções:

- se o parâmetro for o nome de um vetor, o vetor inteiro é passado por deferência;
- se o parâmetro for precedido pelo sinal "arroba" (@) ele é passado por referência.

Para mais detalhes sobre a passagem de parâmetros veja o comando PARAMETERS.

Biblioteca:

CLIPPER.LIB

Exemplos:

* Função Centra( ): a linha seguinte usa a função-de-usuário Centra( ) para centralizar um título através do comando @...SAY:

```
@ 12,Centra("C l i p p e r") SAY "C l i p p e r"

FUNCTION Centra
PARAMETERS tit
RETURN INT((80-LEN(TRIM(tit)))/2)
```

— A função Dpc( ) a seguir, retorna, a partir de uma variável data, o nome do mês e o ano da data passada no formato "MMM/AA", em português.

```
FUNCTION Dpc
PARAMETERS dd
PRIVATE aux
aux= 3 * MONTH(dd) - 2
dc=SUBSTR("JANFEVMARABRMAIJUNJULAGOSETOUTNOVDEZ",aux,3) +;

SUBSTR(DTOC(dd),6,8)
RETURN(dc)

? Dpc(DATE()) && retorna "MAR/88"
```

— A função Mudavar( ) a seguir recebe uma variável passada por referência através do sinal arroba (@). Neste caso as alterações feitas no conteúdo da variável, dentro da função, alteram seu conteúdo original.

```
var=10
? Mudavar(var)
? var
* resulta 10
? Mudavar(@var)
? var
* resulta 100
RETURN

FUNCTION Mudavar
PARAMETERS val
val = val * val
RETURN(val)
```

O exemplo a seguir demonstra a utilização de um função-de-usuário dentro da cláusula VALID do comando @...SAY...GET, com o objetivo de realizar validação dos dados digitados:

```
valor=0
@ 10,20 SAY "Digite um número:" GET valor VALID Vldnu(valor)
READ
RETURN

FUNCTION Vldnu
PARAMETERS numero
RETURN (numero ) 0 .AND. numero ( 100)
```

Veja Também:

PROCEDURE, PARAMETERS, RETURN

Comando IF...ENDIF:

Sintaxe

```
IF <condição>
        .
        .
  <comandos>
        .
        .
 ELSE
        .
        .
  <comandos>
        .
        .
ENDIF
```

Propósito:

O comando IF é um comando de programação estruturada que permite executar condicionalmente um bloco de instruções, também conhecido como desvio condicional. Cada IF deve obrigatoriamente terminar com um ENDIF.

Utilização:

O comando IF permite, através de uma condição, selecionar ou não um bloco de instruções a ser executado. Deve sempre ser finalizado com um ENDIF. Qualquer outro comando de programação estruturada (DO WHILE, DO CASE e FOR...NEXT) dentro de um IF...ENDIF deve ser intercalado corretamente, ou seja seu início e fim devem estar entre o IF e o ENDIF. Intercalações de vários IF's também são permitidas, desde que cada IF possua seu próprio ENDIF.

Após o ENDIF, na mesma linha, podem ser colocados comentários para documentação do programa.

A <condição> é uma expressão lógica de controle. IF <condição> inicia uma decisão condicional sobre uma expressão lógica do tipo A=B, var1>20 ou var1>20 .AND. var2<100, a ser avaliada como verdadeira (.T.) ou falsa (.F.).

Se a <condição> for avaliada como verdadeira (.T.), todas as instruções logo após o IF serão executadas, até ser atingido o ENDIF, ou o ELSE, caso este estiver sendo utilizado. Em seguida, será executada a primeira instrução que vier logo após o ENDIF.

Se a condição for avaliada como falsa (.F.) serão executadas todas as instruções logo a seguir à cláusula ELSE, até ser atingido o ENDIF. No caso de não existir o ELSE, será executada a primeira instrução logo após o

ENDIF. Em qualquer caso, portanto, se a condição for falsa, não será executado o bloco de instruções logo após o IF.
A cláusula ELSE inicia o bloco de comandos que será executado caso a <condição> do IF seja avaliada como falsa (.F.). Se existirem múltiplos IF's em uma estrutura de comandos, um ELSE sempre se refirirá ao IF que imediatamente o precede.

Biblioteca:

CLIPPER.LIB

Exemplo:

* O exemplo a seguir decide se os dados do arquivo Mala.dbf, de mala-direta, serão apresentados na tela ou impressos na impressora:

```
CLEAR
USE Mala INDEX IndAlfa
WAIT "Listagem na Tela ou na Impressora (T/I) ?" TO ti
IF UPPER(ti) = "I"
   LIST nome, endereco, cidade, telefone, cep TO PRINT
ELSE
   DISPLAY ALL nome, telefone
ENDIF
USE
RETURN
```

* Estrutura para intercalação de IF's

```
IF <condição>
.
.    IF <condição>
.    .
.    .    <comandos>
.    .
.    ELSE
.    .
.    .
.    .    IF <condição>
.    .    .
.    .    .<comandos>
.    .    .
.    .    ENDIF
.    .
.    ENDIF
ENDIF
```

Veja Também:

DO CASE...ENDCASE, IF( )/IIF( ) e a seção 5.9.1– Estruturas Básicas de Programação Estruturada.

Comando IF...ELSEIF...ELSE...ENDIF:

Sintaxe:

```
IF <condição>
      .
   <comandos>
      .
ELSEIF <condição>
      .
   <comandos>
      .
ELSEIF <condição>
      .
   <comandos>
      .
 ELSE
      .
   <comandos>
      .
ENDIF
```

Propósito:

O comando IF...ELSEIF...ENDIF é um comando de programação estruturada que seleciona um caminho entre um conjunto de caminhos alternativos para o curso do fluxo de execução do programa. O caminho selecionado é o primeiro para o qual uma condição especificada for avaliada como verdadeira.
Disponível a partir da versão Summer 87.

Utilização:

O comando IF...ELSEIF deve ser utilizado dentro do fluxo de execução dos programas para determinar a escolha de um, entre vários caminhos alternativos a serem executados. o caminho executado será aquele que primeiro for avaliado como verdadeiro dentro das condições ELSEIF especificadas.
É obrigatória a existencia do ENDIF (fim do IF) ao final da estrutura do comando IF...ELSEIF.
A cláusula ELSE é um caminho alternativo caso nenhum ELSEIF tenha sido avaliado como verdadeiro, isto é , se todos os ELSEIF's forem falsos e existir o ELSE, serão executados os comandos que estiverem após ELSE e antes do ENDIF.
Comandos IF...ELSEIF intercalados, isto é, uns dentro dos outros, são permitidos, desde que intercalados corretamente. Ao se incluir pares de outras estruturas de comandos como DO CASE...ENDCASE, IF...ENDIF, DO WHILE...ENDDO e FOR...NEXT dentro de um IF...ELSEIF principal, estes devem iniciar-se e finalizar-se dentro da estrutura IF...ELSEIF...ENDIF principal. Não há limitações quanto ao número de estruturas de comandos que podem ser intercaladas.
A cláusula ELSEIF <condição> define uma declaração condicional a ser avaliada como falsa ou verdadeira. Esta condição deve ser uma expressão lógica como A=B ou var1< 11 etc. Não há limite para o número de ELSEIF's que podem existir dentro de uma estrutura IF...ELSEIF...ENDIF.
Quando a condição de um ELSEIF é avaliada como verdadeira, todos os comandos que a seguem, até o próximo ELSEIF, ELSE ou ENDIF são executados. Neste caso, mais nenhum outro ELSEIF será avaliado. O fluxo de execução passará diretamente para o primeiro comando após o ENDIF.
Se nenhum ELSEIF for avaliado como verdadeiro e não existir a cláusula ELSE, o fluxo de execução do programa passa diretamente para a primeira instrução após o ENDIF.
A estrutura do IF...ELSEIF...ENDIF pode ser sempre utilizada onde houver um certo número de exceções à uma regra de execução a ser seguida. Os IF...ELSEIF <condição> representarão estas exceções, e o ELSE a situação mais comum.
O comando IF...ELSEIF...ENDIF é exatamente equivalente ao comando DO CASE...CASE...ENDCASE.

Biblioteca:

CLIPPER.LIB

Exemplos:

* A rotina a seguir decide sobre o desconto a ser efetuado sobre o salário dos funcionários, de acordo com sua faixa salarial:

```
CLEAR
USE Folhapg INDEX IndNum
DO WHILE .NOT. EOF()
   IF salario ) 12000 .AND. salario ( 20000
         desconto = salario * 0.10
      ELSEIF salario )= 20000 .AND. salario ( 40000
         desconto = salario * 0.15
      ELSEIF salario )= 40000 .AND. salario ( 60000
         desconto = salario * 0.20
      ELSEIF salario )= 60000 .AND. salario ( 90000
         desconto = salario * 0.25
      ELSEIF salario )= 90000 .AND. salario (15000
         desconto = salario * 0.30
      ELSEIF salario )= 150000 .AND. salario ( 200000
         desconto = salario * 0.40
      ELSE
         desconto = salario * 0.50
   ENDIF
   REPLACE desconto WITH desconto - salario/100
   SKIP
ENDDO
USE
RETURN
```

Veja Também:

DO, DO WHILE, IF..ELSE..ENDIF, DO CASE...CASE...ENDDO, FOR...NEXT, IF( )/IF( ) e a seção 5.9.1– Estruturas Básicas de Programação Estruturada.

Comando LOOP:

Sintaxe

LOOP

Propósito:

O comando LOOP retorna o controle do fluxo de execução dos comandos ao início do comando de programação estruturada DO WHILE...ENDDO. O LOOP evita a execução das instruções restantes dentro de um bloco DO WHILE...ENDDO.

Utilização:

O LOOP deve ser utilizado sempre dentro de um bloco de instruções de um comando DO WHILE...ENDDO. Ao ser encontrado ele causa o retorno imediato do fluxo do programa para o início do DO WHILE, sem a necessi-

dade de ser atingido o ENDDO. Os comandos que estiverem entre o LOOP e o ENDDO não serão executados.
Para forçar o retorno ao início do DO WHILE de acordo com uma condição, utilize o LOOP dentro de um IF...ENDIF ou de um DO CASE...ENDCASE.

Biblioteca:

CLIPPER.LIB

Exemplos:

* Neste exemplo é pesquisado o código de um produto a ser apresentado no arquivo de estoque. Se o código não existir é apresentada uma mensagem, retorna-se para o início do DO WHILE e novo código é solicitado.

```
   CLEAR
   USE Estoque INDEX IndCod
+-)DO WHILE .T.
¦     vcod=SPACE(5)
¦     @ 12,22 SAY "Digite o Código do Produto:" GET vcod
¦     READ
¦     SEEK vcod
¦     IF EMPTY(vcod)
¦        EXIT
¦     ENDIF
¦     IF .NOT. FOUND()
¦        @ 14,28 SAY "Produto Não Encontrado !"
¦        FOR i=1 TO 200
¦        NEXT
¦        @ 14,00 CLEAR
+---(--- LOOP
      ENDIF
      @ 14,22 SAY "Produto:"+vcod
      @ 16,22 SAY "Saldo..:"
      @ 16,31 SAY saldo PICTURE "@E 999,999"
      @ 16,39 SAY unidade
   ENDDO
   CLEAR
   USE
   RETURN
```

Veja Também:

DO WHILE...ENDDO e a seção 5.9.1– Estruturas de Programação Estruturada.

Comando NOTE ou *

Sintaxe:

NOTE ou * <texto>

Propósito:

O comando NOTE ou o asterisco (*) no início de uma linha de um programa ou rotina evita que a linha seja executada. Eles são usados para inserir comentários dentro do programa. Estes comentários serão ignorados na compilação, servindo apenas para documentação interna do programa.

Utilização:

Utiliza-se o asterisco (*) ou NOTE, no início de uma linha, sempre que se quiser colocar um comentário ou um título no programa. Tudo o que vier nesta linha, após o asterisco não será compilado pelo Clipper.
Para colocar comentários em linhas que contenham comandos no seu início, utilize o comando &&.
Linhas de comentários não podem ser continuadas para a linha seguinte através do ponto-e-vírgula. Caso seja necessário continuá-las deve ser colocado um comando NOTE ou um asterisco (*) em cada nova linha de comentário.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
CLEAR
USE Estoque INDEX IndCod
* Abertura do Arquivo Estoque indexado por código
DISPLAY ALL
* Mostra todos os registros na tela
USE
* Fecha o Arquivo
RETURN
* Retorna para o DOS
```

Veja Também:

&&

Comando PARAMETERS:

Sintaxe:

PARAMETERS <lista de variáveis>

Propósito:

o comando PARAMETERS declara variáveis locais (privadas) que receberão os dados passados (por valor ou por referência) por um programa que chama uma rotina ou função-de-usuário. É o comando que recebe dados passados através do comando DO ou através dos argumentos de uma função-de-usuário.

Utilização:

Quando se passa parâmetros para uma rotina ou programa através do comando DO <nome da rotina/programa> WITH <lista de parâmetros> ou através dos argumentos de uma função-de-usuário, a rotina ou função chamada deve ter como primeiro comando a ser executado o comando PARAMETERS. Este comando define as variáveis locais que receberão os parâmetros passados.
A <lista de variáveis> deve ser composta pelos nomes, separados por vírgula, de uma ou mais variáveis que receberão os parâmetros passados. A partir da versão Summer 87, o número destas variáveis (recebedoras) não precisa obrigatoriamente coincidir com o número de parâmetros passados. Nas versões anteriores (como Winter 85 ou Autumn 86) o número de parâmetros passados deve necessariamente coincidir com o número de parâmetros recebidos, caso contrário será apresentada uma mensagem de erro.
Parâmetros são definidos como formais ou reais. Parâmetros formais são as variáveis recebedoras especificadas como argumentos do comando PARAMETERS. Parâmetros reais são os argumentos do comando DO...WITH ou da função-de-usuário (comando FUNCTION) que são realmente passados para a rotina ou função chamada.
Recomenda-se que as rotinas e as funções-de-usuário que recebem parâmetros estejam em um arquivo de procedimentos ("procedures files"), para poderem ser utilizadas por qualquer módulo de um sistema. Veja os comandos PROCEDURE E SET PROCEDURE TO.
Quando incluído em uma rotina ou função-de-usuário, o comando PARAMETERS deve obrigatoriamente ser o primeiro comando a ser executado nesta rotina. Um parâmetro passado pode ser qualquer expressão válida (numérica, caractere ou data) do Clipper. A lista de parâmetros declarada no comando PARAMETERS define variáveis locais (privadas) para receber a lista de parâmetros enviados do comando DO...WITH ou função-de-usuário. As variá-

veis locais definidas são automaticamente canceladas quando o fluxo de execução retorna ao programa que chamou a rotina.
Há dois métodos de passagem de parâmetros:

- Por valor, significa que o parâmetro real (variável/expressão passada) é avaliado e o seu resultado é colocado em um posição da memória.
  Quando o comando PARAMETERS é executado, o valor avaliado do parâmetro real é transferido para o parâmetro formal (a variável recebedora).
- Por referência, em contraste, significa que o "ponteiro" de localização do parâmetro real (variável passada) é passado, ao invés de ser passado apenas o seu valor. Mudanças posteriores no parâmetro formal (variável recebedora) irão modificar o conteúdo do parâmetro real.

No Clipper, a partir das versões Summer 87 e posteriores, o número de parâmetros reais passados não precisa coincidir com o número de parâmetros formais esperados. Para determinar o número de parâmetros reais passados utiliza-se a função PCOUNT( ).
A ordem na qual os parâmetros reais são passados será a ordem na qual os parâmetros formais os receberão. Por exemplo:

```
DO <rotina> WITH A,B,C      && no programa que chama.
PARAMETERS P1,P2,P3         && na rotina chamada.
```

O parâmetro formal P1 receberá o parâmetro real A, o parâmetro formal P2 receberá o parâmetro real B e o P3 receberá o C.

Regras para a passagem de Parâmetros:

As seguintes regras se aplicam ao se passar variáveis e vetores como parâmetros a rotinas ou funções-de-usuário:

1 — Variáveis e vetores são sempre passados por referência. Elementos de vetores, expressões, variáveis entre parêntesis e campos de arquivos de dados são sempre passados por valor. Campos de arquivos devem ser sempre passados entre parêntesis.
2 — Parâmetros são passados a funções-de-usuário normalmente por valor; contudo, podem ser passados por referência se a variável for precedida pelo símbolo arroba (@). Vetores são sempre passados por referência; elementos de vetores, entretanto, são sempre passados por valor.

Podem ser passados parâmetros para um programa a partir do Sistema Operacional (DOS)

1 – Os parâmetros passados a partir da linha de comando do DOS são sempre do tipo caracter. Podem ser passadas múltiplas cadeias de caracteres para o programa principal.

2 – Cada cadeia de caracteres passada deve estar separada por um espaço em branco. Um parâmetro delimitado por aspas é passado como uma única cadeia de caracteres. Por exemplo:

C>SISTEMA "Clipper – O Compilador" "Teste"

Neste caso são passados dois parâmetros a partir do DOS: "Clipper – O Compilador" e "Teste".

3 – O programa que recebe os parâmetros a partir do DOS deve testar o número de parâmetros passados através da função PCOUNT( ) para assegurar que os parâmetros essenciais foram passados.

A utilização de parâmetros nas aplicações desenvolvidas em Clipper permite grande flexibilidade e principalmente reduz duplicidades no código dos programas.

Biblioteca:

CLIPPER.LIB

Exemplos:

O exemplo a seguir envidencia a passagem de parâmetros por referência ou por valor:

```
DECLARE vetor[10]
STORE "original" TO var1, var2, vetor[1]
DO Rotina WITH var1, (var2), vetor       && veja abaixo
* var1 é passada por referência
* (var2) é passada por valor
* vetor é passado por referência
*
? var1, var2, vetor[1]
* resultam "novo", "original", "novo"
*
RETURN

PROCEDURE Rotina
PARAMETERS par1, par2, vetpar
STORE "novo" TO par1, par2, vetpar[1]
RETURN
```

O exemplo a seguir evidencia a passagem de parâmetros por valor e por referência a funções-de-usuário:

```
STORE 100 TO var1, var2
Mudavar(@var1, var2)        && vide abaixo
* var1 é passada por referência e var2 por valor
*
? var1, var2
* resultam 200 e 100
*
RETURN

FUNCTION Mudavar
PARAMETERS par1, par2
STORE 200 TO par1, par2
RETURN ""
```

O exemplo a seguir demonstra a passagem de parâmetros a partir do DOS para o programa DOSTESTE:

```
* DOSTESTE.prg
PARAMETERS var1, var2, var3, var4
?? var1, var2, var3,var4

C>DOSTESTE Clipper 1000 20/03/88 "dBASE III"
```

Resulta: Clipper 1000 20/03/88 dBASE III

Atenção: todos os parâmetros passados para um programa a partir do DOS são sempre caracter. O mesmo não ocorre quando os parâmetros são passados de um programa para outro dentro do Clipper.

* O programa abaixo permite consultas aos saldos dos produtos em um arquivo de estoque indexado pelo código do produto. O programa utiliza o arquivo de procedimentos ("procedure") ESTPROC.prg:

```
CLEAR
SET PROCEDURE TO Estproc
USE Estoque INDEX IndCod
continua=SPACE(1)
DO WHILE UPPER(continua)<>"N"
   vcod=SPACE(5)
   @ 12,22 SAY "Digite o Código do Produto:" GET vcod
   READ
   IF EMPTY(vcod)
      EXIT
   ENDIF
   SEEK vcod
   IF .NOT. FOUND()
      DO Mensagem WITH 22,"Produto Não Encontrado !",8
      LOOP
   ENDIF
   CLEAR
   @ 10,20 SAY "Produto:"+vcod+"   "+nome
   @ 12,20 SAY "Saldo..:"
   @ 12,30 SAY saldo PICTURE "@E 999,999"
```

```
    @ 12,38 SAY unidade
    @ 14,20 SAY "Ultima Atualização: "+DTOC(data)
    @ 20,35 SAY "Continua (S/N) ? " GET continua
    READ
ENDDO
DO Mensagem WITH 24,"Fim das Consultas !",4
CLEAR
USE
RETURN


      * Arquivo de procedures ESTPROC.prg
      *
      PRODECURE Mensagem
      *
      * Mostra   uma   mensagem   centralizada   na   linha
        *  especificada da  tela por um determinado tempo, de
        *  acordo com   os  parâmetros linha,  texto e  tempo
      * passados.
      *
      PARAMETERS linha,texto,tempo
      SET COLOR TO W+
      @ linha,40-(LEN(texto)/2) SAY texto
      SET COLOR TO W
      FOR i=1 TO tempo*100 && Dá um tempo para apresentação
      NEXT                 && da mensagem na tela
      @ 24,00 CLEAR
      RETURN
```

Veja Também:

DO, PRIVATE, PUBLIC, PROCEDURE, SET PROCEDURE, PUBLIC, PCOUNT( ).

Comando PROCEDURE:

Sintaxe:

PROCEDURE <nome do "procedimento">

.
.

<comandos>

.
.

RETURN

PROCEDURE <nome do "procedimento">

.
.

<comandos>

.
.

RETURN

Propósito:

O comando PROCEDURE identifica o início de cada rotina (uma "procedure"), a ser incluída em um arquivo de procedimentos ("procedure file") ou em um programa.

Propósito:

O comando PROCEDURE é utilizado para iniciar rotinas dentro de um programa ou arquivo de procedimentos. Uma "procedure" é qualquer bloco de comandos iniciado pelo comando PROCEDURE <nome do procedimento> e finalizado pelo comando RETURN, podendo ocorrer em qualquer parte de um programa ou arquivo de procedures, não podendo, entretanto, ser intercalado com outras "procedures".

As "procedures" são chamadas através do comando DO <nome da "procedure"> WITH <lista de parâmetros>. Ao ser encontrado o comando RETURN no final de uma "procedure", o fluxo de execução retorna para a linha imediatamente seguinte ao comando DO do programa principal que chamou a "procedure".

O <nome do procedimento> é o nome da "procedure". Este nome pode possuir até 10 caracteres de comprimento, mas deve necessariamente começar por uma letra.

Cada nome de "procedure" deve ser único dentro de uma mesma aplicação, ou seja, diferente dos nomes de todas as outras "procedures", programas, funções-de-usuário e arquivos de formato utilizados no sistema.

Um arquivo de procedimentos é um arquivo que contém várias rotinas/"procedures" que serão utilizadas por diversos programas ou módulos de um sistema. Ao invés de repeti-las no corpo dos programas que as necessitam, basta incluí-las em um arquivo de procedimentos como o esquematizado abaixo e chamá-las através do comando DO <nome da rotina/procedure> WITH <lista de parâmetros>.

Arquivos de Procedimentos

```
*******************************
PROCEDURE Mensagem
PARAMETERS
     .
     .
<comandos>
     .
     .
RETURN
*******************************
```

```
PROCEDURES Aviso
PARAMETERS
     .
     .
<comandos>
     .
     .
RETURN
*******************************
PROCEDURE Pesquisa
     .
     .
<comandos>
     .
     .
RETURN
*******************************
PROCEDURE Caberel
PARAMETERS
     .
     .
<comandos>
     .
     .
RETURN
*******************************
* Fim do Arquivo de Procedimentos
*******************************
```

Um Arquivo de Procedimentos deve ser definido no Módulo Principal de um Sistema ou Programa através do comando SET PROCEDURE TO <nome do arquivo de procedimentos>, ou simplesmente compilado separadamente e link-editado com os demais programas de uma aplicação.
No Clipper podem ser declaradas ou utilizadas em uma mesma aplicação tantos arquivos de prodecimentos ("procedures files") quantos forem necessários.

Exemplos:

Por favor veja os arquivos de procedimentos MLDPROC.prg e MLDPESQ.prg do Sistema MLD de Emissão de Mala-Direta no capítulo 10.

Veja Também:

DO, SET PROCEDURE TO, PARAMETERS, o Cap. 8 — Compilando e Executando Programas.

Comando QUIT:

Sintaxe:

QUIT/CANCEL

Propósito:

O comando QUIT ou CANCEL encerra o processamento do programa, fecha todos os arquivos abertos (de qualquer tipo), retornando o controle do computador ao Sistema Operacional.

Utilização:

O comando QUIT deve ser utilizado quando se deseja encerrar a execução de um sistema a partir de um módulo que não seja o módulo principal (programa principal do sistema).
Serão fechados todos os arquivos que estiverem abertos, em qualquer área de trabalho, e o controle do equipamento retornará ao Sistema Operacional.

Biblioteca:

CLIPPER.LIB

Veja Também:

RETURN, CANCEL

Comando RETURN:

Sintaxe:

RETURN [<**expressão**>]

Propósito:

O comando RETURN finaliza uma rotina ou programa, retornando o controle do fluxo de execução ao programa ou rotina que o chamou ou ao Sistema Operacional.

Utilização:

O comando RETURN, quando utilizado em uma rotina ou programa finaliza-o, cancelando todas suas variáveis privadas e retornando o controle do

fluxo de execução à linha imediatamente seguinte ao comando DO do programa ou rotina que o chamou.

Quando um comando RETURN é encontrado no módulo principal de aplicação (programa principal, chamado diretamente a partir do DOS), todos os arquivos são fechados, a aplicação é finalizada e o controle é devolvido ao Sistema Operacional.

Em funções-de-usuário (veja o comando FUNCTION) o comando RETURN finaliza a função e retorna o resultado da <expressão> especificada ao programa que chamou a função. Uma função deve sempre retornar um valor, sendo portanto obrigatória a especificação de uma <expressão> após o comando RETURN para fornecer um valor de retorno à função. Quando não se deseja sejar que uma função retorne algum valor, pode-se especificar o retorno do valor nulo, ou seja: ""duas aspas seguidas (RETURN"").

Pode haver mais de um comando RETURN em quaisquer rotinas, programas ou em funções-de-usuário.

Se o RETURN não for incluído no fim de um programa, o Clipper assume automaticamente o RETURN e devolve o controle ao programa que o chamou ou, se o programa foi chamado a partir do DOS, para o Sistema Operacional.

No Clipper não existe o comando RETURN TO MASTER (presente no dBASE III) ou qualquer outro tipo de RETURN que permita definir o nível do programa de chamada a se retornar. O retorno sempre será feito ao mesmo programa que chamou a rotina ou função-de-usuário, ou então ao Sistema Operacional.

Biblioteca:

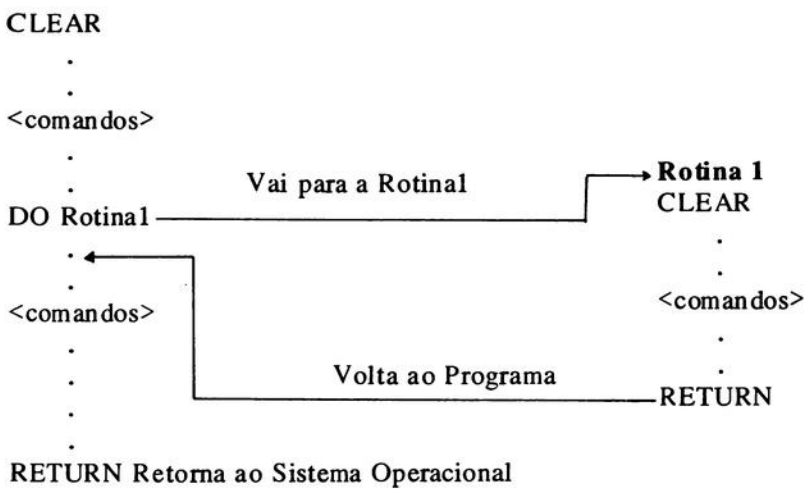
CLIPPER.LIB

Exemplos:

```
PROCEDURE <nome da rotina/procedure>
          .
          .
     <comandos>
          .
          .
RETURN
FUNCTION <nome da função>
          .
          .
     <comandos>
          .
          .
RETURN <expressão de retorno>
```

Esquema de Transição entre Programas:

Programa Principal:



Veja Também:

CANCEL, PRIVATE, PUBLIC, QUIT

Comando RUN/!:

Sintaxe:

RUN/! <**comando do DOS / nome do programa**>

Propósito:

O comando RUN executa, a partir de uma aplicação em Clipper, um comando do Sistema Operacional ou outro "software" qualquer. Ao serem finalizados o controle retorna automaticamente à aplicação em Clipper, sendo executada a instrução imediatamente seguinte ao comando RUN.

Utilização:

Utiliza-se o comando RUN para, a partir de uma aplicação em Clipper, executar qualquer comando ou programa como se o mesmo fosse executado a partir do Sistema Operacional.

O <comando do DOS / nome do programa> pode ser qualquer programa executável a partir do DOS, incluindo os seus comandos residentes e o COMMAND.com.
Para se executar o comando RUN é necessário ter disponível certa quantidade de memória, pois durante a execução do comando do DOS ou do programa chamado, ambos, este e o programa executável em Clipper, estarão simultaneamente na memória RAM. Se o equipamento utilizado não possuir memória suficiente para "executar" os dois simultaneamente, será apresentada uma mensagem indicando memória insuficiente ("Memory Fault" ou "Insuficient Memory"). Para mais detalhes veja a seção 8.3.2 – Utilização da Memória.
Ao se executar o comando RUN, o arquivo "COMMAND.com" do DOS deve estar presente no disco padrão (disco de "boot" ou "default") ou no especificado pelo parâmetro COMSPEC. Pode ser utilizado o comando SET do DOS para dizer ao sistema onde achar o COMMAND.com. Por exemplo:

```
COMSPEC = C:\DOS\COMMAND.COM.
```

Atenção: Não é recomendável que sejam executados programas que ficam residentes na memória. Isto reduzirá a eficiência da aplicação desenvolvida e poderá ter consequências imprevisíveis.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
RUN DIR A:
RUN COPY C:*.DBF A:
! DBASE     * executa o dBASE dentro de uma aplicação em Clipper
RUN BACKUP C: A:
RUN RESTORE A: C:
```

Veja Também:

CALL

Comando TEXT...ENDTEXT:

Sintaxe:

```
TEXT [TO PRINT/TO FILE <nome do arquivo>]
      .
      .
  <texto>
      .
      .
ENDTEXT
```

Propósito:

O comando TEXT...ENDTEXT mostra diretamente blocos de texto na tela, ou opcionalmente os imprime na impressora ou grava em arquivos no disco. Este comando constitui uma forma simples e rápida de apresentar, geralmente na tela, textos exatamente como foram digitados dentro de um programa ou rotina.

Utilização:

Uma das principais utilizações do comando TEXT...ENDTEXT é a elaboração de textos de ajuda ao usuário (HELP's), que serão apresentados caso solicitado.

O <texto> é um bloco de texto entre o comando TEXT e o ENDTEXT que será apresentado na tela exatamente como digitado dentro do programa. Caso a opção TO PRINT seja utilizada, o bloco de texto será impresso. Se for utilizada a opção TO FILE <nome do arquivo> o texto será gravado em um arquivo texto (padrão ASCII) com o nome designado. Se não for especificada cada uma extensão será assumida a extensão padrão (.txt).

A função macro-substituição pode ser utilizada dentro de um bloco de texto definido pelo comando TEXT...ENDTEXT para substituir nomes a serem nele inseridos. Contudo o Clipper não controlará a sobreposição do conteúdo da variável substuída sobre o texto digitado quando os tamanhos forem diferentes.

É muito conveniente utilizar este comando em conjunto com a facilidade de HELP ao usuário que o Clipper possui. Para maiores detalhes consulte o Cap. 7 — A Facilidade de "HELP" (Ajuda) ao Usuário.

Biblioteca:

CLIPPER.LIB

Exemplo:

Este programa ilustra como pode ser implementado o comando TEXT...ENDTEXT dentro de um "Help" ao usuário. O programa Help.prg será acessado pelo Clipper ao ser pressionada a tecla F1 e mostrará na tela o bloco de texto compreendido entre o TEXT e o ENDTEXT:

```
*
* Programa HELP.prg - Generico
*
PARAMETERS prog,lin,var
SEVE SCEEEN
@ 5,0 CLEAR
TEXT
                    HELP - Auxílio ao Usuário

                  Teclas de Controle de Edição
              (Para Inclusão ou Alteração de Dados)

<Enter>  -> Dá entrada nos dados ou efetua uma alteração
<Setas>  -> Movimentam o cursor para a direção indicada
<Home>   -> Movimenta o cursor para o inicio do campo
<End>    -> Movimenta o cursor para o fim do campo
< <-- >  -> Apaga o caracter à esquerda do cursor
<Ins>    -> Liga ou Desliga a de inserção de caracteres
<Del>    -> Apaga o caracter sob o cursor
<Ctrl-T> -> Apaga a palavra à direita do cursor
<Ctrl-Y> -> Apaga todo o conteudo de um campo
<Ctrl-W> -> Finaliza gravando os dados digitados
<Esc>    -> Finaliza sem gravar os dados digitados

              Pressione qualquer tecla para retornar
ENDTEXT
WAIT
RESTORE SCREEN
RETUNR
```

Veja Também:

@...SAY, ?/??, MLCOUNT( ), MEMOLINE( ) e o Cap. 7 – A Facilidade de "HELP" (Ajuda) ao Usuário

5.10. COMANDOS PARA GERAÇÃO DE RELATÓRIOS

Comando LABEL FORM:

Sintaxe

LABEL FORM <nome do arquivo de etiquetas>
[escopo]
[FOR <condição>]
[WHILE <condição>]
[SAMPLE]
[TO PRINT]
[TO FILE <nome do arquivo>]

Propósito:

Gerar um relatório de etiquetas (para endereçamento de correspondência) a partir de um arquivo tipo ".LBL" de definição de etiquetas.

Utilização:

O comando LABEL FORM é utilizado para gerar na impressora, tela ou disco, um relatório de etiquetas a partir dos registros do arquivo que estiver em uso, segundo as definições contidas em um arquivo de formatação de etiquetas (extensão .lbl).

O <arquivo de etiquetas> é o nome de um arquivo gerado pelo utilitário LABEL.exe (ou RL.exe na versão Summer 87) ou mesmo pelo comando LABEL do dBASE III, que possui definições para a emissão de etiquetas a partir do arquivo de dados em uso.

O <escopo> define a quantidade de registros do arquivo de dados a serem processados pelo comando: ALL (todos), NEXT <n> (os próximo n registros a partir do atual) ou RECORD <n> (apenas o registro n). Se não for especificado serão assumidos todos (ALL).

A cláusula FOR especifica uma <condição>, dentro do <escopo> fornecido, para selecionar os registros que deverão ser processados pelo comando. Apenas para os registros que a satisfizerem (a tornarem verdadeira) serão emitidas etiquetas.

A cláusula WHILE seleciona os registros a serem processados, a partir do registro atual, enquanto uma <condição> especificada for satisfeita (permanecer verdadeira). Para tanto o arquivo de dados deverá estar indexado por uma chave relativa à condição especificada.

A cláusula TO PRINT direciona a emissão de etiquetas para a impressora.

A cláusula TO FILE <nome do arquivo> direciona a geração de etiquetas para um arquivo no disco, que será gravado com o nome especificado. Se uma extensão não for especificada será assumida a extensão (.txt).

A opção SAMPLE permite a emissão de um teste das etiquetas, como

sendo linhas de asteriscos (estrelinhas), de acordo com o comprimento dos campos ou dados a serem impressos. Cada teste das etiquetas possuirá o mesmo número de colunas e linhas definido no arquivo (.lbl). Ao final de cada teste será apresentada a mensagem "Do you want more samples ?". Respondendo "Y" (yes – sim) um novo teste será emitido. Respondendo "N" (no – não) as etiquetas, com os dados reais do arquivo, começarão a ser emitidas, de acordo com o <escopo> e condições especificadas. A opção SAMPLE tem a função de permitir o ajuste do formulário de etiquetas na impressora, antes de se iniciar emissão propriamente dita.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
USE Pessoal INDEX Alfa
LABEL FORM Etiqueta FOR depto="FINANÇAS" SAMPLE TO PRINT
```

Para interromper o comando LABEL FORM utiliza-se a função INKEY( ) como parte da condição especificada. No exemplo a seguir, a emissão das etiquetas é finalizada ao ser pressionada a tecla <Esc>, cujo código ASCII é 27.

```
USE Mala
LABEL FORM Etiqueta FOR INKEY( )< >27
```

Veja Também:

REPORT FORM, e o Cap. 9 – "Utilitários do Clipper"

Comando REPORT FORM:

Sintaxe:

```
REPORT FORM <nome do arquivo de relatório>
            [<escopo>]
            [FOR <condição>]
            [WHILE <condição>]
            [HEADING <texto do cabeçalho>]
            [SUMMARY]
            [PLAIN]
            [NOEJECT]
            [TO PRINT]
            [TO FILE <nome do arquivo>]
```

Propósito:

O comando REPORT FORM emite um relatório padrão, em forma de colunas, criado pelos utilitários REPORT.exe ou RL.exe (versão Summer 87) ou pelo comando CREATE REPORT do dBASE III, sobre os registros de um arquivo de dados. As definições de formato do relatório gerado pelo comando REPORT FORM são armazenadas em um arquivo de relatório, com extensão ".FRM".

Utilização:

O comando REPORT FORM emite um relatório padrão criado pelos utilitários REPORT.exe ou RL.exe ou pelo comando CREATE REPORT do dBASE III. As definições do relatório as ser emitido, como cabeçalhos, colunas, totais, subtotais etc, são armazenadas em um arquivo de definição de relatórios, em geral com a extensão (.frm).
O <nome do arquivo de relatório> é o nome do arquivo que contém as definições de formatação do relatório a ser impresso. Se a extensão não for informada (.frm) é assumida.
O <escopo> define a quantidade de registros do arquivo de dados a serem processados pelo comando: ALL (todos), NEXT <n> (os próximo n registros a partir do atual) ou RECORD <n> (apenas o registro n). Se o escopo não for especificado serão assumidos todos os registros (ALL).
A cláusula FOR especifica uma <condição>, dentro do <escopo> fornecido, para selecionar os registros que deverão ser processados pelo comando. Apenas os registros que a satisfizerem (a tornarem verdadeira) serão emitidos no relatório.
A cláusula WHILE seleciona os registros a serem processados, a partir do registro atual, enquanto uma <condição> especificada for satisfeita (permanecer verdadeira). Para tanto o arquivo de dados deverá estar indexado de acordo com a condição desejada.
A cláusula TO PRINT redireciona a saída do comando da tela (padrão normal) para a impressora.
A cláusula TO FILE <nome do arquivo> redireciona a saída do comando, sem o caracter de controle para mudança de página (ASCII 12), para um arquivo do disco cujo nome for especificado. Se uma extensão não for especificada para o nome do arquivo será assumida a extensão (.txt).
Para incluir o caracter ASCII 12, de mudança de página, num arquivo em disco, deve ser utilizado o comando SET PRINTER TO, através das seguintes instruções:

```
SET PRINTER TO <nome do arquivo no disco>
REPORT FORM <nome do relatório> TO PRINT
SET PRINTER TO
```

A cláusula SUMMARY faz com que sejam emitidos apenas os grupos e subgrupos do relatório e seus respectivos totais. Os dados detalhados dos registros são omitidos. É, portanto, gerado um relatório sumarizado.
A cláusula PLAIN suprime a data de emissão, número das páginas (normalmente apresentadas no topo e à esquerda de cada página do relatório) e não realiza a paginação (não há salto de páginas). O título do relatório e o cabeçalho das colunas são apresentados apenas uma única vez no início do relatório.
A cláusula HEADING (cabeçalho) permite a inclusão de um título ou cabeçalho adicional no relatório, acima do título principal. A expressão caractere <texto do cabeçalho> é avaliada apenas uma vez, no início do relatório, mas seu resultado é apresentado no topo de cada página impressa.
A cláusula NOEJECT suprime a ejeção automática de uma página de formulário na impressora antes de se iniciar a impressão, quando a cláusula TO PRINT é usada.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
USE Estoque INDEX Indcord
REPORT FORM Saldos TO PRINT FOR saldo > 0
* Gera um relatório com as definições armazenadas no
* arquivo Saldos.frm, dos produtos do estoque com
* saldo maior que zero.
```

Veja Também:

LABEL FORM, LIST

5.11. COMANDOS PARA CONFIGURAÇÃO DO AMBIENTE

5.11.1. Introdução

Os comandos apresentados nesta seção permitem a configuração do ambiente de trabalho de uma aplicação escrita em Clipper. Seu funcionamento está baseado em definições que estabelecem as características do ambiente operacional da aplicação desenvolvida. Podem ser configuradas tanto características do equipamento (impressora, vídeo, acionadores de discos etc.) como características de arquivos a serem utilizados, relacionamentos, filtros de dados etc.

Os comandos de configuração estão agrupados em duas classes de acordo com o seu modo de operação:

1 — **SET <parâmetro> ON / OFF**
2 — **SET <parâmetro> TO <opção>**

A classe 1, SET <parâmetro> ON/OFF, liga ou desliga uma determinada característica do ambiente de trabalho. Por exemplo: SET BELL ON, liga o sinal sonoro (BEEP) de preenchimento de campos no comando @...GET; SET BELL OFF o desliga.

A classe 2, SET <parâmetro> TO <opção> define, entre as opções possíveis, a característica desejada para determinados parâmetros do ambiente. Por exemplo: SET COLOR TO R/G irá modificar as cores de apresentação de dados na tela de branco (W) com fundo preto (N), que é o padrão, para vermelho (R) com fundo (G) verde; SET DEFAULT TO B irá definir como drive padrão de trabalho o drive B; e assim por diante.

A cada execução de um comando **SET** será alterada uma característica do ambiente de trabalho, até que novamente, através da execução de um outro comando SET ela seja restabelecida ou novamente alterada. Os comandos SET definem, em geral, características globais, ou seja, que valem para todos os outros comandos.

Ao ser utilizado, o Clipper assume um ambiente padrão, que define automaticamente várias características de trabalho. Por exemplo, o padrão do sinal sonoro é desligado (SET BELL OFF), enquanto que o padrão de cores no vídeo é letras brancas no fundo preto. Assim, para cada característica possível de ser definida através dos comandos SET existe um padrão já assumido pelo Clipper.

Nos comandos SET que serão apresentados a seguir, a característica padrão será ressaltada:

- para os comandos do tipo SET ON/OFF, a notação utilizada indica que o estado padrão é o que aparece em letras maiúsculas, enquanto o opcional aparece em letras minúsculas;
- para os comandos do tipo SET TO, a característica padrão é indicada nas explicações sobre o comando.

5.11.2. Comandos de Configuração de Ambiente

Comando SET ALTERNATE:

Sintaxe:

SET ALTERNATE on/OFF
SET ALTERNATE TO <nome do arquivo>

Propósito:

O comando SET ALTERNATE grava toda a saída de dados resultante de operações onde não sejam utilizados comandos de tela-cheia @...SAY...GET ("full-screen"), em um arquivo texto (padrão ASCII), como se fosse um controle de "log" de utilização da aplicação.

Utilização:

Utiliza-se o comando SET ALTERNATE quando se deseja registrar toda a operação efetuada durante a utilização do sistema ou programa em Clipper. Todas as saídas de dados, com exceção das efetuadas pelos comandos @...SAY...GET, serão gravados no arquivo texto designado.
O comando SET ALTERNATE está normalmente desligado, ou seja em OFF.
SET ALTERNATE TO <nome do arquivo> especifica e cria o arquivo texto padrão ASCII que irá gravar o registro das operações. Se uma extensão não for informada será assumida a extensão (.txt) e caso o arquivo já exista será sobreposto pelo novo.
SET ALTERNATE ON inicia (liga) a gravação das subseqüentes saídas de dados, no arquivo texto especificado. SET ALTERNATE OFF e finaliza (desliga), sem contudo fechar o arquivo texto ("ALTERNATE"). Ao ser novamente ligada, a gravação será feita no mesmo arquivo, de forma contínua, após a última saída anteriormente gravada.
Para fechar o arquivo "ALTERNATE" pode-se utilizar os comandos CLOSE ALTERNATE, CLOSE ALL ou SET ALTERNATE TO, sem especificar o nome do arquivo.
Enquanto SET ALTERNATE estiver ligado (ON), e até que o arquivo texto especificado (arquivo "ALTERNATE") seja fechado, as saídas de dados serão nele gravadas seqüencialmente. Pode-se utilizar SET ALTERNATE ON e OFF sem fechar o arquivo com CLOSE ALTERNATE para registrar apenas certas operações do sistema.
Um arquivo "ALTERNATE" não é relativo à uma específica área de trabalho, é global, e por esse motivo não pode haver mais de um arquivo "ALTERNATE" aberto simultaneamente.
Lembre-se que as saídas de dados geradas pelos comandos @...SAY...GET não são gravadas no arquivo "ALTERNATE". Entretanto é possível gravá-las utilizando o comando SET PRINTER TO <nome do arquivo>.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
SET ALTERNATE TO Controle      && Abre o arquivo texto.
SET ALTERNATE ON               && Inicia a gravação.
     .
     .
<comandos>
     .
     .
SET ALTERNATE OFF              && Interrompe a gravação.
     .
     .
<outros comandos>
     .
     .
SET ALTERNATE ON               && Reinicia a gravação.
     .
     .
<mais comandos>
     .
     .
SET ALTERNATE OFF              && Interrompe a gravação.
CLOSE ALTERNATE                && Fecha o arquivo texto.
```

Veja Também:

CLOSE, DISPLAY, LABEL FORM [TO FILE], LIST [TO FILE], REPORT FORM [TO FILE], SET PRINTER TO, TEXT [TO FILE], TYPE [TO FILE], FCLOSE( ), FCREATE( ), FERROR( ), FOPEN( ), FREAD( ), FREADSTR( ), FSEEK( ), FWRITE( )

Comando SET BELL:

Sintaxe:

SET BELL on/OFF

Propósito:

Liga ou desliga o sinal sonoro (BEEP) de aviso emitido durante as operações em tela-cheia (comandos @...SAY...GET).

Utilização:

O sinal sonoro é útil para chamar a atenção de um usuário distraído durante a digitação de dados através de comandos @...SAY...GET, pois o sinal ("beep") ecoará em duas situações se o SET BELL estiver ligado (ON):

- ao se preencher um campo de entrada de dados até sua última posição, indicando que o campo foi completado e o cursor passou para o seguinte;
- ao se digitar algum dado de tipo inválido, por exemplo, caracter onde se esperava numérico.

O sinal sonoro está normalmente desligado (OFF). Para ligá-lo basta executar o comando SET BELL ON. O sinal permanecerá , então, ligado até que seja desligado pelo comando SET BELL OFF.
Uma outra forma de fazer soar o sinal sonoro ("beep") quando se desejar, é executar o comando ?? CHR(7), onde 7 é o código ASCII para o sinal sonoro.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
SET BELL ON
vbell=SPACE(1)
@ SAY "Digite uma letra e ouça o sinal: " GET vbell
READ
SET BELL OFF
```

Veja Também:

SET CONFIRM, CHR( )

Comando SET CENTURY:

Sintaxe:

SET CENTURY on/OFF

Propósito:

Permite a entrada e a apresentação de datas, em variáveis tipo data ou campos tipo data, com o prefixo do século no ano. O padrão é desligado, assumindo-se o século XX (1900 - 1999).

Utilização:

Utiliza-se quando se deseja digitar ou apresentar o ano completo, ou seja, 1988 ao invés de apenas 88, que é o normal.
Quando o SET CENTURY estiver ligado (ON), os dados tipo data terão o ano apresentado com quatro dígitos. Quando desligado (OFF) o ano será apresentado apenas com dois dígitos; neste caso o Clipper assume que se trate do século XX (1900 a 1999).
SET CENTURY está normalmente desligado (OFF), portanto o século do ano não pode ser digitado e nem apresentado nas variáveis ou campos do tipo data.
O padrão é o século 20 (19xx), mas caso outro século seja digitado ou resultar do cálculo com datas, o século correto será armazenado nas variáveis ou campos de arquivos, embora não seja apresentado se o SET CENTURY estiver desligado (OFF). O Clipper suporta uma faixa de datas iniciando-se em 01/01/0100 e finalizando-se em 31/12/2999.
SET CENTURY afeta os comandos de @..SAY...GET de tela-cheia. Quando SET CENTURY estiver desligado o formato de entrada ou apresentação de datas será "99/99/99". Quando SET CENTURY estiver ligado o formato será "99/99/9999", aumentando, portanto o campo de digitação do ano para quatro dígitos.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
? DATE()
12/01/88
SET CENTURY ON
? DATE()
12/01/1988
SET CENTURY OFF
? DATE()
12/01/88
```

Veja Também:

SET DATE, CTOD( ), DATE( ), DTOC( ), DAY( ), MONTH( ), YEAR( )

Comando SET COLOR TO:

Sintaxe:

SET COLOR TO [<**normal**>] [<**realçado**>] [[,<**borda**>] [,<**fundo**>] [,<**neutro**>]]

Propósito:

Permite a definição de cores a serem utilizadas nas apresentações da tela.

Utilização:

O comando SET COLOR TO permite alterar as cores (para monitores coloridos) ou tonalidades (para monitores monocromáticos) nas operações de "desenho" de tela ou apresentação de dados.
As cores padrão do vídeo são:

- **Vídeo Normal:** branco com fundo preto, para apresentação normal dos dados (comando @...SAY e outros como LIST, DISPLAY etc.).
- **Vídeo Realçado:** preto com fundo branco, para realce da entrada de dados (comando @...GET).
- **Borda:** preto para a moldura da tela.
- **Fundo:** opção ainda não disponível para os equipamentos atuais.
- **Vídeo Neutro:** permite que o comando GET atual (onde está o cursor) seja apresentado em vídeo realçado, enquanto que os outros GET's (dentro de um mesmo comando READ) são apresentados nas cores definidas pelo vídeo neutro. O padrão é preto com fundo branco.

O vídeo normal, o vídeo realçado e o neutro podem ser especificados por um par de cores correspondentes à frente (cor dos caractéres apresentados) e ao fundo (cor da tela sobre a qual os caractéres são apresentados). Cada cor é identificada por uma letra ou um número:

Tabela de Cores e Atributos

Cor	Letra	Número
Preto	N	0
Azul	B	1
Verde	G	2
Palha	BG	3
Vermelho	R	4
Magenta	RB	5
Marron	GR	6
Branco	W	7
Cinza	N+	
Amarelo	GR+	
Ausente	X	
Sublinhado	U	
Inverso	I	
+	Alta intensidade	
*	Piscante	

Além das cores, ainda pode ser especificada alta-intensidade, através do sinal "+" e vídeo piscante, através do sinal "*".
Cada argumento de cores deve ser separado por vírgulas e cada conjunto de duas cores, definindo respectivamente a frente e o fundo, deve ser separado por uma barra:

SET COLOR TO

frente/fundo,frente/fundo,frente,frente/fundo,frente/fundo

normal	realçado	borda	fundo	neutro

As cores podem ser especificadas tanto pelos números como pelas letras, como parâmetros. Contudo, não se pode misturar letras e números para especificar as cores desejadas. Os sinais "+" e "*" somente podem ser usados com letras e afetam somente a "frente" do vídeo e não o fundo. Por exemplo:

SET COLOR TO W+*/R,N/W,N,,G/B

Os atributos sublinhado (U) e inverso (I) somente são válidos para vídeo monocromático.
SET COLOR TO sem argumentos restabelece as cores padrões do vídeo, isto é: W/N,N/W,N/W,,,N/W.

Para facilitar a manipulação de cores é interessante construir um conjunto de variáveis , as **"variáveis de cor"**, cujo conteúdo sejam as combinações de cores que você está utilizando. As variáveis padrão de cores facilitarão inclusive a definição de cores quando o seu sistema for executado tanto em equipamentos que possuem vídeo colorido como naqueles com vídeo monocromático.
Para definir uma "variável de cor" armazene nela os argumentos das cores como uma cadeia de caracteres. Em seguida utilize a função & (macro-substituição) quando você quiser definir para o vídeo a cor daquela variável. Por exemplo:

```
vcor1 = "W+/R,N/W"
SET COLOR TO &vcor1
```

Recomendamos, ainda, que sejam dados nomes parecidos às variáveis de cor, por exemplo: vcor1, vcor2 etc. Dessa forma você poderá facilmente "salvar" estas variáveis em um arquivo de configuração de cores, através do comando SAVE TO, e utilizá-las em todos os seus sistemas.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
SET COLOR TO W+/R,W/G,GR,,N/W
```

* Coloca branco intenso com fundo vermelho para o vídeo normal, branco com fundo verde para o vídeo realçado, marrom para a moldura, indefinido para a opção fundo e preto com fundo branco para o vídeo neutro.

```
SET COLOR TO N*/W,W/N,N
```

* Coloca preto piscante com fundo branco para o vídeo normal e branco com fundo preto para o vídeo realçado.

```
SET COLOR TO W+/N, X
```

* Coloca branco intenso e preto no vídeo normal e ausencia de cor no vídeo realçado. Esta combinação é interessante para dar entrada em senhas ("passwords") através do comando @...SAY...GET, sem que a senha digitada seja visível. Por exemplo:

```
SET COLOR TO W+/N, X
senha=SPACE(4)
@ 22,30 SAY "Digite a Senha: " GET senha;
                              VALID Senha(senha,"XYZK")
READ
SET COLOR TO
RETURN

FUNCTION Senha
PARAMETERS vsenha, vchave
IF vsenha()vchave
   SET COLOR TO W+*/N
   @ 24,32 SAY "Senha Inválida, Pressione uma Tecla..."
   SET COLOR TO W+/N,X
   INKEY(0)
   @ 24,00 CLEAR
   RETURN .F.
   * Retorna falso, ou seja, senha inválida.
ENDIF
RETURN .T.
* retorna verdadeiro, ou seja, senha válida.
```

O exemplo seguinte torna o GET atual (onde está o cursor) preto com fundo branco, enquanto os outros aparecerão brancos intensos com o fundo vermelho:

```
corget = "W/N,N/W,,W+/R"
SET COLOR TO &corget
STORE SPACE(30) TO nome1,nome3,nome3
@ 10,10 SAY "Digite o Primeiro Nome:" GET nome1
@ 12,10 SAY "Digite o Segundo Nome.:" GET nome2
@ 14,10 SAY "Digite o Terceiro Nome:" GET nome3
READ
SET COLOR TO
```

Veja Tambem:

SET INTENSITY, ISCOLOR( ), SETCOLOR( )

Comando SET CONFIRM:

Sintaxe:

SET CONFIRM on/OFF

Propósito:

O comando SET CONFIRM é utilizado na edição em tela-cheia (comandos @...SAY..GET) para determinar se o cursor deve mover-se automaticamente

para a próxima variável ou campo editado, quando é atingido o fim do anterior, ou se o usuário precisa teclar <Enter> para confirmar.

Utilização:

Normalmente, quando é preenchido um campo de entrada de dados através de comandos @...GET, o cursor automaticamente passa para o campo seguinte. Neste caso o SET CONFIRM está OFF (desligado), que é o padrão; não é necessária confirmação.
Quando se coloca o SET CONFIRM em ON (ligado), ao terminar o preenchimento de um campo de entrada de dados, obrigatoriamente deve ser digitado <Enter>, ou outra tecla de finalização, para confirmar a passagem do cursor para o próximo campo. SET CONFIRM ON retém o cursor na última posição do campo preenchido até que seja pressionada uma das teclas de finalização listadas a seguir:

Ctrl-Home, Ctrl-End, Seta para Cima, Seta para Baixo, Ctrl-C, PgUp, Ctrl-W, PgDn, Esc ou Enter (Return).

Utiliza-se SET CONFIRM ON em casos onde haja muita incidência de erros de digitação devido à passagem automática para o próximo campo quando do preenchimento do campo anterior.

Biblioteca:

CLIPPER.LIB

Veja Também:

@...SAY...GET, SET BELL

Comando SET CONSOLE:

Sintaxe:

SET CONSOLE ON/off

Propósito:

Liga ou desliga a apresentação de dados gerados pelos comandos no vídeo.

Utilização:

O comando SET CONSOLE afeta todas as saídas ou mensagens enviadas para a tela por todos os comandos do Clipper, com exeção do comandos de tela-cheia: @...SAY...GET, @...PROMPT, @...BOX, @...CLEAR TO, @...TO e CLEAR.
A saída para o vídeo está normalmente ligada, ou seja, SET CONSOLE está ON, que é o padrão.
O SET CONSOLE em OFF desliga apenas a apresentação de dados na tela, não afetando a saída para a impressora. Deve ser utilizado para evitar, por exemplo, a apresentação de caracteres de configuração de impressoras (utilizando-se ?? CHR(nn)) na tela, ou um relatório, gerado pelo comando REPORT FORM, simultaneamente na tela e na impressora.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
USE Mala
SET CONSOLE OFF
SET COLOR TO W*
@ 12,30 SAY "Aguarde... Imprimindo"
SET COLOR TO
LIST nome, endereco, telefone TO PRINT
* Os dados não serão apresentados na tela, apenas
* listados na impressora.
SET CONSOLE ON
CLEAR
RETURN
```

Comando SET CURSOR

Sintaxe:

SET CURSOR ON/off

Propósito:

Ligar ou desligar a apresentação do cursor na tela.

Utilização:

O comando SET CURSOR (disponível apenas nas versões Summer 87 e posteriores) permite que se opte por apresentar ou não o cursor na tela.

Quando o cursor é desligado, através de SET CURSOR OFF, as entradas através do teclado e as apresentações na tela não são afetadas. O cursor apenas não estará sendo mostrado, todas as operações de entrada de dados permanecem normais e dados poderão ser digitados.
SET CURSOR ON, que é o padrão, novamente restabelece a apresentação do cursor, útil para indicar onde deverá ser feita a próxima digitação.
A principal utilização deste comando é suprimir a desagradável apresentação do cursor quando está se "desenhando" uma tela ou apresentando dados para monitoração do usuário. A rigor, os únicos momentos em que o cursor deve ser apresentado em uma aplicação sofisticada é quando está se editando (entrando ou alterando) dados em tela-cheia (com comandos @...SAY...GET), usando o editor de campos memo (a função MEMOEDIT( )), ou em qualquer outro tipo de edição de dados.

Biblioteca:

CLIPPER.LIB

Exemplo:

No exemplo a seguir, é solicitada a confirmação sem a apresentação do cursor:

```
SET CURSOR OFF
confirme=SPACE(1)
SET COLOR TO N/W,N/W
@ 24,60 SAY "Confirme (S/N) ?" GET confirme;
                               VALID confirme$"SNsn"
READ
@ 24,00 CLEAR
SET COLOR TO
SET CURSOR ON
```

Veja Também:

SET CONSOLE

Comando SET DATE:

Sintaxe:

SET DATE AMERICAN/ANSI/BRITISH/ITALIAN/FRENCH/GERMAN

Propósito:

Permite a determinação do formato de entrada e apresentação das expressões que retornam datas ou de variáveis e campos tipo data.

Utilização:

O formato padrão das datas é o AMERICAN (americano), que as apresenta como mes/dia/ano.
Este comando possibilita grande flexibilidade na apresentação das datas. Os formatos possíveis são:

AMERICAN	=	mes/dia/ano
ANSI	=	ano.mes.dia
BRITISH	=	dia/mes/ano
ITALIAN	=	dia-mes-ano
FRENCH	=	dia/mes/ano
GERMAN	=	dia.mes.ano

O comando SET DATE realiza uma definição global, afetando todas as datas apresentadas ou entradas durante a execução do programa. Os formatos mais utilizados no Brasil são o inglês ou francês (BRITISH ou FRENCH) e o italiano (ITALIAN).

Biblioteca:

CLIPPER.LIB

Exemplos:

```
? DATE()
01/19/88
SET DATE BRITISH
? DATE()
19/01/88
SET DATE ITALIAN
? DATE()
19-01-88
```

Veja Também:

SET CENTURY, DATE( ),CTOD( ),DTOC( )

Comando SET DECIMALS:

Sintaxe:

SET DECIMALS TO <exp.numérica>

Propósito:

O comando SET DECIMALS define o número mínimo de casas decimais que deve ser apresentado nos resultados de expressões numéricas ou cálculos realizados pelos comandos e funções do Clipper.

Utilização

O padrão normal de apresentação é de duas casas decimais.
Se o comando SET FIXED estiver desligado (OFF), SET DECIMALS se aplica apenas aos resultados de expressões matemáticas envolvendo divisões e ao resultado das funções SQRT( ), LOG( ) e EXP( ). Se, por outro lado, SET FIXED estiver em ON (ligado) todos os resultados numéricos (de qualquer expressão ou dado) serão apresentados com o número de casas decimais estabelecido por SET DECIMALS.
A <expressão numérica> é um número que determina a quantidade de casas decimais a serem apresentadas.
Para cálculos que não envolvem multiplicação, o número de casas decimais no resultado é o mesmo do número com a maior quantidade de casas decimais.
Para cálculos que envolvem multiplicação, o número de casas decimais no resultado é a soma do número de casas decimais de todos os números sendo multiplicados.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
? 3/4
0.75
? 3.00/4.000
0.750
? SQRT(3.85)
1.96
SET DECIMALS TO 4
? 3/4
0.7500
? SQRT(3.85)
1.9621
```

Veja Também:

SET FIXED

Comando SET DEFAULT:

Sintaxe:

SET DEFAULT TO <drive> [:<diretório>]

Propósito:

Permite estabelecer o acionador de discos ("drive") e o diretório que serão utilizados para todas as operações com arquivos, a menos dos que forem diretamente especificados.

Utilização:

Utiliza-se o comando SET DEFAULT para estabelecer o drive e o diretório onde serão feitas todas as operações de leitura e gravação com arquivos. O drive e opcionalmente o diretório especificados serão o padrão; caso outros sejam utilizados, deverão ser diretamente especificados nos comandos.
O <drive> é a letra que identifica o nome do acionador de discos a ser utilizado, normalmente A, B, C ou D.
O <diretório> identifica o nome do diretório de arquivos que deverá ser utilizado como padrão. Se forem especificados ambos, o drive e o diretório, os dois pontos devem ser incluídos após a letra do drive.
Quando se executa um programa escrito em Clipper, o drive e o diretório assumidos como padrão são os mesmos que estão como padrão para o Sistema Operacional. Para redefini-los utiliza-se o comando SET DEFAULT TO.
O comando SET DEFAULT não checa se o drive ou diretório especificados existem realmente. Eles não são mudados a nível de Sistema Operacional; o Clipper apenas direciona todas as operações de arquivo para o drive e diretório especificados.
SET DEFAULT TO sem argumentos redireciona as operações com arquivos para o último diretório utilizado no drive designado.
O comando SET DEFAULT não constitui a definição de um caminho de pesquisa para o acesso a arquivos. Para definir um caminho de pesquisa utiliza-se o comando SET PATH. SET DEFAULT é utilizado principalmente para especificar onde os arquivos principais do sistema desenvolvido se localizam.

Exemplo:

* Se o programa em Clipper for carregado a partir do drive A, que é padrão para o Sistema Operacional, e todos os arquivos estão armazenados no drive B, utilize o comando SET DEFAULT TO B antes de abrir os arquivos:

* Sem usar o SET DEFAULT

```
CLEAR
SELECT 1
USE B:Estoque INDEX B:IndCod ALIAS EST
SELECT 2
USE B:Produtos INDEX B:IndProd ALIAS Prod

* Usando o SET DEFAULT

CLEAR
SET DEFAULT TO B
SELECT 1
USE Estoque INDEX IndCod ALIAS EST
SELECT 2
USE Produtos INDEX IndProd ALIAS Prod

SET DEFAULT TO C:\DADOS
* Direciona todas as operações com arquivos para o
drive C, no diretório \DADOS.
```

Veja Também:

SET PATH

Comando SET DELETED:

Sintaxe:

SET DELETED OFF / on

Propósito:

O comando SET DELETED determina se os registros marcados pelo comando DELETE serão (ON) ou não (OFF) ignorados no processamento do sistema. Ele filtra os registros marcados para eliminação, ignorando-os no processamento da maioria dos comandos.

Utilização:

Os registros marcados para eliminação pelo comando DELETE são normalmente processados até que sejam definitivamente excluídos pelo comando PACK. Quando, mesmo sem executar o comando PACK deseja-se ignorar os registros deletados no processamento, usa-se o comando SET DELETED ON.

O comando SET DELETED quando em ON (ligado) é muito útil para excluir os registros deletados do processamento sem a necessidade de executar constantes "PACK's", que tornariam o processamento muito lento, principalmente quando os arquivos de dados forem grandes ou possuírem muitos índices associados.

SET DELETED está normalmente OFF, ou seja, os registros deletados participam normalmente de todo o processamento.

Quando SET DELETED estiver em ON, a maioria dos comandos do Clipper ignorará os registros deletados. Contudo, se um registro deletado for referido pelo seu número (GOTO ou RECORD <n>) é processado, mesmo estando deletado.

O comando SET DELETED não tem nenhum efeito sobre os comandos INDEX e REINDEX, que sempre processarão todos os registros do arquivo. Já o comando RECALL ALL é afetado, não desmarcando nenhum registro se o SET DELETED estiver ON.

Para novamente incluir os registros deletados no processamento dos comandos executa-se o comando SET DELETED OFF.

Com o SET DELETED ON, a maioria dos comandos agem como se os registros deletados não existissem. Por exemplo, os comandos LOCATE, LIST, SEEK etc irão ignorá-los.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
CLEAR
USE Mala INDEX IndCod
LOCATE FOR nome="JOSE DA SILVA"
IF FOUND()
   ? RECNO(), nome      && O registro será encontrado
ELSE
   ? "Registro Não Encontrado !
ENDIF
DELETE
LOCATE FOR nome="JOSE DA SILVA"
IF FOUND()
   ? RECNO(), nome      && O registro será encontrado,
ELSE                    && mesmo tendo sido deletado.
   ? "Registro Não Encontrado !"
```

```
ENDIF
SET DELETED ON
LOCATE FOR nome="JOSE DA SILVA"
IF FOUND()
   ? RECNO(), nome
ELSE
   ? "Registro Não Encontrado !"  && O registro Não
ENDIF                             && será encontrado.
SET DELETED OFF
USE
RETURN
```

Veja Também:

DELETED, DELETE( ), RECALL, PACK, INDEX, REINDEX, SET INDEX, USE

Comando SET DELIMITERS:

Sintaxe:

SET DELIMITERS on / OFF
SET DELIMITERS TO [<expressão caracter>/DEFAULT]

Propósito:

Definir caracteres para delimitar a dimensão dos campos de edição de dados através do comando @...GET.

Utilização:

O comando SET DELIMITED está normalmente desligado (OFF). Os campos de edição em tela cheia são delimitados através de vídeo reverso (preto com fundo branco).
SET DELIMITERS ON inclui "dois-pontos" (:) no início e no fim dos campos para delimitar a área (comprimento) de digitação de dados.
Para se determinar outro caractére para delimitar os campos de edição dos comandos @...GET, que não os dois pontos, utiliza-se o comando SET DELIMITER TO <exp.caracter>. A expressão caractére deve conter um ou dois caracteres delimitadores. Se contiver apenas um, este caracter marcará o início e o fim do campo de edição. Se contiver dois, o primeiro marcará o início do campo e o segundo marcará o fim.
O comando SET DELIMITERS deve estar ligado (ON), para que os caracteres definidos pelo SET DELIMITERS TO sejam apresentados.

Para apresentar apenas os caracteres delimitadores para a definição do tamanho do campo de edição, e não o vídeo-reverso, utiliza-se o comando SET INTENSITY que liga (ON) ou desliga (OFF) o vídeo realçado para os comandos @...GET.
Quando são executados comandos @...GET e READ, a última definição de delimitação do campo é utilizada.
Especificando-se SET DELIMITERS TO DEFAULT ou nenhum delimitador retorna-se aos delimitadores padrões, ou seja, os dois pontos.

Biblioteca:

CLIPPER.LIB

Exemplos:

Para delimitar os campos de edição com #:

```
SET DELIMITERS TO "#"
SET DELIMITERS ON
SET INTENSITY OFF
```

Para delimitar o início do campo com [e o fim com] :

```
SET DELIMITERS TO "[]"
SET DELIMITERS ON
SET INTENSITY OFF
var=SPACE(5)
@ 10,20 SAY "Digite o Dado: " GET var
READ
* O resultado será: Digite o Dado: [     ]
* Sem vídeo reverso
SET DELIMITERS OFF
SET INTENSITY ON
RETURN
```

Veja Também:

@...SAY...GET, READ, SET INTENSITY

Comando SET DEVICE:

Sintaxe:

SET DEVICE TO <print / SCREEN>

Propósito:

Determina se os comandos @...SAY serão direcionados para a tela (SCREEN) ou para a impressora (PRINT).

Utilização:

Normalmente os comandos @...SAY são direcionados para a tela, ou seja, o SET DEVICE está em SCREEN, aque é o padrão.
Especificando SCREEN todas as saídas @...SAY serão direcionadas para a tela, independentemente do estado dos campos SET PRINT e SET CONSOLE.
Para redirecionar os comandos @...SAY para a impressora, por exemplo para a geração de relatórios, executa-se o SET DEVICE TO PRINT.
Expecificando-se PRINT como DEVICE (dispositivo), todas as saídas @...SAY serão direcionadas para a impressora ou para o dispositivo especificado pelo comando SET PRINTER TO, que poderá ser uma porta de impressão, um spool de impressão (para utilização em rede) ou um arquivo no disco.
Com o SET DEVICE TO PRINT os comandos @...SAY são enviados para a impressora não sendo apresentados na tela. Os comandos @...GET encontrados serão ignorados.
Se for executado um comando @...SAY requerendo que a impressora retorne uma linha, ocorrerá uma ejeção de página e a impressão será feita na linha determinada só que da página seguinte.
Um comando EJECT também ejeta uma página de formulário, redefinindo o controle interno de linha (PROW( )) e coluna (PCOL( )) da impressora para zero (linha zero e coluna zero). Além disso pode-se especificar novos valores para o controle interno da linha e coluna da impressora através da função SETPRC( ).
Para enviar as saídas dos comandos @...SAY para um arquivo utilize o comando SET PRINTER TO <nome do arquivo> após ter "setado" o SET DEVICE TO PRINT.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
CLEAR
@ 10,20 SAY "Este é um Exemplo"
* será apresentado na tela
SET DEVICE TO PRINT       && dispositivo a impressora
@ 10,20 SAY "Este é um Exemplo"
* será impresso pela impressora
SET DEVICE TO SCREEN      && dispositivo tela
@ 12,20 SAY "Este é outro Exemplo"
* será novamente apresentado na tela
```

Veja Também:

@...SAY, EJECT, SET PRINTER TO, SET PRINT, PROW( ), PCOL( ), SETPRC( )

Comando SET ESCAPE:

Sintaxe:

SET ESCAPE ON /off

Propósito:

Habilita ou desabilita as teclas <Alt> e <C> <Alt-C> pressionadas simultaneamente) para a interrupção da execução de um programa, ou a tecla <Esc> para a finalização de um READ (edição em tela-cheia).

Utilização:

O comando SET ESCAPE está normalmente ligado (ON), permitindo que se interrompa a execução de um programa a qualquer momento apenas pressionando-se simultaneamente as teclas <Alt> e <C>.
Ao serem pressionadas a execução do programa será interrompida e no topo da tela, à direita, aparecerá a pergunta: **QUIT? (Q/A/I)**.
Teclando-se a letra Q (QUIT), serão fechados corretamente todos os arquivos que estiverem abertos e a execução do programa será finalizada. Teclando-se a letra A (ABORT) o programa será cancelado instantaneamente; os arquivos em uso não serão corretamente fechados. Deve-se evitar "Abortar" um programa pois poderá ocorrer perda de dados dos arquivos que estiverem abertos no momento. Finalmente, digitando-se a letra I (IGNORE), o programa voltará a ser normalmente executado, a partir do ponto em que foi interrompido.
Se SET SET ESCAPE estiver em ON (ligado) também será possível terminar um comando READ contornando a execução da cláusula VALID para o comando @...GET atual.
Se for estabelecido SET ESCAPE OFF, a execução do programa não poderá mais ser interrompida através das teclas <ALT-C> e a tecla <Esc> não mais terminará um READ. A única maneira de interromper a execução do programa será desligando o equipamento ou forçando um "boot" (recarga do Sistema Operacional) através das teclas <Ctrl-Alt-Del>. Já a única maneira de finalizar READ com uma cláusula VALID pendente será digitar um dado válido.

O SET ESCAPE OFF é, portanto, útil quando se deseja que o usuário do sistema desenvolvido em Clipper não o interrompa durante um determinado processamento ou burle a entrada de dados válidos.
Para habilitar novamente a interrupção pelas teclas <Alt-C> e a tecla <Esc>, basta executar o SET ESCAPE ON.
Na versão Summer 87, ao se pressionar as teclas <Alt-C> será apresentada no topo da tela a mensagem "Paused...Continue ?" Deve-se teclar "N" (no-não) caso se queira encerrar a execução do programa (equivalente ao "Q" (Quit) das versões anteriores) e "Y" (yes-sim), caso se deseje continuar a execução do programa (equivalente ao "I" (Ignore)). Não é mais permitido "abortar".

Veja Também:

SET KEY, READ

Comando SET EXACT:

Sintaxe:

SET EXACT on/OFF

Propósito:

Determina como a comparação entre duas cadeias de caracteres é avaliada, afetando o resultado de uma condição que estiver sendo avaliada.

Utilização:

SET EXACT normalmente está desligado (OFF). Neste caso as comparações entre cadeias de caracteres (strings) iniciam-se do caractere mais à esquerda e continuam, caractere por caractere, até o fim da cadeia que está do lado direito da operação de comparação. Se as duas cadeias comparadas são iguais até este ponto, o resultado da comparação será verdadeiro (.T.). Deve ser ressaltado que se a cadeia do lado direito da comparação for uma cadeia nula (comprimento zero), nenhuma comparação será feita, entretanto o resultado da mesma será verdadeiro (.T.).
Ligando-se o SET EXACT (SET EXACT ON), as duas cadeias de caracteres devem ser **exatamente iguais,** caractere por caractere (inclusive em seu comprimento, letras maiúsculas e minúsculas) para que a operação de comparação seja avaliada como verdadeira.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
* SET EXACT está em OFF (padrão)

? "JOSE" = "JOSE DA SILVA"
* Resultará falso (.F.)

? "JOSE DA SILVA" = "JOSE"
* Resultará verdadeiro (.T.)

SET EXACT ON
? "JOSE DA SILVA" = "JOSE"
* Resultará falso (.F.)

? "JOSE DA SILVA" = "JOSE DA SILVA"
* Resultará verdadeiro (.T.) pois as cadeias agora são
* exatamente iguais.
```

Veja Também:

FIND, SEEK, LOCATE, DISPLAY, LIST e o operador "==" (duplo igual)

Comando SET EXCLUSIVE:

Sintaxe:

SET EXCLUSIVE ON/off

Propósito:

Em ambiente multiusuário (ou de rede) habilita ou desabilita o acesso restrito a arquivos de dados, de campos memo e de índices.

Utilização:

O comando SET EXCLUSIVE normalmente está em ON (ligado). Neste caso todos os arquivos de dados, arquivos de índice e campos memo associados são abertos em acesso restrito, ou seja, não podem ser abertos e usados (USE) por outros usuários até serem fechados (CLOSE) pelo usuário que os abriu com exclusividade. Tudo se comporta como se o ambiente não fosse multiusuário.

Quando SET EXCLUSIVE está em OFF (desligado), todo o acesso compartilhado (não restrito) deverá ser controlado através do programa desenvolvido, usando as funções RLOCK( ), FLOCK( ) e o comando USE...EXCLUSIVE, pois os dados dos arquivos poderão ser acessados simultaneamente por usuários diferentes.

Biblioteca:

CLIPPER.LIB

Veja Também:

USE...EXCLUSIVE, FLOCK( ), RLOCK( ), NETERR( )

Comando SET FILTER:

Sintaxe:

SET FILTER TO [<condição>]

Propósito:

Filtrar os registros de um arquivo de dados, fazendo com que este pareça possuir apenas os registros que atendam à condição especificada.

Utilização:

O comando SET FILTER é normalmente utilizado quando se deseja restringir o acesso ou o processamento dos registros de um arquivo apenas àqueles que atendam determinada condição. Todos os comandos do Clipper que requerem um arquivo aberto processarão apenas tais registros e os que não atenderem a condição especificada serão ignorados.
A <condição> é uma expressão lógica que identifica ou especifica os registros a serem processados.
O SET FILTER aplica-se apenas ao arquivo de dados aberto na área de trabalho onde o comando foi executado. Um filtro diferente pode ser estabelecido para cada arquivo de dados em cada área de trabalho.
SET FILTER TO, sem a especificação de uma condição desliga o filtro estabelecido, fazendo com que todos os registros passem novamente a ser processados.
O comando SET FILTER, assim como o SET DELETED, não afeta os comandos INDEX e REINDEX, que processarão normalmente todos os regis-

tros do arquivo. Além disso, um determinado registro poderá ser acessado diretamente, mesmo quando não atender a condição do filtro, através do comando GOTO <n> ou da cláusula RECORD <n> na especificação do escopo dos comandos; <n> é o número do registro desejado.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
USE Estoque INDEX IndCod
SET FILTER TO saldo < 100
LIST codigo, nome, saldo TO PRINT
* serão listados apenas os registros cujo saldo for
* menor que 100.
EJECT
GO TOP
DO WHILE .NOT. EOF()
   REPLACE observ WITH "Ponto de Pedido"
   SKIP
ENDDO
* Da mesma forma, serão processados (colocar 'Ponto de
* Pedido" no campo observ) apenas os registros que
* atenderem à condição do filtro.
SET FILTER TO
* Desliga o Filtro
LIST codigo, nome, saldo, observ TO PRINT
* Lista todos os registros na impressora.
```

Veja Também:

SET DELETED

Comando SET FIXED:

Sintaxe:

SET FIXED on/OFF

Propósito:

Determina se o número fixo de casas decimais definido no comando SET DECIMALS TO será (ON) ou não (OFF) exibido para todos os dados numéricos.

Utilização:

O comando SET FIXED está normalmente desligado (OFF), que é padrão. Neste caso o número de casas decimais que são apresentadas nos resultados de cálculos com dados numéricos segue as seguintes regras, dependendo do tipo de operação efetuada:

- *Soma e subtração*: a mesma quantidade de casas decimais do operando com o maior número de dígitos decimais;
- *Multiplicação*: a soma dos dígitos decimais de todos os operandos envolvidos na multiplicação.
- *Divisão*: a quantidade definida pelo comando SET DECIMALS ou duas, que é o padrão.
- *Exponenciação*: a quantidade definida pelo comando SET DECIMALS ou duas, que é o padrão.
- *Funções* (EXP( ), LOG( ) e SQRT( ): a quantidade definida pelo comando SET DECIMALS ou duas, que é o padrão.
- *Função* VAL( ): a mesma quantidade de casas decimais que o operando.

Se o comando SET DECIMALS TO não tiver sido usado preliminarmente, a quantidade padrão de casas decimais será duas.
Quando o SET FIXED está ON (ligado) o número de casas decimais que será apresentado nos resultados de cálculos ou na saída de dados numéricos é o designado pelo comando SET DECIMALS TO.

Biblioteca:

CLIPPER.LIB

Veja Também:

SET DECIMALS, EXP( ), LOG( ), SQRT( ), VAL( )

Comando SET FORMAT:

Sintaxe:

SET FORMAT TO <arquivo de formato de tela>

Propósito:

Permite ativar (abrir) um arquivo de formatação de tela para a entrada ou edição de dados. Desta forma, sempre que um comando READ for executado a rotina de formatação definida será utilizada para a edição dos dados.

Utilização:

O comando SET FORMAT permite a definição de uma tela especialmente formatada com comandos @...SAY...GET, armazenada em um arquivo de formato com extensão (.fmt), para a edição de dados através do comando READ.
O <arquivo de formato de tela> é o nome do arquivo de formato de um programa (extensão .prg) ou de uma rotina (.prg) a ser utilizado para a edição dos dados em tela-cheia.
Um arquivo de formato é idêntico a qualquer outra rotina escrita em Clipper; a única diferença é o método de chamá-lo: ao invés do DO usa-se o SET FORMAT TO.
Para construir um arquivo de formato, em geral, deve-se criar um arquivo com a extensão (.fmt) que contenha comandos @...SAY...GET adequados à edição dos campos de arquivos ou de variáveis desejados. Ao se definir SET FORMAT TO <nome do arquivo .fmt> o comando READ utilizará o formato definido neste arquivo para proceder a edição dos dados.
O arquivo de formato definido permanecerá aberto, fazendo com que todos os READ's encontrados no programa o utilizem para a entrada de dados. Para fechá-lo pode-se utilizar o comando CLOSE FORMAT ou SET FORMAT TO, sem a especificação de arquivo.
No Clipper pode existir apenas um arquivo de formato ativo a cada momento, não podendo ser definido um arquivo de formato para cada área de trabalho. Além disso não é permitida a intercalação de formatos, ou seja, um arquivo de formato não pode possuir um comando SET FORMAT TO entre suas instruções.
Antes de executar um arquivo de formato, o Clipper não limpa a tela, mas permite que outros comandos (não exclusivamente @...SAY...GET) sejam nele incluídos (por exemplo um CLEAR), fornecendo grande flexibilidade.
Finalmente o Clipper não suporta, como o dBASE III, múltiplas páginas de formato. Comandos READ dentro de um arquivo de formato serão simplesmente ignorados.

Compilação: se para compilar os programas-fonte, estiverem sendo utilizados arquivos ".CLP", os arquivos de formato deverão possuir a extensão (.prg) e não a tradicional (.fmt) para serem corretamente incluídos na compilação, pois o arquivo (.clp) localiza apenas arquivos (.prg).

Recomendação: No Clipper não vale a pena utilizar arquivos de formato. Evite-os, pois não trazem nenhuma vantagem. Utilize rotinas (.prg) para formatação dos dados.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
USE Mala
SET FORMAT TO Telamala
DO WHILE LASTKEY()<>27   && 27 = tecla (Esc)
   APPEND BLANK
   READ
ENDDO
RETURN

* Arquivo Telamala.fmt
CLEAR
@ 07,20 SAY "Mala-Direta - Edição de Dados"
@ 08,20 SAY "                              "
@ 10,10 SAY "Nome: " GET nome
@ 12,10 SAY "Endereco: " GET endereco
@ 14,10 SAY "Telefone: " GET telefone
@ 16,10 SAY "Cidade..: " GET cidade
@ 18,10 SAY "Estado..: " GET estado
@ 20,10 SAY "CEP.....: " GET cep
RETURN
```

Veja Também:

@...SAY...GET, READ, CLOSE FORMAT, SET DEVICE, APPEND BLANK.

Comando SET FUNCTION:

Sintaxe:

SET FUNCTION <exp.numérica> TO <exp.caracter>

Propósito:

Permite definir uma cadeia de caracteres que será enviada ao buffer do teclado (como se fosse uma digitação) quando for pressionada uma tecla de função de 2 à 40. Qualquer comando pode ser programado em uma destas teclas; ao ser pressionada a tecla, o comando será executado.

Utilização:

O comando SET FUNCTION permite que as teclas de função sejam definidas para transmitir, ao teclado do computador, uma expressão de comandos, em forma de uma cadeia de caracteres.
A <expressão numérica> define o número da tecla de função a receber a definição.
A <expressão caractere> é uma cadeia de caracteres com até 2.000 caracteres (para a versão Summer 87) que será submetida ao teclado quando a tecla de função correspondente for pressionada em qualquer estado de espera do programa (@...GET, WAIT, ACCEPT, INPUT etc).
A cadeia de caracteres enviada pode conter caracteres de controle, como <Crtl-C> (que é equivalente a um PgDn), para finalizar um comando READ, por exemplo.
O comando SET KEY tem precedência sobre o comando SET FUNCTION se a ambos for assinalada a mesma tecla.
A tecla F1 não pode ser definida pois é associada à facilidade de "HELP" do Clipper, vide Cap. 7.
As teclas de função de 2 a 40 podem ser definidas através deste comando. Portanto, a <expressão numérica> que define o número da tecla somente pode tomar valores entre estes limites.
As teclas de função de 2 a 10 são ativadas pressionando-se a tecla de função correspondente, ou seja, de F2 a F10.
As teclas de função de 11 a 20 são ativadas pressionando-se simultaneamente a tecla de maiúsculas (Shift) e a tecla de função apropriada, ou seja, de Shift-F1 a Shift-F10.
As teclas de função de 21 a 30 são ativadas pressionando-se simultaneamente a tecla de controle (Ctrl) e a tecla de função apropriada, ou seja, de Ctrl-F1 e Ctrl-F10.
Finalmente, as teclas de função de 31 a 40 são ativadas pressionando-se simultaneamente a tecla alternativa (Alt) e a tecla de função apropriada, ou seja, de Alt-F1 a Alt-F10.

Biblioteca:

CLIPPER.LIB

Exemplos:

* Para definir a tecla F2 com a execução do programa "pesquisa.prg" utilize:

```
SET FUNCTION 2 TO "DO Pesquisa"
```

* Pressionando-se F2 o programa pesquisa será executado.

Veja Também:

SET KEY, INKEY( )

Comando SET INDEX:

Sintaxe:

SET INDEX TO <lista de arquivos de índice>

Propósito:

Permite abrir, na área de trabalho atualmente selecionada, os arquivos de índice (tipo .ntx) na ordem expecificada na lista, associados a arquivos de dados (tipo .dbf).

Utilização:

Até sete índices (com os nomes separados por vírgula) podem ser abertos com o comando SET INDEX, desde que eles já tenham sido previamente criados pelo comando INDEX ON para ordenar o arquivo de dados em uso. O comando SET INDEX não cria os índices.

Na versão Summer 87 e posteriores, utilizando-se o DOS 3.3 ou maior, podem ser abertos até 15 arquivos de índice por arquivo de dado. Também neste caso os índices poderão ser índices normais do Clipper (.ntx) ou então compatíveis com o dBASE III Plus (.ndx). O comando SET INDEX assumirá a extensão (.ntx) ou (.ndx) conforme o caso.

A <lista de arquivos de índice> deverá ser composta pelos nomes dos arquivos a serem abertos na área de trabalho atual, separados por vírgulas. Pode ser incluído o nome do drive e ou do diretório como parte do nome do arquivo de índice. A extensão (.ntx) será assumida a menos se outra seja especificada.

O primeiro índice da lista é o índice mestre ou índice de controle. Ele define e controla a ordem de acesso aos registros do arquivo de dados. Contudo, todos os índices ativos (abertos) são automaticamente atualizados quando ocorrer alguma modificação (inclusão, alteração ou exclusão de registros) durante o processamento do arquivo de dados associado.

Para definir outro índice mestre, dentre os índices abertos, sem ter que reabrir os arquivos de índice através do comando SET INDEX, utiliza-se o comando SET ORDER TO.

Ao se executar o comando SET INDEX o ponteiro de registros é posicionado no início do arquivo (primeiro registro lógico) de acordo com o índice mestre (o primeiro da lista).

Os arquivos de índice também podem ser especificados utilizando a função macro-substituição &. Entretanto, cada arquivo de índice deve ter sua macro-

expressão. Se uma das macro-expressões resultar em espaço será ignorada. O comando SET INDEX TO, sem a especificação de um arquivo de índice, fecha todos os índices que estiverem abertos na área de trabalho selecionada. É equivalente ao comando CLOSE INDEX.

Biblioteca:

CLIPPER.LIB

Exemplos:

* Para abrir os índices Indcod, Índnome e Indata associados ao arquivo do Estoque, sendo a ordem de acesso de acordo com índice Indcod (ordem de código), utiliza-se:

```
USE Estoque
SET INDEX TO Indcod, Indnome, Indada
```

* Utilizando a macro-função tem-se:

```
ind1="Indcod"
ind2="Indnome"
ind3="Indata"
USE Estoque
SET INDEX TO &ind1, &ind2, &ind3

SET INDEX TO   && fecha os indices abertos.
```

Veja Também:

CLOSE, INDEX, REINDEX, USE, SET ORDER

Comando SET INTENSITY:

Sintaxe:

SET INTENSITY ON/off

Propósito:

Determina se os campos de entrada/edição de dados criados pelos comandos @...GET serão apresentados em vídeo reverso (realçado) ou em vídeo normal.

Utilização:

Este comando permite selecionar, para os campos de edição, entre dois padrões de atributos de vídeo: vídeo normal ou realçado. Os atributos de vídeo são selecionados pelo comando SET COLOR TO.
SET INTENSITY está normalmente ligado (ON), indicando que os campos de entrada/edição de dados do comando @...GET serão apresentados em vídeo realçado (letras pretas em fundo branco/verde), que é o padrão do atributo vídeo realçado.
Quando o SET INTENSITY está desligado (OFF), o vídeo normal é utilizado também nos campos de entrada/edição de dados. Neste caso os dados editados dentro dos GET's aparecerão da mesma cor que as saídas dos SAY's.
SET INTENSITY não tem efeito nenhum em outros comandos ou funções como MENU...TO, ACHOICE( ) e DBEDIT( ).

Biblioteca:

CLIPPER.LIB

Veja Também:

SET COLOR, @...SAY...GET

Comando SET KEY:

Sintaxe:

SET KEY <exp.numérica> TO [<rotina/procedure>]

Propósito:

Permite que uma rotina/procedure seja executada a partir de qualquer estado de espera do sistema, quando a tecla designada for pressionada.

Utilização:

A <expressão numérica> é o código da tecla desejada, de acordo com a tabela de códigos apresentada na função INKEY( ).
A <rotina ou "procedure"> é um programa a ser executado quando a tecla designada for pressionada pelo usuário.
Se a rotina não for especificada a tecla designada terá sua definição anterior cancelada.

O comando SET KEY permite que, a partir de qualquer estado de espera do sistema, ao se pressionar a tecla designada, executar a rotina especificada. Um estado de espera é definido como o estado de qualquer comando que cause uma pausa na execução do programa como por exemplo: WAIT, READ, ACCEPT, INPUT e MENU...TO (a função INKEY( ) não caracteriza um estado de espera).
No máximo 32 teclas podem ser definidas a cada vez pelo comando SET KEY. A tecla F1 é assumida automaticamente pelo Clipper como sendo Help ao se iniciar a execução de um programa, ou seja: SET KEY 28 TO Help.
Como ocorre no Help.prg (veja o Cap. 7 – A Facilidade de Help ao Usuário) são passados automaticamente três parâmetros à rotina chamada pela tecla pressionada. Os parâmetros são:

- O nome do programa atual: prog (caractere)
- A linha sendo executada: lin (numérica)
- A variável sendo esperada: var (caractere)

Como regra evite utilizar os comandos CLEAR ou READ dentro de uma rotina chamada por uma tecla definida pelo comando SET KEY. Ambos cancelam os GET's pendentes no programa que chamou a rotina. Para limpar a tela utilize os comandos @...CLEAR ou CLEAR SCREEN (este último somente disponível a partir da versão Summer 87
O comando SET KEY tem precedência sobre o comando SET FUNCTION, caso uma mesma tecla seja designada por ambos.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
SET KEY -1 TO Opções
SET KEY -2 TO Inclui
SET KEY -3 TO Altera
SET KEY -4 TO Exclui
SET KEY -5 TO Consulta
SET KEY -6 TO Imprime
SET KEY -7 TO Fim
USE Produtos
DO WHILE .T.
   @ 10,18 TO 14,40 DOUBLE
   @ 12,21 SAY "Tecle F2 para Ver as Opções Disponiveis"
   WAIT
ENDDO
USE
CLEAR
RETURN

PROCEDURE Opcoes
PARAMETERS p1,p2,p3  && prog, lin e var
```

```
SAVE SCREEN
@ 01,00 CLEAR
@ 10,25 TO 17,55 DOUBLE
@ 11,27 SAY "F3 - Inclusão de Dados"
@ 12,27 SAY "F4 - Alteração de Dados"
@ 13,27 SAY "F5   Exclusão de Dados"
@ 14,27 SAY "F6   Consultas na Tela"
@ 15,27 SAY "F7 - Emissão de Relatórios"
@ 16,27 SAY "F8   Fim das Operações"
WAIT
RESTORE SCREEN
RETURN

     * Executando uma rotina a partir da tecla F9:

     CLEAR
     SET PROCEDURE TO TESTPROC
     USE Mala INDEX Indcod
     SET KEY -8 TO Totregs   && F9 mostra o total de
     DO WHILE .T.            && registros do arquivo
        APPEND BLANK
        @ 05,34 SAY "Mala-Direta"
        @ 07,27 SAY "Entrada/Alteração de Dados"
        @ 08,27 SAY "-------------------------"
        @ 11,15 SAY "Nome....:" GET nome     PICTURE "@!"
        @ 13,15 SAY "Endereço:" GET endereco PICTURE "@!"
        @ 15,15 SAY "Cidade..:" GET cidade   PICTURE "@!"
        @ 17,15 SAY "Estado..:" GET estado   PICTURE "!!"
        @ 17,30 SAY "CEP:"      GET cep      PICTURE "@9"
        @ 19,15 SAY "Tecle F9 para ver o Total de Registros"
        READ
   IF EMPTY(nome)
      DELETE
      PACK
      EXIT
   ENDIF
ENDDO
CLEAR
USE
RETURN

* Arquivo de Procedimentos TESTPROC.prg, onde está a
* rotina Totregs, que mostra o número total de
* registros do arquivo:
*
PROCEDURE Totregs
PARAMETERS p1,p2,p3 && Os três parâmetros obrigatórios
@ 21,20 SAY "Número Total de Registros do Arquivo:"
@ 21,57 SAY LASTREC() PICTURE "@E 999,999"
@ 23,22 SAY "Pressione uma Tecla para Continuar !"
WAIT
@ 21,00 CLEAR
RETURN
```

Veja Também:

KEYBOARD, SET FUNCTION, LASTKEY( ), INKEY( )

Comando SET MARGIN:

Sintaxe:

SET MARGIN TO <exp.numérica>

Propósito:

Ajusta, para todas as saídas impressas, a margem esquerda da impressora para um número especificado de espaços.

Utilização:

A margem esquerda padrão na impressora é zero (0). Para se definir outro valor basta especificá-lo através do comando SET MARGIN. A partir daí, todas as saídas impressas passarão a respeitar esta margem.
A <expressão numérica> define a coluna da margem esquerda onde deverão ser iniciadas todas as impressões.
Utiliza-se este comando principalmente para definir a coluna inicial de impressão para os comandos REPORT FORM, LABEL FORM e LIST TO PRINT.

Biblioteca:

CLIPPER.LIB

Exemplo:

* Para definir a margem esquerda de impressão com 20 espaços <caracteres> utiliza-se:

```
SET MARGIN TO 20
USE Mala INDEX Alfa
LIST OFF nome, endereço, telefone TO PRINT
```

Veja Também:

@...SAY...GET, SET DEVICE, SET PRINT

Comando SET MESSAGE TO:

Sintaxe:

SET MESSAGE TO [<exp.numérica> [CENTER]]

Propósito:

Permite especificar a linha da tela onde aparecerão as mensagens definidas nos comandos @...PROMPT...MESSAGE.

Utilização:

O comando SET MESSAGE deve ser utilizado sempre em conjunto com os comandos @...PROMPT...MESSAGE e MENU TO, na criação de menus de opções através de barras luminosas. Ele permite especificar a linha onde aparecerá a mensagem definida para cada opção.
A <expressão numérica> especifica o número da linha (de 1 a 24) onde as mensagens deverão ser apresentadas. Se a expressão for igual a 0 (zero) ou não for especificada, as mensagens não serão apresentadas.
As mensagens sempre se iniciarão na coluna zero da tela, a menos que a opção CENTER, disponível apenas a partir da versão Summer 87, seja especificada. Neste caso as mensagens serão automaticamente centralizadas na linha especificada.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
SET DATE BRITISH
CLEAR ALL
SET CONSOLE OFF
CLEAR
SET DEFAULT TO B:
frame1=CHR(218)+CHR(196)+CHR(191)+CHR(179)+CHR(217)+;
       CHR(196)+CHR(192)+CHR(179)
frame2=CHR(201)+CHR(205)+CHR(187)+CHR(186)+CHR(188)+;
       CHR(205)+CHR(200)+CHR(186)
frame3=frame1+CHR(176)
frame4=frame2+CHR(176)
*
@ 2,0, 4,79 BOX frame1
@ 6,0,23,79 BOX frame4
@ 11,23 SAY " SISTEMA DE EMISSAO DE MALA-DIRETA "
@ 13,23 SAY "        Clipper - Vidal  "
```

```
DO WHILE .T.
   @ 03,01 SAY SPACE(78)
   @ 05,01 SAY SPACE(78)
   mp=1
     SET MESSAGE TO 5 CENTER
   *
   * Define a linha 5 para a apresentação das mensagens centralizadas
   *
   @ 03,02 PROMPT "Inclusões"  Message "Inclusoes de Novos Nomes"
   @ 03,13 PROMPT "Alterações" Message "Alterações de Nomes Cadastrados"
   @ 03,25 PROMPT "Exclusoes"  Message "Exclusões de Nomes Cadastrados"
   @ 03,36 PROMPT "Consultas"  Message "Consultas na Tela ans Nomes"
   @ 03,47 PROMPT "Relatórios" Message "Menu de Relatorios do Sistema"
   @ 03,59 PROMPT "Utilitários" Message "Reorganização dos Arquivos"
   @ 03,73 PROMPT " Fim "      Message "Fim das Operações"
   MENU TO mp
   @ 24,00 CLEAR
   DO CASE
      CASE mp=1
        DO Inclusao
      CASE mp=2
        DO Alteracao
      CASE mp=3
        DO Exclusao
      CASE mp=4
         DO Consulta
      CASE mp=5
         DO Relatorio
      CASE mp=6
         DO Utilit
      CASE mp=7
        confirme=SPACE(1)
        @ 24,60 SAY "Confirme (S/N) ?" GET confirme;
                                  VALID confirme$"SsNn"
        READ
        @ 24,00 CLEAR
        IF confirme()"S"
           LOOP
        ENDIF
        CLEAR
        @ 8,0,14,79 BOX frame4
        @ 14,72 SAY " Vidal "
        SET COLOR TO W+*
        @ 10,30 SAY " Fim das Operações ! "
        SET COLOR TO W+
        @ 12,30 SAY "      Obrigado !     "
        SET COLOR TO W
        EXIT
   ENDCASE
ENDDO
CLEAR
RETURN
```

Veja Também:

@...PROMPT, MENU...TO, SET WRAP

Comando SET ORDER TO:

Sintaxe:

SET ORDER TO [<**exp.numérica**>]

Propósito:

Permite definir um entre os arquivos de índice abertos, como sendo o índice-mestre ou índice de controle (o índice que controlará a ordem de acesso ao arquivo de dados). Também permite remover o controle do acesso ao arquivo de dados de todos os índices abertos, sem contudo fechá-los e reabri-los.

Utilização:

Ao se abrir um arquivo através do comando USE <arquivo> INDEX <lista de índices> ou se definir os índices a serem utilizados através do comando SET INDEX TO <lista de índices>, o primeiro índice da lista será o índice-mestre. Este índice será usado para determinar a ordem de acesso aos registros do arquivo de dados. O comando SET ORDER TO permite redefinir, entre os índices abertos, para um determinado arquivo de dados, qual será o índice-mestre, sem a necessidade de fechar e reabrir todos os arquivos de índice novamente.

A <expressão numérica> especifica o índice que será usado para controlar o acesso aos dados do arquivo, na ordem em que eles foram colocados na lista. Se a expressão valer 2, o segundo índice da lista será o novo índice mestre; se a expressão valer 4, o quarto; e assim por diante. O resultado da expressão numérica deve ser, portanto, um número inteiro entre 0 e 7 (ou entre 0 e 15 para a versão Summer 87), dependendo do número de índices abertos através da lista de índices.

Se o número for zero (0), o arquivo será acessado em sua ordem natural, ou seja, como se não estivesse indexado, de acordo com o número dos registros; contudo, todos os índices abertos continuarão a ser atualizados normalmente.

O comando SET ORDER economiza o tempo de fechamento e reabertura dos arquivos quando é necessário redefinir o índice de controle de acesso aos registros do arquivo de dados.

Após utilizar o comando SET ORDER TO o apontador de registros permanecerá apontando para o mesmo registro, apesar de se estar utilizando um novo índice para controle, permitindo que a ordem de acesso aos registros seja trocada eficientemente.

Para obter-se a chave do índice que está atualmente no controle do arquivo, utiliza-se em conjunto as funções INDEXKEY( ) e INDEXORD( ), disponíveis apenas a partir da versão Summer 87.

Biblioteca:

CLIPPER.LIB

Exemplos:

* No exemplo abaixo são abertos o arquivo Estoque e os índices Indcod (ordem de código), Indnome (ordem de nome) e Indata (ordem de data); sendo que o índice-mestre, de controle de acesso aos dados é o Indcod, ou seja, os registros do arquivo serão acessados pela ordem de código.

```
CLEAR                    1        2        3
USE Estoque INDEX Indcod, Indnome, Indata
```

Para acessar o arquivo Estoque por nome, sem, contudo fechar e reabrir os arquivos, basta especificar a nova ordem de acesso desejada:

```
? INDEXKEY(INDEXORD( ))   && resulta CODIGO

SET ORDER TO 2
```

O índice por nome, o Indnome, é o segundo da lista de índices abertos para o arquivo Estoque. Portanto para acessar o arquivo através dele, basta executar o comando SET ORDER TO 2. Todos os índices da lista permanecerão abertos e sendo atualizados normalmente.

```
? INDEXKEY(INDEXORD( ))   && resulta NOME

SET ORDER TO 3
```

O arquivo do Estoque passa a ser acessado na ordem de data, do índice Indata.

```
? INDEXKEY(INDEXORD( ))   && resulta DATA

SET ORDER TO 0

? INDEXKEY(INDEXORD( ))   && resulta nulo
```

O arquivo de Estoque passa a ser acessado na sua ordem natural de criação (ordem de número de registro), contudo todos os índices da lista permanecem abertos e sendo atualizados.

Veja Também:

INDEX, REINDEX, SET INDEX, USE, SEEK, FIND, INDEXEXT( ), INDEXKEY( ), INDEXORD( )

Comando SET PATH TO:

Sintaxe

SET PATH TO [<rota de acesso aos diretórios>]

Propósito:

Define a rota de diretórios que o Clipper irá seguir para pesquisar e achar arquivos que não estiverem no diretório corrente (padrão).

Utilização:

Utiliza-se comando SET PATH para definir uma "rota" de diretórios a ser seguida pelo Clipper para encontrar arquivos que não estejam no diretório padrão ("default").
A <rota de acesso aos diretórios> especifica a rota de acesso a ser seguida para pesquisar os arquivos no disco e deve ser especificada de acordo com as regras padronizadas adotadas pelo DOS.
SET PATH TO, sem os parâmetros, retorna a rota de pesquisa para o diretório atual.
SET PATH permite o acesso a arquivos em um sistema de diretórios organizados em árvores. Ele especifica a seqüência de diretórios que o Clipper deve pesquisar até localizar o que contêm o arquivo especificado.
O diretório corrente (padrão) é sempre o diretório raíz, ou o definido antes da carga do programa através do DOS. Apenas o diretório corrente é pesquisado para localizar arquivos, a menos que uma rota alternativa tenha sido especificada pelo comando SET PATH. O PATH estabelecido pelo DOS não é lido quando o arquivo é acessado através de um sistema desenvolvido em Clipper.
Uma rota (ou path) é uma lista de diretórios separados por barras invertidas"\", na ordem do diretório de maior nível para o de menor nível, onde os arquivos pesquisados devem estar localizados. Uma lista de rotas é, por sua vez, uma série de rotas separadas por ponto-e-vírgulas.

O comando SET PATH aplica-se apenas a comandos que operam sobre arquivos existentes. Se um comando vai criar um arquivo e este arquivo deve ser localizado em um diretório diferente do atual (padrão), deve-se especificar explicitamente, juntamente com o nome do arquivo o drive e o diretório de desejados, ou utilizar-se o comando SET DEFAULT TO para alterar o diretório padrão.
Uma vez que o ponto-e-vírgula faz parte da sintaxe das rotas, o comando SET PATH não o suporta para dar sua continuação na próxima linha do programa. O SET PATH deve obrigatóriamente estar contido em uma única linha do programa.
O comando SET DEFAULT TO pode ser utilizado para mudar o drive e diretório padrão no qual o Clipper inicialmente pesquisará os arquivos.
A função FILE( ), na verificação da existência de um arquivo no disco, segue a rota definida pelo comando SET PATH TO.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
SET PATH TO A:\MALA\DADOS; C:\CLIPPER\ESTOQUE
USE Estoque
```

* O Clipper irá pesquisar o arquivo Estoque.dbf no diretório padrão, \CLIPPER, não o encontrando irá pesquisar no drive A:\MALA\DADOS e finalmente irá pesquisá-lo no \CLIPPER\ESTOQUE, onde ele realmente está, abrindo-o para ser utilizado.

Veja Também:

SET DEFAULT, DIR, FILE( )

Comando SET PRINT:

Sintaxe:

SET PRINT on / OFF

Propósito:

Permite direcionar todas as saídas de dados, que não sejam realizadas pelo comando @...SAY, simultaneamente para a tela e para a impressora. Liga e desliga a saída de dados não formatados para a impressora.

Utilização:

SET PRINT está normalmente desligado (OFF). Neste caso, todas as saídas que não sejam formatadas pelo comando @...SAY são apresentadas apenas na tela.
Se for definido SET PRINT ON, todas as saídas que não forem realizadas pelo comando @...SAY irão tanto para a tela como para a impressora. Se o comando SET CONSOLE estiver desligado (OFF), as saídas serão apenas impressas e não serão ecoadas na tela.
Para retornar ao padrão, desligando a impressora, utilize SET PRINT OFF.
Para que as saídas formatadas pelo @...SAY também sejam dirigidas para a impressora deve ser utilizado o comando SET DEVICE TO PRINT/ SCREEN.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
USE Mala INDEX Indnome
SET PRINT ON
? CHR(15)
* condensa os caracteres da impressora
SET CONSOLE OFF
* desliga a apresentação na tela
SET COLOR TO W+*
CLEAR
@ 10,27 SAY "Aguarde... Imprimindo os Dados"
SET COLOR TO
@ 12,27 SAY "Tecle (Esc) para interromper !"
DO WHILE .NOT. EOF() .AND. INKEY()<>27
   DO WHILE PROW()<=60 .AND. .NOT. EOF()
      ? nome, endereco, telefone, cidade, estado, cep
      SKIP
   ENDDO
   EJECT
ENDDO
SET PRINT OFF
SET CONSOLE ON
USE
RETURN
```

Veja Também:

EJECT, SET CONSOLE, SET DEVICE, SET PRINTER

Comando SET PRINTER:

Sintaxe:

SET PRINTER TO [<dispositivo>/<arquivo>]

Propósito:

Determina o destino de todas as saídas para a impressora.

Utilização:

O comando SET PRINTER é usado para redirecionar as saídas da impressora para outras saídas que não a padrão. A saída padrão é a porta paralela LPT1:, onde normalmente se conecta uma impressora paralela.

Há três possibilidades de destinação das saídas para impressora:

1 — Envia a saída da impressora ao controle da rede local na qual está conectado o equipamento ou à impressora local. O dispositivo padrão é PRN (impressora conectada à saída paralela LPT1). Nomes possíveis de dispositivos podem ser: LPT1, LPT2, LPT3 (portas paralelas), COM1 e COM2 (portas seriais).

SET PRINTER TO <dispositivo>

2 — Limpa o "spool" de impressão e retorna ao padrão de saída de impressão, ou seja, para o dispositivo padrão PRN.

SET PRINTER TO

3 — Envia todas as saídas de impressão, inclusive as dos comandos @...SAY, para um arquivo, cujo nome deve ser especificado. Se uma extensão não for especificada o Clipper assumirá a extensão padrão (.prn).

SET PRINTER TO <nome do arquivo>

Se o comando SET PRINTER TO direcionar as saídas para um dispositivo não existente será criado um arquivo com o nome do dispositivo especificado. O comando SET PRINTER permite, entre outras coisas:

- Direcionar as saídas de impressão para qualquer porta do equipamento, permitindo o uso de várias impressoras a ele conectadas:
- Direcionar as saídas de impressão para um arquivo e posteriormente transferí-la a um computador remoto via teleprocessamento;

Biblioteca:

CLIPPER.LIB

Exemplos:

Para direcionar as saídas para a porta serial 1:

```
SET PRINTER TO COM1
```

Para direcionar as saídas para o arquivo IMP.txt:

```
SET PRINTER TO IMP.txt
```

```
SET PRINTER TO   && limpa o spool de impressão
```

Veja Também:

SET PRINT, SET DEVICE, @...SAY

Comando SET PROCEDURE:

Sintaxe:

SET PROCEDURE TO [<arquivo de rotinas>]

Propósito:

Abre um arquivo de procedimentos ou rotinas ("procedure file"), contendo rotinas e funções-de-usuário comuns aos módulos de um sistema. Todas as rotinas e funções-de-usuário contidas no arquivo especificado serão compiladas no mesmo módulo-objeto, tornando-se, assim, disponíveis para todos os outros módulos do sistema.

Utilização:

O comando SET PROCEDURE abre um arquivo contendo vários procedimentos (rotinas e funções-de-usuário) que serão compilados e, em geral, tornados disponíveis aos demais módulos do sistema.
O <arquivo de rotinas> é o nome do arquivo de procedimentos a ser utilizado. Se nenhuma extensão for especificada será assumida (.prg).
Um arquivo de procedimentos é um conjunto de rotinas, cada uma iniciada pelo comando PROCEDURE e finalizada pelo comando RETURN, e um conjunto de funções-de-usuário, cada função iniciada pelo comando FUNCTION e finalizada pelo comando RETURN <expressão>.
Cada rotina ou função de um arquivo de procedimentos será compilada e poderá ser utilizada por todos os programas ou módulos de um sistema, evitando assim duplicidade de compilação de rotinas comuns a vários módulos.
Os comandos CLOSE PROCEDURE e SET PROCEDURE TO, sem a especificação do arquivo são ignorados pelo Clipper quando encontrados.
Para se executar uma rotina contida em um arquivo de procedimentos, basta, em qualquer módulo ou programa do sistema, utilizar o comando DO <nome da rotina>.
No Clipper podem ser definidos quantos arquivos de procedimentos forem necessários. Na verdade, o comando SET PROCEDURE TO é desnecessário. Basta compilar os arquivos de procedimentos gerando os respectivos módulos-objeto, e link-editá-los com os demais módulos da aplicação que as rotinas e funções neles contidas poderão ser utilizadas. Além disso, não há limitação para o número de rotinas e funções-de-usuário que um arquivo de procedimentos poderá conter.

Exemplo:

* O comando abaixo abre o arquivo de procedimentos Arqpro.prg, que contém diversas rotinas e funções-de-usuário.

```
SET PROCEDURE TO ARQPRO

* Arquivo ARQPRO.prg

PROCEDURE Um
        .
        .
        .
   <comandos>
        .
        .
        .
RETURN
```

```
*
PROCEDURE Dois
        .
        .
        .
   <comandos>
        .
        .
        .
RETURN
*
FUNCTION XXX
PARAMETERS
        .
        .
   <comandos>
        .
        .
RETURN (valor)
*
PROCEDURE Tres
PARAMETERS
        .
        .
   <comandos>
        .
        .
RETURN
```

Qualquer rotina do arquivo de procedimentos acima poderá ser executada, basta utilizar o comando DO <nome da rotina>.

Veja Também:

PROCEDURE, FUNCTION, DO, PARAMETERS, RETURN, as "procedures" do Sistema de Mala-Direta (Cap. 10).

COMANDO SET RELATION:

Sintaxe:

```
SET RELATION TO [<exp.chave 1>/RECNO( )/<exp.numérica1>
                INTO <alias 1>]
                [,TO <exp.chave 2>/RECNO( )/<exp.numérica2>
                INTO <alias 2>]
                [,...]]
                 ADDITIVE]
```

Propósito:

Relaciona um arquivo de dados em uso na área selecionada a outros arquivos, de acordo com expressões chave que são comuns a cada um deles, ou simplesmente de acordo com o número do registro. Permite, portanto, a ligação direta de um arquivo principal, com outros a ele relacionados.

Utilização:

O comando SET RELATION relaciona um arquivo de dados que está em uso na área de trabalho selecionada a outros arquivos de dados abertos em outras áreas de trabalho. Esta ligação é realizada através de expressões chave que devem ser comuns ao arquivo principal e aos a ele relacionados ou simplesmente através do número do registro de cada arquivo. Os arquivos a serem ligados (INTO) são identificados através do nome de seu "ALIAS".
Através deste comando é possível estabelecer um relacionamento direto entre um arquivo principal e outros a ele relacionados, tornando-se possível o acesso simultâneo e automático aos registros do arquivo principal e aos correspondentes registros dos arquivos relacionados.
O relacionamento definido pelo comando SET RELATION faz com que o apontador de registros se mova nos arquivos relacionados de acordo com o movimento no arquivo principal através de uma chave de ligação, comum a ambos, através do número do registro, ou ainda através de uma expressão numérica. Se não houver um casamento entre as chaves do arquivo principal e do arquivo relacionado, o apontador de registros se posicionará no fim do arquivo relacionado, resultando um registro vazio. A função EOF( ) se tornará verdadeira (.T.) e a função FOUND( ) se tornará falsa (.F.).
O Clipper permite que sejam relacionados até oito arquivos por área de trabalho, ou seja, um arquivo principal aberto em uma determinada área de trabalho pode ligar-se com até oito outros arquivos relacionados abertos em outras áreas de trabalho. Estes, por sua vez podem estar relacionados a outros arquivos. Relacionamentos cíclicos, contudo não são suportados. Não se pode relacionar, direta ou indiretamente um arquivo de dados a ele mesmo.
O <alias> identifica o nome do "alias" dos arquivos a serem relacionados ao arquivo principal cuja área de trabalho está selecionada.
A <expressão chave> é a expressão a ser usada para realizar uma pesquisa (como um SEEK) no arquivo relacionado, cada vez que o apontador de registros se deslocar no arquivo principal. Para isso o arquivo relacionado deve estar obrigatoriamente indexado de acordo com esta expressão.
Portanto, para estabelecer-se o relacionamento através de uma <expressão chave>, a chave deve estar contida no arquivo principal e no arquivo relacionado. O arquivo relacionado deve estar indexado através desta chave e o índice correspondente deve estar como índice mestre, ou seja, o que deter-

mina a ordem de acesso aos registros. Sempre que o apontador de registros do arquivo principal for reposicionado em outro registro, os arquivos a ele relacionados serão posicionados no primeiro registro que possua a expressão chave de ligação coincicente com a do arquivo principal.
O <RECNO( )> (número do registro) liga o arquivo principal ao arquivo relacionado utilizando o número do registro do arquivo principal para executar um GOTO para o mesmo número de registro no arquivo relacionado, cada vez que o apontador de registros se deslocar no arquivo principal. Neste tipo de relacionamento o arquivo relacionado não deve estar indexado.
A <expressão numérica> é uma expressão utilizada para executar também um GOTO no arquivo relacionado, usando para isso o resultado da expressão cada vez que o apontador de registros for deslocado no arquivo principal. Para esse tipo de relacionamento ser executado corretamente o arquivo relacionado não deve estar indexado.
Uma vez relacionados os arquivos, estando selecionada a área de trabalho do arquivo principal, para identificar os campos de cada um dos arquivos relacionados, utiliza-se o seu respectivo alias, através da seguinte sintaxe:

<alias> → <nome do campo>

O relacionamento entre os arquivos é mantido até que seja desconectado através do comando SET RELATION TO, sem a especificação de parâmetros de relacionamento. Sempre que o relacionamento não for mais necessário deve-se desconectá-lo, pois o processamento com o relacionamento ativo é sempre mais lento.
A cláusula ADDITIVE (disponível apenas a partir da versão Summer 87) permite que novos relacionamentos sejam adicionados aos relacionamentos já definidos na área de trabalho atual, não sendo necessário redefini-los novamente. Por exemplo:

```
SELECT 1
USE Produtos INDEX Indprod ALIAS Prodt
SELECT 2
USE Precos   INDEX Indpre  ALIAS Preco
SET RELATION TO codpro INTO Prodt
     .
     .
  (mais tarde...)
     .
SELECT 3
USE Empresas INDEX Indemp ALIAS Empr
SELECT 1
SET RELATION ADDITIVE TO codemp INTO Empr
* o relacionamento inicial é mantido é estabelecido um
* novo
```

O comando SET RELATION ignora o comando SET SOFTSEEK (disponível apenas a partir da versão Summer 87). Mesmo que o SOFTSEEK esteja ligado (ON), para o SET RELATION ele sempre estará desligado (OFF). Isto significa que se o registro correspondente não for encontrado no arquivo relacionado, o apontador de registros se posicionará no fim do arquivo relacionado (EOF( ) se tornará .T.).

Biblioteca:

CLIPPER.LIB

Exemplos:

* O arquivo Notas.dbf possui os campos número do aluno, código da disciplina, número do professor, nota e número de faltas, formando a seguinte estrutura:

Campo	Nome	Tipo	Tamanho	Decimais
1	nroalu	C	5	
2	codisp	C	7	
3	nroprof	C	3	
4	nota	N	4	1
5	faltas	N	2	

* O arquivo Discipli.dbf possui os campos código da disciplina, nome da disciplina e total de aulas previstas, formando a seguinte estrutura:

Campo	Nome	Tipo	Tamanho	Decimais
1	codisp	C	7	
2	disp	C	30	
3	naulas	N	3	

* O arquivo Profs.dbf possui os campos número do professor e nome do professor, formando a seguinte estrutura:

Campo	Nome	Tipo	Tamanho	Decimais
1	nroprof	C	3	
2	prof	C	30	

* Finalmente, o arquivo Alunos.dbf possui os campos número do aluno, nome do aluno, endereço e data da matrícula, formando a seguinte estrutura:

Campo	Nome	Tipo	Tamanho	Decimais
1	nroalu	C	5	
2	aluno	C	30	
3	endere	C	40	
4	datmat	D	8	

Para se obter uma listagem que forneça diretmente o número do aluno (do arquivo Notas), o nome do aluno (do arquivo Alunos), o código da disciplina (do arquivo Notas), o nome da disciplina (do arquivo Discipli), o número do professor (do arquivo Notas), o nome do professor (do arquivo Profs), a nota do aluno (do arquivo Notas) e a porcentagem de aulas assistidas, pode ser utilizado o relacionamento exemplificado a seguir:

```
CLEAR
SELECT 1
USE Notas ALIAS Nota
INDEX ON nota TO Indnota
SELECT 2
USE Discipli ALIAS Disp
INDEX ON codisp TO Indisp
SELECT 3
USE Profs Alias Prof
INDEX ON nroprof TO Indprof
SELECT 4
USE Alunos ALIAS Aluno
INDEX ON nroalu TO Indalu
SELECT Nota
SET RELATION TO codisp  INTO Disp,;
               TO nroprof INTO Prof,;
               TO nroalu  INTO Aluno
LIST nota->nroalu, aluno->aluno, nota->codisp,;
     disp->disp, nota->nroprof, prof->prof,;
     nota->nota,;
     ((disp->naulas - nota->faltas)/disp->naulas)/100;
     TO PRINT
SET RELATION TO
CLOSE DATA
RETURN
```

* O relacionamento entre os arquivos é feito de acordo com as seguintes expressões-chave:

Arquivo Principal	Arquivos Relacionados
notas→nroalu	alunos→nroalu
notas→nroprof	profs→nroprof
notas→codisp	disp→codisp

Veja Também:

INDEX, USE, SELECT, SET INDEX, SET ORDER, SEEK, UPDATE, RECNO( )

Comando SET SCOREBOARD:

Sintaxe:

SET SCOREBOARD ON/off

Propósito:

Liga ou desliga a apresentação de mensagens na primeira linha da tela durante a execução dos programas.

Utilização:

O comando SET SCOREBOARD está normalmente ligado (ON), ou seja, são apresentadas mensagens do Clipper na primeira linha da tela (linha 0) durante a execução dos programas. Estas mensagens estão relacionadas com os comandos READ e MEMOEDIT( ), indicando por exemplo o modo de inserção ligado (<Ins> será apresentado) ou indicando uma entrada de dados inválida (quando se utilizam a cláusula VALID ou a cláusula RANGE).
Para evitar que mensagens do Clipper sejam apresentadas ao usuário durante a execução dos programas compilados, basta desligá-las com SET SCOREBOARD OFF.

Biblioteca:

CLIPPER.LIB

Veja Também:

@...SAY...GET, READ, MEMOEDIT( )

Comando SET SOFTSEEK:

Sintaxe:

SET SOFTSEEK on / OFF

Propósito:

Habilita ou desabilita a pesquisa relativa através dos comandos SEEK e FIND. A pesquisa relativa acha o registro que possui uma chave de valor imediatamente acima, quando a chave exata não é encontrada.
O comando SET SOFTSEEK está disponível apenas a partir da versão Summer 87.

Utilização:

Se o comando SET SOFTSEEK está ligado (ON) e um registro pesquisado através do comando SEEK não é encontrado, o apontador de registros se posicionará no próximo registro do índice que possua uma chave com o valor imediatamente acima da chave pesquisada. Neste caso, tanto a função FOUND( ) como a função EOF( ) tornam-se falsas (.F.). Se não existir nenhum registro com uma chave maior que a pesquisada, aí então o apontador de registros se posicionará no final do arquivo (LASTREC( )+1), a função EOF( ) se tornará verdadeira (.T.) e a função FOUND( ) se tornará falsa (.F.).
Quando o SET SOFTSEEK está ligado (ON), o comando SET EXACT perde seu efeito. Por outro lado, o comando SET RELATION TO ignora o SET SOFTSEEK ON, operando sempre como se o SET SOFTSEEK estivesse desligado (OFF).
Se o SET SOFTSEEK estiver desligado, como é o padrão de operação do Clipper, e uma pesquisa através do comando SEEK não encontrar um registro com a chave pesquisada o apontador de registros se posicionará no final do arquivo (LASTREC( )+1), a função EOF( ) se tornará verdadeira (.T.) e a função FOUND( ) se tornará falsa (.F.).

Biblioteca:

CLIPPER.LIB

Exemplo:

```
USE Mala INDEX Indnome
SEEK "MARIA ALICE"
? FOUND(), EOF()
* O nome não foi achado: FOUND() = falso       (.F.)
*                        EOF()   = verdadeiro (.T.)
SET SOFTSEEK ON
SEEK "MARIA ALICE"
? FOUND(), EOF()
* O nome não foi achado: FOUND() = falso       (.F.)
*                        EOF()   = falso       (.F.)
DISPLAY nome
*
```

```
 retorna 'MARIA APARECIDA', que é a próxima chave no
* arquivo de índice, com valor mais alto que a
* pesquisada.
```

Veja Também:

INDEX, SEEK, SET EXACT, SET RELATION, SET INDEX, SET ORDER, FOUND( ), EOF( )

Comando SET TYPEAHEAD:

Sintaxe:

SET TYPEAHEAD TO <expressão numérica>

Propósito:

Estabelece o tamanho do "buffer" do teclado.
Disponível apenas a partir da versão Summer 87.

Utilização:

O comando SET TYPEAHEAD permite definir o tamanho do "buffer" do teclado.
A <expressão numérica> determina o número de caracteres que o "buffer" do teclado poderá aceitar enquanto aguarda o envio das digitações ao processador. Este número pode variar de um mínimo de zero (0) a um máximo de 32.768 caracteres.

Cuidado: se o buffer do teclado for definido como zero (0), várias teclas como <Alt-C> para interrupção do programa ou <Alt-D> para chamar o "debugador" do Clipper, poderão ser desativadas em determinadas operações.

Biblioteca:

CLIPPER.LIB

Veja Também:

ACCEPT, INPUT, KEYBOARD, READ, SET KEY, LASTKEY( )

Comando SET UNIQUE:

Sintaxe:

SET UNIQUE on/OFF

Propósito:

Determina se todos os registros com o mesmo valor para uma chave serão ou não incluídos em um arquivo de índice (.ntx), permitindo ou não a existência de chaves duplicadas no índice.

Utilização:

O comando SET UNIQUE está normalmente desligado (OFF), ou seja, todos os registros de um arquivo de dados, mesmo aqueles que contenham o mesmo valor para uma chave (chaves duplicadas), serão incluídos em um arquivo de índice criado a partir desta chave sobre o arquivo de dados.
Se for criado um novo arquivo de índice com SET UNIQUE ON (ligado), e vários registros possuirem o mesmo valor para a chave de ordenação, somente o primeiro registro que o Clipper encontrar com este valor será incluído no novo arquivo de índice. Portanto, haverá apenas um único registro para cada valor da chave no arquivo de índice.
Ao se incluir novos registros ou se alterar registros de um arquivo de dados cujo índice foi criado com chave única e o SET UNIQUE estiver ligado (ON), será mantida esta característica. Isto significa que se for incluído um registro cujo valor para a chave do índice já exista, este registro será incluído no arquivo de dados mas não será incluído no arquivo de índice. Da mesma forma, se for alterado um registro e o valor de sua chave passe a ser o mesmo de outro já existente, este registro será removido do arquivo de índice, embora permaneça no arquivo de dados.
Para retornar um arquivo de índice à condição de chave não-única, o índice dever ser reconstruído através dos comandos INDEX ON ou REINDEX e o SET UNIQUE deve estar OFF.

Cuidado: No Clipper, a definição UNIQUE é uma definição global e não relacionada especificamente a um arquivo de índice. Por esse motivo a operação de REINDEXar com o SET UNIQUE ON elimina as chaves não-únicas de todos os arquivos de índice, não importanto se estes anteriormente tiverem sido criados com o SET UNIQUE OFF. Da mesma forma, se um índice criado com o SET UNIQUE ON for atualizado com o SET UNIQUE OFF, a característica de registro único não será mantida.

Veja Também:

INDEX, REINDEX, SEEK, FIND, SET INDEX, USE

Comando SET WRAP:

Sintaxe:

SET WRAP on / OFF

Propósito:

Habilita ou desabilita as teclas <Seta Direita> e <Seta Esquerda> de moverem diretamente a barra luminosa de um menu de opções criado pelos comandos @...PROMPT e MENU TO para a primeira ou última opção do menu.
Este comando só esta disponível a partir da versão Summer 87.

Utilização:

Quando o SET WRAP está ligado (ON) e a barra luminosa do menu está posicionada na última opção, a <Seta Direita> ou a <Seta Esquerda> a movem diretamente para a primeira opção e vice-versa.
Quando o SET WRAP está desligado (OFF), que é o padrão assumido pelo Clipper, as setas direita e esquerda movimentam a barra luminosa de opção em opção, na ordem em que foram colocadas.
Note que este comando somente deve ser utilizado quando os menus criados são "verticais" ou se utilizar a primeira letra de cada opção para a seleção.

Biblioteca:

CLIPPER.LIB

Veja Também:

@...PROMPT...MESSAGE, MENU TO, SET MESSAGE

6

Funções do Clipper

O Clipper possui várias funções pré-programadas extremamente úteis e necessárias ao desenvolvimento de aplicações sofisticadas. Na verdade as funções constituem o **grande poder do Clipper**, sendo um recurso tão importante quanto os comandos.

Todas as funções possuem a forma **F (x, y)**, onde F é o nome que identifica a função e x e y, entre parênteses, são seus argumentos.

Os argumentos das funções são valores sobre os quais a função executará alguma operação e retornará um resultado. Os argumentos podem ser constituídos por uma cadeia de caracteres, uma variável, um número, uma expressão, uma data, uma condição lógica, um vetor etc. Algumas funções possuem um único argumento, outras vários, e existem algumas funções que não possuem argumentos, apenas retornam valores.

Em todo livro usaremos um tipo de notação já tradicionalmente padronizada para a apresentação e explicação das funções do Clipper. As convenções e símbolos desta notação são apresentados a seguir:

- As palavras chave que identificam as funções do Clipper estarão sempre escritas em letras maiúsculas, acompanhadas de um abre e fecha parênteses; por exemplo: SQRT( ) – função raiz quadrada;
- A parte ou argumentos a serem supridos pelo usuário aparecerá entre os sinais < >e estará em letras minúsculas; por exemplo: SQRT(<expressão numérica>);
- Os argumentos opcionais de algumas funções serão apresentados entre colchetes [ ], significando que o usuário poderá ou não incluir este argumento quando da utilização da função, pois ele é opcional; por exemplo: STR((exp.caract>[,<exp.num.1>[<exp.num.2>]]);
- As opções alternativas de algumas funções serão indicadas através da barra /; por exemplo: LASTREC( )/RECCOUNT( );

- Os nomes das funções ou rotinas definidas pelo usuário (funções-de-usuário), ou seja, funções não pertencentes ao Clipper mas criadas pelo usuário através do comando FUNCTION, terão a primeira letra do seu nome apresentada em maiúscula e as demais em minúsculas; por exemplo: Dpc( ).
- Os nomes de arquivos de dados, de índices ou de programas, serão escritos em letras maiúsculas. Os "Aliases" (nomes atribuídos às suas áreas de trabalho) serão apresentados com a primeira letra maiúscula e as restantes minúsculas; por exemplo: PROG1.prg, DADOS.dbf ou ALFA.ntx;

As funções, ao contrário dos comandos, que estão todos contidos na biblioteca de rotinas CLIPPER.lib, podem estar localizadas em diferentes bibliotecas ou módulos-objeto auxiliares, variando inclusive de acordo com a versão do Clipper que você estiver utilizando. Por esse motivo, aconselhamos que você verifique a versão do Clipper que possui e, de acordo com as funções que estiver utilizando, torne disponível a biblioteca ou o módulo-objeto auxiliar adequado ao link-editor, para que este encontre todas as rotinas de funções na fase de geração do módulo executável (veja a seção 8.2.2 – Link-editando com os Utilitários do Clipper para maiores detalhes).

Em cada função indicaremos a biblioteca ou módulo-objeto auxiliar onde a mesma se localiza, de acordo com a versão do Clipper, como resumido a seguir.

Versões Anteriores à Summer 87:

Nas versões anteriores à versão Summer 87 (Autumn 86 e Winter 85) o Clipper vem acompanhado por três bibliotecas e por três módulos-objeto auxiliares (cujo módulo-fonte também é fornecido), onde estão contidas suas funções a saber:

Bibliotecas:

CLIPPER.lib:	funções básicas
DBU.lib:	funções especiais (vetores e outras)
MEMO.lib:	funções para manipulação de campos memo

Módulos Objeto Auxiliares:

EXTENDDB.obj:	funções escritas em Clipper
EXTENDC.obj:	funções escritas em linguagem "C"
EXTENDA.obj:	funções escritas em Assembler

De acordo com as funções que você estiver utilzando em seus programas, a respectiva biblioteca ou módulo-objeto auxiliar deverá estar disponível (incluídos na lista de parâmetros passados) ao link-editor na fase de encadeamento e geração do módulo executável.

Os programas-fonte dos módulos-objeto auxiliares também acompanham o Clipper, possibilitando você estudá-los e até mesmo alterá-los, principalmente no caso do programa EXTENDD.prg, que contém uma série de funções-de-usuário escritas em Clipper através do comando FUNCTION.

As funções contidas nos módulos objeto EXTENDC.obj e EXTENDA.obj precisam ser declaradas como externas ao compilador do Clipper, para que este as defina como símbolos ao link-editor. Para isso deve ser utilizado o comando EXTERNAL, de preferência no módulo principal do seu sistema. Por exemplo:

```
EXTERNAL ISPRINTER( ), HEADER( ), RECSIZE( )
```

Este comando declara como externas as funções acima, contidas nos módulos-objeto auxiliares resumidos a seguir.

Nas versões Winter 85 e Autumn 86 cada módulo-objeto auxiliar contêm as seguintes funções:

Módulo Objeto EXTENDDB.obj e programa fonte EXTENDDB.prg:

```
ALLTRIM( ), AMPM( ),    DAYS( ),    DBF( ),    ELAPTIME( ),
FKLABEL( ), FKMAX( ),   ISALPHA( ),            ISLOWER( ),
ISUPPER( ), LEFT( ),    LENNUM( ),  MOD( ),    NDX( ),
OS( ),      READKEY( ),             RIGHT( ),  SECS( ),
STRZERO( ), SOUNDEX( ),             STUFF( ),  TSTRING( ),
VERSION( ),
```

Módulo Objeto EXTENDC.obj e programa fonte EXTENDC.C:

```
DISKSPACE( ), GETE( ), LUPDATE( ), RECSIZE( ), HEADER( ),
```

Módulo Objeto EXTENDA.obj e programa fonte EXTENDA.ASM:

```
ISPRINTER( )
```

Portanto, se você estiver utilizando as versões anteriores à Summer 87, verifique quais bibliotecas e módulos auxiliares devem estar disponíveis ao link-editor e não se esqueça de utilizar o comando EXTERNAL, de acordo com as funções que estiver utilizando.

Versão Summer 87:

Na versão Summer 87 e provavelmente nas posteriores, todas as funções do Clipper estão contidas em apenas duas bibliotecas:

CLIPPER.lib: funções básicas ou principais
EXTEND.lib: funções avançadas ou auxiliares

Neste caso basta você verificar se as funções que está utilizando se encontram na EXTEND.lib e também torná-la disponível ao link-editor, uma vez que a CLIPPER.lib obrigatoriamente deverá estar disponível.

A seguir são apresentadas todas as funções do Clipper classificadas de acordo com seu propósito, para facilitar a sua localização e consulta. Assim como nos comandos, as funções são apresentadas através dos seguintes tópicos:

Função:	o nome da função
Forma:	a sintaxe de utilização
Propósito:	seu propósito ou função principal
Argumentos:	os argumentos que utiliza
Utilização:	como utilizá-la
Biblioteca:	as bibliotecas onde se encontram
Exemplos:	exemplos práticos ilustrativos
Veja Também:	outras funções ou comandos relacionados a serem consultados.

No tópico "Biblioteca", é apresentada a biblioteca ou módulo-objeto auxiliar onde a função se encontra, de acordo com a versão do Clipper.

6.1. FUNÇÕES NUMÉRICAS

Função: ABS( ) – Valor Absoluto:

Forma:

ABS(<exp.numérica>)

Propósito:

Retorna um número representando o valor absoluto (módulo) do resultado da expressão numérica que é seu argumento.

Argumentos:

<exp.numérica>: qualquer expressão numérica a ser avaliada.

Utilização:

A função ABS( ) retorna sempre um número positivo ou zero para qualquer expressão numérica avaliada.
Permite, entre outras coisas, achar a magnitude de uma expressão numérica e a diferença entre dois números quando o sinal do resultado não é relevante.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
a = 20
b = 100
c = -50
? ABS(a - b) && resulta 80
? ABS(b - a) && resulta 80
? ABS( c )   && resulta 50
```

Função: EXP( ) – Exponencial Neperiano:

Forma:

EXP(<expressão numérica>)

Propósito:

Retorna o valor de **e** (número de Neper = 2,718) elevado a x, onde x é a <expressão numérica> especificada.

Argumentos:

<expressão numérica>: define o valor x para o qual o resultado numérico de e^x deve ser calculado.

Utilização:

A função EXP( ) é utilizada para se obter a exponenciação do número de Neper. A expressão numérica é o valor do expoente, x, na equação e^x. O resultado é o valor de e elevado a x.

EXP( ) é o inverso da função LOG( ).

O valor retornado obedece às definições estabelecidas pelos comandos SET DECIMALS e SET FIXED.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
? EXP(1.000)  && retorna 2.7182818285
x = 1.000
SET DECIMALS TO 3
SET FIXED ON
? EXP( x )    && retorna 2.718
```

Veja Também:

LOG( ), SET DECIMALS, SET FIXED e o operador "^" (exponenciação).

Função: INT( ) – Inteiro:

Forma:

INT(<expressão numérica>)

Propósito:

Converte um número ou o resultado de uma <expressão numérica> em um número inteiro, desprezando todos os dígitos à direita do ponto (vírgula) decimal.

Argumentos:

<expressão numérica>: é a expressão numérica cujo resultado deve ser convertido para um número inteiro.

Utilização:

A função INT( ) é usada para eliminar as casas decimais de um número. Ela não faz um arredondamento, simplesmente "trunca" as casas decimais, gerando um número inteiro.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
? INT(57.235)    && resulta 57
a = 1
b = 3
? INT( a/b )     && resulta 0 (zero)
? INT(.5)        && resulta 0 (zero)
? INT(-199.00)   && resulta -199
```

Veja Também:

ROUND( )

Função: LENNUM( ) – Comprimento de um Número:

Forma:

LENNUM(<expressão numérica>)

Propósito:

Retorna o comprimento (número de dígitos) de um número.

Argumentos:

<expressão numérica>: define o numéro a ter seu comprimento avaliado.

Utilização:

A função LENNUM( ) faz parte das extensões do Clipper, escritas em Clipper, utilizando-se o comando FUNCTION. Seu código-fonte encontra-se no arquivo EXTENDDB.prg (nas versões Winter 85 e Autumn 86), e seu respectivo código-objeto no arquivo EXTENDDB.obj.
Para utilizar a função LENNUM( ) o arquivo EXTENDDB.obj deve ser incluído na lista de módulos-objeto passada ao link-editor.
A função LENNUM( ), de forma semelhante à função LEN( ), retorna o comprimento de um valor numérico, incluindo os dígitos da parte inteira, o ponto decimal e as casas decimais, se existirem.
Pode ser utilizada, por exemplo, na de formatação de tela ou impressora, permitindo um posicionamento adequado de valores numéricos, de acordo com seu comprimento.

Biblioteca:

EXTENDDB.obj (Versões Winter 85 e Autumn 86)

Exemplos:

```
x = 1000
? LENNUM(x)     && retorna 4
y = 3.1416
? LENNUM(y)     && retorna 6
```

Veja Também:

LEN( ), STR( )

Função: LOG( ) – Logarítmo Natural:

Forma:

LOG(<expressão numérica>)

Propósito:

Retorna o logarítmo natural (base e) de um número, ou do resultado de uma <expressão numérica>.

Argumentos:

<*expressão numérica*>: é um número **maior que zero** do qual se deseja determinar o logarítmo natural.

Utilização:

O logarítmo natural tem a base **e** (número de Neper = 2,7183). A função LOG( ) retorna o expoente (x) da seguinte equação:

$$e^{x} = y$$

y é o valor da expressão numérica usada como argumento da função LOG( ); y deve sempre ser maior que zero.
A função LOG( ) retorna o inverso da função EXP( ).
Se o número definido pela <expressão numérica> for menor ou igual a zero serão retornados asteríscos, indicando estouro número ("Numeric Overflow").

Biblioteca:

CLIPPER.LIB

Exemplos:

```
? LOG(2.71828)   && retorna 1.00000
x = 10
y = 3.5
? LOG( x * y )   && retorna 3.56
```

Veja Também:

EXP( ), SET DECIMALS, SET FIXED

Função: MOD( ) – Módulo:

Forma:

MOD(<exp.numérica1>),(<exp.numérica2>)

Propósito:

Retorna um número representando o resto da divisão da <expressão numérica1> pela <expressão numérica2>.

Argumentos:

<*expressão numérica1*>: define o quociente da divisão
<*expressão numérica2*>: define o divisor.

Utilização:

A função MOD( ) como outras, faz parte das extensões do Clipper (versão Autum 86) que foram escritas em Clipper utilizando-se o comando FUNCTION. Seu código-fonte encontra-se no arquivo EXTENDDB.prg e o respectivo código-objeto no arquivo EXTENDDB.obj.
Para utilizá-la, o arquivo EXTENDDB.obj deve ser incluído na lista de módulos-objeto passados ao link-editor.
A função MOD( ) foi incluída nas extensões do Clipper para aumentar sua compatibilidade com o dBASE III Plus, que a possui. Na verdade, no Clipper utiliza-se o operador módulo "%" que também retorna o resto da divisão entre dois números. Deve-se, contudo, prestar certa atenção, pois o resultado

retornado pela função MOD( ) do dBASE difere, em alguns casos, do resultado retornado pelo operador módulo – % – do Clipper.
No Clipper, para obter o resto da divisão (módulo) utilize diretamente o operador "%".
Um exemplo de sua utilização seria para a conversão de unidades, por exemplo, para converter minutos em dias e horas em minutos.
A fórmula da função MOD( ) escrita em linguagem C é:

<expN1> – floor (<expN1>/<expN2>) * <expN2>

Onde a função floor é uma função da biblioteca matemática padrão da linguagem C, que retorna o maior inteiro menor ou igual ao seu argumento.

Biblioteca:

EXTENDDB.prg e EXTENDDB.obj (Versão Autum 86)

Exemplos:

```
? MOD(14,12)   && ou 14%12 resulta 2
? MOD(4,2)     && ou 4%2   resulta 0
? MOD(1,-3)    && ou 1%-3  resulta -2
? MOD(-4,-3)   && ou -4%-3 resulta -1
```

Veja Também:

O operador módulo "%"

Função: ROUND( ) – Arredondamento:

Forma:

ROUND(<exp.numérica1>,<exp.numérica2>)

Propósito:

Arredonda valores numéricos de acordo com um número especificado de casas decimais, retornando o valor arredondado.

Argumentos:

<*exp.numérica 1*>: define o valor a ser arredondado.
<*exp.numérica 2*>: define o número de casas decimais desejado.

Utilização:

A função ROUND( ) arredonda a <exp.numérica 1> de acordo com o número de casas decimais especificado na <exp.numérica 2>.
Se for especificado um valor menor ou igual a zero na <exp.numérica 2>, a parte inteira do número será arredondada: zero indica arredondamento sem casas decimais e um número negativo indica quantos dígitos à esquerda do ponto decimal deverão ser arredondados.
A função ROUND( ) é útil para se obter o arredondamento de valores numéricos com um determinado número de casas decimais, evitando problemas de insconsistência de dados devido à precisão dos cálculos matemáticos efetuados.
Se o comando SET FIXED estiver ON (ligado) o valor arredondado será apresentado com o número de casas decimais definido pelo comando SET DECIMALS. Se SET FIXED estiver OFF (desligado), o valor arredondado será apresentado com o número de casas decimais definidos pela <exp.numérica> ou nenhuma caso esta seja igual ou menor que zero.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
? ROUND(3.141516,3)   && retorna 3.142000
x = 47.29435
? ROUND( x,2 )        && retorna 47.29000
? ROUND( x,1 )        && retorna 47.30000
SET DECIMALS TO 3
SET FIXED ON
? ROUND( x,2 )        && retorna 47.290
? ROUND( x,1 )        && retorna 47.300
? ROUND( x,-1 )       && retorna 50.000
? ROUND( x,-2 )       && retorna 0.000
? ROUND( x,0 )        && retorna 47.000
```

Veja Também:

INT( ), SET DECIMALS, SET FIXED

Função: SQRT( ) – Raíz Quadrada:

Forma:

SQRT(<exp.numérica>)

Propósito:

Retorna a raíz quadrada de um número positivo,especificado pela expressão numérica.

Argumentos:

<*expressão numérica*>: define o valor numérico sobre o qual será calculada e retornada a raíz quadrada. Deve ser obrigatoriamente um número positivo.

Utilização:

A função SQRT( ) retorna um número, que é a raiz quadrada do número ou <expressão numérica>, utilizado como seu argumento. O número de casas decimais do resultado será igual ao que for maior entre o número de casas decimais da expressão e o número de casas decimais padrão, que pode ser definido pelos comandos SET DECIMALS e SET FIXED.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
? SQRT(4)       && retorna 2.00
? SQRT( 2*2)    && retorna 2.00
x = 4.0000
? SQRT( x )     && retorna 4.0000
```

Veja Também:

SET DECIMALS, SET FIXED e o operador da exponenciação "^".

6.2. FUNÇÕES ALFANUMÉRICAS (STRINGS)

Função: ALLTRIM( ) – Retira todos os Brancos:

Forma:

ALLTRIM(<exp.caractere>)

Propósito:

Elimina simultaneamente os espaços em branco do início e do fim de uma cadeia de caracteres. Retorna uma cadeia de caracteres sem os espaços em branco iniciais ou finais.

Argumentos:

<*exp.caractere*>: define a cadeia de caracteres a ter os espaços em branco (do início e do final) removidos.

Utilização:

A função ALLTRIM( ) elimina todos os espaços em branco, tanto do início como do fim de uma cadeia de caracteres.
A função ALLTRIM( ) é equivalente à combinação das funções LTRIM(TRIM(<exp.caractere>)).

Biblioteca:

EXTEND.LIB ou
EXTENDDB.obj e EXTENDDB.prg (nas versões anteriores à Summer 87)

Exemplos:

```
nome= "   Clipper   "
? "***"+nome+"***"  && retorna "***   Clipper   ***
? "***"+ALLTRIM(nome)+"***"
* retorna "***Clipper***", foram eliminados os brancos
* existentes no início e no fim.
```

Veja Também:

TRIM( ), LTRIM( ), RTRIM( )

Função: ASC( ) – Código ASCII:

Forma:

ASC(<exp.caractere>)

Propósito:

Retorna o valor do código ASCII (inteiro de 0 a 255) da posição mais à esquerda da expressão caractere espcificada como argumento.

Argumentos:

<*exp.caractere*>: define a cadeia de caracteres a ser retornado o número do código ASCII.

Utilização:

Utiliza-se a função ASC( ) para se obter o código ASCII do primeiro símbolo, letra ou dígito de uma expressão caractere. Pode ser utilizado onde seja necessário realizar cálculos numéricos com o valor ASCII de um caractere.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
? ASC("Clipper")        && retorna 67, que é o valor do
                        && código ASCII para a letra C.
numero = "123"
? ASC(numero)           && retorna 49 que é o valor do
                        && código ASCII para o número 1.

? ASC("CARLOS")         && retorna 67, que é o valor do
                        && código ASCII para a letra C.
```

Veja Também:

CHR( ), INKEY( )

Função: AT( ) – Pesquisa sub-cadeia (substring):

Forma:

AT(<exp.caractere 1>,<exp.caractere 2>)

Propósito:

Retorna um número que representa a posição inicial de uma expressão caractere dentro de uma segunda expressão, se a primeira estiver nela contida. Retorna zero (0) se a primeira expressão não estiver contida na segunda.

Argumentos:

<*exp.caractere 1*>: é a cadeia de caracteres a ser localizada na segunda expressão.
<*exp.caractere 2*>: é a cadeia de caracteres a ser pesquisada para localizar a primeira expressão.

Utilização:

A função AT( ) é utilizada para se determinar se uma subcadeia de caracteres está contida em outra, e a partir de qual posição.
A <expressão caractere 1>, que supõe-se estar contida na <expressão caractere 2>, é chamada de subcadeia ou substring. Se a subcadeia 1 não estiver contida na <expressão caractere 2>, o resultado da função será zero (0). Se estiver, o resultado será igual à posição onde ela se inicia na <expressão caractere 2>.
Esta função é muito útil para pesquisar o conteúdo de variáveis ou campos dos arquivos de dados, quando estes são conhecidos apenas parcialmente. Além disso, a função determina em que posição a subcadeia pesquisada encontra-se na cadeia completa.
Se você precisar apenas determinar se a <expressão caractere 1> está contida na <expressão caractere 2>, não importando a sua posição, utilize o operador $ ou "substring" (subcadeia).

Biblioteca:

CLIPPER.LIB

Exemplos:

```
? AT("Clipper","Compilador do dBASE III é o Clipper")
* Retorna 29, posição onde se inicia Clipper.

nome = "Clipper - Guia do Usuário"
pesq = "Usu"
? AT(pesq,nome)    && retorna 19, posição onde está Usu
pesq = "dBASE"
? AT(pesq,nome)    && retorna 0 (zero)
```

* A rotina abaixo localiza um nome no arquivo da Mala direta, mesmo conhecendo-o parcialmente e apresenta o nome encontrado no vídeo:

```
USE Mala
DO WHILE .T.
   CLEAR
   vnome=SPACE(40)
   ACCEPT "Digite o Nome a ser Pesquisado: " TO vnome
```

```
   IF EMPTY(vnome)
      EXIT
   ENDIF
   LOCATE FOR vnome$nome

   continua=SPACE(1)
   DO WHILE FOUND() .AND. .NOT. EOF()
      pos=AT(vnome,nome)
      * aqui determinamos a posição para apresentar a
      * parte encontrada sobre o nome completo
      @ 10,10 SAY "Registro No.:"+STR(RECNO())
      @ 12,10 SAY nome
      SET COLOR TO W*+
      @ 12,09+pos SAY vnome
      SET COLOR TO
      @ 16,30 SAY "Continua (S/N) ?" GET continua
      READ
      IF continua()"S"
         EXIT
      ENDIF
      CONTINUE
   ENDDO
ENDDO
USE
CLEAR
RETURN
```

* aqui não interessa a posição, mas apenas se vnome
* está contido em nome

* Se o nome digitado for "SILVA", o programa encontrará o primeiro registro que contiver "SILVA" em qualquer posição no nome; por exemplo: "HUGO DE OLIVEIRA **SILVA** JÚNIOR". O nome será apresentado e a parte do nome pesquisada, que foi encontrada, aparecerá piscando. Em seguida, caso se deseje continuar a pesquisa, o programa localizará, através do comando CONTINUE, o próximo nome que contêm, SILVA.

Veja Também:

RAT( ), STRTRAN( ), SUBSTR( ), LEFT( ), RIGHT( ) e o operador "$"

Função: CHR( ) – Função Número para Caractere:

Forma:

CHR(<expressão numérica>)

Propósito:

Retorna o caractere cujo código ASCII (ou IBM Extended Character Set) é o definido pelo resultado da <expressão numérica>.

Argumentos:

<*expressão numérica*>: deve resultar em um valor entre 0 e 255, correspondente ao código ASCII ou ao IBM Extended Character Set, do caractere a ser fornecido.

Utilização:

A função CHR( ) fornece um meio de usar efeitos especiais oferecidos pela maioria das impressoras e/ou pela maioria dos monitores de vídeo, através da transformação do valor do código ASCII ou IBM-ECS em caracteres que muitas vezes não existem no teclado dos equipamentos. Os caracteres mais utilizados são em geral códigos de controle para impressoras ou símbolos gráficos a serem apresentados nos monitores de vídeo.
Uma outra aplicação típica da função CHR( ) é fazer soar o sinal sonoro do equipamento para alertar o usuário, através do comando:

```
?? CHR(7) && onde 7 é o código do "beep"
```

Em muitos casos é interessante atribuir a uma variável ao código do "beep" para fazê-lo soar:

```
beep = CHR(7)
?? beep
```

Aplicações sofisticadas para a função CHR( ) são, em geral, relacionadas com o controle do teclado, em combinação com o comando KEYBOARD, para forçar a entrada de determinados caracteres no buffer do teclado, ou testar teclas digitadas através da função INKEY( ). (CHR(INKEY( )) = caractere digitado pelo usuário.
Nas versões anteriores CHR(0) é o caractere nulo e em geral não pode ser enviado a nenhum periférico, como impressoras ou monitores. Na versão Summer 87 CHR(0) possui o comprimento 1 e é tratado como qualquer outro caractere, podendo ser enviado a periféricos ou arquivos, incluindo um arquivo de dados.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
? CHR(65)  && retorna "a", cujo código ASCII é 65.

? CHR(24)  && retorna uma seta para cima
```

* Para criar uma moldura com linha gráfica dupla, utilizando-se o comando @...BOX, pode-se utilizar a seguinte rotina:

```
CLEAR
moldura:CHR(201)+CHR(205)+CHR(187)+CHR(186)+CHR(188)+;
        CHR(204)+CHR(200)+CHR(186)
* Os caracteres gráficos gerados acima não possuem
* equivalente no teclado, ou seja, não  podem  ser
* digitados diretamente.
@ 01,00,23,79 BOX moldura
```

* Para compactar e descompactar as letras de uma impressora padrão EPSON (maioria das impressoras nacionais), pode se utilizar a seguinte rotina:

```
SET PRINT ON
?? CHR(15)      && caractér para compactação das letras
? "Impressão compactada: 17,6 caracteres por polegada"
?? CHR(18)      && caractér para retornar ao normal
? "Impressão normal: 10 caracteres por polegada"
SET PRINT OFF
```

* Para fazer soar o três vezes BEEP interno do computador pode-se utilizar:

```
FOR i=1 TO 3
    ? CHR(7)  && código para soar o BEEP do computador
NEXT
```

Veja Também:

ASC( ), INKEY( ), KEYBOARD

Função: ISALPHA( ) – É Alfabético?:

Forma:

ISALPHA(<exp.caractere>)

Propósito:

Retorna o valor lógico verdadeiro (.T.) se a <expressão caractere> especificada é iniciada por um caractere alfabético.

Argumentos:

<expressão caractere>: define a cadeia de caracteres a ser examinada.

Utilização:

A função ISALPHA( ) é utilizada para determinar se uma expressão é iniciada por um caractere alfabético. Pode ser usada, por exemplo, para validar nomes de arquivos fornecidos pelo usuário, pois estes devem neccessariamente se iniciar por uma letra.
Um caractere alfabético é qualquer caractere, maiúsculo ou minúsculo, entre as letras A e Z.

Biblioteca:

EXTEND.LIB ou
EXTENDDB.obj (nas versões anteriores à Summer 87)

Exemplos:

```
? ISALPHA("xyz987")   && retorna verdadeiro .T.
? ISALPHA("XYZ987")   && retorna verdadeiro .T.
? ISALPHA("987XYZ")   && retorna falso      .F.
```

Veja Também:

ISLOWER( ), ISUPPER( ), LOWER( ), UPPER( )

Função: ISLOWER( ) – É Minúscula?:

Forma:

ISLOWER(<exp.caractere>)

Propósito:

Retorna o valor lógico verdadeiro (.T.) se a <expressão caractere> especificada é iniciada por um caractere alfabético minúsculo.

Argumentos:

<*expressão caractere*>: define a cadeia de caracteres a ser examinada.

Utilização:

A função ISLOWER( ) retorna o valor lógico falso (.F.) se um caractere alfabético maiúsculo ou um caractere não alfabético estiver na primeira posição da expressão (cadeia de caracteres) especificada. Retorna verdadeiro (.T.) caso contrário.
Uma caractere alfabético minúsculo é qualquer caractere entre a e z.

Biblioteca:

EXTEND.LIB ou
EXTENDDB.obj (nas versões anteriores à Summer 87)

Exemplos:

```
? ISLOWER("xyz987")  && retorna verdadeiro  .T.
? ISLOWER("XYZ987")  && retorna falso       .F.
? ISLOWER("987XYZ")  && retorna falso       .F.
```

Veja Também:

ISALPHA( ), ISUPPER( ), LOWER( ), UPPER( )

Função: ISUPPER( ) – É Maiúscula?:

Forma:

ISUPPER(<exp.caractere>)

Propósito:

Retorna o valor lógico verdadeiro (.T.) se a <expressão caractere> especificada é iniciada por um caractere alfabético maiúsculo.

Argumentos:

<*expressão caractere*>: define a cadeia de caracteres a ser examinada.

Utilização:

A função ISUPPER( ) retorna o valor lógico falso (.F.) se um caractere alfabético minúsculo ou um caractere não alfabético estiver na primeira posição da cadeia de caracteres especificada; caso contrário retorna verdadeiro (.T.).
Uma caractere alfabético maiúsculo é qualquer caractere entre A e Z.

Biblioteca:

EXTEND.LIB ou
EXTENDDB.obj (nas versões anteriores à Summer 87)

Exemplos:

```
? ISUPPER("xyz987")  && retorna falso    .F.
? ISUPPER("XYZ987")  && retorna verdadeiro  .T.
? ISUPPER("987XYZ")  && retorna falso       .F.
```

Veja Também:

ISALPHA( ), ISUPPER( ), LOWER( ), UPPER( )

Função: LEFT( ) – Subcadeia da Esquerda:

Forma:

LEFT(<exp.caractere>,<exp.numérica>)

Propósito:

Extrai um número especificado de caracteres de uma cadeia de caracteres, iniciando do caractere mais a esquerda.

Argumentos:

<*expressão caractere*>: é a cadeia de caracteres de onde se deseja extrair a subcadeia mais a esquerda.
<*expressão numérica*>: define o número de caracteres a ser extraído.

Utilização:

A função LEFT( ) possibilita extrair a primeira parte (parte à esquerda) de uma cadeia de caracteres com o número de caracteres especificado pela <expressão numérica>.
Se a expressão numérica for igual a 0 (zero) ou negativa será retornada uma cadeia de caracteres nula. Se a expressão numérica for maior que o comprimento total da cadeia de caracteres, será retornada toda a cadeia.
Nas versões anteriores à Summer 87 a função LEFT( ) é escrita em Clipper e encontra-se no programa EXTENDDB.prg cujo módulo-objeto é o EXTENDDB.obj. Para utilizá-la o arquivo EXTENDDB.obj deve ser incluído na lista de módulos-objeto fornecida ao link-editor.

Biblioteca:

EXTEND.LIB ou
EXTENDDB.obj (nas versões anteriores à Summer 87)

Exemplos:

```
? LEFT("Clipper",4)  && retornará "Clip"
```

Veja Também:

AT( ), LTRIM( ), RAT( ), RIGHT( ), RTRIM( ), SUBSTR( ), ALLTRIM( ), TRIM( ), STUFF( )

Função: LEN( ) – Comprimento:

Forma:

LEN(<expressão caractere/vetor>)

Propósito:

Retorna um valor numérico igual ao número de caracteres de uma expressão caractere especificada, ou seja, o comprimento da cadeia de caracteres.
Caso seja especificado o nome de um vetor será retornado o seu número de elementos, definido pelo comando DECLARE.

Argumentos:

<expressão caractere>: é a cadeia de caracteres a ter o comprimento/tamanho determinado.
<vetor>: é o nome do vetor a ter o número de elementos determinado.

Utilização:

A função LEN( ) pode ser utilizada para se determinar o comprimento de uma cadeia de caracteres desconhecida ou a quantidade de elementos de um vetor.
Uma utilização típica da função LEN( ) é para a centralização de um título em um espaço de comprimento conhecido, como por exemplo uma linha da tela ou uma linha da impressora.
O comprimento de uma cadeia nula é zero e o comprimento de um vetor é igual ao número de seus elementos.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
? LEN("Clipper")    && retorna 7
x = "dBASE III Plus"
? LEN( x )          && retorna 14

DECLARE vetor[10]
? LEN(vetor)  && retorna 10, o número de elementos.
```

* Para centralizar uma cadeia de caracteres qualquer na tela pode-se utilizar a construção abaixo, onde um texto será centralizado na linha 12. Lembre-se que a tela possui 80 colunas.

```
texto = "Clipper"
@ 12,40-INT(LEN(TRIM(texto))/2) SAY texto
```

* Nas rotinas Mensagem e Aviso, incluídas em alguns exemplos do livro e no Sistema MLD, a função LEN( ) também é utilizada para obter-se a centralização dos textos.

Veja Também:

DECLARE( ), LENNUM( ), TRIM( ), LTRIM( ), RTRIM( ), ALLTRIM( )

Função: LOWER( ) – Minúsculo:

Forma:

LOWER(<expressão caractere>)

Propósito:

Converter as letras maiúsculas contidas na <expressão caractere> especificada em minúsculas.

Argumentos:

<*expressão caractere*>: define a cadeia de caracteres a ter as letras maiúsculas convertidas para minúsculas.

Utilização:

A função LOWER( ) é utilizada para converter uma <expressão caractere> que possua letras maiúsculas, na mesma expressão só que com letras minúsculas.
Todos caracteres alfabéticos maiúsculos serão convertidos para minúsculos. Qualquer outro caractere será ignorado.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
? LOWER("CLIPPER")    && resulta "clipper"

x = "Compilador do dBASE III"
? LOWER(x)            && resulta "compilador do dbase iii"
? LOWER(x)            && resulta falso (.F.)

? LOWER("Lotus 1-2-3" && resulta "lotus 1-2-3"
```

Veja Também:

ISALPHA( ), ISLOWER( ), ISUPPER( ), UPPER( )

Função: LTRIM( ) — Remove Brancos da Esquerda:

Forma:

LTRIM(<expressão caractere>)

Propósito:

Remove os brancos (caractere espaço em branco) existentes à esquerda de uma expressão caractere.

Argumentos:

<*expressão caractere*>: é a cadeia de caracteres a ter os espaços em branco a esquerda removidos.

Utilização:

A função LTRIM( ) é especialmente interessante de ser utilizada para remover os brancos à esquerda quando se necessita de uma adequada formatação de caracteres. Este pode ser o caso, por exemplo, quando se aplica a função STR( ), que transforma um valor numérico em caractere, trocando os "zeros" à esquerda do número por brancos.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
x = 18.25
? STR(x,10,2)                 && retorna "     18.25"
? LEN(STR(x,10,2))            && retorna 10
? LTRIM(STR(x,10,2))          && retorna "18.25"
? LEN(LTRIM(STR(x,10,2)       && retorna 5

y = "     Clipper"
? LTRIM( y )                  && retorna "Clipper"
```

Veja Também:

LEFT( ), RIGHT( ), SUBSTR( ), STR( ), RTRIM( ), TRIM( ), ALLTRIM( ).

Função: RAT( ) – Pesquisa de SubCadeia:

Forma:

RAT(<exp. caractere 1>,<exp.caractere 2>)

Propósito:

Pesquisa numa cadeia de caracteres definida pela <exp. caractere 2> a última ocorrência de uma subcadeia especificada pela <exp. caractere 1>, retornando a posição onde esta se inicia como um valor numérico.
Disponível a partir da versão Summer 87.

Argumentos:

<exp. caractere 1>: define a subcadeia de caracteres a ser localizada.
<exp. caractere 2>: é a cadeia de caracteres a ser pesquisada.

Utilização:

A função RAT( ) é semelhante à função AT( ). Ao invés de localizar a primeira ocorrência de uma cadeia de caracteres dentro de outra, como é o caso da AT( ), localiza a última, executando sua pesquisa da direita para a esquerda.
Se a subcadeia especificada pela <exp. caractere 1> estiver contida na cadeia especificada pela <exp. caractere 2> RAT( ) retornará a posição inicial da subcadeia dentro da cadeia. Se a subcadeia não for encontrada, RAT( ) retornará zero.

Biblioteca:

EXTEND.LIB

Veja Também:

AT( ), STRTRAN( ), SUBSTR( ) e o operador "$"

Função: REPLICATE( ) – Replicar:

Forma:

REPLICATE(<exp.caracter>,<exp.numérica>)

Propósito:

Repete (replica) uma expressão caractere um especificado número de vezes, retornando a cadeia de caracteres resultantes.

Argumentos:

<*expressão caractere*>: define o caractere ou cadeia de caracteres a ser replicada.
<*expressão numérica*>: define o número de vezes a ser repetida a <expressão caractere> especificada.

Utilização:

A função REPLICATE( ) é muito interessante para a formação de linhas e traços na impressora ou efeitos gráficos no vídeo (molduraas, linhas etc). A expressão resultante (após a repetição) não poderá exceder a 254 caracteres nas versões anteriores ou a 65.535 caracteres ou 64 Kbytes na versão Summer

87. Portanto, a <exp.numérica> deve ser um número menor que 254 (ou 65.535) dividido pelo número de caracteres da <expressão caracteres>.
Se a <expressão numérica> for igual a zero resultará no caractere nulo ("").

Biblioteca:

CLIPPER.LIB

Exemplos:

```
? REPLICATE("-",5)   && resulta      "-----"

? REPLICATE("*",10) && resulta "**********"
x = REPLICATE("-",132)
```

* x conterá uma cadeia de 132 tracinhos, que poderá
* ser utilizada como uma linha de separação na
* impressora.

Função: RIGHT( ) — Subcadeia da Direita:

Forma:

RIGHT(<exp.caractere>,<numérica>)

Propósito:

Retorna um número especificado de caracteres de uma cadeia de caracteres, iniciando do caractere mais à direita.

Argumentos:

<*exp.caractere*>: é a cadeia de caracteres de onde será extraída a subcadeia (sub-string) da direita.
<*exp.numérica*>: define o número de caracteres a ser extraído, a partir do caractere mais à direita da cadeia de caracteres.

Utilização:

A função RIGHT( ) retorna os "n" (determinado pela <expressão numérica >) caracteres mais à direita de uma cadeia de caracteres especificada.
Se a expressão numérica for igual a 0 (zero) ou negativa será retornada uma cadeia de caracteres nula. Se a expressão numérica for maior que o comprimento total da cadeia de caracteres, será retornada toda a cadeia.

A função RIGHT( ) é equivalente à função SUBSTR( ) se o argumento que determina a posição inicial da subcadeia for negativo. Por exemplo: RIGHT("Clipper",3) retorna a mesma subcadeia que SUBSTR("Clipper",-1,3).
Nas versões do Clipper anteriores à Summer 87, a função RIGHT( ) encontra-se no módulo-objeto auxiliar EXTENDDB.obj. Para utilizá-la este módulo deve ser incluído na lista de módulos-objeto passada ao link-editor.

Biblioteca:

EXTEND.LIB
EXTENDDB.obj (nas versões anteriores à Summer 87)

Exemplos:

```
? RIGHT("Clipper",4)  && retornará "pper"
```

* A função RIGHT( ) é útil para gerar um código com zeros à esquerda:

```
codpad    "0000000000"
numero = 27
codnum    RIGHT(codpad+LTRIM(STR(numero,10)),10)
```

* Resulta no código "0000000027"

Veja Também:

AT( ), LTRIM( ), LEFT( ), RTRIM, SUBSTR( ), ALLTRIM( ), TRIM( ).

Função: RTRIM( ) — Brancos da Direita:

Forma:

RTRIM(<exp.caractere> **)**

Propósito:

Remove os brancos (caractere espaço em branco) existentes à direita de uma expressão caractere.

Utilização:

A utilização da função RTRIM( ) é idêntica à função TRIM( ).

Exemplos:

```
x = "Clipper      "
? RTRIM( x )              && retorna "Clipper"
? LEN(x)                  && retorna 13
? LEN(RTRIM(x))           && retorna 7
```

Veja Também:

LEFT( ), RIGHT( ), SUBSTR( ), STR( ), LETRIM( ), TRIM( ), ALLTRIM( ).

Função SPACE( ) – Espaço:

Forma:

SPACE(<exp.numérica>)

Propósito:

Gera uma cadeia de caracteres formada por um número especificado de espaços em branco.

Argumentos:

<*expressão numérica*>: define o número de espaços a serem retornados na cadeia de caracteres. O número máximo na versão Summer 87 é 65.535 (64 K) e nas anteriores 254.

Utilização:

A utilização mais comum para esta função é a inicialização de variáveis caractere que deverão receber dados de um determinado comprimento. A variável é, portanto, inicializada com o número de brancos previsto pelo comprimento máximo que os dados deverão ter.
Se a <exp.numérica> que especifica o número de brancos de cadeia a ser gerada for igual a zero, será gerada uma cadeia de caracteres nula ("").
Ao se solicitar uma entrada de dados através do comando @...GET, deve-se previamente inicializar as variáveis a serem lidas. A função SPACE ( ) é útil para a inicialização de qualquer variável caractere ou data.
A função SPACE( ) pode também ser utilizada com o objetivo de formatação de dados, adicionando-se espaços para a centralização ou justificação de textos, títulos etc.

Biblioteca:

CLIPPER.LIB

Exemplos:

* Para criar e ler uma variável com 40 caracteres de comprimento através do comando @...GET:

```
vnome = SPACE(40)
@ 12,10 SAY "Digite o Nome:" GET vnome PICTURE "@!"
READ
```

* Como vnome foi gerada com 40 brancos, o comando @...GET irá apresentar em vídeo reverso um espaço de 40 posições para a entrada dos dados.

```
? "Clipper"

Resulta:

Clipper

? SPACE(10)+"Clipper"

Resulta:

          Clipper
```

* O exemplo a seguir mostra a função-de-usuário Cursor, que pode ser construída para posicionar o cursor em uma determinada posição da tela sem apagar o que lá está sendo apresentado (posicionamento não-destrutivo do cursor):

```
FUNCTION Cursor
PARAMETERS lin,col
@ lin,col SAY SPACE(0)    && gera uma cadeia nula
RETURN ""   && deve retornar também o caractér nulo
```

Veja Também:

REPLICATE( )

Função: STR – "String" ou Cadeia:

Forma:

STR(<exp.num. 1>[,<exp.num. 2>[<exp.num. 3>]])

Propósito:

Converte uma expressão numérica em uma cadeia de caracteres.

Argumentos:

<*exp.numérica 1*>: é a expressão numérica a ser convertida para uma cadeia de caracteres.
<*exp.numérica 2*>: é o comprimento da cadeia de caracteres a ser retornada, incluindo as casas decimais, o ponto decimal e o sinal de menos se houver.
<*exp.numérica 3*>: define o número de casas decimais a ser retornado.

Utilização:

Utiliza-se a função STR( ) para converter valores ou <expressões numéricas> em cadeias de caracteres. Ela é muito útil na geração de códigos numéricos sequenciais e para a impressão de dados numéricos concatenados (justapostos) com dados caracteres.
Se a <exp.numérica 2> não for especificada, será automaticamente assumida como igual a 10 posições onde serão incluídos os espaços em branco à esquerda do número, o número propriamente dito, o ponto, as casas decimais e o sinal de menos, se houver.
Por outro lado, através da <exp.numérica 2> se for especificado um comprimento menor que o da parte inteira da expressão numérica a ser convertida, serão retornados asteríscos no lugar do número. Ainda, se o comprimento definido pela <exp.numérica 2> não for suficiente para todo o número e os dígitos decimais, a cadeia de caracteres retornada será arredondada para a quantidade máxima possível de dígitos decimais.
Se o número de decimais através da <exp.numérica 3> não for especificado, o número será arredondado para um inteiro. Se forem especificadas menos casas decimais que as existentes na expressão numérica a ser convertida, o Clipper arredondará o resultado para o número de casas decimais especificadas.
A função STR( ) aplicada à funções DAY( ) e MONTH( ), retorna uma cadeia de caracteres com comprimento 3. STR( ) aplicada a YEAR( ) retorna uma cadeia de caracteres com comprimento 5.

Biblioteca:

CLIPPER.LIB

Exemplos:

* Para converter o valor 1234.345 para uma cadeia de caracteres:

```
? STR(1234.345,8,3)   && resulta "1234.345" (caractér)

x := 1234.345
? STR(*x*10 )         && resulta "     12343"
? STR( x*10,5 )       && resulta "12343"
? STR( x*10,7,1 )     && resulta "12343.4"
```

Veja Também:

STRZERO( ), VAL( ), SUBSTR( ), TYPE( )

Função: STRTRAN( ) – Pesquisa e Substituição de Cadeias:

Forma:

STRTRAN(<exp.caractere 1>,<exp.caractere 2>[,<exp.caractere 3>]
[,<exp.numérica 1>][,<exp.numérica 2>])

Propósito:

Pesquisar e substituir uma subcadeia de caracteres por outra, retornando a cadeia modificada.
Disponível a partir da versão Summer 87.

Argumentos:

<exp.caractere 1>: é a cadeia de caracteres a ser pesquisada.
<exp.caractere 2>: é a subcadeia de caracteres a ser localizada.
<exp.caractere 3>: é a subcadeia de caracteres pela qual a cadeia localizada será substituída. Se não for especificada, todas as ocorrências da subcadeia a ser localizada serão substituídas pelo caractere nulo ("").
<exp.numérica 1>: define o número da ocorrência que deverá ser substituída. Se não especificada será assumido o valor 1 (a primeira ocorrência).
<exp.numérica 2>: define o número de ocorrências a ser substituído. Se não especificada todas as ocorrências serão substituídas.

Utilização:

A função STRTRAN( ) realiza uma pesquisa de uma subcadeia, especificada pela <exp.caractere 2>, em uma cadeia, especificada pela <exp.caractere 1>. Quando uma ocorrência é encontrada é substituída pela subcadeia especificada pela <exp.caractere 3>. Caso as expressões numéricas 1 e 2 não sejam especificadas, todas as ocorrências da <exp.caratere 2> encontradas na <exp.

caractere 1> serão substituídas pela <exp.caractere 3>. Se as expressões numéricas 1 e 2 forem especificadas, apenas um determinado número de ocorrências será substituído.
O comando SET EXACT não tem influência sobre a operação da função STRTRAN( ).

Biblioteca:

CLIPPER.LIB

Exemplo:

```
texto="Ser ou não Ser, heis a questão"
? STRTRAN(texto,"Ser", "Clippar")
```

Resultado:

Clippar ou não Clippar, heis a questão.

Veja Também:

AT( ), RAT( ), SUBSTR( ), e o operador "$"

Função: STRZERO( ) – Zero na Frente de Números:

Forma:

STRZERO(<exp.num. 1>[,<exp.num. 2>[,<exp.num. 3>]])

Propósito:

Retorna uma expressão caractere a partir de uma <expressão numérica>, substituindo os brancos à esquerda por zeros.

Argumentos:

<exp.numérica 1>: define o valor a ser convertido.
<exp.numérica 2>: define o comprimento do número a ser convertido.
<exp.numérica 3>: define o número de casas decimais.

Utilização:

A função STRZERO( ) faz parte das extensões do Clipper escritas em Clipper. Seu código fonte encontra-se no arquivo EXTENDDB.prg. Para utilizá-la deve ser incluído o arquivo EXTENDDB.obj na lista de módulos-objeto passada ao link-editor.
A função STRZERO( ) é extremamente útil para a formação de códigos caracteres seqüenciais a partir de dados numéricos. É utilizada para, após a incrementação de códigos seqüenciais numéricos, transformá-los em caractere, sendo os brancos da esquerda substituídos por zeros.

Biblioteca:

EXTENDDB.obj (Versões Winter 85 e Autumn 86)

Exemplos:

```
vcod=10
vcod=vcod+1
? vcod    && retorna o valor 11
vcod=STRZERO(vcod,5,0)
? vcod    && retorna a cadeia "00011"
```

Veja Também:

LEN( ), LTRIM( ), STR( )

Função STUFF( ) – Substitui-cadeia:

Forma:

STUFF(<exp.caractere 1>,<exp.numérica 1>,
<exp.numérica 2>,<exp.caractere 2>)

Propósito:

Retorna a <expressão caractere 1>, com a parte que vai da posição definida pela <expressão numérica 1> até a <expressão numérica 2>, substituída pela <expressão caractere 2>.

Argumentos:

<*expressão caractere 1*>: define a cadeia de caracteres que será substituída.
<*expressão numérica 1*>: define a posição inicial da substituição na cadeia de caracteres que será substituída.
<*expressão numérica 2*>: define o número de caracteres a serem substituídos na <expressão caractere 1>.
<*expressão caractere 2*>: é a cadeia que substituirá os caracteres definidos na <expressão caractere 1>.

Utilização:

A função STUFF( ) permite que uma parte especificada de uma cadeia de caracteres (definida pela <expressão caractere 1>) seja substituída por um determinado número de caracteres (definido pela <exp.numérica 2>) de uma outra cadeia de caracteres (definida pela <expressão caractere 2>).
A partir dessa operação, a função STUFF( ) pode realizar as seguintes operações:

Inserção: Se a <exp.numérica 2> for igual a zero, nenhum caractere é removido da <exp.caractere 1> e toda a <exp.caractere 2> é inserida na <exp. caractere 1> a partir da posição especificada pela <exp. numérica 1>.

Substituição: Se for especificada uma <exp.caractere 2> com o mesmo número de caracteres definido pela <exp.numérica 2>, a <exp.caractere 2> substituirá os caracteres da <exp.caractere 1>, a partir da posição definida pela <exp.numérica 1>.

Eliminação: Se a <exp.caractere 2> for uma cadeia de caracteres nula (""), o número de caracteres especificados pela <exp.numérica 2> será removido da <exp.caractere 1>.

Troca e Inserção: Se a <exp.caractere 2> possuir um número maior de caracteres que o definido pela <exp.numérica 2>, todos os caracteres, a partir da posição indicada pela <exp.numérica 1>, serão substituídos e os restantes inseridos.

Troca e Eliminação: Se o tamanho da <exp.caractere 2> for menor que o número de caracteres especificados na <exp.caractere 2>, todos os caracteres na <exp.caractere 1> são eliminados a partir do final da <exp.caractere 2> até atingir o número de caracteres da <exp. numérica 2>.

Troca e Eliminação do Restante: Se a <exp.numérica 2> for maior que o tamanho da <exp.caractere 1>, a <exp.caractere 2> será inserida a partir da posição definida pela <exp.numérica 1> e o restante da <exp.caractere 1> será eliminado.

Nas versões anteriores à Summer 87, o código-fonte da função STUFF( ), escrito em Clipper, encontra-se no arquivo EXTENDDB.prg e o respectivo código-objeto no arquivo EXTENDDB.obj. Para utilizá-la o arquivo EXTENDDB.obj deve ser incluído na lista de módulos-objeto a ser encadeada pelo link-editor.

Biblioteca:

EXTEND.LIB
EXTENDDB.obj (nas versões anteriores à Summer 87)

Exemplos:

Substituição:
? STUFF("Clipper",4,2,"ff") && retorna "Cliffer"

Eliminação:
? STUFF("Clipper",3,4,"") && retorna "Cli"

Inserção:
? STUFF("Clipper",3,0,"pp") && retorna "Clipppper"

Substituição e inserção:
? STUFF("ABCDEFG",3,2,"123") && retorna "AB123EFG"

Substituição e eliminação:
? STUFF("ABCDEFG",3,4,"123") && retorna "AB123G"

Substituição e eliminação do restante:
? STUFF("ABCDEFG",3,20,"123") && retorna "AB123"

Veja Também:

AT( ), LEFT( ), RIGHT( ) RAT( ), STRTRAN( ), SUBSTR( )

Função: SUBSTR( ) — Sub-String (Subcadeia):

Forma:

SUBSTR(<exp.caractere>,<exp.num. 1>[,<exp.num. 2>])

Propósito:

Extrai uma parte especificaca (subcadeia) de uma cadeia de caracteres.

Argumentos:

<exp.caractere>: é a cadeia de caracteres a ter uma subcadeia extraída. O tamanho máximo que uma cadeia de caracteres pode ter no Clipper (versão Summer 87) é 65.535 bytes (64 K) ou 254 bytes nas versões anteriores.
<exp.numérica 1>: é a posição inicial na <exp.caractere 1> da subcadeia a ser extraída. Se for positiva será contada da esquerda para a direita. Se for negativa será contada da direita para a esquerda. Deve ser obrigatoriamente informada.
<exp.numérica 2>: é o número de caracteres da subcadeia a ser retornada, contados a partir da posição inicial especificada pela <exp.numérical>. Se for omitida a subcadeia começará na posição inicial determinada e irá até o fim da cadeia especificada pela <exp.caractere>. Se for maior que o número de caracteres existentes do início da subcadeia ao fim da cadeia, será ignorada.

Utilização:

Utiliza-se a função SUBSTR( ) em muitos casos práticos quando se deseja extrair de uma cadeia de caracteres original apenas uma determinada parte (uma subcadeia). Um exemplo típico de utilização é quando se trabalha com códigos compostos, como em geral são os números-código de contas contábeis, onde cada parte (subcadeia) identifica um grupo ou subgrupo de contas.

Biblioteca:

CLIPPER.LIB

Exemplos:

* Para extrair "11" do número da conta "2.22.11.21.00":

```
? SUBSTR("2.22.11.21.00",6,2)   && resulta "11

x = "Clipper   o Compilador do dBASE III"
? SUBSTR(x,13,10)    && resulta "Compilador"
y = SUBSTR(x,27)     && resulta "dBASE III"
z = SUBSTR(x,-1,9)   && resulta "dBASE III"
```

Veja Também:

AT( ), LEFT( ), RIGHT( ), RAT( ), STR( ), LEN( ), STUFF( ).

Função: TRIM( ) – Remove Brancos do Final:

Forma:

TRIM(<exp.caractere>) ou RTRIM(<exp.caractere>)

Propósito:

Remove os espaços em branco restantes à direita de uma expressão caractere, retornando uma cadeia de caracteres sem os espaços em branco.

Argumentos:

<expressão caractere>: é a cadeia de caracteres da qual se deseja remover os espaços em branco restantes à sua direita.

Utilização:

A função TRIM( ) é muito utilizada para eliminar os espaços restantes de um campo ou variável caractere quando o seu comprimento real é menor que o originalmente previsto. É muito útil quando se necessita concatenar strings e não se deseja incluir os brancos restantes entre elas.
Uma aplicação típica para o comando TRIM( ) é a formatação de nomes e endereços.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
CLEAR
vcidade=SPACE(20)
vestado=SPACE(2)
@ 10,20 SAY "Digite o Nome da Cidade.: " GET vnome
@ 12,20 SAY "Digite a Sigla do Estado: " GET vestado
READ
@ 14,20 SAY vcidade+"   "+vestado
* resulta, por exemplo: "SAO PAULO                SP"
* resulta, por exemplo: "RIO DE JANEIRO           RJ"
@ 16,20 say TRIM(vcidade)+"   "+vestado
* resulta, por exemplo: "SAO PAULO   SP"
* resulta, por exemplo: "RIO DE JANEIRO   RJ"

x = SPACE(20)
@ 12,20 SAY "Digite um Nome: " GET x
READ
```

```
* Se o nome digitado for, por exemplo, Clipper
? LEN(x)          && resulta 20
? LEN(TRIM(x))    && resulta 7
```

Veja Também:

RTRIM( ), LTRIM( ), ALLTRIM( ), LEN( ), SUBSTR( )

Função: UPPER( ) – Maiúscula:

Forma:

UPPER(<exp.caractere>)

Propósito:

Converte letras minúsculas em maiúsculas em expressões ou cadeias de caracteres.

Argumentos:

<expressão caractere>: define a cadeia de caracteres a ter as letras minúsculas convertidas para minúsculas.

Utilização:

A função UPPER( ) é utilizada para converter uma expressão caractere que possua letras minúsculas, na mesma expressão, só que com as letras minúculas convertidas para maiúsculas.
A função UPPER( ) é interessante para transformar qualquer resposta do usuário, tipo caractere, em letras maiúsculas. Dessa forma, as respostas podem ser testadas para efeito de tomada de decisão, não importando se foram fornecidas em letras minúsculas ou maiúsculas.

Biblioteca

CLIPPER.LIB

Exemplos

```
? UPPER("clipper")    && resulta "CLIPPER"

x   "compilador do dbase iii"
? UPPER(x)          && resulta "COMPILADOR DO DBASE III"

? UPPER(x)          && resulta falso (.F.)
```

Veja Também:

ISALPHA( ), ISLOWER( ), ISUPPER( ), LOWER( )

Função: VAL( ) – Caractere para Numérico:

Forma:

VAL(<exp.caractere>)

Propósito:

Converte uma expressão ou uma cadeia de caracteres formada por números em seu valor numérico.

Argumentos:

<expressão caractere>: define a expressão caractere formada por dígitos numéricos a ser convertida.

Utilização:

A função VAL( ) é utilizada para converter cadeias de caracteres formadas por dígitos numéricos em seu valor numérico. É útil quando se necessita realizar cálculos com expressões caracteres que contenham números.
Se a expressão a ser convertida consistir de outros caracteres que não sejam numéricos ou brancos à esquerda de números, o valor resultante será zero (0).
A função VAL( ) finaliza a conversão quando um segundo ponto decimal, o primeiro caractere não numérico ou o final da expressão for encontrado.
A parte decimal dos números convertidos será apresentada de acordo com o comando SET DECIMALS.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
x = "123" && "123" é uma cadeia de caracteres
y = 100
? x + y    && resulta em erro ("Data type in conflict")
? VAL(x) + y    && resulta 223
? VAL(x)        && resulta o valor numérico 123
? VAL("XYZ")    && resulta zero (0)
```

```
z = 123.23
SET FIXED ON
SET DECIMALS TO 1
? VAL( z )      && resulta 123.2
w = VAL( z )
? w   && resulta 123.2; na memória é armazenado 123.23
SET DECIMALS TO 2
? w   && resulta 123.23
```

Veja Também:

SET DECIMALS, SET FIXED, STR( ), STORE, SUBSTR( ), TYPE( )

6.3. FUNÇÕES PARA CONTROLE DE DATA E HORA

Função: CDOW( ) – Nome do Dia da Semana:

Forma:

CDOW(<expressão data>)

Propósito:

Retorna uma cadeia de caracteres representando o nome (em inglês) do dia da semana de uma <expressão data>.

Argumentos:

<expressão data>: é uma expressão que retorna uma data sobre a qual se deseja converter o nome do dia da semana.

Utilização

A função CDOW( ) ("Caracter Day-Off-Week") retorna "Sunday", "Monday", "Tuesday", "Wednesday", "Friday" e "Saturday"; conforme for o dia da semana da <expressão data especificada>. A expressão data poderá ser uma variável de memória, um campo, uma expressão com datas ou a data do sistema (DATE( )).
Uma data nula retorna uma cadeia de caracteres nula ("").
Para obter-se o nome do dia da semana em português, pode ser utilizado o comando FUNCTION, que permite a construção de funções-de-usuário. Veja o comando FUNCTION para maiores informações.

Biblioteca

CLIPPER.LIB

Exemplos:

```
SET DATE BRITISH
? DATE()          && retorna 29/01/88
? CDOW(DATE())    && retorna "Tuesday"
```

? "The Cristmans will be on a" +CDOW("25/12/88")
* Retorna: "The Cristmans will be on a Sunday"
* Ou seja, "O Natal será num Domingo"

Veja Também:

DOW( ), CMONTH( ), MONTH( ), DAY( ), YEAR( ), CTOD( ), DTOC( ), DTOS( ), DATE( ), FUNCTION

Função: CMONTH( ) – Nome do Mês:

Forma:

CMONTH(<expressão data>)

Propósito:

Retorna uma cadeia de caracteres representando o nome (em inglês) do mês de uma <expressão data>.

Argumentos:

<expressão data>: é uma expressão que resulta um valor tipo data do qual quer se extrair o nome do mês.

Utilização:

A função CMONTH( ) ("Caractere Month") retorna o nome (em inglês) do mês contido no resultado da <expressão data> especificada. A expressão data poderá ser uma variável de memória, um campo, uma expressão ou a data do sistema (DATE( )).

Uma data nula retorna o caractere nulo ("").
Para obter-se o nome do mês em português, pode ser utilizado o comando FUNCTION, que permite a definição de funções-de-usuário. Veja o comando FUNCTION para maiores detalhes e exemplos.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
SET DATE BRITISH
? DATE()            && retorna 26/01/88
? CMONTH(DATE())    && retorna "January"
```

? "The Cristmans is in "+CMONTH("25/12/88")
* Retorna: "The Cristmans is in December"
* Ou seja: "O Natal é em Dezembro"

Veja Também:

CDOW( ), DOW( ), MONTH( ), DAY( ), YEAR( ), CTOD( ), DTOC( ) DTOS( ), DATE( )

Função: CTOD( ) – Caractere para Data:

Forma:

CTOD(<expressão caractere>)

Propósito:

Converte uma data que foi entrada ou armazenada como uma cadeia de caracteres para uma data do tipo data. Permite a criação de variáveis tipo data, uma vez que pelo teclado somente podem ser digitados valores do tipo caractere, numéricos ou lógicos.
Retorna, portanto, um valor tipo data a partir de um valor tipo caractere.

Argumentos:

<*expressão caractere*>: é uma cadeia de caracteres consistindo de números representando uma data, em geral no formato "99/99/99", respectivamente

mes/dia/ano (padrão do Clipper) ou dia/mes/ano (se o comando SET DATE BRITISH tiver sido utilizado).
Se forem especificados apenas dois dígitos para o ano, será assumido o século XX (1900 a 1999).
Para especificar uma data nula ou vazia utiliza-se CTOD(SPACE(8)) ou CTOD(" / / ").

Utilização:

A função CTOD( ) é muito importante, pois somente através dela pode-se criar diretamente variáveis tipo data, através da transformação de uma variável tipo caractere. A única outra forma de se obter valores tipo data é através de arquivos de dados que contenham em sua estrutura campos do tipo data.
O formato da cadeia de caracteres (string) a ser convertida para data segue normalmente o padrão americano, ao seja "mm/dd/aa", mas este formato pode ser alterado comandos SET DATE e SET CENTURY.
Para tranformar para o formato "dd/mm/aa", usado no Brasil, utiliza-se SET DATE BRITISH ou SET DATE FRENCH. Para transformar para o formato "dd/mm/aaaa", utiliza-se o SET DATE BRITISH e em seguida SET CENTURY ON.
A cadeia de caracteres utilizada pela função CTOD( ) para representar uma data pode variar entre "01/01/0100" até "31/12/2999.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
SET DATE BRITISH
data = "29/02/88"
? data    DATE()
* resulta erro, pois data é uma variável tipo caracter

? CTOD(data)    DATE()   && resulta 34
? CTOD(data)+1           && resulta 01/03/88
? CTOD("30/02/88") && resulta erro, não existe tal data

dd = "01/04/88"
? TYPE(dd)
* resulta C, indicando que dd é uma variável caracter
dd = CTOD(dd)
? TYPE(dd)
* resulta D, indicando que dd é uma variável tipo data
```

Para preencher campos data de um arquivo através do comando REPLACE é necessário também utilizar a função CTOD( ):

```
USE Mala
REPLACE ALL data WITH CTOD("01/04/88")
```

Veja Também:

SET DATE, SET CENTURY, DTOC( ), DTOS( ), CDOW( ), DOW( ), CMONTH( ), MONTH( ), DAY( ), YEAR( ), DATE( ).

Função: DATE( ) – Data do Sistema Operacional:

Forma:

DATE( )

Propósito:

Retorna a data atual do Sistema Operacional, no formato padrão americano, ou seja mm/dd/aa, como um valor tipo data.

Argumentos

Nenhum

Utilização:

A função DATE( ) sempre fornece a data atual do Sistema Operacional como um valor tipo data.
A data do Sistema Operacional deve ser obrigatoriamente definida no DOS. O valor da função DATE( ) não pode ser alterado internamente. Para acertar a data do Sistema estando em uma aplicação criada com o Clipper, utiliza-se o comando "RUN date", que invoca o comando date do DOS.
Para mudar o formato padrão de DATE( ) para o utilizado no Brasil (dd/mm/aa) utiliza-se o comando SET DATE BRITISH ou SET DATE FRENCH. Para obter o século (dd/mm/aaaa), utiliza-se em seguida o comando SET CENTURY ON.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
SET DATE BRITISH
? DATE()  && mostra a data atual 26/01/88
? DATE()   CTOD("08/03/58")
```

* realiza o cálculo da diferença entre as duas datas,
* mostrando quandos dias de vida o autor tem de acordo
* com a data atual do Sistema Operacional.

```
datasis = DATE()
```

* armazena a data do sistema na variável datasis

```
? datasis  && mostra a data atual 26/01/88

? CDOW(DATE())  && retorna "Tuesday"
```

Veja Também:

SET CENTURY, SET DATE, CDOW( ), DOW( ), CMONTH( ), MONTH( ), DAY( ), YEAR( ), CTOD( ), DTOC( ), DTOS( ), RUN e o comando date do DOS.

Função: DAY( ) — Dia:

Forma:

DAY(<expressão data>)

Propósito:

Retorna o valor numérico do dia do mês de uma expressão ou de uma variável tipo data.

Argumentos:

<*expressão data*>: é valor tipo data a se obter o dia do mês.

Utilização:

A <expressão data> pode ser uma variável de memória (criada através da função CTOD( )), um campo tipo data de um arquivo de dados, uma expressão envolvendo datas ou a data do sistema — função DATE( ).
A função DAY( ) retorna um número entre 0 e 31 dependendo o dia do mês em questão.

Se a <expressão data> resultar uma data nula (vazia), a função DAY( ) retorna zero (0). Além disso, no Clipper se o mês for fevereiro, o dia 29 e o ano não for bissexto a função DAY( ) também retornará zero(0).
DAY( ) é útil quando se torna necessário utilizar o dia de uma data para realizar cálculos aritméticos.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
SET DATE BRITISH
? DATE              && retorna 26/01/88
? DAY(DATE())       && retorna o valor numérico 26

dia = DAY(DATE())
? dia               && retorna 26
? dia+100           && retorna 126
```

Veja Também:

CDOW( ), DOW( ), CMONTH( ), MONTH( ), YEAR( ), CTOD( ), DTOC( ), DTOS( ), DATE( )

Função: DOW( ) – Dia da Semana:

Forma:

DOW(<expressão data>)

Propósito:

Retorna um número entre 0 e 7, representando o dia da semana de uma variável ou <expressão data>.

Argumentos:

<expressão data>: define a data a ser retornado o valor do dia da semana.

Utilização:

A função DOW( ) é utilizada para se obter o dia da semana de uma determinada data. Pode ser utilizada em conjunto com o comando FUNCTION para

a criação de uma função que retorna em português o nome do dia da semana de uma data qualquer.
DOW( ) é útil também para se verificar o dia da semana e construir rotinas que somente aceitem datas que sejam dias úteis (excluam sábados — 7 e domingos — 1)
Domingo é representado por 1 e Sábado por 7. Se <expressão data> resultar em uma data vazia será retornado zero (0).

Biblioteca:

CLIPPER.LIB

Exemplos:

```
SET DATE BRITISH
? DATE()          && retorna 26/01/88
? DOW(DATE())     && retorna 3 ou seja Terça-Feira

ds = DOW(DATE())
? ds              && retorna o valor numérico 3
```

Veja Também:

CDOW( ), CMONTH( ), MONTH( ), DAY( ), YEAR( ), CTOD( ), DTOC( ), DTOS( ), DATE( ), FUNCTION

Função: DTOC( ) — Data para Caractere:

Forma:

DTOC(<expressão data>)

Propósito:

Converte uma espressão do tipo data em uma cadeia de caracteres.

Argumentos:

<*expressão data*>: define a data a ser convertida.

Utilização:

A função DTOC( ) retorna uma cadeia de caracteres representando uma data a partir da <expressão data> especificada. A data retornada seguirá as definições estabelecidas através dos comandos SET DATE e SET CENTURY.

Com SET DATE BRITISH será retornada uma cadeia de caracteres do tipo "dd/mm/aa" ou seja dia, mes e ano. Se SET CENTURY estiver ligado (ON) a cadeia retornada terá o formato "dd/mm/aaaa".
Se for especificada uma data vazia será retornada uma cadeia de oito brancos (dez com SET CENTURY ON).
Se for necessário criar uma chave de índice onde uma data for ser composta com uma cadeia de caracteres deve ser utilizada a função DTOS( ) ao invés de DTOC( ).
A função DTOC( ) é o inverso da função CTOD( ).

Biblioteca:

CLIPPER.LIB

Exemplos:

```
SET DATE BRITISH
? DATE()              && retorna 26/01/88
? DTOC(DATE())        && retorna "26/01/88"
data = DATE()
*
? "A data de hoje é "+data
* Retorna erro (Data type in conflict), pois não se
* pode concatenar uma cadeia de caracteres com uma
* variável tipo data.
*
? "A data de hoje é "+DTOC(data)
* Retorna "A data de hoje é 26/01/88"
? TYPE(data)    && retorna D, pois data é do tipo data
? TYPE(DTOC(data)) && retorna C, houve a conversão

dc = DTOC(DATE())
? dc                  && retorna "26/01/88"
? TYPE(dc)            && retorna C
```

Veja Também:

SET DATE, SET CENTURY, CDOW( ), DOW( ), CMONTH( ), MONTH( ), DAY( ), YEAR( ), CTOD( ), DTOS( ), DATE( ), TYPE( )

Função: DTOS( ) — Data Para Caractere Invertida:

Forma:

DTOS(<expressão data>)

Propósito:

Retorna uma cadeia de caracteres no formato "aaaammdd" (ano-mes-dia) equivalente à data especificada na <expressão data>.

Argumentos:

<*expressão data*>: define a data a ser convertida para o formato "aaaammdd".

Utilização:

A função DTOS( ) pode ser utilizada para formar a chave de um índice onde é necessário compor uma <expressão caractere> com uma <expressão data> (caractere e ordem cronológica) simultaneamente. A data será transformada em uma cadeia de caracteres no formato "aaaammdd" (ano-mes-dia) permitindo a classificação cronológica e a composição com cadeias de caracteres.
Note que DTOS( ) independe dos comandos SET DATE e SET CENTURY.
Quando a <expressão data> retornar uma data vazia DTOS( ) retornará uma cadeia de seis brancos.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
SET DATE BRITISH
data=CTOD("25/12/88")
? DTOS(data)     && retorna "19881225"
```

* Para indexar um arquivo pelo campo data de cadastramento (tipo data) e pelo campo nome (caractere com 30 posições), utiliza-se:

```
USE Mala
INDEX ON DTOS(data)+nome TO Indatnom
```

* O arquivo estará classificado por data de cadastramento e, dentro de uma mesma data, por ordem alfabética.

Veja Também:

INDEX, CDOW( ), DOW( ), MONTH( ), DAY( ), YEAR( ), CTOD( ), DTOC( ), DATE( )

Função: ELAPTIME( ) – Tempo Decorrido:

Forma:

ELAPTIME(<hora inicial caractere>,<hora final caractere>)

Propósito:

Retorna uma cadeia de caracteres mostrando a diferença, em segundos, entre duas cadeias de caracteres que representam uma hora inicial e uma hora final, no formato hh:mm:ss considerando um dia de 24 horas.
Versões Winter 85 e Autumn 86.

Argumentos:

<*hora inicial caractere*>: é a primeira marcação do tempo, no formato caractere "hh:mm:ss".
<*hora final caractere*>: é a segunda marcação do tempo no formato caractere "hh:mm:ss"

Utilização:

A função ELAPTIME( ) faz parte das extensões do Clipper escritas em Clipper. Seu código-fonte encontra-se no arquivo EXTENDDB.prg e o respectivo código-objeto no arquivo EXTENDDB.obj. Para utilizá-la o arquivo ETENDDB.obj deve ser incluído na lista de módulos-objeto a serem encadeados pelo link-editor.
A função ELAPTIME( ) retorna a diferença, em segundos, de duas cadeias de caracteres no formato hh:mm:ss que representam horas. Se a hora inicial for maior que a hora final a função ELAPTIME( ) assumirá que houve mudança de dia.
As horas válidas são cadeias de oito caracteres que assumem valores entre "00:00:00" e "23:59:59".
Esta função é útil para se determinar o tempo decorrido entre duas operações efetuadas pela aplicação desenvolvida, ou por dois eventos provocados pelo usuário do sistema.

Biblioteca:

EXTENDDB.obj (nas versões anteriores à Summer 87)

Exemplos:

```
? TIME( )                          && retorna 14:00:00
hora = "15:00:00"
? ELAPTIME(hora,TIME( ))           && retorna 3600 (o número de
                                   && segundos em uma hora.
```

Veja Também:

SECONDS( ), TIME( ), DAYS( ), TSTRING( )

Função: MONTH( ) – Mês:

Forma:

MONTH(<expressão data>)

Propósito:

Retorna um número entre 0 e 12 representando o mês de uma <expressão data>.

Argumentos:

<*expressão data*>: define a data a ser retornado o número do mês.

Utilização:

A <expressão data> pode ser uma variável data, um campo data, uma expressão envolvendo datas ou a data do sistema – DATE( ).
A função MONTH( ) é muito útil em aplicações que requeiram cálculos com os meses e para criar uma função-de-usuário que retorne os nomes dos meses em português através do comando FUNCTION.
Se a <expressão data> especificada resultar em uma data nula ou vazia, será retornado o valor zero.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
SET DATE BRITISH
? DATE()          && retorna 26/01/88
? MONTH(DATE())   && retora o valor numérico 1.
data = CTOD("15/10/88")
? MONTH(data)     && retorna o valor numérico 10.
mes = MONTH(data)
? mes             && retorna o valor numérico 10
```

Veja Também:

CDOW( ), DOW( ), CMONTH( ), DAY( ), YEAR( ), CTOD( ), DTOC( ), DTOS( ), DATE( )

Função SECONDS( ) – Segundos:

Forma:

SECONDS( )

Propósito:

Retorna um valor representando o número de segundos decorridos desde as 00:00 horas (meia-noite).

Argumentos:

Nenhum

Utilização:

A função SECONDS( ) retorna um valor entre 0 e 86399 (24 horas × 60 minutos × 60 segundos), que representa o número de segundos passados desde a meia noite (00:00 horas) até a hora atual, baseando-se na marcação de 0 a 24 horas e na hora do Sistema Operacional.
Pode ser utilizada para se determinar o tempo decorrido, em segundos, entre duas marcações de hora, com o objetivo de se avaliar a performance de determinada operação durante a execução de um programa.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
? TIME()       && retorna 09:00:00
? SECONDS()    && retorna 32400

t1=SECONDS()
USE Mala INDEX Indcod
REINDEX
t2=SECONDS()
? "Tempo de reindexação:",t2-t1
```

* retorna a quantidade de segundos gasta na reindexação do arquivo.

Veja Também:

ELAPTIME( ), TIME( ), SECS( )

Função: TIME( ) – Hora do Sistema:

Forma:

TIME( )

Propósito:

Retorna a hora do Sistema Operacional como uma cadeia de caracteres, no formato hora caractere: hh:mm:ss.

Argumentos:

Nenhum

Utilização:

A função TIME( ) retorna a hora atual do sistema como uma cadeia de oito caracteres sempre que for executada. O valor da hora caractere pode variar entre "00:00:00" e "23:59:59". Para fazer cálculos é necessário transforma-la em valor numérico, ou utilizar diretamente as funções ELAPTIME( ) ou SECONDS( ).
Para acertar a hora do sistema é necessário utilizar o comando **time** do DOS. Estando em uma aplicação feita em Clipper, para acertar a hora utilize o comando "RUN time" para invocar o time do DOS.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
? TIME()  && retorna 17:58:12
hora = TIME()
? "A hora atual é: "+hora
* Retorna: A hora atual é 17:59:10
? TYPE(TIME())  && retorna C, pois TIME() é caracter.
```

Veja Também:

SECONDS( ), DATE( ), RUN e time do DOS

Função: TSTRING( ) – Segundos para Hora:

Forma:

TSTRING(<exp.numérica>)

Propósito:

Retorna uma cadeia de oito caracteres representando a hora, a partir de uma determinada quantidade de segundos.

Argumentos:

<*exp.numérica*>: define a quantidade de segundos a ser transformada.

Utilização:

A função TSTRING( ) faz parte das extensões do Clipper escritas em Clipper. Seu código-fonte encontra-se no arquivo EXTENDDB.prg e seu respectivo módulo-objeto no arquivo EXTENDDB.obj.
Para utilizá-la o arquivo EXTENDDB.obj deve ser incluído na lista de módulos-objeto passada ao link-editor.
A função TSTRING( ) retorna uma hora no formato hh:mm:ss (hora caractere) a partir de uma quantidade de segundos especificada pela <expressão numérica>.
Pode ser utilizada, por exemplo, em conjunto com as funções SECONDS( ), ELAPTIME( ) e TIME( ) em aplicações onde é necessário a cronometragem e aferição de tempos de processamento.

Biblioteca:

EXTENDDB.obj (nas versões anteriores à Summer 87)

Exemplos:

```
? TSTRING(3600)        && retorna "01:00:00"
? TSTRING(43200)       && retorna "12:00:00"
```

Veja Também:

TIME( ), SECS( ), ELAPTIME( )

Função: YEAR( ) – Ano:

Forma:

YEAR(<expressão data>)

Propósito:

Retorna o valor numérico do ano de uma expressão data.

Argumentos:

<*expressão data*>: é a data a partir da qual se deseja obter o valor do ano.

Utilização:

A função YEAR( ) é usada para se obter o valor do ano, a partir de uma data ou expressão data. Este valor poderá, ser utilizado, por exemplo, para a realização de cálculos aritméticos.
A <expressão data> pode ser uma variável ou um campo tipo data. O ano será retornado no formato aaaa, incluindo os dígitos do século, como por exemplo 1988.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
SET DATE BRITISH
? DATE()     && retorna 26/01/88
? YEAR(DATE())  && retorna 1988
ano = YEAR(DATE())
? ano    && retorna o valor numérico 1988
? YEAR(DATE())+22  && retorna 2000
```

Veja Também:

SET CENTURY, CDOW( ), DOW( ), CMONTH( ), DAY( ), MONTH( ), CTOD( ), DTOC( ), DTOS( ), DATE( )

6.4. FUNÇÕES PARA MANIPULAÇÃO DE VETORES

Função: ACHOICE( ) – Menu Instantâneo:

Forma:

ACHOICE(<exp.num.1>,<exp.num.2>,<exp.num.3>,<exp.num.4>,
<vetor 1>[,<vetor 2>[,<exp.caractere>[,<exp.num.5>[,
<exp.num.6>]]]])

Propósito:

Cria e executa um menu de opções instantâneo ("pop-up menu"), semelhante aos ou FrameWork ou Windows, usando um vetor que contém, em seus elementos, as opções a serem apresentadas ao usuário. Retorna um valor numérico de acordo com a opção escolhida.
Disponível a partir da versão Summer 87.

Argumentos:

<exp.numérica 1 a 4>: são respectivamente as coordenadas do canto superior esquerdo e do canto inferior direito da janela do menu a ser apresentado.
<vetor 1>: é um vetor tipo caractere, cujos elementos são as opções a serem apresentadas ao usuário.
<vetor 2>: é um vetor paralelo ao primeiro com valores lógicos em seus elementos, um para cada opção do menu, definidas pelo <vetor 1>. Se um de seus elementos for falso (.F.), a opção correspondente do menu não estará disponível. É possível, entretanto, especificar este argumento como um único

valor lógico. Se o valor for verdadeiro (.T.) todas as opções do menu estão disponíveis, caso contrário, todas as opções do menu não estarão disponíveis.
<*exp.caractere*>: define o nome de uma função-de-usuário (sem parênteses ou argumentos) a ser executada quando uma determinada tecla de exceção for pressionada (Alt-Letra ou Ctrl-Letra).
<*exp.numérica 5*>: é o número que especifica o elemento da opção inicial. Se não for especificado, será assumido o elemento número 1 do <vetor 1>.
<*exp.numérica 6*>: define a linha inicial relativa da janela do menu. Se não for especificada, será assumida como a primeira opção, equivalente à posição zero (0).

Utilização:

A função ACHOICE( ) retorna a posição do elemento ou opção escolhida dentro do vetor de opções. Quando o usuário fizer uma escolha o menu é automaticamente finalizado e retorna a posição do elemento escolhido do vetor 1 (vetor de opções). Se a escolha for abortada o menu é finalizado e retornará o valor zero.
ACHOICE( ) opera de maneira diferente se for ou não especificada uma função-de-usuário.
Se não for especificada uma função-de-usuário através da <expressão caractere>, ACHOICE( ) mostra a lista de opções disponíveis dentro das coordenadas da tela definidas e em seguida executa as seguintes operações, dependendo da tecla que for pressionada:

Tecla	Ação
Seta para Cima	Sobe um elemento (ou opção)
Seta para Baixo	Desce um elemento (ou opção)
Home ou Ctrl-PgDn	Primeiro elemento (ou opção)
End ou Ctrl-PgUp	Último elemento (ou opção)
PgUp	Sobe o número de elementos definidos pela janela
PgDn	Desce o número de elementos definidos pela janela
Return ou Enter	Seleciona um elemento (ou opção) retornando sua posição
Primeira Letra	Seleciona o elemento (opção) iniciado pela letra pressionada
Esc Seta Esquerda Seta Direita	Abortam a seleção, retornando zero

Se o número de opções do <vetor 1> for superior ao número de linhas na janela do menu, as opções "rolarão" quando o usuário tentar ultrapassar os limites da janela.
Quando for especificada uma função-de-usuário através da <exp.caractere> o funcionamento da função ACHOICE( ) é alterado.
Primeiramente algumas teclas não serão mais processadas por ACHOICE( ) e o controle passará para a função-de-usuário, que deverá realizar as operações correspondentes.
As teclas listadas a seguir serão processadas por ACHOICE( ) quando for definida uma função-de-usuário, gerando um resultado nulo (modo 0 — espera), enquanto qualquer outra tecla será tratada como exceção (modo 3) e será passada para a função definida:

Tecla	Ação
Seta para Cima	Sobe um elemento (opção)
Seta para Baixo	Desce um elemento (opção)
PgUp	Sobe o número de elementos definidos para a janela do menu
PgDn	Desce o número de elementos definidos para a janela do menu.
Ctrl-PgDn	Posiciona no primeiro elemento
Ctrl-PgUp	Posiciona no último elemento

Note que as teclas Home, End, Enter e Esc também geram uma exceção, que poderá ser convenientemente tratada pela função-de-usuário, cujo nome deve ser definido pela <expressão caractere>.
Quando a função ACHOICE( ) executa a função-de-usuário são passados automaticamente três parâmetros: o modo, o atual elemento do vetor de opções e a sua posição relativa dentro da janela do menu.
O modo indica os seguintes estados de ACHOICE( ) quando a tecla de exceção foi pressionada:

Modo	Estado
0	Nada (aguardando escolha)
1	O cursor tentou ultrapassar a primeira opção
2	O cursor tentou ultrapassar a última opção
3	Uma tecla de exceção foi pressionada
4	Nenhuma opção selecionável

Após a função-de-usuário executar as operações apropriadas, de acordo com o modo da função ACHOICE( ) ou de acordo com a última tecla pressionada, deve ser retornado um valor à ACHOICE( ) instruindo-a sobre que operação fazer. Os seguintes valores de retorno são possíveis:

Valor	Ação
0	Abortar a seleção, retornando zero
1	Fazer uma seleção, retornando o número do elemento onde está posicionado o cursor
2	Continuar a seleção
3	Ir para o elemento (opção) cujo primeiro caractere é igual à última tecla pressionada

Cores:

As opções são apresentadas na cor padrão (as mesmas usadas para o comando @...SAY), a barra luminosa de opções (cursor apontador) é apresentada na cor realçada (as mesmas usadas para o comando @...GET), e as opções não disponíveis são apresentadas na cor neutra. Por exemplo:

```
SET COLOR TO G+/N,  R/B  ,,,W/N
              padrão,realçada,,,neutra
```

apresentará um menu verde intenso sobre fundo preto, a barra seletora será vermelha sobre letras azuis e as opções não disponíveis serão brancas sobre fundo preto.
Veja o comando SET COLOR TO para maiores detalhes.
ACHOICE( ) pode ser executada com todas as opções não disponíveis, para apenas apresentar ao usuário uma lista de opções, não permitindo, entretanto, efetuar uma escolha. Neste caso é interessante atribuir à cor neutra a mesma cor da padrão.

Intercalação:

O Clipper permite executar várias funções ACHOICE( ) intercaladas, capacitando-o a criar vários níveis de menus hierárquicos, uns sendo chamados pelos outros, dando um sensacional efeito de janelas. Divirta-se!

Janelas:

Para salvar e reapresentar à região da tela onde os menus de ACHOICE( ) serão apresentados, sem a necessidade de reconstruí-la, devem ser utilizadas as funções SAVESCREEN( ) e RETSCREEN( ) respectivamente.

Biblioteca:

EXTEND.LIB

Exemplo:

```
DECLARE op[5]
op [1] = "Inclusão"
op [2] = "Alteração"
op [3] = "Exclusão"
op [4] = "Consulta"
op [5] = "Relatórios"

escolha = ACHOICE(10,10,15,20,op)
```

* Será apresentado um menu com cinco opções, em uma janela iniciando-se nas coordenadas 10,10 e finalizando-se nas coordenadas 15,20. O número da opção escolhida será armazenado na variável escolha.

Veja Também:

@...PROMPT, MENU TO, SET MESSAGE, DBEDIT( ), SAVESCREEN( ), RETSCREEN( )

Função: ACOPY( ) — Cópia de Vetores:

Forma:

ACOPY(<vetor 1>,<vetor 2>
[<exp.num.1>[,<exp.num.2>[,<exp.num.3>]]])

Propósito:

Copiar os elementos de um vetor para outro vetor.
Disponível a partir da versão Summer 87.

Argumentos:

<*vetor 1*>: é o vetor a ser copiado.
<*vetor 2*>: é o vetor destino da cópia
<*exp.numérica 1*>: define o número do primeiro elemento do <vetor 1> a ser copiado.
<*exp.numérica 2*>: define o número de elementos do <vetor 1> a serem copiados, a partir do elemento definido pela <exp.numérica 1>.
<*exp.numérica 3*>: é o número do primeiro elemento do <vetor 2> a partir de onde a cópia deverá ser iniciada.

Utilização:

A função ACOPY( ) constitui um meio incrivelmente rápido e eficaz para se copiar um vetor inteiro ou parte dele em outro vetor. A cópia poderá iniciar-se a partir de um determinado elemento e realizar-se para um determinado elemento do vetor destino.

Biblioteca:

EXTEND.LIB

Exemplo:

```
DECLARE vetor1[10], vetor2[10]
ADIR("*.dbf",vetor1)
ADIR("*.prg",vetor2)
ACOPY(vetor1,vetor2,1,2)
FOR i = 1 TO 10
    ? vetor1[i], vetor2[i]
NEXT
```

Veja Também:

ADEL( ), ADIR( ), AFIELDS( ), AFILL( ), AINS( ), ASCAN( ), ASORT( ), LEN( )

Função: ADEL( ) – Eliminação de Elemento:

Forma:

ADEL(<vetor>,<exp.numérica>)

Propósito:

Elimina um elemento do vetor cujo número é especificado pela expressão numérica.

Argumentos:

<*vetor*>: é o nome do vetor a ter o elemento eliminado.
<*exp.numérica*>: define a posição (número) do elemento a ser eliminado.

Utilização:

Esta função deve ser utilizada quando se deseja eliminar um elemento qualquer de um vetor, deslocando todos os outros, de ordem maior, uma posição para cima.
O conteúdo do elemento eliminado do vetor, na posição especificada, será perdido (vetor[posição] = valor perdido).
Todos os outros elementos do vetor, da posição de onde foi eliminado o elemento, até o final do vetor (vetor[LEN(vetor)]) serão transladados uma posição para a esquerda. O último elemento terá seu conteúdo indefinido (vetor [LEN(vetor)] = indefinido.
LEN(vetor) retorna o número de elementos do vetor definido no comando DECLARE..
A função DEL( ) é equivalente a:

```
DECLARE vetor[20]
n=LEN(vetor)      && i é igual a 20
pos=7
FOR i=pos TO n-1
    vetor[i] = vetor[i+1]
NEXT
? vetor[pos]    && passa a ter o valor de pos+1
vetor[n] = ""   && o elemento 20 é indefinido.
```

Biblioteca:

EXTEND.LIB ou DBU.LIB (versões anteriores a Summer 87)

Exemplos:

```
DECLARE vetor 10
vetor[1]  = "A"
vetor[2]  = "B"
vetor[3]  = "C"
vetor[4]  = "D"
vetor[5]  = "E"
```

```
vetor[6]  = "X"     && valor estranho, deve ser eliminado
vetor[7]  = "F"
vetor[8]  = "G"
vetor[9]  = "H"
vetor[10] = "I"

ADEL(vetor,6)       && elimina o elemento indefinido na
                    && sexta posição do vetor.

? vetor[5]          && resulta "E"
? vetor[6]          && resulta "F"
? vetor[7]          && resulta "G"
? vetor[10]         && resulta indefinido
```

Veja Também:

DECLARE, ACHOICE( ), ACOPY( ), ADIR( ), AFIELDS( ), AFILL( ), AINS( ), ASCAN( ), ASORT( ), LEN( )

Função: ADIR( ) – Vetor Diretório:

Forma:

ADIR(<seleção do diretório>[,<vetor1>[,<vetor2>,
[,<vetor3>[,<vetor4>[,<vetor5>]]]]])

Propósito:

Preenche uma série de vetores com informações sobre o diretório de um disco, incluindo os nomes dos arquivos e retornando o número de arquivos que atendem a uma seleção especificada.

Argumentos:

<seleção do diretório>: é uma máscara tipo DOS para selecionar um conjunto de arquivos do disco. Os caracteres de substituição padrão são o asterísco (*) e a interrogação (?). Se não for especificado será assumido "*.*".
<vetor 1>: define o nome do vetor a receber os nomes dos arquivos selecionados. Cada elemento será do tipo caractere.
<vetor 2>: define o nome do vetor a receber o tamanho (em bytes) dos arquivos do <vetor 1>. Cada elemento é do tipo numérico.
<vetor 3>: define o nome do vetor a receber das datas de atualização/criação dos arquivos selecionados no <vetor 1>. Cada elemento é do tipo data.

<*vetor 4*>: define o nome do vetor a receber a hora de criação/atualização dos arquivos selecionados pelo <vetor 1>. Cada elemento é do tipo caractere.
<*vetor 5*>: define o nome do vetor a receber os atributos dos arquivos selecionados. Cada elemento é do tipo caractere. Os possíveis valores são: R (read only), H (hidden), S (system), D (directory) e A (archive). Reporte-se ao manual do DOS para maiores detalhes sobre o significado dos atributos dos arquivos.
Se o <vetor 5> não for especificado apenas arquivos normais (que não sejam H, S e D) serão incluídos na lista.
Nas versões anteriores à Summer 87, apenas o <vetor 1> é suportado.

Utilização:

ADIR( ) é uma função útil para construir rotinas para a manutenção de arquivos.
A função ADIR( ) retorna o número de arquivos, que atendem à seleção de diretório especificada. Além disso, se forem especificados um ou mais nomes de vetores estes serão preenchidos com informações de diretório sobre os arquivos selecionados. Através destes vetores pode-se manipulá-las e em seguida apresentá-las ao usuário da forma desejada.
Para criar vetores com as informações de diretório estes devem ser antes declarados através do comando DECLARE. Para isso pode-se utilizar a função ADIR( ) para definir o número de elementos (equivalentes aos arquivos selecionados) de cada vetor, utilizando-a como argumento no comando DECLARE. Por exemplo:

```
DECLARE dir_dbf[ADIR("*.dbf")]
```

Os vetores terão preenchidos cada um de seus elementos com o nome, tamanho, data de atualização, hora de atualização e atributos dos arquivos, até que todos os arquivos selecionados tenham sido listados ou todos os elementos do vetor tenham sido preenchidos.

Biblioteca:

EXTEND.LIB
DBU.LIB (versões anteriores à Summer 87)

Exemplo:

```
DECLARE vedir[ADIR("*.OBJ")]
* número de arquivos que possui a extensão.OBJ

ADIR("*.OBJ",vedir)
* os elementos de vedir contêm, agora, os nomes dos arquivos .OBJ.
```

Veja Também:

DECLARE, ACHOICE( ), ACOPY( ), ADEL( ), AFIELDS( ), AFILL( ), AINS( ), ASCAN( ), ASORT( ), LEN( )

Função: AFIELDS( ) – Estrutura dos Campos:

Forma:

AFIELDS([<vetor1>[,<vetor2>[,<vetor3>[,<vetor4>]]]])

Propósito:

Preencher uma série de vetores com as características dos campos de um arquivo de dados em uso (nome dos campos, tipo de dados, tamanho, e casas decimais), retornando o número de campos do arquivo.
Disponível a partir da versão Summer 87.

Argumentos:

<*vetor 1*>: define o nome do vetor a ser preenchido com os nomes dos campos. Cada elemento será do tipo caractere.
<*vetor 2*>: define o nome do vetor a ser preenchido com o tipo de dados dos campos do <vetor 1>. Cada elemento será do tipo caractere.
<*vetor 3*>: define o nome do vetor a ser preenchido com o tamanho dos campos. Cada elemento será do tipo numérico.
<*vetor 4*>: define o nome do vetor a ser preenchido com o número de casas decimais dos campos. Cada elemento será do tipo numérico.

Utilização:

A função AFIELDS( ) retorna o número de campos de um arquivo de dados ou o comprimento do menor vetor especificado como argumento; o menor valor entre eles. Se nenhum parâmetro for especificado ou se não houver um arquivo de dados em uso na área de trabalho selecionada, AFIELDS( ) retornará zero.
AFIELDS( ) é muito útil para fornecer informações sobre a estrutura de arquivos de dados em rotinas de gerenciamento de arquivos. Os quatro vetores passados como argumentos terão seus elementos preenchidos respectivamente com o nome, tipo, tamanho e casas decimais dos campos do arquivo que estiver aberto na área de trabalho selecionada, sendo retornado ainda o número de campos do arquivo.

Para preencher os vetores com apenas alguns atributos dos campos, ignorando outros, podem ser criadas "variáveis fantasma". Por exemplo, para obter apenas o nome e o tamanho dos campos do arquivo, utiliza-se a seguinte construção:

```
DECLARE cnome[FCOUNT( )],ctipo[FCOUNT( )]
fantasma = "" && valor nulo
AFIELDS(cnome, fantasma, ctipo)
```

Biblioteca:

EXTEND.LIB

Exemplo:

```
CLEAR
USE Mala
n = FCOUNT()
* FCOUNT() conta o número de campos do arquivo
DECLARE cnome[n], ctipo[n], ctamanho[n], cdecimais[n]
AFIELDS(cnome, ctipo, ctamanho, cdecimais)
@ 10,30 SAY "Estrutura do Arquivo"
@ 12,20 SAY "Nome do Campo   Tipo   Tamanho   Decimais"
FOR i=1 TO n
    @ 12+i,18 SAY i
    @ 12+i,22 SAY cnome[i]
    @ 12+i,37 SAY ctipo[i]
    @ 12+i,44 SAY ctamanho[i]
    @ 12+i,55 SAY cdecimais[i]
NEXT
RETURN
```

Veja Também:

DECLARE, ACHOICE( ), ACOPY( ), ADEL( ), ADIR( ), AFILL( ), AINS( ), ASCAN( ), ASORT( ), LEN( ), FCOUNT( )

Função: AFILL( ) — Preenchimento de Vetor:

Forma:

AFILL(<vetor>,<expressão>[,<exp.num.1> ,<exp.num.2>|])

Propósito:

Preenche os elementos de um vetor com um mesmo conteúdo especificado.

Argumentos:

<*vetor*>: é o nome do vetor a ser preenchido.
<*expressão*>: é a expressão com a qual os elementos do vetor serão preenchidos. Poderá ser uma expressão de qualquer tipo de dado.
<*expressão*>: é o número (posição) do primeiro elemento do vetor a ser preenchido; se não especificado será assumido o padrão (default) 1 (primeiro elemento).
<*exp.numérica 2*>: é o número de elementos a ser preenchido; se não especificado será assumido o padrão (default) todos, do primeiro ao último elemento do vetor.

Utilização:

Quando um vetor é declarado (comando DECLARE), seus elementos, apesar de definidos, possuem um conteúdo nulo. Para preencher todos os elementos do vetor, rápida e eficientemente, com um determinado conteúdo, utiliza-se a função AFILL( ).
A função AFILL(vetor,0) é equivalente, só que muito mais rápida e eficiente, a:

```
FOR i=1 TO n            && n é o número do último elemento.
    vetor[i] = 0        && zero é o valor a ser preenchido.
```

Todos os elementos desde o início (vetor[inicio] até o final do contador (vetor[inicio+contador-1]) serão preenchidos com a mesma expressão especificada.
A expressão pode ser qualquer expressão válida do tipo caractere, numérica ou data.

Biblioteca:

EXTEND.LIB
DBU.LIB (versões anteriores à Summer 87)

Exemplo:

* Para se preencher todos os elementos de um vetor com zeros (0) e de outro com espaços em branco pode-se utilizar:

```
DECLARE vet1[20],vet2[30]
AFILL(vet1,0)
AFILL(vet2,SPACE(30))
```

* Para se preencher alguns elementos de um vetor com zeros (0), espaços em branco e asteriscos, pode-se utilizar:

```
DECLARE vet3[30]
AFILL(vet3,0,1,10)
AFILL(vet3,SPACE(20),11,20)
AFILL(vet2,"*",21,30)
```

Veja Também:

DECLARE, ACHOICE( ), ACOPY( ), ADEL( ), ADIR( ), AFIELDS( ), AINS( ), ASCAN( ), ASORT( ), LEN( )

Função: AINS( ) – Inserção de Elemento:

Forma:

AINS(<vetor>,<exp.numérica>)

Propósito:

Permite a inserção de um novo elemento, de valor indefinido, em uma posição especificada de um vetor.

Argumentos:

<*vetor*>: é o nome do vetor a ter um novo elemento inserido.
<*exp.numérica*>: define o número da posição do elemento a ser inserido.

Utilização:

Esta função deve ser utilizada quando se deseja inserir um novo elemento em uma determinada posição de um vetor, deslocando todos os outros, de ordem maior, uma posição para baixo (direita).
O novo elemento inserido no vetor, na posição especificada passa a ter um conteúdo indefinido (vetor[posição] = valor indefinido); deve ser, portanto, definido após a inserção.
Todos os outros elementos do vetor, da posição onde foi inserido o novo elemento até o final do vetor (vetor[LEN(vetor)]), serão transladados uma posição para a direita (baixo). O último elemento será perdido.
LEN(vetor) é o número de elementos do vetor definido no comando DECLARE.
A função AINS( ) é equivalente a:

```
DECLARE vetor[20]
n=LEN(vetor)      && n é igual a 20
pos=7
FOR i=n TO (pos-1) STEP -1
    vetor[i] = vetor[i-1]
NEXT
vetor[pos] = ""  && passa a ser indefinido
* o elemento 20 é perdido.
```

Biblioteca:

EXTEND.LIB
DBU.LIB (versões anteriores à Summer 87)

Exemplo:

```
DECLARE vetor 10
vetor[1]     = 1
vetor[2]     = 2
vetor[3]     = 3
vetor[4]     = 5  && foi "pulado" o número 4
vetor[5]     = 6
vetor[6]     = 7
vetor[7]     = 8
vetor[8]     = 9
vetor[9]     = 10
vetor[10]    = 11

AINS(vetor,4)          && insere um elemento indefinido na
                       && quarta posição do vetor.

vetor 4      = 4
? vetor[3]         && resulta 3
? vetor[4]         && resulta 4
? vetor[5]         && resulta 5
? vetor[10]        && resulta 10
```

Veja Também:

DECLARE, ACHOICE( ), ACOPY( ), ADEL( ), ADIR( ), AFIELDS( ), AFILL( ), ASCAN( ), ASORT(0, LEN( )

Função: ASCAN( ) – Pesquisa de Elemento:

Forma:

ASCAN(<vetor>,<expressão>|,<exp.num.1>[,<exp.num.2>]])

Propósito:

Pesquisar um valor específico dentro dos elementos de um vetor, retornando a posição (número) do elemento encontrado ou zero, caso não seja encontrado.

Argumentos:

<*vetor*>: é o nome do vetor a ser pesquisado.
<*expressão*>: é o conteúdo (chave) a ser pesquisado. Pode ser uma expressão de qualquer tipo de dado.
<*exp.numérica 1*>: define o número do primeiro elemento a ser pesquisado; se não especificado o padrão (default) será igual a 1.
<*exp.numérica 2*>: define o número de elementos a ser pesquisado, a partir do elemento definido pela <exp.numérica 1>; se não especificado o padrão (default) será todos os elementos do vetor, do início definido até o último elemento.

Utilização:

Utiliza-se a função ASCAN( ) quando se deseja rapidamente descobrir o número do elemento de um vetor que possui um determinado conteúdo. A função ASCAN( ) retorna o número do primeiro elemento que possui o conteúdo pesquisado ou 0 (zero) se o conteúdo não for encontrado.
A função ASCAN( ) é equivalente a seguinte função-de-usuário:

```
FUNCTION Pesqvet
PARAMETERS vetor, expressão, inicio, contador
FOR i= inicio TO (inicio+contador-1)
    IF vetor[i] = expressao
       RETURN(i)
    ENDIF
NEXT
RETURN(0)
```

Biblioteca:

EXTEND.LIB
DBU.LIB (versões anteriores à Summer 87)

Exemplo:

```
* Para pesquisar o conteúdo "Clipper" em um vetor:

n = ASCAN(vetor,"Clipper")
? n,vetor[n]    && resulta o número do elemento do
                && vetor e o seu conteúdo "Clipper"
```

Veja Também:

DECLARE, ACHOICE( ), ACOPY( ), ADEL( ), ADIR( ), AFIELDS( ), AFILL( ), AINS( ), ASORT( ), LEN( )

Função: ASORT( ) – Classificação de um Vetor:

Forma:

ASORT(<vetor>, [<exp.numérica 1>[,<exp.numérica 2>]])

Propósito:

Ordenar os conteúdos dos elementos de um vetor em ordem ascendente. Disponível a partir da versão Summer 87.

Argumentos:

<*vetor*>: é o nome do vetor a ser ordenado.
<*exp.numérica 1*>: é o número do primeiro elemento a participar da ordenação. Se omitido será assumido como sendo igual a 1.
<*exp.numérica 2*>: é o número de elementos a serem ordenados, contados a partir do elemento definido pela <expressão numérica 1>. Se for omitido, todos os elementos do vetor, a partir do elemento definido pela <expressão numérica 1> serão ordenados.

Utilização:

A função ASORT( ) é um meio simples e rápido de se ordenar os elementos de um vetor em ordem ascendente.
Todos os elementos do vetor a serem ordenados devem obrigatoriamente ser do mesmo tipo de dado, isto é, ou numéricos, ou caracteres, ou data. Não poderá haver mistura de tipos diferentes de dados para se realizar a ordenação.

Biblioteca:

EXTEND.LIB

Exemplo:

```
DECLARE ordenado[4]
ordenado[1]="A"
ordenado[2]="D"
ordenado[3]="C"
ordenado[4]="B"
ASORT(ordenado)
FOR i=1 TO 4
    ? ordenado[i]
NEXT

* Retorna: A
           B
           C
           D
```

Veja Também:

DECLARE, ACHOICE( ), ACOPY( ), ADEL( ), ADIR( ), AFIELDS( ), AFILL( ), AINS( ), ASCAN( ), LEN( )

6.5. FUNÇÕES PARA CONTROLE DO AMBIENTE

As funções para controle do ambiente de operação do Clipper retornam diversas informações que permitem o controle de dispositivos (impressora, vídeo, drives, etc), fluxo de programaçãqo, e outras. São muito importantes para a eficiência e sofisticação das aplicações desenvolvidas.

Função: COL( ) – Coluna:

Forma:

COL( )

Propósito:

Retorna o número da coluna do vídeo onde está posiconado atualmente o cursor.

Argumentos:

Nenhum

Utilização:

A função COL( ) facilita o desenvolvimento de rotinas para a apresentação de dados na tela, principalmente quando é necessário o controle e a mudança sucessiva de colunas.
A função COL( ) retorna sempre um valor numérico entre 0 e 79 (número das colunas do vídeo) que poderá ser armazenado em uma variável.
COL( ) pode ser utilizada como endereçamento relativo. Por exemplo:

@ 05,COL( )+8 SAY "Teste"

Posiciona o cursor na linha 5, oito colunas para a direita de sua posição atual.
A função COL( ) (coluna), utilizada em conjunto com a função ROW( ) (linha) permite o estabelecimento de coordenadas relativas do vídeo, a serem utilizadas com todas as variações do comando arrôba "@".

Biblioteca:

CLIPPER.LIB

Exemplos:

* Para finalizar um DO WHILE, quando se atingir a coluna 75:

```
DO WHILE COL()<75
   @ 05,COL()+1 SAY "*"
ENDDO
```

* Para mover o cursor relativamente através das colunas pode ser utilizado:

```
@ 02,COL()   SAY "Titulo 1"
@ 02,COL()+3 SAY "Titulo 2"
@ 02,COL()+3 SAY "Titulo 3"
@ 02,COL()+5 SAY "Titulo 4"
```

Cada título ficará separado por três colunas em branco, sendo que entre o Título 3 e o Título 4 haverão 5 colunas em branco.

Veja Também:

@...SAY...GET, @...BOX, @...CLEAR...TO, @...TO, ROW( ), PROW( ), PCOL( )

Função: DISKSPACE( ) — Espaço livre no Disco:

Forma:

DISKSPACE([<exp.numérica >])

Propósito:

Retorna um número inteiro representando o número de bytes livres no disco definido pela <expressão numérica>.

Argumentos:

<expressão numérica>: é o número do acionador de discos (drive) a ser pesquisado. O drive A: é o número um, o B: é o número dois, o C: é o número três, e assim por diante. Se este argumento não for especificado será assumido o drive corrente ("default").

Utilização:

Utiliza-se a função DISKPACE( ), entre outras aplicações, em conjunto com as funções RECSIZE( ), LASTREC( ) ou RECCOUNT( ) e HEADER( ), em rotinas para fazer Back-Up's (cópias de segurança) dos arquivos de dados (.dbf) em disquetes. Através delas é possível verificar quantos registros de um arquivo poderão ser copiados para um disquete e quanto espaço disponível ainda há no disquete.
Nas versões anteriores à Summer 87, a função DISKSPACE( ) encontra-se no módulo-objeto auxiliar EXTENDC.obj. Para utilizá-la este módulo deve ser incluído na lista dos arquivos-objeto que devem ser link-editados. Também deve ser utilizado o comando EXTERNAL, no módulo principal do sistema, para declarar a função DISKSPACE( ) como uma função externa ao compilador.

```
EXTERNAL DISKSPACE( )
```

Na versão Summer 87, a função DISKSPACE( ) faz parte da biblioteca EXTEND.lib, que deve estar presente na lista de bibliotecas passada ao link-editor.

Biblioteca:

EXTEND.LIB ou
EXTENDC.obj (nas versões anteriores à Summer 87)

Exemplo:

* Abaixo encontra-se um exemplo de uma rotina simples para se fazer back up em disquetes. A cada disquete que for completado (não possuir mais espaço disponível) a rotina solicicitará um novo.

```
USE Arquivo
disc=1
DO WHILE .NOT. EOF()
    @ 23,20 SAY "Coloque o disquete No."+STR(i,2)
    @ 23,42 SAY "no Drive B:"
    @ 24,26 SAY "Tecle (Enter) quando pronto..."
    WAIT ""
    COPY NEXT (DISKSPACE(2)-HEADER())/RECSIZE() TO B:Backup
    SKIP
    disc=disc+1
ENDDO
CLEAR
@ 23,33 SAY "Arquivo Salvo !"
USE
RETURN
```

Veja Também:

HEADER( ), RECSIZE( ), LASTREC( ), RECCONT( ), LUPDATE( )

Função: DOSERROR( ) – Erro do DOS:

Forma:

DOSERROR( )

Propósito:

Determina o número do último erro de DOS ocorrido, retornando-o como um valor numérico.
Disponível a partir da versão Summer 87.

Argumentos:

Nenhum

utilização:

A função DOSERROR( ) deve ser utilizada para a construção de rotinas de recuperação de erros, ocorridos durante o processamento da aplicação desenvolvida em Clipper. Ao ocorrer um erro do DOS, como, por exemplo, "Arquivo não encontrado", que possui o código de erro 2, o programa poderá ser continuado e as providências necessárias tomadas.
Esta função, juntamente com outras, permite que você construa sua própria rotina de recuperação de erros de execução ("run time errors"). Com esse propósito, a versão Summer 87 vem acompanhada de um exemplo através das rotinas ERRORSYS.prg e ALTERROR.prg, que são documentadas pelo arquivo ERROR.doc. Caso você queira construir sua própria rotina de recuperação de erros, recomendamos que este exemplo seja cuidadosamente estudado.

Biblioteca:

CLIPPER.LIB

Veja Também:

ERRORLEVEL( ), FERROR( ), NETERR( )

Função: EMPTY( ) — Vazio:

Forma:

EMPTY(<expressão>)

Propósito:

Retorna o valor verdadeiro (.T.) se a <expressão> especificada resultar em um valor vazio, ou falso (.F.) caso contrário.

Argumentos:

<expressão>: é uma expressão de qualquer tipo de dado a ser verificada.

Utilização:

Utiliza-se a função EMPTY( ) para testar se uma expressão caractere, numérica, data ou lógica é vazia.

A função EMPTY( ) retornará verdadeiro (.T.) se a expressão ou conteúdo de uma variável ou campo for vazia, de acordo com os seguintes critérios:

- Caractere: uma cadeia nula ou somente brancos (espaços em branco);
- Numérica: igual a zero (0);
- Data: uma data em branco: CTOD(SPACE(8)) ou nula;
- Lógica: falsa (.F.).

Biblioteca:

CLIPPER.LIB

Exemplo:

```
vnome=SPACE(40)
@ 12,10 SAY "Digite seu Nome: " GET vnome PICTURE "@!"
READ
IF EMPTY(vnome)     && se o nome for deixado em branco o
   RETURN           && programa é finalizado.
ENDIF
@ 14,10 SAY "Seu nome é "+vnome
```

Função: ERRORLEVEL( ) — Nível de Erro do Dos:

Forma:

ERRORLEVEL([<expressão numérica>])

Propósito:

Retorna o nível atual de erros definido para o DOS e opcionalmente pode definir um novo valor a substituir o atual.
Disponível a partir da versão Summer 87.

Argumentos:

<*expressão numérica*>: define, opcionalmente, o novo nível de erro a ser estabelecido para o DOS. Pode variar entre 0 e 255.

Utilização:

A função ERRORLEVEL( ) é utilizada em rotinas para a recuperação de erros onde pode-se definir o nível de erros a serem reportados pelo DOS.

Cada rotina de erro deverá tratar um determinado nível de erro do DOS. Esta função, juntamente com outras, permite que você construa sua própria rotina de recuperação de erros de execução ("run time errors"). Com esse propósito, a versão Summer 87 vem acompanhada de um exemplo através das rotinas ERRORSYS.prg e ALTERROR.prg, que são documentadas pelo arquivo ERROR.doc. Caso você queira construir sua própria rotina de recuperação de erros, recomendamos que este exemplo seja cuidadosamente estudado.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
niverr = ERRORLEVEL( )
* armazena em niverr o nível atual de erros do DOS

ERRORLEVEL(1)
* define como 1 o atual nível de erros do DOS
```

Função: FILE( ) — Arquivo:

Forma:

FILE(<expressão caractere>)

Propósito:

Retorna o valor lógico verdadeiro (.T.) se o arquivo especificado existe no diretório padrão do Clipper ou dentro da rota de diretórios definida pelo comando SET PATH, caso contrário retorna falso (.F.).

Argumentos:

<expressão caractere>: define o nome e a extensão do arquivo a ser localizado.

Utilização:

Utiliza-se a função FILE( ) para verificar se um arquivo existe num determinado diretório. Se o arquivo existir, a função FILE( ) retorna .T. (verdade), caso contráro retorna .F. (falso).

Esta função é muito útil para verificar a existência, no drive e diretório especificados, de todos os arquivos necessários à uma aplicação, ou se não existem arquivos com nomes duplicados em diferentes diretórios.
Se o drive e o diretório não forem explicitamente informados o Clipper irá pesquisar o arquivo no diretório padrão. Caso o comando SET PATH tenha sido utilizado, o arquivo será pesquisado também na rota de diretórios definida.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
IF FILE("C:Mala.dbf") .AND. FILE("C:Indcod.ntx")
   USE Mala.dbf INDEX Indcod
ELSE
   ? "Arquivos não encontrados no drive !"
   RETURN
ENDIF
```

Veja Também:

SET DEFAULT, SET PATH

Função: GETE( ) – Variáveis Ambientais do DOS:

Forma:

GETE(<expressão caractere>)

Propósito:

Retorna a cadeia de caracteres contida na variável ambiental do Sistema Operacional definida pelo seu nome na <expressão caractere>.

Argumentos:

<expressão caractere>: define o nome da variável ambiental do DOS requerida.

Utilização:

A função GETE( ) permite ao programador ter acesso às variáveis ambientais do Sistema Operacional.

GETE( ) localiza a variável ambiental do DOS especificada e retorna o seu conteúdo na forma de uma cadeia de caracteres.
Se o nome da variável, definido pela <expressão caractere>, não for encontrado, será retornada uma cadeia de caracteres nula.
Consulte o manual do DOS para maiores informações sobre os comandos SET e as variáveis ambientais.
Nas versões anteriores à Summer 87, a função GETE( ) encontra-se no módulo-objeto auxiliar EXTENDC.obj. Para utilizá-la nos seus sistemas, você deverá declará-la externa, através do comando EXTERNAL, no módulo principal, e incluir o arquivo EXTENDC.obj na lista de módulos-objeto passada ao link-editor.

```
EXTERNAL GETE( )
```

Biblioteca:

EXTEND.LIB
EXTENDC.obj (nas versões anteriores à Summer 87)

Exemplo:

```
* Para determinar os conteúdos do comando SET do DOS pode-se utilizar:

? GETE("PATH") && retorna "C:\VIDAL\CLIPPER"
? GETE("COMSPEC") && retorna "C:\command.com
```

Veja Também:

DISKSPACE( ), HEADER( ), RECSIZE( ), LUPDATE( ), SET PATH

Função: ISCOLOR( ) – Vídeo Colorido:

Forma:

ISCOLOR( )

Propósito:

Retorna verdadeiro (.T.) se uma placa adapatadora para monitor colorido e gráfico ("color graphics adapter card") está instalada no equipamento.

Argumentos:

Nenhum

Utilização:

Utiliza-se a função ISCOLOR( ) para testar se o equipamento onde se executará o programa ou aplicação desenvolvida em Clipper possui ou não placa adaptadora para monitor colorido e gráfico instalada. Se possuir ISCOLOR( ) retorna verdadeiro (.T.), caso contrário retorna falso (.F.).
Se o equipamento possuir uma placa para vídeo colorido e gráfico, pode-se fazer uso de cores ou de tons (em monitores monocromáticos), através do comando SET COLOR TO, e de caracteres gráficos através da função CHR( ).

Biblioteca:

CLIPPER.LIB

Exemplo:

```
IF ISCOLOR( )
  SET COLOR TO R/B,G/GR  && vermelho/azul,verde/marrom
ELSE
  SET COLOR TO W/N,N/W  && branco/preto,preto/branco
ENDIF
```

Veja Também:

SET COLOR TO

Função: ISPRINTER( ) – Verifica Impressora:

Forma:

ISPRINTER( )

Propósito:

Retorna verdadeiro (.T.) se a impressora conectada ao computador, através da porta paralela LPT1:, está em linha e pronta para impressão.

Argumentos:

Nenhum

Utilização:

Utiliza-se a função ISPRINTER( ) para verificar se a impressora conectada ao computador, através da porta paralela, está em linha (on-line) e pronta para impressão.
Se a impressora estiver em linha e pronta para impressão ISPRINTER( ) retornará verdadeiro (.T.), caso contrário falso (.F.).
Antes de enviar algo para a impressora é conveniente, portanto, testá-la com a função ISPRINTER( ), só enviando se ISPRINTER( ) retornar verdadeiro (.T.)
Nas versões anteriores à Summer 87, a função ISPRINTER( ) está contida no módulo-objeto auxiliar EXTENDA.OBJ. Para utilizá-la, deve-se declará-la como externa, através do comando EXTERNAL ISPRINTER( ), E incluir o arquivo EXTENDA.OBJ na lista de módulos objeto passada ao link-editor.

Biblioteca:

EXTEND.LIB
EXTENDA.obj (nas versões anteriores à Summer 87)

Exemplo:

```
USE Mala INDEX Indnome
EXTERNAL ISPRINTER()
DO WHILE .T.
   IF ISPRINTER()
      LIST nome,telefone TO PRINT
      USE
      RETURN
   ELSE
      @ 12,25 SAY "A Impressora Não está Ativa !
      @ 14,23 SAY "Verifique e pressione uma tecla..."
      WAIT ""
   ENDIF
ENDDO
USE
RETURN
```

Veja Também:

SET PRINT, SET DEVICE

Função: MEMORY( ) – Memória disponível:

Forma:

MEMORY(<expressão numérica>)/MEMORY(0)

Propósito:

Retorna a quantidade de memória disponível no equipamento em Kbytes.

Argumentos:

<expressão numérica>: deverá sempre retornar o valor zero (0).

Utilização:

A função MEMORY(0) retorna, em Kbytes, o espaço disponível, na memória do equipamento, para a manipulação de dados: variáveis, vetores, arquivos abertos etc.

Biblioteca:

CLIPPER.LIB

Veja Também:

Seção 8.3.2 – Utilização da Memória

Função: NETERR( ) – Erro na Rede:

Forma:

NETERR( )

Propósito:

Determina se os comandos USE, USE...EXCLUSIVE ou APPEND BLANK não puderam ser executados em um ambiente multiusuário de rede. Retorna verdadeiro (.T.) se houve falha e falso (.F.) caso tenham sido bem sucedidos.

Argumentos:

Nenhum

Utilização:

A função NETERR( ) retorna verdadeiro (.T.) se os seguintes comandos não puderam ser executados quando se está utilizando o sistema em um ambiente multiusuário.

Comando	Causa da Falha
USE	USE EXCLUSIVE executado por outro usuário.
USE...EXCLUSIVE	USE EXCLUSIVE ou USE executado por outro usuário.
APPEND BLANK	FLOCK( ) executado por outro usuário ou outro APPEND BLANK simultâneo.

Biblioteca:

CLIPPER.LIB

Exemplo

```
USE Mala
IF NETERR()
   ? "Arquivo em uso exclusivo por outro usuário !!"
   WAIT "Pressione qualquer tecla..."
   RETURN
ENDIF
```

Veja Também:

USE, USE...EXLUSIVE, APPEND BLANK, FLOCK( ), RLOCK( )

Função: NETNAME( ) – Nome do Usuário:

Forma:

NETNAME( )

Propósito:

Retorna a identificação do usuário atual através de uma cadeia de caracteres.

Argumentos:

Nenhum

Utilização:

A função NETNAME( ) retorna a identificação do usuário ou da estação de trabalho atual através de uma cadeia de 15 caracteres. Se a identificação do usuário não foi definida ou se não se estiver utilizando uma rede compatível como a "IBM-PC Network", será retornado o caractere nulo ("").

Biblioteca:

CLIPPER.LIB

Exemplo:

```
? NETNAME( ) && retorna o nome da estação de trabalho
```

Veja Também:

NETERR( )

Função: PCOL( ) – Coluna da Impressora:

Forma:

PCOL( )

Propósito:

Retorna o valor numérico representativo da coluna onde está atualmente posicionada a cabeça de impressão da impressora, ou seja, a próxima coluna a ser impressa.

Argumentos:

Nenhum

Utilização:

A função PCOL( ) retorna um valor numérico indicando a próxima coluna da impressora a ser impressa. O valor retonado por PCOL( ) pode ser normalmente armazenado em uma variável numérica.
Uma das maiores vantagens da função PCOL( ) é permitir endereçamento relativo na impressora. Por exemplo, @ 05,PCOL( )+5 posiciona a impressora cinco colunas além de sua posição atual.
O comando EJECT (mudança de página na impressora) retorna o valor de PCOL( ) para zero, isto é, reposiciona a impressora no início de uma nova página (lógica) do formulário. Caso seja necessário alterar o valor de PCOL( ) sem executar o comando EJECT (que causa uma ejeção de página física na impressora) pode ser utilizada a função SETPRC( ).
Utiliza-se a função PCOL( ) para a formatação de relatórios através do comando arrôba @...SAY. PCOL( ) pode indicar, relativamente, a coluna da impressora onde os dados deverão ser impressos. Se usada em conjunto com a função PROW( ) (linha da impressora), podem ser indicadas as cordenadas (linha e coluna) da onde os dados deverão ser impressos.
O número representativo da coluna da impressora, para a maioria das impressoras nacionais, pode variar de 0 (zero) a 240, de acordo com a compactação de caracteres utilizada. Consulte o manual da impressora acoplada ao seu equipamento para verificar a quantidade máxima de colunas que podem ser impressas e quais as configurações possíveis. Utilize a funçào CHR( ) para enviar códigos de controle que reconfiguração à impressora. Por exemplo, o código de controle CHR(15) na impressora Emilia PC permite a obtenção de 220 colunas de impressão, enquanto que o CHR(18) faz voltar ao normal, isto é, 132 colunas.

Bibliteca:

CLIPPER.LIB

Exemplos:

```
SET DEVICE TO PRINT
@ 01,PCOL()+10 SAY "Clipper   Compilador do dBASE III"
* Irá imprimir na linha 1 da impressora, 10 colunas à
* direita da coluna atual.
SET DEVICE TO SCREEN

x = PCOL()  && armazena em x o valor coluna da
            && impressora
```

* Como exemplo de listagem na impressora, considere o arquivo Mala.dbf que possui os seguintes campos:

nome — nome do destinatário, com 40 caracteres
ende — endereço, com 50 caracteres
cep — cep, com 5 caracteres
cid — cidade, com 20 caracteres
est — estado, com 2 caracteres

```
USE Mala INDEX Indnome
EJECT
SET DEVICE TO PRINT
DO WHILE .NOT. EOF()
   @ PROW(), 01 SAY "NOME"
   @ PROW(), 43 SAY "ENDERECO"
   @ PROW(), 95 SAY "CEP"
   @ PROW(),102 SAY "CIDADE"
   @ PROW(),122 SAY "ESTADO"
   @ PROW()+1,01 SAY REPLICATE("=",132)
   DO WHILE PROW()<60 .AND. .NOT. EOF()
      @ PROW()+1,01 SAY nome
      @ PROW(),PCOL()+2 SAY ende
      @ PROW(),PCOL()+2 SAY cep
      @ PROW(),PCOL()+2 SAY cid
      @ PROW(),PCOL()+2 SAY est
      SKIP
   ENDDO
   EJECT
ENDDO
SET DEVICE TO SCREEN
USE
RETURN
```

Veja Também:

PROW( ), SETPRC( ), COL( ), ROW( ), SET DEVICE, @...SAY, CHR( )

Função: PCOUNT( ) – Contagem de Parâmetros:

Forma:

PCOUNT( )

Propósito:

Retorna o número de parâmetros reais que foram passados a partir da linha de comando do Sistema Operacional, ou de um programa para outro.

Argumentos:

Nenhum

Utilização:

A função PCOUNT( ) retorna o número de parâmetros reais (realmente passados) que foram enviados de um programa para a rotina ou função-de-usuário chamada. Se nenhum parâmetro foi passado, PCOUNT( ) retorna zero.
Utiliza-se a função PCOUNT( ), para determinar-se quantos parâmetros estão sendo passados de um programa para uma rotina ou para uma função-de-usuário, com o intuito de verificar se o número mínimo de parâmetros esperado foi efetivamente passado.
O valor resultante de PCOUNT( ) pode ser normalmente armazenado em uma variável.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
* Programa para abrir um arquivo:
PARAMETERS arq
IF PCOUNT()=0
   ACCEPT "Digite o nome do arquivo:" TO arq
ENDIF
arq=arq+".dbf"
IF FILE(arq)
   USE &arq
ELSE
   ? "Arquivo nao encontrado !"
ENDIF
```

Veja Também:

PARAMETERS, PROCEDURE, DO...WITH, FUNCTION

Função: PROCLINE( ) – Linha da Rotina (procedure):

Forma:

PROCLINE( )

Propósito:

Retorna o número da linha da rotina ou programa que atualmente está sendo executada, de acordo com a numeração de linhas de seu código-fonte, numeradas a partir de seu início.

Argumentos:

Nenhum

Utilização:

Utiliza-se esta função para verificar qual linha do programa está sendo executada. Sua aplicação principal refere-se à correção de erros do programa (depuração ou "debug"), pois é possível saber em que linha é efetuada cada operação.
PROCLINE( ) também pode ser útil para tomadas de decisão em programas muito parametrizados, como o HELP.prg (veja a "facilidade" HELP do Clipper no Cap. 7).
Entretanto, se o programa tiver sido compilado sem incluir o número das linhas (opção –l do compilador do Clipper), o número de linha retornado pela função PROCLINE( ) é imprevisível e não terá valor real.

Biblioteca:

CLIPPER.LIB

Exemplo:

* A função PROCLINE( ) pode ser incluída em diversos pontos do programa para mostrar seu fluxo de execução (qual linha está sendo executada a cada instante):

```
USE Mala INDEX Indnome
? PROCLINE()  && retorna o número 2
LIST
? PROCLINE()  && retorna o número 4

? PROCLINE(),"variavel = ", var
```

* mostra o valor que possui a variável "var" em uma
* determinada linha do programa.

Veja Também:

PROCNAME( )

Função: PROCNAME( ) – Nome da Rotina (procedure):

Forma:

PROCNAME( )

Propósito:

Retorna o nome da rotina ou programa que está atualmente sendo executado através de uma cadeia de caracteres.

Argumentos:

Nenhum

Utilização:

Permite identificar o nome do programa que está sendo executado em um determinado instante. É uma função útil para efeito de correções nos programas ou para tomada de decisões em sistemas muito parametrizados, como o HELP.prg (veja "A Facilidade" HELP do Clipper" no Cap. 7).

Biblioteca:

CLIPPER.LIB

Exemplo:

? "O programa que está sendo executado é: ",PROCNAME( )

Veja Também:

PROCLINE( )

Função: PROW( ) – Linha de Impressão:

Forma:

PROW( )

Propósito:

Retorna um valor numérico representativo da linha (lógica) onde está atualmente posicionada a cabeça de impressão da impressora, em relação ao topo da página do formulário.

Argumentos:

Nenhum

Utilização:

A função PROW( ) retorna o número da linha onde está atualmente posicionada a cabeça de impressão da impressora, a partir do topo (lógico) do formulário.
A principal utilização da função PROW( ) é a formatação de relatórios na impressora através do comando @...SAY. A função PROW( ) pode indicar, relativamente, a linha da impressora onde os dados deverão ser impressos. Se usada em conjunto com a função PCOL( ) (coluna da impressora), podem ser indicadas as coordenadas (linha e coluna) onde os dados deverão ser impressos.
O comando EJECT retorna o valor de PROW( ) a zero (topo da página), uma vez que EJECT ejeta uma página. Para alterar o valor de PROW( ) sem executar um comando EJECT utiliza-se a função SETPRC( ).
O valor de PROW( ) pode ser normalmente armazenado em uma variável.
O número representativo da linha da impressora, varia de acordo com o número de linhas do formulário que está sendo utilizado. Para o formulário padrão de 11 polegadas, pode variar de 0 (zero) a 66 ou 88, de acordo com o entrelinhamento que estiver sendo utilizado na impressora (6 ou 8 linhas por polegada). Consulte o manual da impressora acoplada ao seu equipamento para verificar os entrelinhamentos disponíveis. Utilize a funçào CHR( ) para enviar códigos de controle para reconfigurar a impressora. Por exemplo, o código de controle CHR(27)+"0" na impressora Emilia PC permite o entrelinhamento a 8 linhas por polegada, enquanto que o CHR(27)+"2" faz voltar ao normal, isto é, 6 linhas por polegada.
Uma das maiores vantagens da função PROW( ) é permitir o endereçamento relativo na impressora. Por exemplo, @ PROW( )+5,PCOL( ) posiciona a cabeça de impressão cinco linhas abaixo de sua posição atual.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
SET DEVICE TO PRINT
@ PROW()+10,20 SAY "Clipper   Compilador do dBASE III"
* Irá imprimir na coluna 20 da impressora, 10 linhas
* abaixo da linha atual.
SET DEVICE TO SCREEN

x = PROW()  && armazena em x o valor linha da
            && impressora

* Como  exemplo de  listagem na impressora, considere o
arquivo Mala.dbf que possui os seguintes campos:

       nome    nome do destinatário, com 40 caracteres
       ende    endereço, com 50 caracteres
       cep     cep, com 5 caracteres
       cid     cidade, com 20 caracteres
       est     estado, com 2 caracteres

USE Mala INDEX Indnome
EJECT
SET DEVICE TO PRINT
DO WHILE .NOT. EOF()
   @ PROW(), 01 SAY "NOME"
   @ PROW(), 43 SAY "ENDERECO"
   @ PROW(), 95 SAY "CEP"
   @ PROW(),102 SAY "CIDADE"
   @ PROW(),122 SAY "ESTADO"
   @ PROW()+1,01 SAY REPLICATE("=",132)
   * Incremento de uma linha na impressora
   DO WHILE PROW()<60 .AND. .NOT. EOF()
      @ PROW()+1,01 SAY nome
      * Incremento de uma linha na impressora
      @ PROW(),PCOL()+2 SAY ende
      @ PROW(),PCOL()+2 SAY cep
      @ PROW(),PCOL()+2 SAY cid
      @ PROW(),PCOL()+2 SAY est
      SKIP
   ENDDO
   EJECT
ENDDO
SET DEVICE TO SCREEN
USE
RETURN
```

Veja Também:

SET PRINT, SET DEVICE, @...SAY, PCOL( ), COL( ), ROW( ), CHR( ), SETPRC( ), EJECT

Função: READEXIT( ) – Saída do READ:

Forma:

READEXIT([<expressão lógica>])

Propósito:

Habilita ou desabilita as teclas <seta para cima> e <seta para baixo> como teclas de finalização de um comando READ, retornando verdadeiro (.T. – teclas habilitadas) ou falso (.F. – teclas desabilitadas).
Disponível a partir da versão Summer 87.

Argumentos:

<expressão lógica>: se for verdadeira (.T.) habilita as setas para cima e para baixo como finalizadoras de um READ; caso contrário as desabilita. O padrão é falso (.F.), ou seja, teclas desabilitadas.

Utilização:

A função READEXIT( ) é utilizada para se determinar ou não se as teclas <seta para cima> ou <seta para baixo> poderão ser utilizadas para finalizar um READ.

Biblioteca:

CLIPPER.LIB

Veja Também:

@...SAY...GET, READ, READINSERT( )

Função: READINSERT( ) – Indicador de Modo:

Forma:

READINSERT([<expressão lógica>])

Propósito:

Retorna se o modo de inserção de caracteres na edição está ligado (.T. – verdadeiro) ou desligado (.F. – falso).
Disponível a partir da versão Summer 87.

Argumentos:

<*expressão lógica*>: habilita ou desabilita o modo de inserção de caracteres. Se for verdadeira (.T.), o modo de inserção é ligado; se for falsa (.F.), o modo de inserção é desligado. O padrão é desligado, ou seja falso (.F.).

Utilização:

A função READINSERT( ), se não for especificado nenhum argumento, retorna verdadeiro (.T.) se o modo de inserção está ligado ou falso (.F.) se está desligado.
Se for especificado um argumento verdadeiro (.T.) (através da <exp.lógica>, o modo de inserção será ligado; se for especificado falso (.F.), o modo de inserção será desligado.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
modins=READINSERT(.T.) && liga o modo de inserção
REPLACE obs WITH MEMOEDIT(obs,6,0,23,79)
* entra no modo de edição da função MEMOEDIT( ) com o
* modo de inserção ligado
READINSERT(modins)
* retorna ao valor original do modo de inserção,
* ligado ou desligado
```

Veja Também:

READ, @...SAY...GET, MEMOEDIT( ), READEXIT( )

Função: RESTSCREEN( ) – Recupera Área da Tela:

Forma:

RESTSCREEN(<exp.num.1>,<exp.num.2>,<exp.num.3>,<exp.num.4>,<exp.caractere>)

Propósito:

Reapresenta uma região da tela previamente salva em uma área da tela especificada.
Disponível a partir da versão Summer 87.

Argumentos:

<exp.num.1 a exp.num.4>: definem as coordenadas do canto superior esquerdo e canto inferior direito da tela, onde a região da tela salva deverá ser reapresentada.
<expressão caractere>: é uma cadeia de caracteres contendo os dados (a tela) a serem reapresentados, normalmente definida pela função SAVESCREEN( ) e armazenada em uma variável.

Utilização:

A função RESTSCREEN( ) é utilizada para reapresentar uma região da tela que tenha sido previamente salva através da função SAVESCREEN( ). A posição da tela onde a região salva deverá ser reapresentada poderá ser a mesma ou diferente da originalmente salva. Se for especificada uma posição diferente, a nova região deverá possuir o mesmo tamanho (área) da região salva; caso contrário, o resultado será destrutivo.
Não se pode utilizar o comando RESTORE SCREEN para reapresentar regiões de tela salvas através da função SAVESCREEN( ). O resultado também será destrutivo.

Biblioteca:

EXTEND.LIB

Exemplo:

```
tela1 = SAVESCREEN(6,0,23,79)
* a região foi salva na variável tela1
```

```
<apresentação de outra tela na região salva>
        .
        .
RESTSCREEN(6,0,23,79,tela1)
* reapresentação da região salva em tela1
```

Veja Também:

RESTORE SCREEN, SAVE SCREEN, SAVESCREEN( )

Função: ROW( ) – Linha do Vídeo:

Forma:

ROW( )

Propósito:

Retorna o número da linha do vídeo onde está atualmente posicionado o cursor.

Argumentos:

Nenhum

Utilização:

A função ROW( ) retorna um número entre 0 e 24 indicando a linha da tela onde está posicionado o cursor. Zero é a primeira linha da tela e 24 a última.
O comando CLEAR, que limpa a tela, retorna o valor da função ROW( ) a zero e, após a execução de um comando READ, o valor de ROW( ) passa a valer 23.
A função ROW( ) facilita o desenvolvimento de rotinas para a apresentação de dados na tela, principalmente quando é necessário controle e a mudança sucessiva de linhas.
Pode ser utilizado endereçamento relativo. Por exemplo:

```
@ ROW( )+8,35 SAY "Teste"
```

Posiciona o cursor na coluna 35, oito linhas abaixo de sua posição atual.
A função ROW( ), utilizada em conjunto com a função COL( ) (coluna do

cursor) permite o estabelecimento das coordenadas do vídeo para o comando arrôba @...SAY...GET.

Biblioteca:

CLIPPER.LIB

Exemplos:

* Para finalizar um DO WHILE, quando se atingir a linha 23:

```
DO WHILE ROW( )<23
    @ROW( )+1,40 SAY "*"
ENDDO
```

* Para mover o cursor relativamente através das linhas e colunas pode ser utilizado:

```
@ ROW( ),COL( )          SAY "Título 1"
@ ROW( )+02,COL( )+3     SAY "Título 2"
@ ROW( )+02,COL( )+3     SAY "Título 3"
@ ROW( )+02,COL( )+5     SAY "Título 4"
```

Cada título ficará separado por duas linhas e três colunas em branco, sendo que entre o Título 3 e o Título 4 haverão 5 colunas em branco.

Veja Também:

@...SAY...GET, COL( ), PROW( ), PCOL( )

Função: SAVESCREEN( ) – Salva Região da Tela:

Forma:

SAVESCREEN(<exp.num.1>,<exp.num.2>,<exp.num.3>,<exp.num.4>)

Propósito:

Salva uma região da tela especificada em uma variável de memória tipo caractere.
Disponível a partir da versão Summer 87.

Argumentos:

<exp.num.1 a exp.num.4>: definem as coordenadas do canto superior esquerdo e canto inferior direito da região da tela a ser salva.

Utilização:

A função SAVESCREEN( ) é utilizada para salvar uma região da tela em uma variável de memória. SAVESCREEN( ) retorna a região especificada da tela como uma cadeia de caracteres com até 4.000 bytes de comprimento. A região da tela salva poderá depois ser reapresentada através da função RESTSCREEN( ).
A utilização típica das funções SAVESCREEN( ) e RESTSCREEN( ) é permitir salvar e reapresentar regiões da tela onde são apresentados menus instantâneos ("pop-up menus").
Não se pode utilizar o comando SAVE SCREEN para salvar apenas uma região da tela.

Biblioteca:

EXTEND.LIB

Exemplo:

```
tela1 = SAVESCREEN(6,0,23,79)
* a região foi salva na variável tela1
          .
          .
<apresentação de outra tela na região salva>
          .
          .
RESTSCREEN(6,0,23,79,tela1)
* reapresentação da região salva em tela1
```

Veja Também:

RESTORE SCREEN, SAVE SCREEN, RESTSCREEN( ), ACHOICE( )

Função: SETCANCEL( ) – Liga ou Desliga <Alt–C>:

Forma:

SETCANCEL([<expressão lógica>])

Propósito:

Habilita ou desabilita a interrupção da execução do programa através das teclas <Alt-C> pressionadas simultaneamente.
Disponível a partir da versão Summer 87.

Argumentos:

<*expressão lógica*>: se for verdadeira (.T.) habilita as teclas <Alt-C>, se for falsa (.F.) desabilita-as. O padrão é (.T.) verdadeiro.

Utilização:

A função SETCANCEL( ) retorna o estado das teclas <Alt-C> se um argumento não for especificado (ligadas ou desligadas), e o estado anterior se um argumento for especificado.

Biblioteca:

CLIPPER.LIB

Veja Também:

ALTD( ), SET ESCAPE

Função: SETCOLOR( ) – Define cores:

Forma:

SETCOLOR([<expressão caractér>])

Propósito:

Retorna a definição de cores atual ou anterior e opcionalmente define as cores da próxima tela a ser construída.
Disponível a partir da versão Summer 87.

Argumentos:

<*expressão caractere*>: é uma cadeia de caracteres contendo as definições de vídeo normal, realçado, borda, fundo e neutro (de acordo com a sintaxe do comando SET COLOR), a serem utilizadas para apresentação dos próxi-

mos dados na tela. SETCOLOR( ) suporta apenas letras na definição das cores a serem utilizadas, números não são válidos.

Utilização:

A função SETCOLOR( ) retorna uma cadeia de caracteres representando a nova definição de cores estabelecida se a <expressão caractere> para definição de cores for especificada. Se a <expressão caractere> não for especificada SETCOLOR( ) retorna a atual definição de cores.
SETCOLOR( ) sem argumentos, ao contrário do comando SET COLOR TO não reestabelece as cores do vídeo para o padrão.
Veja o comando SET COLOR TO para uma explicação completa da forma de definição de cores do vídeo e das cores válidas.

Biblioteca:

EXTEND.LIB

Exemplo:

```
cores = SETCOLOR( )
* armazena a atual definição de cores na variável
* cores

SETCOLOR("W+/R,N/W,,,W+/R")
* define novas cores para o vídeo, equivalente a:
* SET COLOR TO W+/R,N/W,,,W+/R

corantiga = SETCOLOR("BR+/N,R+/N")
* armazena a cor que estava sendo utilizada na
* variável corantiga e define novas cores

SETCOLOR(corantiga)
* retorna as cores antigas
```

Veja Também:

SET COLOR TO, ISCOLOR( )

Função: SETPRC( ) — Define Posição de Impressão:

Forma:

SETPRC(<exp.numérica 1>,<exp.numérica 2>)

Propósito:

Redefine os valores internos das funções PROW( ) (linha) e PCOL( ) (coluna) de impressão para os novos valores especificados.

Argumentos:

<*exp.numérica 1*>: define a nova posição interna da linha da impressora (PROW( )).
<*exp.numérica 2*>: define a nova posição interna da coluna da impressora (PCOL( )).

Utilização:

A função SETPRC( ) pode ser extremamente útil para enviar códigos de configuração à impressora sem alterar a posição (linha e coluna) da cabeça de impressão, relativa ao controle interno do programa.
Após o envio de códigos de controle para a impressora, os valores de controle interno da posição da linha e coluna de impressão são alterados. Pode-se retorná-los ao valores originais através da função SETPRC( ).
Além disso, utilizando-se esta função, é possível inibir a mudança de página automática de algumas impressoras.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
SET PRINT ON
lin = PROW()
col = PCOL()
?? CHR(15)+chr(14)   && compacta e expande a impressão
SETPRC(lin,col)      && retorna à posição original
```

Veja Também:

SET PRINT, SET DEVICE TO PRINT, PROW( ), PCOL( )

Função: TYPE( ) — Tipo:

Forma:

TYPE(<expressão caractere>)

Propósito:

Avalia a <expressão caractere> especificada, e retorna uma letra maiúscula indicando o tipo de dado do resultado: caractere (C), numérico (N), data (D), lógico (L), um campo memo (M), um vetor (A), ou indefinido (U).

Argumentos:

<*expressão caractere*>: define uma cadeia de caracteres que resulta no dado a ser avaliado. Pode incluir o nome de um campo de arquivo (incluindo o alias), o nome de uma variável, ou uma expressão de qualquer tipo.

Utilização:

A função TYPE( ) determina o tipo de dado de uma expressão qualquer, retornando:

C	–	caractere
N	–	numérico
D	–	data
L	–	lógico
M	–	campo memo
A	–	vetor
U	–	indefinida
UE	–	erro de sintaxe
UI	–	erro indeterminado

Pode ser utilizada como um teste para verificar a existência de uma variável ou a validade de uma expressão.
A função TYPE( ), entretanto, não é capaz de avaliar macro-substituições (&) corretamente.
Os tipos UE, indicando um erro de sintaxe na expressão especificada e UI, indicando um erro que não pode ser determinado foram implementados a partir da versão Summer 87.

Biblioteca:

CLIPPER.LIB

Exemplos:

* Para avaliar se uma variável existe e qual é o seu tipo:

```
x = "Clipper"
y = 1000
DECLARE v[10]
d = DATE()

? TYPE("x") ou      && retorna C
? TYPE("Clipper")

? TYPE("y")         && retorna N
? TYPE("v")         && retorna A
? TYPE("d") ou      && retorna D
? TYPE("DATE()")
? TYPE("1+2=3")     && retorna L
? TYPE("w")         && retorna U (não definida)

? TYPE("x/y") && retorna U (caracter não pode ser
              && dividido por um número.

? TYPE("y/2") && retorna N, é uma expressão válida.

? TYPE([USU()]) && retorna UI se a função-de-usuário
                && USU() não tiver sido definida.
```

Nota: Colchetes são equivalentes a aspas como delimitadores.

```
? TYPE([SUBSTR(4,5)]) && retorna UE, erro de sintaxe
```

6.6. FUNÇÕES PARA CONTROLE DE ARQUIVOS DE DADOS

Função: ALIAS( ) – Alias:

Forma:

ALIAS(<exp.numérica>)

Propósito:

Retorna o nome do "alias" de uma determinada área de trabalho onde há um arquivo de dados aberto.

Argumentos:

<exp.numérica>: é o número da área de trabalho a ser retornado o alias.

Utilização:

A função ALIAS( ) é utilizada para se saber o nome do alias de uma determinada área de trabalho selecionada, onde um arquivo de dados está em uso.

A <expressão numérica> pode variar de 1 até 10 ou de 1 a 255 na verão Summer 87, indicando o número da área de trabalho que se deseja saber o alias. Se não for especificada, o alias da área de trabalho atualmente selecionada é retornado como resposta.
Se não houver um arquivo de dados em uso na área de trabalho especificada será retornado um alias nulo ("").

Biblioteca:

CLIPPER.LIB

Exemplos:

```
SELECT 1
USE SCMP INDEX SCMP1 ALIAS Produtos
SELECT 2
USE SCMH INDEX SCMH1 ALIAS Movimto
SELECT 3
USE SCMR INDEX SCMR1 ALIAS Receitas
? ALIAS(1)   && retorna 'Produtos'
? ALIAS(2)   && retorna "Movimto"
? ALIAS(3)   && retorna "Receitas
? ALIAS()    && retorna "Receitas" pois é a area atual
             && mente selecionada
```

Veja Também:

SELECT, USE, SELECT( )

Função: BOF( ) – Início do Arquivo:

Forma:

BOF( )

Propósito:

Retorna verdadeiro (.T.) quando for feita uma tentativa de mover o apontador de registros para antes do primeiro registro do arquivo de dados atualmente em uso.

Argumentos:

Nenhum

Utilização:

A função **BOF( )** ("beginning of file") é utilizada para verificar se, através da movimentação reversa em um arquivo (do seu fim para o início), o seu início foi atingido, retornando verdadeiro (.T.). Pode-se evitar, assim, executar o comando SKIP −1, quando o apontador de registros já estiver posicionado no primeiro registro lógico do arquivo em uso.
Se o arquivo não estiver índexado o primeiro registro será o de número 1. Caso contrário, o primeiro registro será o primeiro registro lógico (de acordo com a ordem) determinado pelo índice.
Uma aplicação típica da função BOF( ), por exemplo, é em rotinas de paginação de tela, que permitem que as páginas sejam avançadas ou retrocedidas através dos registros de um arquivo de dados. A função BOF( ) pode ser utilizada para verificar se o topo do arquivo foi atingido.
Se o arquivo de dados não contiver registros, ambas funções, BOF( ) e EOF( ) (fim-de-arquivo), retornarão verdadeiro (.T.).

Biblioteca:

CLIPPER.LIB

Exemplo:

```
USE Estoque
? BOF()         && retorna falso (.F.), pois o apontador
                está no primeiro registro.
USE Estoque
SKIP -1
? BOF()         && retorna verdadeiro (.T.), pois o
                && apontador foi para o início do arquivo

* Para percorrer o arquivo em ordem inversa e parar ao
atingir o seu início, pode-se usar:

USE Estoque
GO BOTTOM
DO WHILE .NOT. BOF()
   ? produto, saldo
   SKIP -1
ENDDO
USE
RETURN
```

Veja Também:

EOF( ), SKIP

Função: DBEDIT( ) – Edição de Arquivo de Dados:

Forma:

DBEDIT([<exp.num1>[,<exp.num2>[,<exp.num3>[,<exp.num4>]]]]
[,<vetor1>][,<exp.caractere>][,<vetor2>][,<vetor3>]
[,<vetor4>][,<vetor5>][,<vetor6>][,<vetor7>])

Propósito:

Apresenta e edita os registros de um ou mais arquivos de dados abertos em áreas de trabalho diferentes, através de uma janela a ser definida, de forma semelhante ao comando "BROWSE" do dBASE. Permite a visualização e edição em tela cheia ("full-screen") de um ou mais arquivos de dados. A aparência da tela, bem como muitas teclas de função podem ser customizadas de acordo com a coveniência da aplicação desenvolvida.
Nas versões anteriores do Clipper, a função DBEDIT( ), apesar de existir, é bem mais simples. Aqui a apresentaremos em sua forma mais sofisticada, como foi implementada na versão Summer 87.

Argumentos:

<exp.numérica 1> a <exp.numérica 4>: são respectivamente as coordenadas superior esquerda e inferior direita da janela a ser construída pela DBEDIT( ). Qualquer um ou todos estes argumentos podem ser especificados, os que não forem serão assumidos como sendo as extremidades da tela.
<vetor 1>: é um vetor contendo<expressões caractere> representando os nomes dos campos dos arquivos a serem editados ou expressões de qualquer tipo de dado envolvendo estes campos. Se este argumento não for especificado o Clipper assumirá todos os campos do arquivo aberto na área de trabalho atualmente selecionada.

<expressão caractere>: é o nome de uma função-de-usuário a ser executada quando uma tecla de exceção for pressionada. O nome da função deve ser especificado sem os parêntesis ou seus argumentos. o funcionamento da DBEDIT( )é afetado dependendo ou não da presença deste argumento.

<vetor 2>: é um vetor de <expressões caractere> a ser utilizado como máscara de formatação das colunas a serem editadas – PICTURE's. Se for especificada apenas uma <expressão caractere> ao invés de um vetor, todas as colunas serão editadas através da mesma máscara (PICTURE) de formatação.

<vetor 3>: é um vetor de <expressões caractere> que fornecerá o cabeçalho das colunas a serem editadas.

<*vetor 4*>: é um vetor de <expressões caractere> usado para desenhar linhas separando os cabeçalhos da área de edição dos campos/colunas de dados. Se for especificada apenas uma <expressão caractere> ao invés de um vetor, o mesmo caractere será utilizado para separar toda a linha.

<*vetor 5*>: é um vetor de <expressões caractere> usado para desenhar linhas separando as colunas editadas entre si. Se for especificada apenas uma < expressão caractere> ao invés de um vetor, o mesmo caractere será utilizado para separar todas as colunas.

<*vetor 6*>: é um vetor de <expressões caractere> usado para desenhar linhas separando os rodapés da área de edição dos campos/colunas de dados. Se for especificada apenas uma <expressão caractere> ao invés de um vetor, o mesmo caractere será utilizado para separar toda a linha do rodapé.

<*vetor 7*>: é um vetor de <expressões caractere> usado para apresentar rodapés nas colunas editadas. Para forçar um rodapé possuir mais de uma linha, pode ser incluído um ponto-e-vírgula onde se deseja que ocorra a quebra da linha. Se for especificada uma <expressão caractere> ao invés de um vetor, todos os rodapés terão o mesmo caractere especificado.

Nota: para obter um padrão (default) em um determinado argumento, enquanto definindo outros que vêm em seguida à este, deve ser passada uma definição ou um valor impróprio (tipo de dado incorreto, por exemplo). O valor nulo ("") não pode ser passado pois eliminará os cabeçalhos e os rodapés.

Utilização:

A função DBEDIT( ) é bastante poderosa. Ela permite a edição em tela-cheia de um ou mais arquivos de dados através de uma janela formatada. Vetores, definem os dados e campos a serem editados em forma tabular, os cabeçalhos e os rodapés das colunas de edição. Todas as teclas tradicionais de movimentação do cursor são aceitas pela DBEDIT( ), permitindo completa "navegação" pelos dados editados.
Se uma função-de-usuário for especificada, DBEDIT( ) a executará quando uma tecla de exceção for pressionada, passando à função o seu modo de operação atual ("status" atual) e o número do elemento do vetor de campos onde está posicionado o cursor. A função-de-usuário deverá, através do exame dos parâmetros recebidos e do valor da última tecla pressionada pelo usuário (utiliza-se a função LASTKEY( )), executar a ação apropriada e em seguida retornar um valor para DBEDIT( ). Esta, por sua vez, também executará uma determinada operação, de acordo com o valor recebido. Além disso, ainda é possível retornar uma tecla determinada para DBEDIT( ) através do comando KEYBOARD.

Os modos passados à função-de-usuário que identificam o estado ("status") da DBEDIT( ) são os seguintes:

Modo	Descrição
0	Neutro, qualquer movimento do cursor; nenhuma pendência a ser processada.
1	Tentativa de ultrapassar o início do arquivo.
2	Tentativa de ultrapassar o fim do arquivo.
3	O arquivo de dados editado está vazio.
4	Foi pressionada uma tecla de exceção.

Os valores que a função-de-usuário deve retornar à DBEDIT( ) para ser tomada uma ação são os seguintes:

Valor	Descrição
0	Finalizar a DBEDIT( ); fim da edição dos dados.
1	Continuar normalmente a DBEDIT( ).
2	Forçar que os dados sejam reapresentados, reconstruindo a janela de edição e, continuando a edição através da DBEDIT( ).

DBEDIT( ) não constitui um estado de espera e por esse motivo, rotinas não podem ser executadas através do comando SET KEY, enquanto os dados estiverem sendo editados através da DBEDIT( ), a menos que a função-de-usuário crie um estado de espera.
É possível intercalar várias funções DBEDIT( ), permitindo assim a apresentação simultânea ao usuário de múltiplas janelas de edição de dados na mesma tela, efeito semelhante à um "Windows".

Biblioteca:

EXTEND.LIB e DBU.LIB (versões anteriores à Summer 87)

Exemplo:

O exemplo a seguir mostra uma janela de edição (tipo "BROWSE") que inclui todos os campos do arquivo de mala-direta Mala.dbf. Ao se pressionar <Enter> com o cursor sobre um determinado campo DBEDIT( ) entrará em modo de edição neste campo. Pressionando <Esc> DBEDIT( ) será finalizada.

```
USE Mala
DBEDIT( )
```

Neste caso DBEDIT( ) é praticamente equivalente ao comando BROWSE do dBASE III.

Veja Também:

@...SAY...GET, READ, ACHOICE( )

Função: DBF( ) – Arquivo de Dados:

Forma:

DBF( )

Propósito:

Retorna o nome do arquivo de dados em uso (aberto) na área de trabalho atualmente selecionada.
Apenas para as versões Winter 85 e Autumn 86.

Argumentos:

Nenhum

Utilização:

A função DBF( ) faz parte das extensões escritas em Clipper; seu código-fonte encontra-se no arquivo EXTENDDB.prg e seu respectivo módulo-objeto no arquivo EXTENDDB.obj.
É simulada através da função ALIAS( ), que retorna o nome do "alias" da área de trabalho atualmente selecionada, onde está aberto o arquivo de dados. Esta simulação será válida quando não for atribuído um "alias" ao se abrir o arquivo, pois nesse caso o Clipper assume como "alias" o próprio nome do arquivo. Para incluí-la em suas aplicações o arquivo EXTENDDB.obj deve ser incluído na lista de módulos-objeto a serem encadeados pelo link-editor.
A função ALIAS( )/DBF( ) é útil por informar o nome ou "alias" do arquivo de dados que está em uso, especialmente nas aplicações onde a definição do arquivo a ser aberto é feita pelo próprio usuário do sistema.
Se não houver arquivo de dados aberto na área selecionada, a função ALIAS/( )/DBF( ) irá retornar uma cadeia de caracteres nula.

Biblioteca:

Módulo-fonte EXTENDDB.prg
Módulo-objeto EXTENDDB.obj

Exemplo:

```
CLEAR
ACCEPT "Digite o Nome do Arquivo Desejado:" TO arq
arq=arq+".dbf"
IF FILE(arq)
   USE &arq
ELSE
   ? "Arquivo Não Encontrado !
ENDIF
? DBF()    && retorna o nome do arquivo em uso.

USE C:\Clipper\Clientes
? DBF()    && retorna "Clientes
```

Veja Também:

USE, ALIAS( )

Função: DBFILTER( ) – Filtro Ativo:

Forma:

DBFILTER( )

Propósito:

Retorna a expressão do filtro de registros atualmente ativo (através do comando SET FILTER), na área de trabalho selecionada.
Disponível a partir da versão Summer 87.

Argumentos:

Nenhum

Utilização:

A função DBFILTER( ) retorna, como uma cadeia de caracteres, a condição do filtro estabelecido através do comando SET FILTER, na área de trabalho

atualmente selecionada. Se não houver filtro estabelecido DBFILTER( ) retorna uma cadeia nula ("").

Biblioteca:

CLIPPER..LIB

Exemplo:

```
USE Mala INDEX Indcod
SET FILTER TO estado="SP"
? DBFILTER()
* retorna [estado='SP']
```

Veja Também:

DBRELATION( ), DBRSELECT( ), SET FILTER

Função: DBRELATION( ) – Relação Ativa:

Forma:

DBRELATION(<expressão numérica>)

Propósito:

Retorna a expressão definida para o relacionamento de arquivos através do comando SET RELATION na área de trabalho selecionada.
Disponível a partir da versão Summer 87.

Argumentos:

<expressão numérica>: é a n-ésima posição do relacionamento a ser retornado, dentro da lista dos relacionamentos estabelecidos.

Utilização:

A função DBRELATION( ) retorna uma cadeia de caracteres contendo a n-ésima expressão de relacionamento definida pelo comando SET RELATION, de acordo com o valor da <expressão numérica> especificada: se for 1 será retornado o primeiro relacionamento estabelecido, se for 2 o segundo, e assim por diante.

Se na área de trabalho atualmente selecionada não houver relacionamento ativo DBRELATION( ) retorna uma cadeia de caracteres nula (“”).
DBRELATION( ) adequadamente utilizada com DBRSELECT( ) e DBFILTER( ) permite que se construa um sistema equivalente ao “VIEW” do dBASE III Plus ou ao utilitário DBU.exe do Clipper, cujo código-fonte acompanha a versão Summer 87.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
SELECT 3
USE Produtos INDEX Indprod ALIAS Prod
SELECT 2
USE Empresas INDEX Indemp ALIAS Empr
SELECT 1
USE Precos INDEX Indpre ALIAS Preco
SET RELATION TO codpro INTO Prod, TO codemp INTO Empr
? DBRELATION(1)
* Retorna "CODPRO"
? DBRELATION(2)
* Retorna "CODEMP"
```

Veja Também:

DBFILTER( ), DBRSELECT( ), SET RELATION e o utilitário DBU.exe no Cap. 9.

Função: DBRSELECT( ) – Área Relacionada:

Forma:

DBRSELECT(<expressão numérica>)

Propósito:

Determina a área de trabalho relacionada, através do comando SET RELATION, com a área de trabalho atual.
Disponível a partir da versão Summer 87.

Argumentos:

<expressão numérica>: é a n-ésima posição da lista de relacionamentos definida na área de trabalho atual através do comando SET RELATION.

Utilização:

A função DBRSELECT( ) retorna o número da área de trabalho relacionada, de acordo com o valor da <expressão numérica> especificada: se for igual a 1 será retornado o número da primeira área de trabalho relacionada, se for igual a 2 o número da segunda, e assim por diante.
Se não houver nenhuma relação estabelecida na área de trabalho selecionada DBRSELECT( ) retornará zero.
DBRSELECT( ) deve ser utilizada em conjunto com as funções DBRELATION( ) e DBFILTER( ) para criar um sistema equivalente ao "VIEW" do dBASE III Plus ou ao utilitário DBU.exe do Clipper, cujo código-fonte acompanha a versão Summer 87.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
SELECT 3
USE Produtos INDEX Indprod ALIAS Prod
SELECT 2
USE Empresas INDEX Indemp ALIAS Empr
SELECT 1
USE Precos INDEX Indpre ALIAS Preco
SET RELATION TO codpro INTO Prod, TO codemp INTO Empr
? DBRSELECT(1)
* Retorna 3
```

Veja Também:

SET RELATION, DBFILTER( ), DBRELATION( ) e o utilitário DBU.exe no capítulo 9.

Função: DELETED( ) – Deletado:

Forma:

DELETED( )

Propósito:

Identifica os registros que estão marcados para deleção (exclusão). Retorna verdadeiro (.T.) se o registro onde atualmente está posicionado o apontador de registros está marcado para eliminação (deleção); caso contrário, retorna falso (.F.).

Argumentos:

Nenhum

Utilização:

A função DELETED( ) é útil pois permite identificar os registros que estão marcados para deleção em um arquivo, e a partir disso processá-los de acordo com o desejado. Por exemplo fornecer um relatório, indicando quais os registros estão ativos e quais os deletados; ou evitar que registros deletados sejam processados.

Biblioteca:

CLIPPER.LIB

Exemplos:

* Para mostrar na tela todos os registros deletados de um determinado arquivo pode-se usar o comando DISPLAY:

DISPLAY FOR DELETED( )

* Para determinar se um registro está deletado:

```
USE Mala
DO WHILE .T.
   INPUT "Digite o número do Registro:" TO reg
   IF reg=0 .or. reg>LASTREC()
      EXIT
   ENDIF
  GOTO reg
  IF DELETED()
     ? "Registro Deletado"
  ELSE
     ? "Registro Não Deletado"
  ENDIF
ENDDO
USE
RETURN
```

Veja Também:

SET DELETED, DELETE, RECALL, PACK

Função: DESCEND( ) – Descendente:

Forma:

DESCEND(<expressão>)

Propósito:

Criar um índice e fazer uma pesquisa através do comando SEEK em ordem descendente.
Disponível a partir da versão Summer 87.

Argumentos:

<*expressão*>: é uma expressão de qualquer tipo de dado a ter seu valor invertido.

Utilização:

A função DESCEND( ) retorna o mesmo tipo de dado que o definido pela <expressão> em sua forma complementar (inverso).
DESCEND( ) é especialmente desenhada para indexar um arquivo em ordem inversa (descendente), através do comando INDEX e realizar pesquisas sobre este arquivo através do comando SEEK.

Biblioteca:

EXTEND.LIB

Exemplo:

O exemplo a seguir cria um índice em ordem cronológica descendente de acordo com a data de admissão dos funcionários:

```
USE Pessoal
INDEX ON DESCEND(datadm) TO Indadm

* datadm é um campo tipo data que contém a data de
* admissão do funcionário.
```

Para realizar pesquisas sobre um arquivo de índice criado em ordem descendente deve ser utilizada a função DESCEND( ). Por exemplo:

```
USE Pessoal INDEX Indadm
SET DATE BRITISH
SEEK DESCEND(CTOD("06/04/88"))
IF FOUND()
   ? numero, nome, datadm, depto, salario
ELSE
   ? "Data não encontrada !"
   WAIT "Pressione qualquer tecla !"
ENDIF
RETURN
```

Veja Também:

INDEX, SEEK

Função: EOF( ) – Fim do Arquivo:

Forma:

EOF( )

Propósito:

Retorna verdadeiro (.T.) se o apontador de registros estiver no fim do arquivo de dados atualmente em uso ou se uma tentativa foi feita para ultrapassar o fim do arquivo; caso contrário, retorna falso (.F.).

Argumentos:

Nenhum

Utilização:

A função EOF( ) ("end of file") é utilizada para verificar se, através da movimentação progressiva pelos registros de um arquivo (do início para o fim), o seu final foi atingido. Pode-se evitar, assim, executar o comando SKIP, quando o apontador de registros já estiver posicionado no final do arquivo em uso.

Quando EOF( ) torna-se verdadeira (.T.), o apontador de registros é posicionado após o último registro (LASTREC( )+1). Qualquer outra tentativa de deslocar o apontador de registros ultrapassando o último registro retorna o mesmo resultado sem entrar em condição de erro.

Se o arquivo de dados não contém registros EOF( ) retornará verdadeiro (.T.).

Biblioteca:

CLIPPER.LIB

Exemplo:

```
USE Estoque
GO BOTTOM
? EOF()        && retorna falso (.F.), pois o apontador
               está no último registro.
USE Estoque
GO BOTTOM
SKIP
? EOF()        && retorna  verdadeiro  (.T.),  pois  o
               && apontador foi para o final do arquivo
```

*** Para percorrer o arquivo em ordem progressiva e parar ao atingir o seu final, pode-se usar:**

```
USE Estoque
DO WHILE .NOT. EOF()  && quando EOF() for verdadeira o
   ? produto, saldo   && DO WHILE será finalizado.
   SKIP
ENDDO
USE
RETURN
```

Veja Também:

FIND, SEEK, LOCATE, SKIP, BOF( ), RECNO( ), LASTREC( ), FOUND( )

Função FCOUNT( ) – Número de Campos:

Forma:

FCOUNT( )

Propósito:

Retorna a número de campos do arquivo aberto na área de trabalho atualmente selecionada.

Argumentos:

Nenhum

Utilização:

A função FCOUNT( ) é utilizada para obter-se o número de campos de um arquivo que esteja aberto na área de trabalho atualmente selecionada.
Se não houver um arquivo de dados aberto FCOUNT( ) retorna zero (0).
FCOUNT( ) é útil em aplicações onde se deseja independência da estrutura de dados, ou seja, aplicações que possam trabalhar com qualquer arquivo de dados, onde obviamente não se conhece "a priori" a quantidade de campos que possuirão.

Biblioteca:

CLIPPER.LIB

Exemplo:

O exemplo a seguir apresenta os nomes dos campos de um arquivo desconhecido pelo programa, aberto na área de trabalho selecionada:

```
CLEAR
ACCEPT "Digite o Nome do Arquivo:" TO arq
USE &arq
FOR i = 1 TO FCOUNT()
    ? FIELD(i)
NEXT
USE
RETURN

Para determinar o número de campos do arquivo Mala.dbf:

USE Mala
? FCOUNT() && retorna 7
```

Veja Também:

FIELDNAME( )/FIELD( ),TYPE, AFIELD( )

Função: FIELDNAME( ) – Nome do Campo:

Forma:

FIELDNAME(<exp.numérica>)/FIELD(<exp.numérica>)

Propósito:

Retorna o nome do campo do arquivo de dados em uso, cujo número é indicado pela <expressão numérica>.

Argumentos:

<expressão numérica>: é um número definindo a posição do campo dentro da estrutura do arquivo de dados.

Utilização:

A função FIELDNAME( ) ou apenas FIELD( ) é utilizada para se obter o nome (em letras maiúsculas) do n-ésimo campo do arquivo de dados em uso. O número do campo é especificado pela <expressão numérica>.
Se o número da <expressão numérica> for maior que o número de campos do arquivo, uma cadeia de caracteres nula será retornada. Entretanto, a função FCOUNT( ) pode ser utilizada para se determinar o número de campos que possui o arquivo, evitando-se assim ultrapassá-lo.
Esta função é interessante para ser utilizada em aplicações onde se procure independência da estrutura de dados.
A função vetorial AFIELDS( ) produz um resultado similar mas expandido, pois permite armazenar em vetores toda a estrutura de um arquivo de dados.

Biblioteca:

CLIPPER.LIB

.Exemplo:

* O exemplo a seguir permite obter todos os nomes dos campos de um arquivo de dados:

```
USE Mala
? FCOUNT()        && retorna 3
FOR i=1 TO FCOUNT()
    ? FIELDNAME(i)
NEXT
```

* Onde:

? FIELDNAME(1) && retorna "CODIGO"
? FIELDNAME(2) && retorna "NOME"
? FIELDNAME(3) && retorna "ENDEREÇO"

* E assim por diante.

Veja Também:

COPY STRUCTURE EXTENDED, FCOUNT( ), AFIELDS( ), LASTREC( ), TYPE( ), LEN( )

Função: FLOCK( ) – Bloqueia Arquivo:

Forma:

FLOCK( )

Propósito:

Inibir ou impedir o acesso de outros usuários a um arquivo de dados compartilhado em um ambiente multiusuário. FLOCK( ) retorna verdadeiro (.T.) se o acesso foi restringido; ou falso (.F.), caso contrário.

Argumentos:

Nenhum

Utilização:

Quando se está operando em ambiente multiusuário, a função FLOCK( ) impede o acesso de outros usuários ao arquivo aberto na área de trabalho atualmente selecionada.
Se o acesso ao arquivo for restringido FLOCK( ) retorna verdadeiro (.T.); caso contrário retorna (.F.).
O arquivo permanecerá restrito (bloqueado) até que seja fechado ou tiver o acesso liberado através do comando UNLOCK ou da função RLOCK( ).

Biblioteca:

CLIPPER.LIB

Exemplo:

```
USE Mala
FLOCK()
IF FLOCK()
   ZAP
ENDIF
```

Veja Também:

USE...EXCLUSIVE, SET EXCLUSIVE, UNLOCK, RLOCK( )

Função: FOUND( ) – Encontrado:

Forma:

FOUND( )

Propósito:

Retorna verdadeiro (.T.) se o último comando de pesquisa executado, SEEK, FIND, LOCATE ou CONTINUE, encontrou um registro que possua a chave pesquisada ou satisfaça condição solicitada. Retorna falso (.F.) se a última operação de pesquisa não foi bem sucedida.

Argumentos:

Nenhum

Utilização:

A função FOUND( ) é usada para determinar se um dos comandos de pesquisa, SEEK, FIND, LOCATE ou CONTINUE, foi bem sucedido ou não. É muito útil para determinar que caminho alternativo deve tomar o fluxo de execução de um programa caso uma pesquisa tenha ou não encontrado o registro procurado.
FOUND( ) retorna verdadeiro (.T.) se o último comando de pesquisa executado foi bem sucedido; e falso (.F.) caso contrário.
Se o apontador de registros for movido por qualquer outro comando (SKIP, GOTO, etc) que não sejam os comandos SEEK, FIND, LOCATE ou CONTINUE, o resultado da função FOUND( ) será sempre .F. (falso).
Cada área de trabalho selecionada possui um controle próprio ("flag") indicando o valor atual da função FOUND( ). Se houver um relacionamento estabelecido através do comando SET RELATION, a função FOUND( ) na área relacionada será verdadeira (.T.) quando o registro relacionado foi encontrado (efetivou-se o relacionamento), e falsa (.F.) caso contrário.

Biblioteca:

CLIPPER.LIB

Exemplos:

* Utilização da função FOUND( ) com um arquivo não indexado:

```
USE Estoque
LOCATE FOR SUBSTR(codigo,1,2)="BE"
DO WHILE FOUND()
   ? codigo, produto, saldo
   CONTINUE
ENDDO
```

* Utilização da função FOUND( ) com um arquivo indexado:

```
USE Estoque INDEX Indcod
SEEK "BE"
IF FOUND()
   DO WHILE SUBSTR(codigo,1,2)="BE"
      ? codigo, produto, saldo
      SKIP
   ENDDO
ELSE
  ? "Código Não Encontrado !"
ENDIF
CLOSE DATABASE
RETURN
```

Veja Também:

SEEK, FIND, LOCATE, CONTINUE, EOF( ), SET RELATION, SET SOFTSEEK

Função: HARDCR( ) — Quebra de linha:

Forma:

HARDCR(<exp.caracter>)

Propósito:

Retorna a expressão caracter determinada com todos os códigos de fim de linha lógico (CHR(141)) substituidos por códigos de fim de linha físico (CHR(13) = Enter). Seu objetivo é apresentar adequadamente campos memo contendo fins de linha lógicos ("soft carriage returns").

Argumentos:

<*expressão caractere*>: é a cadeia de caracteres ou campo memo a ser convertido.

Utilização:

A função HARDCR( ) é utilizada para permitir a apresentação de conteúdos de campo memo com a quebra automática de linha (CHR(141)), gerada pela função MEMOEDIT( ), substituída por <Enter> (CHR(13)), como se fosse uma cadeia de caracteres normal. Dessa forma um campo memo poderá ser normalmente apresentado pelos comandos ?,@...SAY, REPORT FORM etc. HARDCR( ) retorna uma cadeia de caracteres até 65.535 bytes (64 K) de comprimento.

Biblioteca:

EXTEND.LIB
MEMO.LIB (versões anteriores à Summer 87)

Exemplo:

```
* Suponho um arquivo que possua o campo memo obs:

? HARDCR(obs)

* Apresenta o conteúdo do campo memo com a mesma
* formatação de linha gerada pela função MEMOEDIT( )
```

Veja Também:

?/??, @...SAY, REPORT FORM, MEMOEDIT( ), MEMOLINE( ), MEMOREAD( ), MEMOTRAN( ), MEMOWRIT( ), MLCOUNT( )

Função: HEADER( ) – Cabeçalho do Arquivo de Dados:

Forma:

HEADER( )

Propósito:

Retorna um valor numérico representando o tamanho, em bytes, do cabeçalho do arquivo de dados em uso na área de trabalho selecionada.

Argumentos:

Nenhum

Utilização:

A função HEADER( ) determina o tamanho, em bytes, do cabeçalho do arquivo que está aberto na área de trabalho atualmente selecionada.
Sua utilização específica refere-se à construção de rotinas de Back-Up de arquivos de dados (.dbf), onde se torna necessária a separação de um arquivo em diversos disquetes, devido ao seu tamanho. Neste caso é preciso determinar o espaço ocupado pelo cabeçalho dos arquivos, a fim de verificar se o espaço ainda disponível no disco é suficiente para copiar um determinado número de registros.
No cabeçalho dos arquivos de dados ficam armazenadas informações sobre sua estrutura, data da última atualização e o número atual de registros. O tamanho do cabeçalho dos arquivos pode, entretanto, ser calculado por aproximação através da fórmula:

32 * <número de campos> + 35

A função HEADER( ) em geral é utilizada em conjunto com as funções RECCOUNT( ), RECSIZE( ), e DISKSPACE( ) para a construção de rotinas de Back-up, como a exemplificada a seguir, sem a necessidade de se recorrer ao comando BACKUP do DOS, quando se torna necessário segmentar um arquivo em diversos disquetes.
Nas versões anteriores à Summer 87 a função HEADER( ) está contida no módulo-objeto auxiliar EXTENDC.obj. Para utilizá-la este módulo deve ser incluído na lista de módulos-objeto a ser passada para o link-editor. Além disso, deve ser utilizado o comando EXTERNAL, no módulo principal do sistema desenvolvido, para declará-la como externa para o compilador do Clipper, vide a seguir.

EXTERNAL HEADER

Na versão Summer 87 a função HEADER( ) está contida na biblioteca EXTEND.lib, que deve estar incluída na lista de bibliotecas passadas ao link-editor.

Biblioteca:

EXTEND.LIB
EXTENDC.obj (nas versões anteriores à Summer 87)

Exemplo:

```
EXTERNAL HEADER, RECSIZE, DISKSPACE
* São todas rotinas escritas em "C" que devem ser
```

* declaradas externas ao compilador do Clipper, caso você
* esteja utilizando uma versão anterior à Summer 87.

```
USE Arquivo
vdisc=0
DO WHILE .NOT. EOF()
   vdisc=vdisc+1
   @ 23,22 SAY "Coloque Back-Up No."+STR(vdisc,2)
   @ 23,43 SAY "no drive A"
   @ 24,22 SAY "     Tecle (Enter) quando pronto..."
   WAIT ""
   COPY NEXT (DISKSPACE(1)-HEADER())/RECSIZE() TO A:Backup
   * determina a quantidade de registros que poderão ser
   * copiados para o disquete sem "enche-lo".
   SKIP
ENDDO
CLEAR
@ 23,33 SAY "Arquivo Salvo !"
USE
RETURN

USE Estoque
? HEADER()     && resulta 348
* Número total de bytes do arquivo Estoque.dbf
? (RECSIZE()*LASTREC())+HEADER()+1
* resulta: 21384
```

Veja Também:

RECSIZE( ), DISKSPACE( ), LASTREC( ), COPY TO

Função: INDEXEXT( ) — Índice Externo:

Forma:

INDEXEXT( )

Propósito:

Determina se a aplicação foi link-editada utilizando o módulo-objeto NDX.obj, que permite a utilização índices compatíveis com os do dBASE III Plus.
Disponível a partir da versão Summer 87.

Argumentos:

Nenhum

Utilização

A função INDEXEXT( ) é utilizada para determinar se uma aplicação está utilizando índices externos, compatíveis com os índices (.ndx) do dBASE III Plus, ou os índices (.ntx) normais do Clipper.
INDEXEXT( ) retorna "NDX" se estiverem sendo utilizados índices do dBASE III Plus ou "NTX" se estiverem sendo utilizados os índices normais do Clipper.
Para serem utilizados índices compatíveis com os do dBASE III Plus, o sistema deve ser link-editado juntamente com o módulo-objeto NDX.obj (veja a seção 8.2.2– Link-editando com os utilitários do Clipper.

Biblioteca:

CLIPPER.LIB

Exemplo:

Para pesquisar a existência de um índice e criá-lo, caso não exista, deve-se verificar a extensão correta conforme o caso de se estar utilizando índices do dBASE III ou do Clipper:

```
IF .NOT. FILE("Alfa."+INDEXEXT( ))
   INDEX ON nome TO Alfa
ENDIF
```

Veja Também:

INDEXKEY( ), INDEXORD( )

Funções: INDEXKEY( ) – Chave do Indice:

Forma:

INDEXKEY(<expressão numérica>)

Propósito:

Retorna a expressão-chave de um arquivo índice (.ntx) ou (.ndx), onde a <expressão numérica> determina o índice desejado.

Argumentos:

<*expressão numérica*>: especifica a n-ésima posição do índice desejado na lista dos índices abertos na área de trabalho selecionada através dos comandos USE...INDEX <lista de índices> ou SET INDEX TO <lista de índices>. Se for especificado zero, será assumido o índice mestre (índice de controle), ou seja o índice que está definindo a ordem de acesso aos registros do arquivo, não importando a sua posição na lista.

Utilização:

A função INDEXKEY( ) retorna a expressão-chave de um arquivo de índices associado à um arquivo de dados aberto na área de trabalho selecionada.
A <expressão numérica> identifica o índice desejado a partir da lista de índices abertos juntamente com o arquivo de dados. Zero irá retornar a expressão-chave do índice mestre (o primeiro índice da lista ou o definido pelo comando SET ORDER TO), que controla a ordem de acesso ao arquivo.
Uma expressão nula ("") será retornada caso o número definido na expressão numérica ultrapasse o número de índices abertos para o arquivo.
Pode-se utilizar a função INDEXKEY( ) para verificar a chave dos índices criados ou para escolher qual índice deverá ser utilizado.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
USE Mala INDEX Indcod,Indnome,Indcep
? INDEXKEY(1)    && retorna codigo
? INDEXKEY(2)    && retorna nome
? INDEXKEY(3)    && retorna cep
? INDEXKEY(0)    && retorna codigo
SET ORDER TO 3
? INDEXKEY(0)    && retorna cep
```

Veja Também:

USE, INDEX, SET INDEX TO, SET ORDER TO, INDEXEXT( ), INDEXORD( )

Função: INDEXORD( ) – Posição do Índice Mestre:

Forma:

INDEXORD( )

Propósito:

Determina a posição do índice de controle (índice mestre) na lista de índices abertos pelo comando USE...INDEX <lista de índices> ou SET INDEX TO <lista de índices>, na área de trabalho selecionada.
O índice mestre é o índice que determina a ordem de acesso aos registros do arquivo de dados.
Disponível a partir da versão Summer 87.

Argumentos:

Nenhum

Utilização:

A função INDEXORD( ) retorna um número indicando a posição do índice de controle do arquivo de dados na lista de índices abertos. Se for retornado o valor zero significará que não há índice de controle e o arquivo de dados está sendo acessado em ordem natural (ordem do número dos registros).
INDEXORD( ) é útil para verificar (e eventualmente armazenar numa variável) o último índice de controle utilizado, quando for necessário acessar o arquivo posteriormente por esta mesma ordem.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
USE Mala INDEX Indcod, Indnome, Indcep
mestre=INDEXORD()
? mestre                && resulta 1
SET ORDER TO 2
? INDEXORD()            && resulta 2
SET ORDER TO mestre
? INDEXORD              && resulta 1
```

Veja Também:

USE, SET INDEX, SET ORDER, INDEXEXT( ), INDEXKEY( )

Função: LASTREC( )/RECCOUNT( ) – Último Registro:

Forma:

LASTREC( ) ou RECCOUNT( )

Propósito:

Retorna o número do último registro físico do arquivo de dados em uso.

Argumentos:

Nenhum

Utilização:

A função LASTREC( ) é equivalente à função RECCOUNT( ), ambas retornam o número do último registro físico do arquivo de dados aberto na área de trabalho selecionada.
Comandos que selecionam registros como SET FILTER ou SET DELETED não afetam o resultado de LASTREC( ).
LASTREC( ) retorna zero se não houver nenhum arquivo em uso (aberto) na área de trabalho selecionada ou se o arquivo estiver vazio.
A função LASTREC( ) pode ser usada, por exemplo, para apresentar a quantidade de registros de um arquivo ou para não permitir que um registro de número maior que o último tente ser acessado.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
USE Mala
? LASTREC()    && retorna o número do último registro.
INPUT "Digite o Número do Pegistro Desejado:" TO reg
IF reg > LASTREC()
   ? "Registro não Existente"
ELSE
   GOTO reg
   ? codigo, nome, ende, telef, cep
ENDIF
```

Veja Também:

RECNO( ), FIELD( )

Função: LUPDATE( ) – Última Data de Atualização:

Forma:

LUPDATE( )

Propósito:

Retorna a data na qual o arquivo de dados aberto na área de trabalho atualmente selecionada foi pela última vez modificado e fechado, correspondendo à última data de sua atualização.

Argumentos:

Nenhum

Utilização:

A função LUPDATE( ) retorna a data (no formato data) na qual o arquivo de dados atualmente em uso foi atualizado pela última vez. Essa informação pode ser muito útil ao usuário de uma aplicação desenvolvida em Clipper, pois lhe permitirá saber qual foi a última vez que alterou os dados do seu arquivo.
A data de atualização do arquivo somente é modificada após o fechamento do arquivo e se este tiver sofrido alguma alteração: inclusão de registros, modificação de dados de algum registro, deleção de registros, exclusão de registros etc.
Se não houver um arquivo de dados em uso na área de trabalho selecionada. LUPDATE( ) retornará uma data em branco.
Nas versões anteriores à Summer 87 a função LUPDATE( ) está contida no módulo-objeto auxiliar EXTENDC.obj. Para utilizá-la deve-se declará-la externa ao compilador através do comando EXTERNAL e o arquivo EXTENDC.obj deve ser incluído na lista de módulos-objeto passada ao link-editor.

Biblioteca:

EXTEND.LIB
EXTENDC.obj (nas versões anteriores à Summer 87)

Exemplo:

```
* EXTERNAL LUPDATE()
* declara a função como externa nas versões anteriores
* à Summer 87
SET DATE BRITISH
USE Estoque
? LUPDATE()          && retorna 15/02/88
? TYPE((LUPDATE)) && retorna D (tipo data)
```

Função: MEMOEDIT( ) — Edição de campos memo nas versões Anteriores à Summer 87

Forma:

MEMOEDIT(<exp.caractere>,
[<exp.num.1>,<exp.num2>,<exp.num.3>,<exp.num.4>]
[,<exp.lógica>])

Propósito:

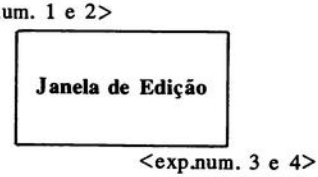
Permite a edição de campos memo ou de uma cadeia de caracteres, em uma "janela" especificada na tela, com diversos recursos de formatação.

Argumentos:

<expressão caractére>: define o nome do campo memo ou a cadeia de caracteres a ser editada.

<exp.numérica 1>,<exp.numérica 2>: definem as coordenadas do canto superior esquerdo da janela a ser criada para a edição. Se omitidas, será assumida a coordenada 01,00.

<exp.numérica 3>,<exp.numérica 4> definem as coordenadas do canto inferior direito da janela a ser criada para a edição. Se omitidas, será assumida a coordenada 24,79.



<*expressão lógica*>: determina se um campo memo será editado (alterado e regravado) ou apenas apresentado. Se for especificado verdadeiro (.T.), o campo memo é apresentado em modo de edição. Se for especificado falso (.F.)o campo memo é apenas apresentado, não sendo permitida a edição. Se o campo memo for apenas apresentado (for especificado .F.) não será possível realizar uma rolagem de linhas na tela ("scroll"), e apenas parte dele poderá ser apresentada.
Se a <expressão lógica> não for especificada será assumido verdadeiro (.T.) e o campo será editado.

Utilização:

A função MEMOEDIT( ) abre uma janela, de dimensões determinadas na tela, através da qual poderão ser editados campos memo ou cadeias de caracteres extensas.
Uma vez especificada a janela, e editado o campo memo ou a variável caractere, diversas teclas de controle de edição poderão ser utilizadas para alterá-lo como a seguir:

Tecla	Função
Seta para Cima	Move o cursor uma linha para cima
Seta para Baixo	Move o cursor uma linha para baixo
Seta para Esquerda	Move um caractere para a esquerda
Seta para Direita	Move um caractere para a direita
^→	Move uma palavra para a esquerda
^→	Move uma palavra para a direita
Home	Move para o início da linha atual
End	Move para o final da linha atual
^Home	Move para o início do campo memo
^End	Move para o final do campo memo
PgUp	Move uma janela para cima
PgDw	Move uma janela para baixo
^PgUp	Move para o início da janela atual
^PdDw	Move para o final da janela atual
^W	Finaliza a edição, gravando o campo
Esc	Aborta a edição, não gravando
^Y	Elimina a linha do cursor
^T	Elimina a palavra à direita
^B	Reformata o campo memo na janela de edição es-especificada

O símbolo ^ indica que a tecla Ctrl (control) deve ser pressionada simultaneamente com a outra tecla indicada. A função MEMOEDIT( ) deve ser utilizada em conjunto com o comando REPLACE quando se deseja atualizar o conteúdo de um campo memo no arquivo de dados.
Caso a função MEMOEDIT( ) não satisfaça as necessidade de edição de sua aplicação, as funções MEMOREAD( ), MEMOTRAN( ) e MEMOWRIT( ) permitem que seja utilizado o editor de textos de sua preferência para substituí-la. Você poderá gerar, a partir de um campo memo, o texto a ser editado (com MEMOWRIT( )) editá-lo com o editor de textos desejado, e a seguir gravá-lo novamente no campo memo (com MEMOREAD( )).
Caso os caracteres de controle para a mudança de linha gerados pela função MEMOEDIT( ) prejudiquem a utilização de um outro editor de textos ou a emissão de relatórios, utilize as funções MEMOTRAN( ) ou HARDCR( ) para substituí-los.

Biblioteca:

MEMO.LIB (nas versões anteriores à Summer 87)

Exemplo:

* Supondo que no arquivo Mala.dbf exista um campo memo chamado obs:

```
USE Mala INDEX Indcod
REPLACE obs WITH MEMOEDIT(obs,7,10,22,69,.T.)
```

* Abre uma janela iniciando-se na linha 7, coluna 10 e finalizando-se na linha 22, coluna 69 para a edição com atualização (.T.) do campo memo obs.

```
MEMOEDIT(obs,7,10,22,69,.F.)
```

* Abre a mesma janela, só que apenas mostra o conteúdo do campo memo obs. A expressão lógica foi especificada falsa (.F.).

Para a edição de uma variável caracter utiliza-se:

```
notas=SPACE(200)
notas=MEMOEDIT(notas,10,10,20,69,.T.)
```

Função: MEMOEDIT( ) – Edição de campos memo na versão Summer 87 e posteriores

Forma:

MEMOEDIT(<exp.caractere 1>,
[<exp.num.1>,<exp.num2>,<exp.num.3>,<exp.num.4>]
[,<exp.lógica>][,<exp.caractere 2>][,<exp.num.5>]
[,<exp.num.6>][,<exp.num.7>][,<exp.num.8>]
[,<exp.num.9>][,<exp.num.10>])

Propósito:

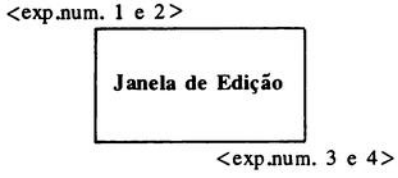
Permite a edição de campos memo ou de uma cadeia de caracteres em uma "janela" especificada na tela, com diversos recursos de formatação e definição de uma função-de-usuário para o controle.
MEMOEDIT( ) retorna o campo memo (ou cadeia de caracteres) modificada se a edição for finalizada com <Ctrl-W> ou o original se a edição for finalizada com <Esc>.

Argumentos:

<expressão caractere>: define o nome do campo memo ou a cadeia de caracteres a ser editada.

<exp.numérica 1>,<exp.numérica 2>: definem as coordenadas do canto superior esquerdo da janela a ser criada para a edição. Se omitidas será assumida a coordenada 01,00.

<exp.numérica 3>,<exp.numérica 4>: definem as coordenadas do canto inferior direito da janela a ser criada para a edição. Se omitidas será assumida a coordenada 24,79.



<expressão lógica>: determina se um campo memo será editado (alterado e regravado) ou apenas apresentado. Se for especificado verdadeiro (.T.), o campo memo entra em modo de edição. Se for especificado falso (.F.) o campo memo é apenas apresentado, não sendo permitida a edição. Se o

campo memo for apenas apresentado (for especificado .F.) será possível realizar uma rolagem de linhas na tela ("browse mode") para visualizar todo o seu conteúdo.
Se a <expressão lógica> não for especificada será assumido verdadeiro (.T.) e o campo será editado.

<exp.caractere 2>: define o nome de uma função-de-usuário a ser executada sempre que uma tecla de exceção (ou controle) for pressionada pelo usuário. O nome deve ser especificado sem os parênteses ou argumentos (apenas o nome da função).

<exp.numérica 5>: determina o comprimento da linha de edição. Se o resultado da <exp.numérica 5> for maior que a largura da janela definida por (<exp.numérica 4>-<exp.numérica 2>-1), o texto na janela se deslocará horizontalmente ("scroll" horizontal). Se a <exp.numérica 5> não for especificada será assumido o valor resultante da <exp.numérica 4>—<exp.numérica 2>—1, ou seja a largura da janela definida.

<exp.numérica 6>: determina o tamanho da tabulação a ser utilizada, permitindo tabulações reais. Se não especificada será assumido o valor de 4 espaços.

<exp.numérica 7>: determina a linha inicial do campo memo (ou texto sendo editado), onde deverá ser colocado o cursor.

<exp.numérica 8>: determina a coluna inicial do campo memo (ou texto sendo editado), onde deverá ser colocado o cursor.

<exp.numérica 9>: determina a linha inicial onde deverá ser colocado o cursor, relativamente à posição da janela de edição definida. Se não especificada será assumido zero.

<exp.numérica 10>: determina a coluna inicial onde deverá ser colocado o cursor, relativamente à posição da janela de edição definida. Se não especificada será assumido zero.

Nota: Para todos os argumentos opcionais que não forem utilizados deverá ser passado um argumento fantasma, caso sejam utilizados argumentos que são especificados mais para o final da lista.

Utilização:

A função MEMOEDIT( ) é um editor de textos de propósito geral que pode ser utilizado nos sistemas escritos em Clipper. Suporta um grande número de opções de formatação, além de poder ser incluída uma função-de-usuário (vide o comando FUNCTION) permitindo a redefinição das teclas de controle de edição e qualquer outra atividade de edição.

Uma vez especificada a janela de edição, e editado o campo memo ou a variável caracter, as seguintes teclas de controle de edição poderão ser utilizados para alterá-lo:

Tecla	Função
Seta para Cima	Move o cursor uma linha para cima
Seta para Baixo	Move o cursor uma linha para baixo
Seta para Esquerda	Move um caractere para a esquerda
Seta para Direita	Move um caractere para a direita
^←	Move uma palavra para a esquerda
^→	Move uma palavra para a direita
Home	Move para o início da linha atual
End	Move para o final da linha atual
^Home	Move para o início do campo memo
^End	Move para o final do campo memo
PgUp	Move uma janela para cima
PgDw	Move uma janela para baixo
^PgUp	Move para o início da janela atual
^PgDw	Move para o final da janela atual
^W	Finaliza a edição, gravando o campo
Esc	Aborta a edição, não gravando
^Y	Elimina a linha do cursor
^T	Elimina a palavra à direita
^B	Reformata o campo memo na janela de edição especificada.

O símbolo ^ indica que a tecla Ctrl (control) deve ser pressionada simultaneamente com a outra tecla indicada.
Dependendo do valor especificado para a <exp.lógica> dois modos de apresentação do texto são suportados pela função MEMOEDIT( ). Se ela for verdadeira (.T.) MEMOEDIT( ) automaticamente entrará no modo de edição, caso contrário MEMOEDIT( ) entrará no modo de apresentação de dados ("browse").
No modo de apresentaçãpo, todas as teclas de "navegação" (deslocamento do cursor) estão ativas e possuem as mesmas funções do modo de edição, com uma exceção:

- no modo de edição a "rolagem" da tela ("scroll") está desligada e as setas para cima e para baixo deslocam o cursor uma linha para cima ou para baixo respectivamente.

- no modo de apresentação a "rolagem" da tela ("scroll") está ligada e as setas para cima e para baixo "rolam" o conteúdo da MEMOEDIT( ) uma janela para cima ou uma janela para baixo, respectivamente.

Função-de-Usuário:

Se for especificada uma função-de-usuário através da <exp.caractere 2>, ao ser pressionada uma tecla de exceção, MEMOEDIT( ) passará a ela automaticamente três parâmetros: modo, linha e coluna.

- **O parâmetro Modo**: indica o estado atual da função MEMOEDIT( ), dependendo da última tecla pressionada pelo usuário, ou da última ação tomada anteriormente pela função-de-usuário. São possíveis os seguintes valores para o modo:

Modo	Descrição
0	Aguardando (neutro)
1	Tecla reconfigurada ou sem função (dados inalterados)
2	Tecla reconfigurada ou sem função (dados alterados)
3	Início da edição ou apresentação

O modo 3 é o início da edição ou apresentação dos dados. Quando uma função-de-usuário é especificada MEMOEDIT( ) a chama imediatamente após ser executada. Neste ponto sua função poderá configurar diversas opções de edição da MEMOEDIT( ): rolagem, modos de inserção ou sobrescrito, teclas especiais etc. MEMOEDIT( ) chamará a função-de-usuário repetidamente, permanecendo em estado de inicialização até que seja retornado pela função-de-usuário o código zero (RETURN 0). Retornando zero o campo memo ou o texto será apresentado ou entrará no modo de edição (de acordo com a <exp.lógica> especificada), assumindo as configurações efetuadas pela função-de-usuário.
Os modos 0, 1 e 2 processam as teclas que forem pressionadas pelo usuário. O modo zero (neutro) é assumido quando não há nenhuma tecla de controle a ser processada. Dentro deste modo são geralmente atualizadas a linha e a coluna onde está atualmente posicionado o cursor. MEMOEDIT( ) chama a função-de-usuário toda vez que uma tecla de exceção, ou seja, as teclas de controle (Ctrl e Alt) e teclas de função forem pressionadas. Como estas teclas não são processadas pela MEMOEDIT( ) a função-de-usuário poderá reconfigurá-las para executar uma determinada operação.

- **O parâmetro Linha:** indica a linha da janela de edição definida, na qual está atualmente posicionado o cursor, quando a função-de-usuário é chamada. A linha inicial é a linha zero.

- **O parâmetro Coluna:** indica a coluna da janela de edição definida, na qual está atualmente posicionado o cursor, quando a função-de-usuário é chamada. A coluna inicial é a coluna zero.

Quando os modos passados pela MEMOEDIT( ) forem iguais a 1,2, ou 3, a função-de-usuário poderá retornar um valor (código) instruindo MEMOEDIT( ) sobre qual ação tomar em seguida. Os possíveis valores a serem retornados à MEMOEDIT( ) e suas consequências são os seguintes:

Valor	Ação da MEMOEDIT( )
0	Nenhuma ação ou ações padrão.
1 a 31	Executa a ação requisitada, de acordo com o valor da tecla especificada: por exemplo, se for 22, que equivale à tecla <Ins> o modo de inserção será ativado ou desativado, se for 2 equivale a Ctrl-B (reformatação) e se for 23 equivale a Ctrl-W (sair e salvar).
32	Ignora a tecla pressionada (desabilitando-a).
33	Processa a tecla pressionada como sendo um dado (inserindo-a no texto), possibilitando o recurso de acentuação.
34	Habilita ou desabilita a quebra lógica de linha ("word-wrap") sem cortar palavras.
35	Habilita ou desabilita a rolagem da tela.
100	Desloca o cursor para a próxima palavra.
101	Desloca o cursor para o canto inferior direito da janela.

Atenção: as teclas de movimentação do cursor, <Enter>, retrocesso, tabulação, <Del> e todas as teclas de caracteres não podem ser desativadas.

Possibilidade de Acentuação:

Através do valor de retorno 33 pode ser construída uma função-de-usuário que, entre outras coisas, permita que o texto seja acentuado. Para isso basta definir que ao se pressionar por exemplo a tecla de controle <Alt> e outra <tecla qualquer> seja inserido um c-cedilha ou um a com til na posição onde está o cursor. Da mesma forma poderão ser definidos os outros caracteres especiais da língua portuguesa.

Quebra-de-linha:

A quebra-de-linha ("word-wrap") é habilitada ou desabilitada através do retorno do valor 34 pela função-de-usuário. O padrão é ligado, e nesse caso MEMOEDIT( ) insere um controle de mudança de linha lógico ("soft carriage return") após a palavra mais próxima da borda direita da janela de edição, ou ao máximo comprimento de linha definido, mudando de linha sem truncar palavras. Quando a quebra-de-linha está desligada MEMOEDIT( ) rola o texto além da janela definida até que seja atingido o fim da linha definida. Neste ponto o usuário deverá pressionar <Enter> (inserindo uma mudança de linha física – "hard carriage return") para mudar para a próxima linha de edição.
Uma vez que o controle de linha lógico ("soft carriage return") pode interferir na impressão de relatórios, ou na edição do texto através de outros processadores, podem ser utilizadas as funções HARDCR( ) e MEMOTRAN( ) para substituí-los por caractéres de controle mais adequados.

Reformatação de Parágrafos:

Ao se pressionar <Ctrl-B> ou se a função-de-usuário retornar o código 2, o parágrafo no qual se localiza o cursor será reformado, não importando se a quebra-de-linha está ligada ou desligada. A operação de reformatação de parágrafos é idêntica à realizada pelo WordStar ou SideKick ao serem pressionadas as teclas <Ctrl e B>.

Tabulação:

Quando for especificado o tamanho da tabulação através da

<exp.numérica 6>,

MEMOEDIT( ) insere o caractere de controle de tabulação (Hex 09H) no texto, na posição em que a tecla <Tab> foi pressionada. Se o argumento de tabulação não foi pressionado, MEMOEDIT( ) insere quatro espaços em branco. O tamanho da tabulação definido pela <exp.numérica 6> é global para todo o texto.
A função MEMOEDIT( ) pode ser utilizada em conjunto com o comando REPLACE quando se deseja automaticamente atualizar o conteúdo de um campo memo no arquivo de dados.
Finalmente, caso a função MEMOEDIT( ) não satisfaça as necessidades de edição de sua aplicação, as funções MEMOREAD( ), MEMOTRAN( ) e MEMOWRIT( ) permitem que seja utilizado o editor de textos de sua preferência para substituí-la.

Você poderá gravar o texto a ser editado em um arquivo no disco com a função MEMOWRIT( ), a partir de um campo memo. Em seguida poderá editá-lo com o editor de textos desejado. Para retorná-lo ao campo memo utilize a função MEMOREAD( ).

Biblioteca:

EXTEND.LIB

Veja Também:

MEMOEDIT( ), (versões anteriores à Summer 87), MEMOLINE( ), MEMOREAD( ), MEMOTRAN( ), HARDCR( ), MEMOWRIT( ), MLCOUNT( ), REPLACE

Função: MEMOLINE( ), Linha do Campo Memo:

Forma:

MEMOLINE(<exp.caractere>,<exp.num. 1>,<exp.num. 2>)

Propósito:

Extrair uma linha de texto formatada a partir de uma <expressão caractere> ou de um campo memo.
Retorna uma cadeia de caracteres correspondente à linha especificada extraída.
Disponível a partir da versão Summer 87.

Argumentos:

<expressão caractere>: é o nome do campo memo ou da cadeia de caracteres a ter uma linha formatada extraída.
<exp.numérica 1>: é o número máximo desejado de caracteres (ou posições) por linha.
<exp.numérica 2>: é o número da linha a ser extraída.

Utilização:

A função MEMOLINE( ) extrai linhas de texto a partir de cadeias de caracteres longas ou de campos memo de arquivos de dados. Ela baseia-se no número de caracteres desejados por linha.

Esta função fornece grande capacidade e flexibilidade para a apresentação ou impressão formatada de campos memo.
Se o fim de uma linha cortar uma palavra no meio, esta palavra será tranferida para a próxima linha.
Caso uma linha possua menos carateres do que o especificado pela <expressão numérica 1>, será completada com caracteres em branco.
Se o número da linha, especificado pela <expressão caractere 2> for maior que a quantidade total de linhas da cadeia de caracteres ou do campo memo em questão, será retornada uma "string" nula ("") pelo MEMOLINE( ).
A função MEMOLINE( ) deve ser utilizada em conjunto com a função MLCOUNT( ) que conta o número de linhas de um campo memo.

Biblioteca:

EXTEND.LIB

Exemplo:

Impressão formatada de um campo memo.

```
USE Mala
SET DEVICE TO PRINT
larg = 60
col  = 10
lins = MLCOUNT(obs, larg)
* determina o número de linhas do campo memo obs, de
* acordo com largura (larg=60) especificada.
FOR i = 1 TO lins
    @ PROW()+1,col SAY MEMOLINE(obs,larg,i)
    * formata a i-ésima linha do campo memo.
NEXT
SET DEVICE TO SCREEN
USE
RETURN
```

Veja Também:

HARDCR( ), MEMOEDIT( ), MEMOREAD( ), MEMOTRAN( ), MEMOWRIT( ), MLCOUNT( )

Função: MEMOREAD( ) – Leitura de um Arquivo Texto:

Forma:

MEMOREAD(<expressão caractere>)

Propósito:

Lê o conteúdo de um arquivo-texto (padrão ASCII) contido no disco, retornando-o em uma cadeia de caracteres:

Argumentos:

<*expressão caractere*>: define o nome do arquivo-texto a ser lido do disco. Deve ser incluída a extensão e opcionalmente o drive e o diretório onde o mesmo se localiza.

Utilização:

A função MEMOREAD( ) lê e retorna para uma variável o conteúdo de um arquivo texto (padrão ASCII) que exista no disco, através de uma cadeia de caracteres. Sua principal utilização é ler arquivos-texto do disco e transformá-los em conteúdos de campos memo de arquivos de dados (.dbf), ou em conteúdos de variáveis de memória tipo caractere.
O tamanho máximo de arquivo que pode ser lido pela função MEMOREAD( ) é de 65.535 bytes (64K), equivalente ao tamanho máximo que uma variável caractere ou campo memo pode assumir no Clipper.
Através deste comando pode-se ler qualquer arquivo-texto do disco, armazená-lo em campos memo, editá-lo ou formatá-lo através das funções MEMO( ) e suas variações.

Biblioteca:

EXTEND.LIB ou
MEMO.LIB (nas versões anteriores à Summer 87)

Exemplo:

* Suponho o arquivo Mala.dbf, que contém o campo memo obs em sua estrutura, e o arquivo-texto dados.txt, que contém dados a serem lidos para o campo memo em questão, pode-se utilizar a função MEMOREAD( ) para realizar tal tarefa:

```
USE Mala INDEX Indcod
INPUT "Digite o Codigo:" TO vcod
SEEK vcod
REPLACE obs WITH MEMOREAD("dados.txt")
```

Para atribuir um arquivo texto à uma variável:

```
vtexto = MEMOREAD("dados.txt")
```

Para ler um arquivo texto do disco, editá-lo e novamente gravá-lo, ou seja, construir um editor de textos tipo "MODIFY COMMAND" utilize a seguinte rotina:

```
PROCEDURE Editor
PARAMETERS arquivo
MEMOWRIT(arquivo, MEMOEDIT(MEMOREAD(arquivo)))
RETURN
```

(Dá para sentir o poder do Clipper, não dá !?)

Veja Também:

REPLACE, HARDCR( ), MEMOEDIT( ), MEMOLINE( ), MEMOTRAN( ), MEMOWRIT( ), MLCOUNT( )

Função: MEMOTRAN( ) – Tranforma campo memo:

Forma:

MEMOTRAN(<exp.caractere 1>
[,<exp.caractere 2>[,<exp.caractere 3>]])

Propósito:

Retorna uma expressão caractere com os controles de mudança de linha, gerados pela edição através da função MEMOEDIT( ), substituídos por outros que forem especificados.

Argumentos:

<expressão caractere 1>: é a cadeia de caracteres ou campo memo a ter os caracteres de mudança de linha substituídos.
<expressão caractere 2>: é o novo caractere que deve entrar no lugar da mudança de linha física ("hard carriage return/line feed").
<expressão caractere 3>: é o novo caractere que deve entrar no lugar da mudança de linha lógica ("soft carriage return/line feed").

Utilização:

A função MEMOTRAN( ) permite que sejam trocados os caracteres de controle de edição (mudanças de linha) gerados pela função MEMOEDIT( )

por outros que sejam especificados. É utilizada, por exemplo, para gerar um arquivo texto normal a partir do conteúdo de um campo memo.
Esta função pode ser utilizada também para permitir a formatação da impressão de campos memo, onde se deseja imprimi-los como se fossem cadeias normais de caracteres, eliminando assim o caracteres de controle de edição gerados pela função MEMOEDIT( ).
O comando REPORT FORM, por exemplo, requer que os campos memo sejam tranformados, através da substituição dos caracteres de controle de mudança de linha, para imprimí-los corretamente. Não sendo especificadas as expressões 2 e 3 o MEMOTRAN( ) realiza automaticamente a transformação de acordo com a necessidade do REPORT FORM.
Se a função MEMOTRAN( ) for especificada dentro de um arquivo de definição de relatório (.frm) a ser impresso via REPORT FORM e não for utilizada em qualquer outro programa do sistema, é necessário declará-la como externa através do comando EXTERNAL. Dessa forma tanto o compilador como o link-editor a tratarão corretamente, de acordo com suas necessidades.
Os caracteres de edição da função MEMOEDIT( ) serão substituídos de acordo com o seguinte, se não forem determinados caracteres específicos:

Mudança de linha física ("hard carriage return/line feed") através de CHR(13)+CHR(10) serão substituídos por um ponto-e-vírgula.
Mudança de linha lógica ("soft carriage return/line feed") através de CHR(141)+CHR(10) serão substituídos por um espaço em branco.

Se forem determinados caracteres específicos, os controles da função MEMOEDIT( ) serão substituídos da seguinte forma:

Mudança de linha física ("hard carriage return/line feed") através de CHR(13)+CHR(10) serão substituídos pelo caractere especificado na <expressão caractere 2>.
Mudança de linha lógica ("soft carriage return/line feed") através de CHR(141)+CHR(10) serão substituídos pelo caractere especificado na <expressão caractere 3>.

Biblioteca:

EXTEND.LIB ou
MEMO.LIB (nas versões anteriores à Summer 87)

Exemplo:

* Para eliminar todos os caracteres de controle gerados pelo comando MEMOEDIT( ) de um campo memo chamado obs, substituindo-os por espaços, e gerar uma cadeia de caracteres "limpa", pode-se utilizar:

vobs = MEMOTRAN(obs," "," ")

* Para eliminar os caracteres de formatação de uma cadeia de caracteres, editada através da função MEMOEDIT( ), a ser armazenada em um campo memo:

REPLACE notas WITH MEMOTRAN(notas, " "," ")

Veja Também:

REPLACE, HARDCR( ), MEMOEDIT( ), MEMOLINE( ), MEMOREAD( ), MEMOWRIT( ), MLCOUNT( )

Função: MEMOWRIT( ) - Grava arquivo texto:

Forma:

MEMOWRIT(<exp.caractere 1>,<exp.caractere 2>)

Propósito:

Grava um arquivo-texto no disco, a partir de uma cadeia de caracteres ou campo memo especificado, e retorna o valor lógico verdadeiro (.T.) se a operação for bem sucedida, ou falso (.F.), caso contrário.

Argumentos:

<*exp.caractere 1*>: é o nome do arquivo a ser gravado. Deve ser incluída a extensão e opcionalmente o drive e diretório correspondentes, caso não sejam o atual.

<*exp.caractere 2*>: é a cadeia de caracteres a ser gravada no arquivo especificado pela <expressão caractere 1>.

Utilização:

A função MEMOWRIT( ) permite gravar um arquivo-texto no disco a partir de um campo memo ou de uma cadeia de caracteres extensa. O arquivo gravado (no padrão ASCII) passa a ter o conteúdo do campo ou da variável especificada. Se a operação for bem sucedida MEMOWRIT( ) retorna o valor lógico .T. (verdadeiro), caso contrário .F. (falso), indicando que houve algum erro.

Esta função deve ser utilizada em conjunto com as outras funções tipo MEMO( ), quando atinge o máximo de seu poder, permitindo até mesmo a criação de um editor de textos dentro do Clipper.

Biblioteca:

EXTEND.LIB ou
MEMO.LIB (nas versões anteriores à Summer 87)

Exemplo:

* Suponha o campo memo obs, do arquivo Mala.dbf. Para gravar o seu conteúdo em um arquivo-texto chamado dados.txt utiliza-se:

```
USE Mala INDEX Indcod
INPUT "Digite o Código Desejado:" TO vcod
SEEK vcod
arqtxt = MEMOWRIT("dados.txt",obs)
IF arqtxt  && arqtxt = .T., gravação foi bem sucedida
   RUN (nome do editor de textos) dados.txt
    * utilize o seu editor de textos preferido para
   * editar o arquivo.
   REPLACE obs WITH MEMOREAD("dados.txt")
    * substitui o conteúdo do campo memo pelo arquivo
   * editado.
ELSE
   ? "Erro de gravação"
ENDIF
```

Veja Também:

HARDCR( ), MEMOEDIT( ), MEMOLINE( ), MEMOREAD( ) MEMOTRAN( ), MLCOUNT( )

Função: MLCOUNT( ) – Linhas de Campo Memo:

Forma:

MLCOUNT(<expressão caractere>,<expressão numérica>)

Propósito:

Retorna o número de linhas de uma cadeia de caracteres extensa ou de um campo memo, de acordo com o número de caracteres por linha especificado. Disponível a partir da versão Summer 87.

Argumentos:

<expressão caractere>: é a cadeia de caracteres extensa ou campo memo a ter suas linhas contadas.

<expressão numérica>: define o número de caracteres a ser considerado por linha.

Utilização:

A função MLCOUNT( ) conta o número de linhas em uma cadeia de caracteres extensa ou em um campo memo, baseando-se no número especificado de caracteres por linha. Se uma palavra tiver que ser quebrada ao final de uma linha será transferida para a linha seguinte.
MLCOUNT( ) deve ser utilizada em conjunto com a função MEMOLINE( ) para formatar um campo memo ou uma cadeia de caracteres extensa a ser enviada para a impressora. Para isso deve-se inicialmente utilizar MLCOUNT( ) para determinar o número de linhas a serem impressas. Em seguida, utilizando MEMOLINE( ), extrair cada linha do campo memo até que todas tenham sido impressas.

Biblioteca:

EXTEND.LIB

Exemplo:

```
numlin = MLCOUNT(memo, 60)
? numlin
```

Veja Também:

HARDCR( ), MEMOEDIT( ), MEMOLINE( ), MEMOREAD( ), MEMOTRAN( ), MEMOWRIT( )

Função: NDX( ) – Arquivo de Indice

Forma:

NDX(<exp.numérica>)

Propósito:

Manter a compatibilidade com o dBASE III Plus; supõe retornar o nome do n-ésimo índice aberto em conjunto com um arquivo de dados.

Utilização:

A função NDX( ) foi apenas simulada nas extensões do Clipper (versões Winter 85 e Autumn 86) para manter a compatibilidade com o dBASE III Plus que a possui. Seu código-fonte encontra-se no arquivo EXTENDDB.prg e o respectivo código-objeto no arquivo EXTENDDB.obj.
Para utilizá-la o arquivo EXTENDDB.obj deve ser incluído na lista de módulos-objeto passada ao link-editor.
A função NDX( ) apenas retorna "NTX" e o número do índice especificado. Sua utilização é unicamente para simular a compatibilidade entre o Clipper e o dBASE III Plus.

Exemplo:

```
USE Mala INDEX Indcod, Indnome, Indcep
? NDX(1)    && retorna NTX1
? NDX(2)    && retorna NTX2
? NDX(3)    && retorna NTX3
```

Veja Também:

INDEXKEY( ), USE, SET INDEX, SET ORDER, INDEX

Função: RECCOUNT( ) – Número de Registros:

Forma:

RECCOUNT( )

Propósito:

Retorna o número total de registros existentes no arquivo de dados em uso na área de trabalho selecionada.

Utilização:

A função RECCOUNT( ) é idêntica à função LASTREC( ) e permite determinar o número total de registros existentes em um arquivo. É muito mais

rápida que executar um tradicional GO BOTTOM e um reg=RECNO( ) para determinar o número total de registros de uma arquivo.
Se nenhum arquivo estiver em uso RECCOUNT( ) retornará zero.
RECCOUNT( ) retorna o número total de registros, mesmo que o SET DELETED estiver ON ou um filtro estiver estabelecido através do comando SET FILTER.
RECCOUNT( ) ou LASTREC( ), em conjunto com DISKSPACE( ), RECSIZE( ) e HEADER( ) podem ser utilizadas para criar uma rotina de Back-Up de arquivos, que funcione mesmo quando um arquivo tiver que ser segmentado em vários disquetes.

Exemplo:

```
USE Mala INDEX Indcod
? RECCOUNT( )  && retorna o número total de registros
               && do arquivo Mala.
```

Veja Também:

LASTREC( ), RECSIZE( ), DISKPACE( ), HEADER( ), RECNO( )

Função: RECNO( ) – Número do registro:

Forma:

RECNO( )

Propósito:

Retorna um valor inteiro, representando o número do registro onde está atualmente posicionado o apontador de registros do arquivo de dados aberto na área de trabalho selecionada.

Argumentos:

Nenhum

Utilização:

A função RECNO( ) é utilizada para se ter acesso ao número do registro do arquivo em uso onde está posicionado o apontador de registros.
Se um arquivo não contiver registros RECNO( ) será igual a 1 (um) e tanto a função EOF( ) como a BOF( ) serão verdadeiras (.T.).

Se em uma operação de movimentação pelo arquivo o apontador de registros passar além do último registro, função EOF( ) (fim de arquivo) será verdadeira (.T.) e RECNO( ) retornará o número do último registro acrescido de 1 (LASTREC( )+1). Se, pelo contrário, o apontador de registros passar além do primeiro registro do arquivo, a função BOF( ) (início de arquivo) será verdadeira (.T.) e RECNO( ) retornará 1 (um).
O valor do número do registro pode ser normalmente armazenado em uma variável.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
USE Mala
? RECNO()     && retorna 1, pois o arquivo acabou de ser
              && aberto
? BOF()       && retorna .F. (falso)
SKIP -1
? RECNO()     && retorna 1
? BOF()       && retorna .T. (verdadeiro)
GO BOTTOM
? RECNO()     && retorna, por exemplo, 100
? EOF()       && retorna .F. (falso)
SKIP
? RECNO()     && retorna 101 (um a mais)
? EOF()       && retorna .T. (verdadeiro)


GOTO 10
reg = RECNO()
? RECNO()     && retorna 10
? reg         && retorna 10
```

Veja Também:

RECCOUNT( ), LASTREC( ), BOF( ), EOF( )

Função: RECSIZE( ) – Tamanho do Registro

Forma:

RECSIZE( )

Propósito:

Retorna em número de bytes, o tamanho do registro do arquivo de dados em uso na área de trabalho selecionada.

Argumentos:

Nenhum

Utilização:

A função RECSIZE( ) retorna em número de bytes o tamanho do registro do arquivo de dados que estiver em uso na área de trabalho atualmente selecionada.
Utilizando-a em conjunto com as funções LASTREC( ), DISKSPACE( ) e HEADER( ) é possível se determinar a quantidade de registros de um arquivo que pode ser copiada para um disquete sem que seja ultrapassada sua capacidade de armazenamento. Com estas funções pode-se construir rotinas Back-Up (cópias de segurança) de arquivos que ocupem diversos disquetes.
RECSIZE( ) determina o tamanho de cada registro somando o tamanho de cada campo e adicionando 1 caractere ao total (correspondente ao asterisco que indica quando um registro está marcado para eliminação). Se multiplicarmos o valor de RECSIZE( ) pela quantidade de registros do arquivo retornada por LASTREC( ), obteremos o espaço total ocupado pelos registros do arquivo no disco. Para determinar o tamanho total do arquivo basta adicionarmos ao produto obtido o tamanho do seu cabeçalho, fornecido pela função HEADER( ):

tamanho do arquivo = (RECSIZE( )*LASTREC( ))+HEADER( )

Nas versões anteriores à Summer 87, a função RECSIZE( ) está contida no módulo-objeto auxiliar EXTENDC.OBJ. Para utilizá-la o arquivo EXTENDC.obj deve ser incluído na lista de módulos-objeto passada ao link-editor e deve ser utilizado o comando EXTERNAL para declará-la como sendo externa ao compilador do Clipper.

Biblioteca:

EXTEND.LIB
EXTENDC.obj (nas versões anteriores à Summer 87)

Exemplos:

```
USE Mala
? RECSIZE()  && retorna, por exemplo 304 bytes

* Exemplo de uma rotina para Back-up de arquivos em
vários disquetes, sem utilzar o comando BACKUP do DOS:

EXTERNAL HEADER, RECSIZE, DISKSPACE
 * São todas rotinas escritas em "C", que devem ser
 * declaradas externas ao compilador do Clipper se você
* estiver utilizando uma versão anterior à Summer 87
USE Arquivo
DO WHILE .NOT. EOF()
   @ 23,22 SAY "Coloque um disquete Back Up no drive A"
   @ 24,22 SAY "     Tecle (Enter) quando pronto..."
   WAIT ""
   COPY NEXT (DISKSPACE(1)-HEADER())/RECSIZE() TO A:Backup
   * determina a quantidade de registros que poderão ser
   * copiados para o disquete sem "enche-lo".
   SKIP
ENDDO
CLEAR
@ 23,33 SAY "Arquivo Salvo !"
USE
RETURN
```

Veja Também:

DISKSPACE( ), LASTREC( ), HEADER( ), FIELD( )/FIELDNAME( )

Função: RLOCK( )/LOCK( ) – Registro Restrito a Acesso:

Forma:

RLOCK( ) ou LOCK( )

Propósito:

Restringe o acesso ao registro onde está atualmente posicionado o apontador de registros em um ambiente multiusuário de rede, retornando verdadeiro (.T.) se a restrição foi realizada ou falso (.F.), caso contrário.

Argumentos:

Nenhum

Utilização:

A função RLOCK( ) restringe o acesso ao registro corrente da área de trabalho selecionada aos outros usuários da rêde, retornando verdadeiro (.T.) se a restrição foi bem sucedida ou (.F.) caso contrário.
O registro permanecerá restrito ao usuário que executou o comando RLOCK( ) até que outro registro tenha seu acesso restringido ou sejam executados os comandos UNLOCK, CLOSE DATABASE ou a função FLOCK( ).
RLOCK( ) restringe o acesso a apenas um registro do arquivo de cada vez. Ao se restringir o acesso a outro registro o anterior será automaticamente liberado.
A função RLOCK( ) não restringe o acesso a registros de arquivos abertos em outras áreas de trabalho, mesmo que haja uma relação estabelecida entre eles pelo comando SET RELATION.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
USE Mala INDEX Indcod
SEEK "12234"
IF FOUND()
   IF RLOCK()
      DELETE
   ELSE
      ? "Acesso Compartilhado, impossível eliminar !"
   ENDIF
ELSE
      ? "Código Não Encontrado !"
ENDIF
RETURN
```

Veja Também:

USE...EXCLUSIVE, SET EXCLUSIVE, UNLOCK, FLOCK( )

Função: SELECT( ) – Área Selecionada:

Forma:

SELECT([<exp.caractere>])

Propósito:

Retorna o número da área de trabalho atualmente selecionada ou o número da área cujo nome do "alias" foi especificado.
Disponível a partir da versão Summer 87.

Argumentos:

<expressão caractere>: define, opcionalmente, o nome do "alias" da área desejada.

Utilização:

A função SELECT( ) retorna um valor numérico entre 0 e 254 representando o número da área de trabalho atualmente selecionada. Se não houver um arquivo de dados em uso na área selecionada, a função SELECT( ), da mesma forma, retornará o número desta área.
Caso seja especificada através da <expressão caracter> o nome do "alias" da área desejada, a função SELECT( ) retornará o número da área de trabalho que possui aquele "alias". Se o alias não existir SELECT( ) retornará zero.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
SELECT 1
USE Mala ALIAS Direta
? SELECT()    && retorna 1
SELECT 4
? SELECT()    && retorna 4
? SELECT("Direta")
* retorna 1
```

Veja Também:

SELECT, USE, ALIAS( )

Função: SOUNDEX( ) – Índice pela semelhança sonora:

Forma:

SOUNDEX(<exp.caractere>)

Propósito:

Retorna um código no formato A9999 a partir de uma cadeia de caracteres (normalmente um nome) que é semelhante para cadeias com sons parecidos.

Argumentos:

<*expressão caractere*>: é a cadeia de caracteres a ser convertida para o código sonoro da função SOUNDEX( ).

Utilização:

A função SOUNDEX( ) retorna um código de cinco caracteres (no formato A999) a partir de uma expressão caractere (usualmente um nome). É usada para classificação e pesquisa de nomes com sons parecidos, quando não se conhece extamente o nome procurado.
Em aplicações de Mala-Direta, por exemplo, para se achar nomes ou endereços que tenham sons parecidos pode-se indexar o arquivo utilizando como chave o código gerado pela função SOUNDEX( ) e em seguida pesquisar nomes sem conhecê-los precisamente.
Nas versões anteriores à versão Summer 87, a função SOUNDEX( ) está contida no módulo-objeto auxiliar EXTENDDB.obj, que contém funções escritas em Clipper. Para utilizá-la, este módulo deverá estar presente na lista de módulos-objeto passada ao link-editor.

Biblioteca:

EXTEND.LIB
EXTENDDB.obj (nas versões anteriores à Summer 87)

Exemplos:

```
USE Mala
INDEX on SOUNDEX(nome) TO Indsom

vnome="JOSE
SEEK SOUNDEX(vnome)
? FOUND(), nome          && retorna .T. JOSE

vnome="JOSUE"
SEEK SOUNDEX(vnome)
? FOUND(), nome          && retorna .T. JOSE

SEEK("JOSEF")
? FOUND(), nome          && retorna .T. JOSE

* Todos geram o mesmo "código soundex".
```

Biblioteca:

CLIPPER.LIB
EXTENDDB.obj (nas versões anteriores à Summer 87)

Veja Também:

INDEX, SEEK, FIND, LOCATE, SET SOFTSEEK

Função: UPDATED( ) – READ Atualizado:

Forma:

UPDATED( )

Propósito:

Retorna verdadeiro (.T.) se o último comando READ alterou algum dado nos @...GET que foram editados.

Argumentos:

Nenhum

Utilização:

A função UPDATED( ) retorna o valor lógico verdadeiro (.T.) se no último comando READ, os campos ou variáveis que estavam sendo editados pelo comando @...SAY...GET à ele associados, tiverem sido alterados. Caso nenhum dado tenha sido alterado, o resultado da função UPDATED( ) será falso (.F.).
Cada vez que um comando READ é executado, inicialmente o valor da função UPDATED( ) é retornado para falso (.F.). Em seguida, qualquer @...GET que for alterado alternará esse valor para verdadeiro (.T.).
Ao se finalizar um comando READ, qualquer teste posterior poderá verificar o valor de UPDATED( ) para determinar se os dados foram alterados ou não pelo último READ executado.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
CLEAR
SET DATE BRITISH
USE Mala INDEX Indcod, Indnome
DO WHILE .T.
   vcod = SPACE(4)
   vnome=SPACE(40)
   vende=SPACE(40)
   vcida=SPACE(20)
   vesta=SPACE(2)
   vcep=SPACE(5)
   @ 05,32 SAY "Alteração de Dados"
   @ 06,32 SAY "_____________________"
   @ 10,20 SAY "Codigo..: " GET vcod  PICTURE "9999"
   @ 12,20 SAY "Nome....: " GET vnome PICTURE "@!"
   @ 14,20 SAY "Endereco: " GET vende PICTURE "@!"
   @ 16,20 SAY "Cidade..: " GET vcida PICTURE "@!"
   @ 18,20 SAY "Estado..: " GET vesta PICTURE "!!"
   @ 18,35 SAY "CEP: "      GET vcep  PICUTRE "99999"
   CLEAR GETS
   @ 10,20 SAY "Codigo..: " GET vcod PICTURE "9999"
   READ
   IF EMPTY(vcod)
      CLOSE DATABASE
      CLEAR
      RETURN
   ENDIF
   @ 23,00 CLEAR
   SEEK vcod
   IF .NOT. FOUND()
      @ 23,30 SAY "Codigo Não Cadastrado !"
      LOOP
   ENDIF
   vnome=nome
   vende=ende
   vcida=cida
   vesta=esta
   vcep = cep
   @ 12,30 GET vnome PICTURE "@!"
   @ 14,30 GET vende PICTURE "@!"
   @ 16,30 GET vcida PICTURE "@!"
   @ 18,30 GET vesta PICTURE "!!"
   @ 20,32 GET vcep  PICTURE "99999"
   READ
   IF UPDATED()
      REPLACE nome WITH vnome,ende WITH vende
      REPLACE cida WITH vcida,esta WITH vesta
      REPLACE cep  WITH vcep
   ENDIF
ENDDO
RETURN
```

Veja Também:

READ, @...SAY...GET

Função: USED( ) – Arquivo em uso:

Forma:

USED( )

Propósito:

Determinar se um arquivo de dados está em uso (aberto) na área de trabalho atualmente selecionada.
Disponível a partir da versão Summer 87.

Argumentos:

Nenhum

Utilização:

A função USED( ) retorna verdadeiro (.T.) se existir um arquivo de dados em uso (aberto) na área de trabalho atualmente selecionada e falso (.F.) caso contrário.
Ela é especialmente útil em ambiente multiusuário para verificar se um arquivo compartilhado foi aberto. A função NETERR( ) também poderá ser utilizada com o mesmo objetivo.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
SET EXCLUSIVE OFF
SELECT 3
USE Mala
IF USED()
     SET INDEX TO Indcod
ELSE
     ? "Impossível usar o arquivo.... Pressione uma Tecla"
     INKEY(0)
ENDIF
RETURN
```

Veja Também:

USE, SELECT, NETERR( ), SELECT( )

6.7. FUNÇÕES ESPECIAIS

Função: & – Macro Substituição:

Forma:

& <nome da variável caractere>

Propósito:

Embora a Macro não seja uma função propriamente dita, ela é assim considerada por conveniência. Seu propósito é substituir o nome de uma variável tipo caractere, diretamente pelo seu conteúdo em qualquer lugar que apareça.

Argumentos:

<nome da variável caractere>: é o nome da variável tipo caractere cujo conteudo deve ser substituído.

Utilização:

A função & (Macro) é um dos mais interessantes recursos de programação do Clipper (e também do dBASE); ela realiza a substituição do nome de uma variável caractere diretamente pelo seu conteúdo, permitindo grande flexibilidade de programação, principalmente na substituição de parâmetros.
A função Macro somente pode ser utilizada com variáveis tipo caractere.
Uma Macro pode ser utilizada para substituir constantes, nomes de variáveis, expressões completas (incluindo funções e operadores), títulos, condições no comando DO WHILE e Macros recurssivas (Macros dentro de Macros).
O Clipper necessita receber diretamente toda a sintaxe de um comando quando um programa é compilado, para poder validá-la. Portanto, comandos e argumentos dos comandos (opções separadas por vírgulas por exemplo) não são permitidos em Macros. Toda linha de um programa obrigatoriamente deve conter um comando válido e toda a pontuação (vírgulas) que faz parte da sintaxe do comando, caso contrário o compilador do Clipper retornará uma mensagem de erro de sintaxe.
Uma Macro, portanto, não pode substituir comandos ou partes de comandos, mas poderá ser usada para substituir parâmetros de comandos.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
* Utilização correta de Macros:

USE Mala INDEX Indnome
vnome="VIDAL"
FIND &vnome  && O Clipper irá entender FIND VIDAL

             * No Clipper o comando FIND deve sempre
             obrigatoriamente ser utilizado com a função
             Macro; contudo é sempre preferível
             utilizar-se o comando SEEK.

nome="Clipper"
titulo="&nome, Compilador do dBASE III"
? titulo && retorna "Clipper, Compilador do dBASE III"

arq="Estoq"
USE &arq     && O Clipper irá entender USE Estoq
             && &arq foi substituido por Estoq

cond="saldo<100"
LIST codigo,produto,saldo FOR &cond
* O Clipper irá entender:
* LIST codigo,produto,saldo FOR saldo<100

Utilização incorreta de Macros:

comando="USE Estoq"
&comando
* O Clipper irá acusar erro de Sintaxe

cond=codigo,produto,saldo FOR saldo<100"
LIST &cond
* O Clipper irá acusar erro de Sintaxe
```

Função: ALTD( ) — "Debugador":

Forma:

ALTD(<expressão numérica>)

Propósito:

Invoca (executa) o "Debugador" (depurador de erros) do Clipper ou habilita ou desabilita o uso das teclas <Alt-D> para invocá-lo.
Disponível a partir da versão Summer 87.

Argumentos:

<expressão numérica>: é um número inteiro de 0 a 3 que define o modo de chamada do debugador.

Utilização:

O "Debugador" do Clipper (veja a seção 9.2) é um utilitário extremamente útil e interessante. Ao ser link-editado juntamente com o sistema desenvolvido facilita a correção de erros, através dos diversos recursos que oferece. A função ALTD( ) define o modo de utilização do debugador de acordo com o argumento especificado:

Valor do Argumento	Descrição
0	Desabilita <Alt-D> para invocar o debugador
1	Habilita <Alt-D> para invocar o debugador
2	Invoca o debugador na ocorrência de um erro
3	Invoca o debugador apresentando as variáveis

Se nenhm argumento for especificado a função ALTD( ) invoca o debugador trazendo a última tela apresentada.

Biblioteca:

CLIPPER.LIB

Veja Também:

SETCANCEL( ), SET ESCAPE

Função: FCLOSE( ) – Fechar Arquivo DOS:

Forma:

FCLOSE(<expressão numérica>)

Propósito:

Fechar um arquivo aberto, gravando os buffers do DOS no disco.

Esta função permite acesso de baixo nível aos arquivos DOS e por esse motivo deve ser utilizada com extremo cuidado, requerendo conhecimento detalhado sobre o funcionamento do Sistema Operacional.
Disponível a partir da versão Summer 87.

Argumentos:

<*expressão numérica*>: identifica o arquivo sendo manipulado, devendo seu valor ser previamente obtido a partir das funções FOPEN( ) ou FCREATE( ).

Utilização:

A função FCLOSE( ) permite acessar diretamente as rotinas de manipulação de arquivos de baixo nível do DOS. Deve ser utilizada em conjunto com as funções FOPEN( ) e FCREATE( ).
FCLOSE( ) retorna falso (.F.) se um erro ocorrer durante a operação de fechamento do arquivo, caso contrário retorna verdadeiro (.T.), indicando que a gravação foi bem sucedida.

Biblioteca:

EXTEND.LIB

Exemplo:

```
arq = FCREATE("TESTE",0)
FCLOSE(arq)
```

Veja Também:

FCREATE( ), FERROR( ), FOPEN( ), FREAD( ), FREADSTR( ), FSEEK( ), FWRITE( )

Função: FCREATE( ) — Criação de Arquivos DOS:

Forma:

FCREATE(<expressão caractere> [,<expressão numérica>])

Propósito:

Criar um novo arquivo ou truncar um arquivo existente deixando-o com 0 bytes.
Esta função permite acesso de baixo nível aos arquivos DOS e por esse motivo deve ser utilizada com extremo cuidado, requerendo conhecimento detalhado sobre o funcionamento do Sistema Operacional.
Disponível a partir da versão Summer 87.

Argumentos:

<*expressão caractere*>: é o nome do arquivo a ser criado.

<*expressão numérica*>: é um atributo de arquivo do DOS. Se omitido será considerado zero. Os atributos possíveis são:

Valor	Atributo	Descrição
0	Normal	Leitura e Gravação
1	Apenas Leitura	Tentativa de gravação retorna um erro
2	Escondido	Não é apresentado no diretório
4	Sistema	Não é apresentado no diretório

Utilização:

A função FCREATE( ) permite acessar as rotinas de baixo nível do DOS para a manipulação de arquivos. Retorna o número de manipulação de arquivos do DOS do novo arquivo criado. Este número está compreendido entre zero e 65.535. Se um erro ocorrer será retornado o valor −1.
Quando um novo arquivo é criado através de FCREATE( ), permanecerá aberto no modo 2 do DOS (compartilhado e permitindo leitura e gravação).
Como FCREATE( ) age diretamente sobre o DOS, os comandos SET DEFAULT e SET PATH são ignorados. Para criar arquivos em outros diretórios do DOS, que não o atual, estes devem ser explicitamente especificados.
Para a utilizarão das outras funções de manipulação de arquivos é necessário armazenar o número do arquivo para o DOS. Por esse motivo uma variável deve sempre ser criada para receber o valor retornado por FCREATE( ), para posterior identificação e uso do arquivo criado.

Biblioteca:

EXTEND.LIB

Veja Também:

FCLOSE( ), FERROR( ), FOPEN( ), FREAD( ), FREADSTR( ), FSEEK( ), FWRITE( )

Função: FERROR( ) – Erro de Arquivo DOS:

Forma:

FERROR( )

Propósito:

Testa se houve erro do DOS após o término da operação de uma função de manipulação de arquivos, retornando um valor numérico representando o código de erro do DOS.
Esta função permite acesso de baixo nível aos arquivos DOS e por esse motivo deve ser utilizada com extremo cuidado, requerendo conhecimento detalhado sobre o funcionamento do Sistema Operacional.
Disponível a partir da versão Summer 87.

Argumentos:

Nenhum

Utilização:

A função FERROR( ) retorna o número do erro do DOS após a última operação de arquivo realizada. Se não ocorrer erro será retornado o valor zero (0).

Biblioteca:

EXTEND.LIB

Exemplo:

```
arq = FCREATE("TESTE.arq")
IF FERROR()()0
  ? "Erro na Criação do Arquivo. Código DOS:",FERROR()
ENDIF
```

Veja Também:

FCLOSE( ), FCREATE( ), FOPEN( ), FREAD( ), FREADSTR( ), FSEEK( ), FWRITE( )

Função: FOPEN( ) – Abre Arquivo DOS:

Forma:

FOPEN(<exp.caractere> ,<exp.numérica>)

Propósito:

Abre um arquivo, retornando o respectivo número de manipulação do DOS. Esta função permite acesso de baixo nível aos arquivos DOS e por esse motivo deve ser utilizada com extremo cuidado, requerendo conhecimento detalhado sobre o funcionamento do Sistema operacional.
Disponível a partir da versão Summer 87.

Argumentos

<expressão caractere>: define o nome do arquivo a ser aberto, incluindo o diretório, se não for o atual.
<expressão numérica>: é o modo de abertura do DOS, indicando como o arquivo poderá ser acessado:

Modo de Abertura	Operação
0	Apenas leitura
1	Apenas escrita
2	Leitura e escrita

Se não for especificado será assumido o modo zero (0).

Utilização:

A função FOPEN( ) abre um arquivo, de acordo com o modo do DOS especificado, e retorna o número de manipulação do arquivo entre 0 e 65.535. Se ocorrer um erro será retornado o valor -1.
Como o número de manipulação do arquivo é necessário para identificar o arquivo para as outras funções de arquivo DOS, o valor retornado pela função

FOPEN( ) sempre deverá ser armazenado em uma variável de memória para posterior utilização.
Como FOPEN( ) é uma função que acessa diretamente o DOS, os parâmetros definidos pelos comandos SET DEFAULT e SET PATH são ignorados. Apenas o diretório atual do DOS será pesquisado, a menos que o nome do diretório seja explicitamente especificado.

Biblioteca:

EXTEND.LIB

Exemplo:

```
arq = FOPEN("TESTE.arq")
IF FERROR()()0
  ? "Erro de Abertura, Código DOS:",FERROR()
ENDIF
```

Veja Também:

FCLOSE( ), FCREATE( ), FERROR( ), FREAD( ), FREADSTR( ), FSEEK( ), FWRITE( )

Função: FREAD( ) - Lê Arquivo DOS:

Forma:

FREAD(<exp.num.1>@<variável caractere>,<exp.num.2>)

Propósito:

Ler caracteres de um arquivo para uma variável de memória tipo caractere, retornando o número de bytes lidos.
Esta função permite acesso de baixo nível aos arquivos do DOS e por esse motivo deve ser utilizada com extremo cuidado, requerendo conhecimento detalhado sobre o funcionamento do Sistema Operacional.
Disponível a partir da versão Summer 87.

Argumentos:

<exp.numérica 1>: é o número de manipulação do arquivo a ser lido, obtido a partir das funções FOPEN( ), FCREATE( ), ou pré-definido pelo DOS.

<variável caractere>: é o nome de uma variável caractere existente, passada por referência (através do prefixo @), para ser utilizada como um buffer. O tamanho (comprimento) desta variável deve ser no mínimo o mesmo definido pela <exp.numérica 2>.

<exp.numérica 2>: é o número de bytes a serem lidos para o buffer, inicindo-se da localização atual do ponteiro do DOS. O valor retornado pela função FREAD( ), quando bem sucedida a operação, deve ser igual ao valor da <exp.numérica 2>.

Utilização:

A função FREAD( ) lê diretamente, através do DOS, caracteres de um arquivo para uma variável de memória, retornando o número de bytes corretamente lidos. Se houver algum erro de leitura ou for encontrado o fim-do-arquivo será retornado o valor zero (0).
FREAD( ) lê o arquivo a partir da posição atual do ponteiro de arquivos do DOS. FREAD( ) lê qualquer tipo de caracter, incluindo caracteres de controle, o caractere nulo ("") ou caracteres com código ASCII/IBM-ECS acima de 128.
Para reposicionar o ponteiro de arquivos do DOS sem fazer uma leitura utiliza-se a função FSEEK( ).

Biblioteca:

EXTEND.LIB

Exemplo:

O exemplo a seguir lê um bloco do DOS com 128 bytes:

```
bloco  = 128
buffer = SPACE(512)
arq    = FOPEN("TESTE.arq")
*
IF FERROR()0
   bytes=FREAD(arq,@buffer,bloco)
   IF bytes()bloco
      ? "Erro de leitura !"
   ENDIF
ENDIF
```

Veja Também:

FCLOSE( ), FCREATE( ), FERROR( ), FOPEN( ), FREADSTR( ), FSEEK( ), FWRITE( )

Função: FREADSTR( ) – Lê Caracteres de Arquivo DOS:

Forma:

FREADSTR(<exp.numérica 1>,<exp.numérica 2>)

Propósito:

Ler caracteres de um arquivo DOS.
Esta função permite acesso de baixo nível aos arquivos DOS e por esse motivo deve ser utilizada com extremo cuidado, requerendo conhecimento detalhado sobre o funcionamento do Sistema Operacional.
Disponível a partir da versão Summer 87.

Argumentos:

<exp.numérica 1>: é o número de manipulação do arquivo DOS, obtido a partir das funções FOPEN( ), FCREATE( ), ou pré-definido pelo DOS.
<exp.numérica 2>: é o número de bytes a serem lidos iniciando-se a partir da posição atual do ponteiro de arquivos do DOS. Este número pode ser positivo ou negativo, dependendo da direção (para frente ou para trás) a ser lida, a partir da atual posição do ponteiro de arquivos do DOS.

Utilização:

A função FREADSTR( ) retorna uma cadeia de caracteres de até 65.535 bytes (64K). Se for retornado um valor nulo ("") indicará que houve erro ou que foi encontrado o fim-do-arquivo.
READSTR( ) lê, a partir da atual posição do ponteiro de arquivos do DOS, o número de bytes (caracteres) especificados pela <exp.numérica 2> ou até que um caractere nulo (ASCII 0) seja encontrado. Como acontece em FREAD( ), todos os caracteres são lidos, incluindo os de alto-nível (acima do ASCII 128) e os de controle.

Biblioteca:

EXTEND.LIB

Exemplo:

O exemplo a seguir apresenta o valor ASCII dos primeiros 32 bytes lidos de um arquivo texto, TESTE.txt:

```
arq = FOPEN("TESTE.txt")
IF FERROR()<>0
   ? "Erro na abertura do Arquivo !"
   RETURN
ELSE
   buffer = FREADSTR(arq,32)
   ? "Tamanho: ", LEN(buffer)
   ?
   FOR i = 1 TO LEN(buffer)
       ?? TRANSFORM(ASC(SUBSTR(buffer,i,1)),"99")
   NEXT
   FCLOSE(arq)
ENDIF
RETURN
```

Veja Também:

FCLOSE( ), FCREATE( ), FERROR( ), FOPEN( ), FREAD( ), FSEEK( ), FWRITE( )

Função: FSEEK( ) — Posicionamento do Ponteiro de Arquivos do DOS:

Forma:

FSEEK(<exp.num.1>,<exp.num.2>[,<exp.num.3>])

Propósito:

Movimentar o ponteiro de arquivos do DOS para uma nova posição em um arquivo.
Esta função permite acesso de baixo nível aos arquivos DOS e por esse motivo deve ser utilizada com extremo cuidado, requerendo conhecimento detalhado sobre o funcionamento do Sistema Operacional.
Disponível a partir da versão Summer 87.

Argumentos

<exp.numérica 1>: é o número de manipulação do arquivo, obtido a partir das funções FOPEN( ), FCREATE( ) ou pré-definido pelo DOS.

<exp.numérica 2>: é o número de bytes a ser deslocado o ponteiro de arquivos, a partir da posição atual, de acordo com a <exp.numérica 3>. Este número poderá ser positivo ou negativo, dependendo da direção a ser movido o ponteiro (para frente ou para trás).

<exp.numérica 3>: define o método de deslocamento do ponteiro, sendo indicado por um dos valores a seguir. O método padrão assumido pelo Clipper é o zero (0).

Método	Descrição
0	A partir do início do arquivo.
1	A partir da posição atual.
2	A partir do fim do arquivo.

Utilização:

A função FSEEK( ) é utilizada para deslocar o ponteiro de arquivos do DOS, retornando sua nova posição (em número de bytes) relativa ao início do arquivo.

Biblioteca:

EXTEND.LIB

Exemplo:

```
arq = FOPEN("TESTE.txt")
*
* Determinando o tamanho do arquivo
tamanho = FSEEK(arq,0,2)
*
* Retornar à posição original do ponteiro
FSEEK(arq,0)
```

Veja Também:

FCLOSE( ), FCREATE( ), FERROR( ), FOPEN( ), FREAD( ), FREADSTR( ), FWRITE( )

Função: FWRITE( ) – Escrever um buffer em um Arquivo DOS:

Forma:

FWRITE(<exp.num.1>,<variável caractere>[,<exp.num.2>])

Propósito:

Escrever/gravar um variável buffer em um arquivo DOS especificado, retornando o número de bytes gravados.

Esta função permite acesso de baixo nível aos arquivos DOS e por esse motivo deve ser utilizada com extremo cuidado, requerendo conhecimento detalhado sobre o funcionamento do Sistema Operacional.
Disponível a partir da versão Summer 87.

Argumentos:

<*exp.numérica 1*>: é o número de manipulação do arquivo obtido a partir das funções FOPEN( ), FCREATE( ), ou pré-definido pelo DOS.
<*variável caractere*>: é uma variável de memória caractere já existente a ser utilizada como um buffer de saída.
<*exp.numérica 2*>: define o número de bytes a serem gravados da variável de buffer no arquivo, a partir da atual posição do ponteiro de arquivos do DOS. Se não especificada todo o conteúdo da variável de buffer será gravado/escrito no arquivo.

Utilização:

A função FWRITE( ) grava caracteres em um arquivo a partir de uma variável utilizada como buffer, retornando o número de bytes escritos.
Se o valor retornado for igual a zero indicará que ocorreu algum erro durante a operação. Este erro poderá ser determinado através de FERROR( ) que retornará o código de erro do DOS correspondente ao erro ocorrido. Quando a gravação é bem sucedida o valor retornado por FWRITE( ) deverá ser igual ao valor da <exp.numérica 2>.

Biblioteca:

EXTEND.LIB

Exemplo:

O exemplo a seguir mostra a leitura de caracteres a partir de um arquivo (LE.txt) e a gravação dos mesmos caracteres em outro (GRAVA.txt):

```
buffer = SPACE(512)
arqent = FOPEN("LE.txt")
* Criação do Arquivo
arqsai = FCREATE("GRAVA.txt")
IF FERROR()<>0
   ? "Erro de criação, Código DOS:", FERROR()
   RETURN
ENDIF
* Leitura do Arquivo
FREAD(arqent,@buffer,512)
IF FERROR()<>0
   ? "Erro de leitura, Código DOS:", FERROR()
   RETURN
```

```
ENDIF
* Gravação no Arquivo
FWRITE(arqsai,buffer,512)
IF FERROR()<>0
   ? "Erro de gravação, Código DOS:", FERROR()
ENDIF
FCLOSE(arqent)
FCLOSE(arqsai)
RETURN
```

Veja Também:

FCLOSE( ), FCREATE), FERROR( ), FOPEN( ), FREAD( ), FREADSTR( ), FSEEK( )

Função: IF( )/IIF( ) – Decisão Condicional:

Forma:

IF(<exp.lógica>,<exp.1>,<exp.2>)
ou
IIF(<exp.lógica>,<exp.1>,<exp.2>)

Propósito:

Permite o processamento condicional de uma expressão ou de outra, dependendo de uma condição especificada. Retorna o resultado de uma das duas <expressões>, dependendo da avaliação da <expressão lógica> definida.

Argumentos:

<*expressão lógica*>: define uma expressão lógica a ser avaliada como verdadeira (.T.) ou falsa (.F.).
<*expressão.1*> é a expressão a ser processada caso o resultado da <expressão lógica> seja verdadeiro (.T.).
<*expressão 2*>: é a expressão a ser processada caso o resultado da <expressão lógica> seja falso (.F.).
As expressões 1 e 2 poderão ser expressões de qualquer tipo de dado (caractere, numérico ou data), não precisando ser ambas do mesmo tipo de dado.

Utilização:

A função IF( ) ou IIF( ) é utilizada para o processamento condicional de uma <expressão>, de acordo com uma condição especificada através de uma

<expressão lógica>. Se a <expressão lógica> for avaliada como verdadeira (.T.) a <expressão 1> será processada; caso contrário, se a <expressão lógica> for avaliada como falsa (.F.) a <expressão 2> será processada.
A <expressão lógica> pode ser qualquer expressão lógica válida, como por exemplo: valor<0, A=B, etc.
As <expressões 1 e 2> poderão ser quaisquer expressões (do tipo: numérica, caractere ou data) válidas. Ambas podem ser, inclusive, de tipos diferentes de dados. O valor resultante será equivalente ao tipo de dado da expressão processada.
A função IF( )/IIF( ) é uma das mais poderosas e flexíveis do Clipper e tem inúmeras aplicações vantajosas:

- Fornece um meio de avaliar uma condição dentro de uma expressão. Através dela é possível converter-se expressões lógicas em outro tipo de dado. Por exemplo:

```
IF(DELETED( ),"Deletado","Ativo")
```

- É também muito útil para a geração de relatórios, por exemplo, evitando que zeros sejam impressos:

```
@PROW( ),PCOL( ) SAY IF(valor>0,valor,"  -")
```

- Fornece um meio de forçar um relatório de etiquetas (LABEL FORM) imprimir dados em branco, através da inclusão do seguinte tipo de instrução em sua definição:

```
IF(EMPTY(<exp.caractere>), CHR(255), <exp.caractere>)
```

- Finalmente, a utilização da função IF( )/IIF( ) ao invés do comando IF...ENDIF, quando possível, economiza tempo de processamento.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
x = IIF(coef)0,valor/coef,0)
y = IF(coef)=0,"Não Negativo","Negativo")
```

* A função IIF( ) pode ser utilizada para substituir a estrutura da programação abaixo:

```
IF sexo="F"
   nome="Sra. "+nome
ELSE
   nome="Sr. "+nome
ENDIF
```

* É equivalente a:

```
nome = IIF(sexo="F","Sra. "+nome,"Sr. "+nome)
```

Veja Também:

IF...ENDIF, DO CASE...ENDCASE

Função: INKEY( ) – Tecla Digitada:

Forma:

INKEY([<expressão numérica>])

Propósito:

Lê um caractere a partir do teclado, ou seja, lê a tecla digitada, retornando seu código ASCII.
Se a expressão numérica não for especificada retorna o valor numérico do código ASCII da mais recente tecla pressionada, incluindo as teclas de controle. A execução do programa não é interrompida.
Se a expressão numérica for igual a zero (0) faz uma pausa na excução do programa até que uma tecla seja pressionada, retornando, em seguida, o seu valor ASCII.
Se a expressão numérica for um número inteiro "n" maior que zero (0) causa uma pausa de "n" segundos na execução do programa esperando que uma tecla seja pressionada.

Argumentos:

<expressão numérica>: especifica o número de segundos que a função INKEY( ) aguardará até que uma tecla seja pressionada. Se for especificado zero o programa será interrompido até que uma tecla seja pressionada. O tempo que a função INKEY( ) aguarda é baseado no relógio do Sistema Operacional, não sendo relacionado com a velocidade do "clock" do microprocessador do equipamento.

Utilização:

A função INKEY( ) permite a determinação da mais recente tecla pressionada pelo usuário, retornando um inteiro entre -39 e 386 representando o seu valor ASCII ou IBM-ECS (o mesmo valor retornado pela função LASTKEY( )). Se o buffer do teclado estiver vazio (nenhuma tecla foi digitada) INKEY( ) retorna zero.

Como também são incluídas as teclas de controle, ela é muito útil para a criação de rotinas que fazem uso destas teclas, como por exemplo rotinas de paginação de tela ou movimentação do cursor.

Quando for pressionada uma sequencia de teclas, INKEY( ) irá retornar o valor ASCII da primeira tecla da seqüência.

INKEY( ) retorna valores para todas as teclas de função e para as combinações Alt-função, Ctrl-função, Alt-letra e Ctrl-letra.

Abaixo seguem-se valores retornados pela função INKEY( ) para algumas teclas de controle, que poderão ser utilizadas para sofisticar suas rotinas:

Tecla	Valor
→	4
→	19
Para cima	5
Para baixo	24
^→	2
^→	26
Enter	13
Ins	22
Del	7
BackSpace	8
Home	1
End	6
PgUp	18
PgDn	3
+	43
—	45
^Home	29
^End	23
^PgUp	31
^PgDw	30
F1	28
F2 a F10	−1 a −9
Shift−F1 a Shift−F10	−10 a −19
Ctrl-F1 a Ctrl-F10	−20 a −29
Alt-F1 a Alt-F10	−30 a −39

A função INKEY( ) sempre retorna automaticamente valor da mais recente tecla digitada, sem fazer uma pausa para esperar que a tecla seja pressionada. Para conseguir isto utiliza-se INKEY(0). O argumento zero (0) na função INKEY( ) (INKEY(0)) faz com que a execução do programa seja interrompida até que uma tecla seja pressionada, de maneira semelhante ao comando WAIT. Quando, for pressionada uma tecla, seu valor ASCII é captado pela função INKEY( ) e a execução do programa continua normalmente. Pode-se, por exemplo, fazer uso do valor captado da tecla armazenando-o em uma variável: tk = INKEY(0).
Se o argumento da função INKEY( ) for um número inteiro maior que zero, será feita uma pausa na execução do programa para aguardar o pressionamento de uma tecla por um número de segundos igual ao número especificado. Assim INKEY(10) causa uma pausa de 10 segundos na execução do programa à espera de uma digitação do usuário.
A principal utilização da função INKEY( ) é, portanto, de captar o valor da tecla pressionada para utilizá-lo como uma condição para decisões de processamento nos comandos DO CASE...ENDCASE, IF...ELSEIF...ENDIF ou IF...ENDIF. Além disso INKEY( ) também pode ser utilizada para finalizar comandos como LIST, LABEL FORM e REPORT FORM, se for incluída na especificação das condições das cláusulas WHILE ou FOR, como ilustrado:

```
LIST nome, endereço, telef TO PRINT WHILE INKEY( ) < > 27
(o comando é interrompido pressionando-se Esc = 27)
REPORT FORM Relatorio FOR INKEY( ) < > 13)
(o comando é interrompido pressionando-se Enter = 13)
```

Para facilitar a manutenção dos programas aconselha-se a criar um conjunto de variáveis que, através da função CHR( ), armazenem os códigos das teclas de controle que estiverem sendo utilizadas na aplicação. Durante o processamento o resultado da função INKEY( ) pode ser comparado com estas variáveis para determinar a ação a tomar utilizando a expressão CHR(INKEY( )).

Biblioteca

CLIPPER.LIB

Exemplos:

* O exemplo a seguir permite a consulta a registros paginando-os na tela, para frente e para trás:

```
**********************
PROCEDURE Consulta
**********************
@ 24,10 SAY "(PgUp) -> Anterior   (PgDn) -> Próximo   "+;
"(Esc) -> Finaliza"
DO WHILE .T.
   tk=INKEY(0)  && faz uma pausa para aguardar a tecla.
   DO CASE
      CASE tk=18 && código da tecla PgDn
           SKIP -1
           EXIT
      CASE tk=3  && código da tecla PgUp
           SKIP
           EXIT
      CASE tk=27 && código da tecla Esc
           EXIT
   ENDCASE
ENDDO
@ 24,00 CLEAR
RETURN
```

* O exemplo abaixo permite, com a utilização da função INKEY( ), interromper a impressão pressionando-se a tecla <Esc>:

```
************************
PROCEDURE Escprint
************************
   PRIVATE    esc
   STORE .F. TO fim
   esc = INKEY()       && não interrompe a execução.
   DO WHILE esc<>0 .AND. esc<>27
      esc = INKEY()
   ENDDO
   IF esc = 27
      SET DEVICE TO SCREEN
      SET CONSOLE ON
      ?? CHR(7)        && faz soar o "beep"
      SET CONSOLE OFF
      resp=SPACE(1)
      @ 24, 20 SAY "Deseja Interromper a Impressão (S/N)?";
      GET resp PICTURE "!"
      @ 24,00 CLEAR
      SET DEVICE TO PRINT
      IF resp="N"
         RETURN
      ENDIF
      fim = .T.
      @ PROW()+2, 0 SAY REPLICATE("-",132)
      @ PROW()+2,25 SAY "* * * IMPRESSAO INTERROMPIDA * * *"
      @ PROW()+2, 0 say REPLICATE("-",132)
      EJECT
      @ PROW() ,PCOL() SAY CHR(27)+"@"
      SET DEVICE TO SCREEN
      RETURN
   ENDIF
   SET DEVICE TO PRINT
RETURN
```

O exemplo abaixo permite que sejam determinados todos os valores de retorno da função INKEY( ), para qualquer tecla que for pressionada, finalizando-se com Esc:

```
tk=0
DO WHILE LASTKEY()<>27
   @ 12,30 SAY "Pressione uma Tecla !"
   tk = INKEY(0)
   @ 14,30 SAY "Caracter.........: "+CHR(tk)
   @ 16,30 SAY "Código...........: "+LTRIM(STR(tk))
ENDDO
RETURN
```

Veja Também:

SET KEY, LASTKEY( ), READKEY( ), CHR( ), CLEAR TYPEAHEAD, KEYBOARD

Função: LASTKEY( ) — Última Tecla Pressionada:

Forma:

LASTKEY( )

Propósito:

Retorna um valor numérico (código ASCII) da última tecla pressionada, incluindo as teclas de controle, a partir de um estado de espera do sistema.

Argumentos:

Nenhum.

Utilização:

A função LASTKEY( ) retorna um número entre -39 e 386, identificando o código ASCII da última tecla pressionada em um estado de espera do sistema. Os comandos que geram estado de espera são os seguintes: ACCEPT, INPUT, MENU TO, READ e WAIT.

A função LASTKEY( ) é bem semelhante à função INKEY( ), a principal diferença reside no fato de que quando é digitada uma sequência de teclas, INKEY( ) retorna o código ASCII da primeira tecla da sequência, enquanto que LASTKEY( ) retorna o código ASCII da última.

A função LASTKEY( ) é especialmente útil para determinar como um comando READ foi completado, especialmente por permitir verificar através

de qual tecla de controle (Esc, Ctrl-W, PgDw etc.) ele foi finalizado. Além disso, se estiver sendo usada uma cláusula VALID pode-se permitir abandonar o GET atual dependendo da tecla pressionada.

Veja os valores retornados para as teclas de controle na função INKEY( ). Baseando-se no valor retornado por LASTKEY( ), pode-se tomar várias decisões em uma rotina ou função-de-usuário para se tomar a ação apropriada em função da última tecla pressionada.

Contudo, ao contrário do dBASE III Plus, para se saber se ao finalizar um GET o usuário alterou ou não os dados pode ser utilizada a função UPDATE( ) e não necessariamente a função LASTKEY( ).

Ao contrário da função INKEY( ), a função LASTKEY( ) não possui argumentos.

Biblioteca:

CLIPPER.LIB

Exemplos:

* Atualizando um arquivo, dependendo da última tecla digitada na edição dos dados:

```
CLEAR
SET DATE BRITISH
USE Mala INDEX Indcod, Indnome
DO WHILE .T.
   vcod = SPACE(4)
   vnome=SPACE(40)
   vende=SPACE(40)
   vcida=SPACE(20)
   vesta=SPACE(2)
   vcep=SPACE(5)
   @ 05,32 SAY "Alteração de Dados"
   @ 06,32 SAY "--------------------"
   SET COLOR TO W/N,W+/R
   @ 10,20 SAY "Codigo..: " GET vcod  PICTURE "9999"
   @ 12,20 SAY "Nome....: " GET vnome PICTURE "@!"
   @ 14,20 SAY "Endereco: " GET vende PICTURE "@!"
   @ 16,20 SAY "Cidade..: " GET vcida PICTURE "@!"
   @ 18,20 SAY "Estado..: " GET vesta PICTURE "!!"
   @ 18,35 SAY "CEP: "      GET vcep  PICUTRE "99999"
   @ 20,35 SAY "(Esc) Cancela as Alterações"
   SET COLOR TO
   CLEAR GETS
   @ 10,20 SAY "Codigo..: " GET vcod PICTURE "9999"
   READ
   IF EMPTY(vcod)
      CLOSE DATABASE
      CLEAR
      RETURN
   ENDIF
   @ 23,00 CLEAR
```

```
    SEEK vcod
    IF .NOT. FOUND()
       @ 23,30 SAY "Codigo Não Cadastrado !"
       LOOP
    ENDIF
    vnome=nome
    vende=ende
    vcida=cida
    vesta=esta
    vcep = cep
    @ 12,30 GET vnome PICTURE "@!"
    @ 14,30 GET vende PICTURE "@!"
    @ 16,30 GET vcida PICTURE "@!"
    @ 18,30 GET vesta PICTURE "!!"
    @ 20,39 GET vcep  PICTURE "99999"
    READ
    * Se a tecla usada para finalizar a alteração dos
    * dados for diferende de Esc (ASCII 27), atualiza-
    * se o arquivo.
    IF LASTKEY()()27
       REPLACE nome WITH vnome,ende WITH vende
       REPLACE cida WITH vcida,esta WITH vesta
       REPLACE cep  WITH vcep
    ENDIF
ENDDO
RETURN
```

Veja Também:

KEYBOARD, INKEY( ), READKEY( ), READ, CHR( ), ACCEPT, WAIT, INPUT

Função: MAX( ) — Máximo:

Forma:

MAX(<expressão 1, expressão 2>)

Propósito:

Retorna o maior valor entre duas <expressões> numéricas ou de datas especificadas.

Argumentos:

<*expressão 1*>: é a primeira expressão numérica ou data a ser comparada.
<*expressão 2*>: é a segunda expressão numérica ou data a ser comparada.

Utilização:

A função MAX( ) é uma forma rápida e eficiente de se obter o maior entre dois valores numéricos ou entre duas datas resultantes de duas <expressões> a serem comparadas.
As duas <expressões> devem ser do mesmo tipo; são avaliadas e comparadas, resultando sempre a maior.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
? MAX(1000,500)     && retorna 1000
x=100
y=20
z=MAX(x,y)          && z=100

data=CTOD("01/01/89")
? MAX(DATE(),data)
* retorna 01/01/89 se DATE() for menor ou vice-versa.
```

Veja Também:

MIN( ), IIF( )/IF( )

Função: MIN( ) – Mínimo:

Forma:

MIN(<expressão 1, expressão 2>)

Propósito:

Retorna o menor valor entre duas <expressões> numéricas ou datas especificadas.

Argumentos:

<expressão 1>: é a primeira expressão numérica ou data a ser comparada.
<expressão 2>: é a segunda expressão numérica ou data a ser comparada.

Utilização:

A função MIN( ) é uma forma rápida e eficiente de se obter menor entre dois valores ou entre duas datas resultantes das <expressões> especificadas. As duas expressões devem ser do mesmo tipo; são avaliadas e comparadas, resultando sempre a menor.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
? MIN(1000,500)    && retorna 500
x=100
y=20
z=MAX(x,y)         && z=20

data=CTOD("01/01/89"
? MAX(DATE(),data)
* retorna DATE() ou 01/01/89 se DATE() for maior
```

Veja Também:

MAX( ), IIF( )/IF( )

Função: NEXTKEY( ) – Próxima Tecla:

Forma:

NEXTKEY( )

Propósito:

Lê a próxima tecla digitada pelo usuário sem removê-la do buffer de teclado, retornando o código ASCII correspondente.
Disponível a partir da versão Summer 87.

Argumentos:

Nenhum

Utilização:

A função NEXTKEY( ) retorna um número entre -39 e 386, identificando o código ASCII da próxima tecla pressionada pelo usuário. Os valores retornados são equivalentes aos das funções INKEY( ) e LASTKEY( ). Se o buffer do teclado estiver vazio, NEXTKEY( ) retonará o valor zero (0).
Como INKEY( ) e LASTKEY( ) retorna valores para todas as funções e teclas de controle. Veja a tabela apresentada na função INKEY( ) para verificar alguns dos principais.
A função NEXTKEY( ) permite antecipar ações que utilizam determinadas teclas para chamar rotinas ou executar determinadas operações.

Biblioteca:

EXTEND.LIB

Exemplo:

```
KEYBOARD CHR(27)  && equivale a teclar Esc
? NEXTKEY(), LASTKEY()  && retornam: 27 e  0
? INKEY(), LASTKEY()    && retornam: 27 e 27
```

Veja Também:

INKEY( ), LASTKEY( )

Função: READVAR( ) – Lê Variável:

Forma:

READVAR( )

Propósito:

Retorna o nome da variável do último comando GET ou MENU TO executado, ou uma cadeia de caracteres nula, se não existir nenhum GET ou MENU TO pendentes.

Argumentos:

Nenhum

Utilização:

A função READVAR( ) retorna o nome (em letras maiúsculas) da variável sendo esperada pelo comando GET ou MENU TO ou o caracter nulo ("") se não houver nenhum pendente.
Esta função é útil para se determinar qual a variável que está sendo aguardada em um comando GET ou em um comando MENU TO para efeito de depuração de erros, construção de Help's ao usuário, construção de rotinas especiais etc. Obtendo-se a informação pode-se desviar o fluxo de execução do programa de acordo com a variável esperada e tomar a ação apropriada. READVAR( ) é especialmente útil em rotinas que utilizam o comando SET KEY.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
nome=SPACE(30)
salario=0.00
SET -8 TO Helpvar
@ 10,10 SAY "Digite o Nome...:" GET nome
@ 12,10 SAY "Digite o Salario:" GET salario
@ 14,20 SAY "Tecle F9 para um Help !"
READ
RETURN

PROCEDURE Helpvar
@ 16,00 CLEAR
IF READVAR()="NOME"
   @ 16,20 SAY "Digite o Nome do Funcionario !!"
ELSE
   @ 16,15 SAY "Digite o Salario Mensal do Funcionario"
ENDIF
WAIT "Pressione qualquer tecla para retornar..."
@ 16,00 CLEAR
RETURN
```

Veja Também:

SET KEY, @...SAY...GET, MENU TO.

Função: SCROLL( ) – Rolamento da Tela:

Forma:

SCROLL(<exp.num1>,<exp.num2>,<exp.num3>,<exp.num4>,<exp.num5>)

Propósito

Permite determinar uma janela (área) da tela a ser rolada("scroll") para cima, para baixo ou apagada.
Disponível a partir da versão Summer 87.

Argumentos:

<exp.numérica 1> e *<exp.numérica 2>*: identificam a coordenada do canto superior esquerdo da janela a ser rolada.
<exp.numérica 3> e *<exp.numérica 4>*: identificam a coordenada do canto inferior direito da janela a ser rolada.
<exp.numérica 5>: determina o número de linhas a serem roladas. Se for um número maior que zero, causará o rolamento para cima de acordo com o número especificado. Se for um número menor que zero causará o rolamento para baixo. Zero limpa a área especificada.

Utilização:

A função SCROLL( ) é utilizada para "rolar" instantaneamente os dados apresentados em uma determinada região da tela, para cima ou para baixo, simulando o recurso de janelas, disponível nos mais modernos "softwares". Ela pode ser útil quando da apresentação de uma lista de dados na tela (em geral os registros de um arquivo de dados), para mostrar os dados anteriores ou posteriores à posição atual que não cabem na tela. Neste caso você poderá construir uma rotina que realize o "rolamento" dos dados, isto é, ao ser pressionada uma determinada tecla um novo dado é apresentado no topo ou no pé da janela definida e todos os outros são rolados para cima ou para baixo.

Biblioteca:

EXTEND.LIB

Exemplo:

* O exemplo abaixo permite o rolamento de uma determinada janela na tela:

```
CLEAR
@ 04,29 TO 13,51 DOUBLE
@ 05,31 SAY "Teste do Rolamento"
@ ROW()+3,30 SAY "Linha 111111111111"
@ ROW()+1,30 SAY "Linha 222222222222"
@ ROW()+1,30 SAY "Linha 333333333223"
@ ROW()+1,30 SAY "Linha 444444444224"
@ ROW()+1,30 SAY "Linha 555555555225"
@ 23,20 SAY "Seta Para Cima Sobe, Seta Para Baixo Desce"
@ 24,34 SAY "(Esc) Finaliza"
```

```
DO WHILE INKEY()<>27  && 27 é o código do Esc
   INKEY(0)
   DO CASE
      CASE INKEY()=5        && 5 é o código da seta para cima
           SCROLL(8,30,12,50,1)   && rola uma linha para cima
      CASE INKEY()=24     && 24 é o codigo da seta para baixo
           SCROLL(8,30,12,50,-1) && rola uma linha para baixo
   ENDCASE
ENDDO
CLEAR
RETURN
```

Veja Também:

@...CLEAR...TO, @...BOX, @...TO

Função: TONE( ) – Tom:

Forma:

TONE(<exp.numérica 1>,<exp.numérica 2 >)

Propósito:

Faz soar o alto-falante do equipamento de acordo com o tom (frequência) e tempo especificados.
Disponível a partir da versão Summer 87.

Argumentos:

<exp.numérica 1>: determina a frequência do som.
<exp.numérica 2>: determina a duração da emissão do som, medida em incrementos de 1/18 de segundos. Para um segundo, portanto, deve ser especificado 18.

Utilização:

A função TONE( ) "toca" o alto-falante do equipamento de acordo com a frequência e a duração especificadas pelos argumentos.
A duração <exp.numérica 2> é medida em um dezoitoavos de segundo (18 equivale a um segundo). A frequência é medida em hertz (ciclos por segundo). Frequências abaixo de 20 hertz são inaudíveis.

Biblioteca:

EXTEND.LIB

Exemplo:

O exemplo a seguir mostra uma rotina que emite um som para caracterizar o fim de uma operação ("than, than, than, thaaaaaam!"):

```
PROCEDURE THAN
TONE(300,8)
TONE(260,10)
TONE(280,9)
TONE(200,25)
RETURN
```

Veja Também:

SET BELL, CHR( )

Função: TRANSFORM( ) – Transforma:

Forma:

TRANSFORM(<expressão>,<expressão caractere>)

Propósito:

Permite a formatação através de máscaras (PICTURE's) de expressões de qualquer tipo de dados, retornando uma cadeia de caracteres formatada.

Argumentos:

<*expressão*>: é uma expressão tipo caractere, data ou numérica a ser formatada.
<*exp.caractere*>: define o formato da cadeia de caracteres resultante.

Utilização:

Utiliza-se a função TRANSFORM( ) sempre que for conveniente a formatação de dados sem a utilização do comando @...SAY...PICTURE. Por exemplo, para a obtenção de saídas de dados formatados a partir dos comandos ?, LIST, DISPLAY, REPORT FORM e LABEL FORM. Note-se, contudo, que a função TRANSFORM( ) retorna uma cadeia de caracteres, não permitindo que os totais e subtotais dos relatórios gerados pelo comando REPORT FORM sejam formatados.

As máscaras de formatação possíveis de serem especificadas através da <expressão caractere> são as mesmas disponíveis para a cláusula PICTURE do comando @...SAY...GET.

Biblioteca:

CLIPPER.LIB

Exemplos:

```
? TRANSFORM(1234.54,"999,999.99")   && resulta 1,234.54
? TRANSFORM(1234,"@E 999,999.99")   && resulta 1.234,00
? TRANSFORM("clipper","!!!!!!!")    && resulta "CLIPPER"
? TRANSFORM("Clipper","@R X X X X X X X")
* resulta "C L I P P E R"
```

Veja Também:

@...SAY...GET...PICTURE, LIST, DISPLAY, ?, REPORT FORM, LABEL FORM

Função: WORD( )

Forma:

WORD(<expressão numérica>)

Propósito:

Converter parâmetros numéricos do comando CALL do tipo dupla precisão (DOUBLE) para inteiro (INT).

Argumentos:

<*expressão numérica*>: é o valor numérico a ser convertido para tipo inteiro (INT).

Utilização:

Utiliza-se a função WORD( ) para converter parâmetros numéricos de dupla precisão (DOUBLE) para inteiros (INT), com o objetivo de reduzir tempo de processamento de uma rotina externa em "C" chamada através do comando CALL. O valor convertido deve estar entre 32.767 e -32.767.

Biblioteca:

CLIPPER.LIB

Exemplo:

```
CALL rotC WITH WORD(500)
```

Veja Também:

CALL

7

A Facilidade de "Help" ao Usuário

Todos os analistas, programadores ou profissionais que desenvolvem sistemas (programas) sabem que uma das grandes dificuldades encontradas para a implantação e operação de suas "obras primas" é o treinamento dos usuários. Um excelente "software" ou sistema pode ser rejeitado pelos usuários se for difícil de ser entendido ou operado. Quanto maior a facilidade de operação de um "software" e quanto maiores forem os recursos que ele oferecer para o esclarecimento de dúvidas dos usuários, tanto mais e melhor ele será utilizado. Não é a toa que "softwares" consagrados como o dBASE III, o Lotus 1-2-3, o WordStar, o SideKick e todos os seus concorrentes oferecem a facilidade de "HELP", ou seja, ajuda ao usuário diretamente na tela do computador durante sua utilização.

A "interface" de relacionamento computador × usuário ou sistema × usuário, é um dos quesitos que atualmente selecionam "softwares" de boa qualidade. Quanto mais fácil de ser entendida, mais completa e agradável for a interface com o usuário, melhor será o "software".

O "HELP" consiste no fornecimento de informações, diretamente através da tela do computador, que auxiliam o usuário a esclarecer dúvidas e a descobrir como proceder em situações específicas durante a utilização do "software". Até você, sem dúvida nenhuma, já utilizou o **HELP** de algum "software". Será que não seria interessante oferecer também aos usuários dos sistemas que você desenvolve em Clipper um "HELP"?

O Clipper, através de sua "Facilidade de Criação de HELP", permite que sejam construídas e acessadas telas de auxílio aos usuários das aplicações com ele desenvolvidas. Este recurso, além de ser muito fácil de ser implementado nos programas, permite que você sofistique suas aplicações, fornecendo a seus usuários um sistema de alto nível profissional.

Para criar um HELP em seus sistemas, basta que você escreva um programa chamado **HELP.prg** que, através de mensagens e recursos de controle de tela, forneça ao usuário informações para o esclarecimento geral de dúvidas ou para

definir procedimentos em determinadas situações de operação. O HELP.prg deverá ser depois compilado e link-editado com os demais programas de sua aplicação. A partir daí, para invocar o "HELP" basta o usuário pressionar a tecla <F1>, em qualquer estado de espera do programa, causado pelos comandos WAIT, INPUT, ACCEPT, MENU TO ou READ. O Clipper automaticamente chamará o HELP.prg enviando-lhe três parâmetros: o nome do programa sendo executado (prog), número da linha sendo executada (lin) e o nome da variável sendo aguardada (var).

Pressionar a tecla <F1> durante a execução de um programa, em qualquer estado de espera. será equivalente a executar o seguinte comando:

DO Help WITH prog, lin, var

Onde os parâmetros prog, lin e var indicam:

1 — **prog** — Fornece uma variável caractere contendo o nome da rotina ou programa a partir do qual o HELP.prg foi invocado. Todas as letras são passadas em maiúsculas, portanto qualquer comparação a ser feita para identificar este nome deve ser feita em letras maiúsculas.

2 — **lin** — Fornece uma variável numérica contendo o número da linha do programa-fonte que está sendo executada, em geral uma linha contendo um comando READ, ou um dos comandos que caracterizam um estado de espera no processamento.

3 — **var** — Fornece uma variável caractere contendo o nome da variável que está sendo solicitada como entrada ao usuário, através de um comando READ. Todas as letras do nome desta variável são passadas em maiúsculas. Qualquer comparação para identificar o nome da variável que está sendo esperada deverá ser em letras maiúsculas. Comandos de espera como WAIT, INPUT e ACCEPT não retornam, entretanto, valores para este parâmetro, apenas o comando READ.

Os nomes destes três parâmetros não precisam necessariamente ser prog, lin e var. Podem ser quaisquer nomes, mas sempre fornecerão ao HELP.prg as informações descritas acima e **deverão estar** sendo esperados por ele, mesmo que não sejam utilizados, pois o Clipper sempre os enviará.

Através destes três parâmetros pode-se identificar a situação de espera específica na qual o usuário se encontra e fornecer-lhe uma tela de "HELP" que contenha as informações a ela adequadas.

Compilação:

Como o HELP.prg não será explicitamente chamado por um programa de sua aplicação, o compilador do Clipper não irá compilá-lo automaticamente junto com os outros programas. O HELP.prg deverá, portanto, ser compilado isoladamente (ou através de um arquivo ".CLP"). Por exemplo:

C:\clipper>CLIPPER HELP

O módulo-objeto gerado, HELP.obj, deverá posteriormente ser incluído na lista de módulos-objeto passado ao link-editor, para ser incluído no módulo executável da aplicação.

Por exemplo, supondo que os módulos-objeto de sua aplicação sejam PROG1, PROG2 e PROG3, para link-editá-los juntamente com o HELP através do PLINK86 deve ser executado o seguinte comando:

C:\clipper>PLINK86 FI PROG1,PROGR2,PROG3,HELP

Para maiores detalhes sobre a compilação e a link-edição veja o Cap. 8.

Chamadas recursivas:

Chamadas recursivas do HELP, ou seja chamar o HELP dentro do próprio HELP, pode causar problemas para o seu controle do programa. Isto, entretanto, poderá ser evitado se as seguintes instruções forem colocadas no início do seu programa HELP.prg:

```
IF prog="HELP"
   RETURN
ENDIF
```

Ou seja, se o programa que chamou o HELP foi o próprio HELP, o controle retornará a ele mesmo.

Exemplos:

Um programa HELP.prg genérico poderá, por exemplo, possuir a seguinte estrutura:

```
*************************************************
* HELP.prg   Programa de Help para o Sistema .....
*************************************************
PARAMETERS prog,lin,var     && recebe os parâmetros enviados
IF prog="HELP"              && pelo Clipper - obrigatório.
   RETURN      && se o Help for chamado pelo Help retorna.
ENDIF
```

```
SAVE SCREEN
*  salva a tela que está sendo apresentada para reapresentá-
*  la quando o usuário sair do HELP.
@ 01,00 CLEAR
* limpa a tela sem limpar os GET's pendentes ao ser invocado
* o Help
SET COLOR TO W+
@ 03,28 SAY "HELP - Auxílio ao Usuário"
SET COLOR TO
DO CASE
   CASE prog="PROG1"    && nome do programa
      DO CASE
         CASE linha=20  && número da linha sendo executada
            DO CASE
               CASE var="VCODIGO" && variável esperada
                     .
                     .
                  <Mensagens do Help>
                     .
                     .
               CASE var="VNOME" && variável esperada
                     .
                     .
                  <Mensagens do Help>
                     .
                     .
               OTHERWISE
                  @ 12,24 "Desculpe, Não Há HELP disponível"
            ENDCASE
         CASE linha=30  && número da linha sendo executada
            DO CASE
               CASE var="VCODIGO" && variável esperada
                     .
                     .
                    <Mensagens do Help>
                     .
                     .
               CASE var="VNOME" && variável esperada
                     .
                     .
                    <Mensagens do Help>
                     .
                     .
               OTHERWISE
                  @ 12,24 "Desculpe, Não Há HELP disponível"
            ENDCASE
            OTHERWISE
               @ 12,24 "Desculpe, Não Há HELP disponível"
         ENDCASE
   CASE prog="PROG2"  && nome do programa sendo executado
      DO CASE
         CASE linha=25  && número da linha sendo executada
            DO CASE
               CASE var="VPRODUTO" && variável esperada
                     .
                     .
                    <Mensagens do Help>
                     .
                     .
               CASE var="VVALOR" && variável esperada
```

```
                    .
                    .
                   <Mensagens do Help>

                    .
               OTHERWISE
                  @ 12,24 "Desculpe, Não Há HELP disponível"
            ENDCASE
         CASE linha=50  && número da linha sendo executada
            DO CASE
               CASE var="VSALDO" && variável esperada
                    .
                    .
                   <Mensagens do Help>
                    .
                    .
               CASE var="VQUANT" && variável esperada
                    .
                    .
                   <Mensagens do Help>
                    .
                    .
               OTHERWISE
                  @ 12,24 "Desculpe, Não Há HELP disponível"
            ENDCASE
         OTHERWISE
            @ 12,24 "Desculpe, Não Há HELP disponível"
      ENDCASE
   OTHERWISE
      @ 12,24 "Desculpe, Não Há HELP disponível"
ENDCASE
SET COLOR TO W+
@ 24,20 SAY "Pressione Qualquer Tecla para Retornar !"
SET COLOR TO
RESTORE SCREEN
* recupera e reapresenta, instantaneamente,tela original que
* estava  sendo  apresentada  ao usuário, antes de ter sido
* invocado o Help.
RETURN
* retorna ao programa que chamou o Help.prg, exatamente onde
* o usuário estava.
*********************
* Fim do HELP.prg
*********************
```

Os comandos CLEAR e READ nunca deverão ser utilizados dentro do programa HELP.prg porque eles cancelam todos os GET's que estiverem pendentes antes do HELP ser acionado pelo usuário. Utilize os comandos @...linha, coluna...CLEAR ou CLEAR SCREEN para limpar a tela e WAIT, ACCEPT, MENU TO ou INKEY(0) para a entrada de dados dentro do HELP.prg.

Uma forma alternativa ou suplementar para fornecer HELP ao usuário é construir um HELP genérico, sem a utilização dos parâmetros automaticamente passados pelo Clipper. Para isso deve ser criada, no programa principal de sua aplicação, uma variável pública denominada, por exemplo, codhelp. Atribua à essa variável valores de acordo com o módulo ou seção do programa em que o usuário

estiver. Desta maneira, ao ser invocado o HELP, você poderá testar o valor da variável codhelp e, de acordo com ele, fornecer as informações adequadas.

Escreva o seu HELP.prg de forma a definir as telas de mensagens ao usuário de acordo com cada valor assumido pela variável codhelp. Ao ser pressionada a tecla <F1>, o HELP.prg será invocado e serão apresentadas as mensagens correspondentes ao valor armazenado naquele momento na variável codhelp.

Não se esqueça, entretanto, que mesmo neste caso, onde não são utilizados os parâmetros prog, lin e var, o Clipper os passará ao HELP.prg, que deverá, portanto, esperá-los.

O exemplo abaixo ilustra um HELP.prg construído utilizando a técnica da variável codhelp.

```
*********************************
* Programa HELP.prg - Genérico
*********************************
PARAMETERS prog,linha,var
IF prog="HELP"
   RETURN
ENDIF
SAVE SCREEN TO tela
@ 05,00 CLEAR
SET COLOR TO W+
@ 05,28 SAY "HELP - Auxílio ao Usuário"
SET COLOR TO W
DO CASE
   CASE codhelp=1
        @ 08,25 SAY "Teclas de Contrôle de Edição"
        @ 10,04 SAY "[Enter] -> Muda de linha ou parágrafo"
        @ 10,44 SAY "[↑ ->]  -> Palavra para a direita"
        @ 11,04 SAY "[Setas] -> Movimentam o cursor"
        @ 11,44 SAY "[↑<--]  -> Palavra para a esquerda"
        @ 12,04 SAY "[Home]  -> Início da linha"
        @ 12,44 SAY "[↑Home] -> Início do Texto"
        @ 13,04 SAY "[End]   -> Fim da linha"
        @ 13,44 SAY "[↑End]  -> Fim do Texto"
        @ 14,04 SAY "[<--]   -> Apaga à esquerda"
        @ 14,44 SAY "[PgUp]  -> Página para cima"
        @ 15,04 SAY "[Ins]   -> Liga/Desliga Inserção"
        @ 15,44 SAY "[PgDn]  -> Página para baixo"
        @ 16,04 SAY "[Del]   -> Apaga letra sob o cursor"
        @ 16,44 SAY "[↑PgUp] -> Início da página"
        @ 17,04 SAY "[↑T]    -> Apaga a palavra à direita"
        @ 17,44 SAY "[↑PgDn] -> Fim da página"
        @ 18,04 SAY "[↑Y]    -> Elimina a linha do cursor"
        @ 18,44 SAY "[↑B]    -> Reformata o texto"
        @ 19,04 SAY "[↑W]    -> Finaliza gravando o texto"
        @ 19,44 SAY "[Esc]   -> Finaliza sem gravar"
        @ 20,30 SAY "↑ -> Pressione a Tecla Ctrl e a Letra"
   CASE codhelp=2
        @ 07,27 SAY "Teclas de Contrôle de Edição"
        @ 08,24 SAY "Para Inclusão ou Alteração de Dados"
        @ 10,10 SAY "[Enter]  -> Dá entrada nos dados ou
                                 efetua uma alteração"
        @ 11,10 SAY "[Setas]  -> Movimentam o cursor para a
                                 direção indicada"
```

```
        @ 12,10 SAY "[Home]    -> Movimenta o cursor para o
                                  início do campo"
        @ 13,10 SAY "[End]     -> Movimenta o cursor para o
                                  fim do campo"
        @ 14,10 SAY "[<---]    -> Apaga o caractere à
                                  esquerda do cursor"
        @ 15,10 SAY "[Ins]     -> Liga ou Desliga o modo de
                                  inserção de caracteres"
        @ 16,10 SAY "[Del]     -> Apaga o caractere sob o
                                  cursor"
        @ 17,10 SAY "[↑T]      -> Apaga a palavra à direita
                                  do cursor"
        @ 18,10 SAY "[↑Y]      -> Apaga todo o conteúdo de um
                                  campo"
        @ 19,10 SAY "[↑W]      -> Finaliza gravando os dados
                                  digitados"
        @ 20,10 SAY "[Esc]     -> Finaliza sem gravar os
                                  dados digitados"
        @ 21,10 SAY "↑         -> Pressione a tecla [Ctrl] e
                                  a letra correspondente"
    CASE codhelp=3
        MEMOEDIT(MEMOREAD("Help1.txt"),7,2,22,79,.F.)
        * a função MEMOREAD() lê um arquivo texto qualquer
        * gravado no disco, no caso o Help1.txt. Este
        * arquivo,que contém um texto de informações para o,
        * usuário é, então,apresentado em uma janela através
        * da função MEMOEDIT(). Esta,além de mostrar o texto
        * numa janela definida, permite que o usuário faça
        * "rolamento" vertical linha por linha para lê-lo.
        *  Para maiores detalhes consulte as funções MEMO():
        * MEMOEDIT() e MEMOREAD() na seção 5.6.
    CASE codhelp=4
        MEMOEDIT(MEMOREAD("Help2.txt"),7,2,22,79,.F.)
ENDCASE
@ 24,20 SAY "Pressione qualquer tecla para retornar"
INKEY(0)
RESTORE SCREEN FROM tela
RETURN
```

Veja outros exemplos práticos no HELP.prg do Sistema de Mala-Direta, no Cap. 10 — Um Sistema de Informações Construído com o Clipper.

8

Compilando e Executando Programas

8.1. COMPILANDO OS PROGRAMAS

8.1.1. Utilizando o Compilador

O compilador do Clipper (CLIPPER.exe) converte o código-fonte dos programas, escritos na "linguagem Clipper", para módulos em código-objeto (código de máquina). Os programas de código-fonte são identificados pelo seu nome e pela extensão ".PRG", idêntica ao dBASE III, por exemplo "PROG1.prg". Os módulos em código-objeto são identificados pelo mesmo nome do programa (ou programas) fonte que lhes deu origem e pela extensão ".OBJ", por exemplo "PROG1.obj". Uma vez compilados os programas-fonte, os módulos-objeto resultantes devem ser encadeados ou "link-editados" com rotinas de biblioteca do Clipper para produzir um módulo executável. O módulo executável é o programa resultante do processo de compilação e link-edição que pode ser executado diretamente sob o Sistema Operacional. Recebe a extensão ".EXE", por exemplo "PROG1.exe".

O Clipper permite que sejam compilados programas individuais, arquivos de procedimentos ("procedure files") e funções definidas pelo usuário (através do comando FUNCTION), para gerar os respectivos módulos em código-objeto. Os módulos em código-fonte, ou seja, os diferentes programas que formam uma aplicação ou sistema podem ser compilados individualmente, em grupos pré-definidos ou todos de uma só vez. A forma de compilar um sistema completo, composto por vários programas e arquivos de procedimentos, dependerá de como ele estiver estruturado e de como deverá estar estruturado o módulo executável resultante. Por esse motivo a forma de se estruturar um sistema ou aplicação é fundamental para efeito de compilação e execução. Dela dependerão a rapidez, a eficiência e a facilidade de manutenção do sistema desenvolvido, tanto no que se refere à compilação dos programas-fonte, à link-edição dos módulos-objeto e à sua execução.

Os arquivos dos programas em código-fonte a serem compilados devem ser gerados utilizando-se o editor de textos de sua preferência, por exemplo o Notepad do SideKick, o WordStar etc. Não deve ser utilizado o "MODIFY COMMAND" do dBASE, pois este editor, além de ter capacidade para editar arquivos de no máximo 4.096 bytes, insere quebras de linha lógicas (caractere CHR(141)), que não são aceitas pelo Clipper.

O sistema desenvolvido pode utilizar arquivos de definição de relatórios padronizados e de etiquetas. Os arquivos de definição de relatórios podem ser criados pelo comando REPORT do dBASE III ou pelos utilitários do Clipper (veja a seção 9.1.3 – O Utilitário REPORT.exe e 9.1.5 – O Utilitário RL.exe), que geram arquivos com a extensão ".FRM". Os arquivos de definição de relatórios de etiquetas podem ser criados pelo comando LABEL do dBASE III ou pelos utilitários do Clipper (veja a seção 9.1.4 – O Utilitário LABEL.exe e 9.1.5 – O Utilitário RL.exe), que geram arquivos com a extensão ".LBL". Entretanto, tanto os arquivos de definição de relatórios (.frm) quanto os de etiquetas (.lbl) não precisam ser compilados.

Para utilizar o compilador do Clipper, que se encontra no arquivo "Clipper.exe", você deverá estar no diretório onde este arquivo está localizado (por exemplo clipper) e digitar a seguinte linha de comando no DOS:

C: clipper>CLIPPER [d:] [\dir\] <nome do arquivo>

Onde:

[d:] é o nome do drive onde o programa a ser compilado se encontra, se não for informado será assumido o drive atual;

[\dir\] é o nome do diretório onde o programa a ser compilado se encontra, se não for informado será assumido o diretório atual;

<nome do arquivo> é o nome do programa-fonte a ser compilado; não é necessário informar a extensão, sendo assumido (.prg).

Por exemplo, se o programa principal (que chama outros subprogramas) a ser compilado chama-se "PROG1.prg" e encontra-se no drive C: no diretório \progs, você deverá fornecer os seguintes comandos na linha de comando do DOS:

```
C: >CD \clipper
C:\clipper>CLIPPER C:\progs\PROG1
```

O comando acima irá gerar no diretório atual (\clipper) um único módulo em código-objeto com o mesmo nome do seu programa-fonte e a extensão ".OBJ", ou seja, o arquivo "PROG1.obj". Todos os programas (.prg), arquivos de procedimentos (.prg) e arquivos de formato (.fmt) que forem chamados pelo programa

principal "PROG1.prg" (através dos comandos DO, SET PROCEDURE TO ou SET FORMAT TO) serão automaticamente compilados pelo Clipper, sendo incluídos no módulo-objeto gerado.

Por exemplo, se o PROG1.prg chama o arquivo de procedimentos PROGPROC.prg e os programas PROG2.prg e PROG3.prg, todos os quatro serão compilados e o resultado será gravado em um único módulo-objeto denominado "PROG1.obj".

Se você estiver trabalhando com um disco rígido, deverá, portanto, colocar todos os programas que fazem parte de seu sistema no mesmo diretório, por exemplo o diretório \progs, pois o compilador irá procurá-los lá.

Por outro lado, se você estiver trabalhando com dois drives, coloque o compilador (CLIPPER.exe) no drive A: e todos os programas a serem compilados no drive B:, e execute o comando abaixo:

A>CLIPPER B:PROG1

O módulo em código-objeto, PROG1.obj, será gravado no drive A:.

Durante o processo de compilação, o compilador irá apresentar na tela a versão do Clipper que você está utilizando e indicará, linha por linha (ou de cem em cem linhas na versão Summer 87), todos os programas que estão sendo compilados. Caso sejam detectados erros no código-fonte, serão apresentadas mensagens indicando o tipo de erro cometido, o nome do programa e a respectiva linha onde cada erro se encontra.

Os erros mais comuns que costumam ser cometidos são:

- *Erros de Sintaxe dos Comandos*: nos quais o Clipper indicará a linha e o nome do programa onde se encontram e provavelmente qual foi exatamente o erro cometido. Neste caso você deverá editar novamente o seu programa e corrigir os erros apontados e recompilá-los até que estejam livres de erros.
- *Erro de Localização*: o Clipper não encontrou um programa ou rotina que deveria ser compilada, apresentando a seguinte mensagem: "Cannot Open Assumed External". Neste caso você deverá criar ou colocar o programa solicitado no mesmo diretório, juntamente com os outros programas que fazem parte da sua aplicação. Entretanto, quando se compila separadamente os módulos-fonte de uma aplicação, esta mensagem não constitui um erro, mas apenas um alerta de que o programa ou rotina apontada deverá estar presente em outro módulo.

Se você desejar, poderá utilizar o comando PATH do DOS para redirecionar a localização de arquivos para outros diretórios, evitando, assim, os erros de localização. Recomendamos, entretanto, que todos os programas-fonte requeridos para a compilação de uma aplicação sejam colocados num mesmo diretório.

Um recurso interessante é a gravação das mensagens do compilador em um arquivo-texto no disco. Ao invés de apenas observar as mensagens da compilação

no vídeo, você pode gravá-las em arquivo para posteriormente poder revê-las com calma ou imprimi-las, o que facilitará o processo de correção dos eventuais erros detectados. Para isso utiliza-se a opção de redirecionamento de saída do DOS, bastando adicionar o sinal ">" e o nome do arquivo a ser gravado, após a linha de comando para a execução do compilador.

Por exemplo, para criar um arquivo de mensagens de compilação "ERRO.txt", digite a seguinte linha de comando no DOS:

```
C:\clipper>CLIPPER\PROG1 >ERRO.TXT
```

Com isso, as mensagens da compilação não serão mais apresentadas na tela, mas será gravado um arquivo-texto denominado "ERRO.txt", que poderá ser depois visto ou impresso através do comando TYPE do DOS:

```
C:\clipper>TYPE ERRO.TXT
```

8.1.2. Opções do Compilador

O compilador do Clipper na versão Summer 87 oferece sete opções, sendo que as quatro primeiras (−l, −m, −p e −s) também estão disponíveis nas versões anteriores. No modo padrão, visto na seção anterior, nenhuma opção foi especificada, sendo produzido um único módulo em código-objeto a partir do seu programa ou código-fonte principal.

Cada uma das opções pode ser utilizada, de acordo com sua conveniência, isoladamente ou em conjunto, bastando para isso incluí-las na linha de comando do compilador ao invocá-lo através do DOS. A seguir apresentaremos cada uma delas.

8.1.2.1. Eliminação da Numeração das Linhas: −l

A opção **−l** permite que os números das linhas dos programas-fonte não sejam gravados no módulo-objeto. Com isso o tamanho do módulo-objeto (arquivo .OBJ) será reduzido em 3 bytes para cada linha do programa fonte que contiver um comando.

Para especificar esta opção, deve ser incluído o sinal −l (obrigatoriamente em letras minúsculas) na linha de comando do compilador no DOS. O exemplo a seguir irá compilar o PROG1.prg sem a numeração das linhas.

```
C:\clipper>CLIPPER \progs\PROG1 -l
```

A vantagem de se eliminar a numeração das linhas é reduzir o tamanho dos módulos-objeto e, conseqüentemente, dos módulos executáveis, gerados a partir do seu código-fonte. Contudo, se ocorrer um erro durante a execução do sistema, o Clipper não mais poderá informar o número da linha onde ele foi detectado,

dificultando, assim, a depuração da aplicação. Além disso, a operação do "debugador" do Clipper ficará muito prejudicada pois o número das linhas é fundamental para fornecer informações básicas. Se for sua intensão utilizar o debugador não utilize a opção −1.

8.1.2.2. Compilação de um Único Programa ou Módulo: −m

Ao compilar um programa ou módulo-fonte, o Clipper automaticamente irá compilar todos os programas que são por ele chamados através dos comandos DO, SET PROCEDURE TO ou SET FORMAT TO, gerando um único módulo-objeto (arquivo .OBJ) para este conjunto de programas.

A opção **−m** permite que seja compilado um único programa (ou módulo) isoladamente, e não todos os outros que são por ele chamados, gerando, desta forma, um módulo-objeto referente apenas ao programa especificado. Caso você utilize esta opção, não se esqueça de compilar todos os módulos que fazem parte de sua aplicação.

A opção −m é útil quando você, por exemplo, deseja recompilar apenas um módulo que foi alterado, sem ter a necessidade de recompilar todos os outros que permaneceram intactos. Lembre-se, entretanto, que para gerar o módulo executável, todos os módulos-objeto de todos os programas que compõem o seu sistema (mesmo que individualmente) devem ser fornecidos ao link-editor, caso contrário será apresentada uma mensagem de erro, indicando símbolo indefinido. Esse erro ocorre quando você esquece de compilar um programa que foi chamado por algum outro programa de sua aplicação, ou quando o respectivo módulo-objeto não foi incluído na lista passada ao link-editor.

O exemplo abaixo compila exclusivamente o módulo PROG1.prg e não os que são chamados por ele.

```
C:\clipper>CLIPPER \progs\PROG1 −m
```

O sinal **−m** deve obrigatoriamente estar em letra minúscula.

8.1.2.3. Pausa Para Troca de Discos: −p

Se você estiver utilizando discos flexíveis para compilar seus programas (o que não recomendamos), pode haver a necessidade de trocá-los durante a compilação, caso todos os programas-fonte a serem compilados não caibam em um único disquete.

A opção **−p** faz uma pausa a cada programa compilado, o que permitirá a troca de disquetes quando necessário.

O exemplo a seguir ilustra a utilização desta opção para compilar o programa PROG1.prg no drive B:.

```
C:\clipper>CLIPPER B:PROG1 −p
```

8.1.2.4. Apenas Checagem da Sintaxe do Programa: −s

A quarta opção do compilador suprime a geração do módulo em código-objeto (arquivo .OBJ). Esta opção é útil quando se deseja utilizar o Clipper apenas para checar os erros de sintaxe de um programa (ou conjunto de programas). Como não há geração do módulo-objeto, esta compilação é executada com muito mais rapidez.

Para especificar esta opção deve ser incluído o sinal **−s** (obrigatoriamente em letra minúscula) na linha de comando do compilador. Por exemplo:

```
C:\clipper>CLIPPER \progs\PROG1 −s
```

Não será gerado o módulo-objeto PROG1.obj, apenas serão apresentadas as mensagens do compilador e os erros que eventualmente forem encontrados.

8.1.2.5. Criação do Módulo Objeto em Outro Diretório: −o

Esta opção determina a gravação do arquivo de módulo-objeto em outro diretório, que não o atual. Para especificar esta opção deve ser incluído o sinal **−o** (em letra minúscula) e o diretório onde se deseja que o módulo-objeto gerado seja gravado. Por exemplo:

```
C:\clipper>CLIPPER \progs\PROG1 −o \clipper\objetos
```

8.1.2.6. Supressão da Apresentação do Número das Linhas: −q

Esta opção permite que seja suprimida a apresentação na tela dos números das linhas dos programas que estão sendo compilados, não afetando, contudo, a numeração das linhas no código-objeto que será gerado. Para utilizá-la basta especificar **−q** em letra minúscula, como no exemplo a seguir:

```
C:\clipper>CLIPPER \progs\PROG1 −q
```

8.1.2.7. Criação de Arquivos de Trabalho em Outro Drive: −t

Durante o processo de compilação, o Clipper cria e utiliza arquivos de trabalho temporários. Esta opção permite que a criação destes arquivos seja direcionada para outro drive, sendo especialmente útil quando você estiver utilizando um disco virtual na memória do seu computador.

No processo de compilação os arquivos de trabalho podem ser redirecionados para o disco virtual através da opção **−t** (em letra minúscula) e da letra designadora do drive em questão. Por exemplo:

C:\clipper\progs\PROG1 -t D:

O comando acima irá utilizar o disco virtual identificado como drive D: para a criação e manipulação dos arquivos de trabalho temporários utilizados pelo compilador, acelerando sensivelmente o processo de compilação.

8.1.3. Compilando uma Lista Especificada de Programas

Normalmente você cria os programas (.prg), os arquivos de procedimentos (.prg), os arquivos de formatação de tela (.fmt) e os arquivos de função-de-usuário (.prg) de um sistema inteiro em um determinado diretório do seu disco rígido ou em um disquete. Sempre o programa principal de um sistema irá chamar outros programas através do comando DO, que por sua vez chamarão outros, e assim por diante. Ao se compilar este programa o Clipper automaticamente irá compilar de uma única vez todos os programas ou arquivos de procedimentos que são direta ou indiretamente referenciados por ele, gerando um único módulo-objeto. Caso ocorra um erro em apenas um dos programas, utilizando esta técnica você deverá, após corrigir o erro, compilar os todos os programas novamente, mesmo aqueles que já estavam corretos. Procedendo desta forma, o dispêndio de tempo para corrigir os erros poderá ser enorme.

Ao estruturar uma aplicação ou sistema desenvolvido em Clipper, é conveniente dividi-lo em módulos. Cada um destes módulos será chamado por um módulo principal (ou gerenciador) e possuirá seu próprio conjunto de programas-fonte. Para atender à essa necessidade, o Clipper permite que os seus programas, arquivos de procedimentos ("procedures") sejam compilados em grupos, de acordo com sua necessidade, sendo criado um módulo-objeto separado para cada grupo.

Além disso há uma série de outras razões, entre elas criação de "overlays" (discutida na seção 8.2.3 — Construindo "Overlays") e maior clareza, rapidez e eficiência na depuração de erros, que justificam a técnica de compilar um programa ou um grupo de programas separadamente, isto é, em módulos.

Uma outra situação que independe de você, mas que justifica a compilação de um programa ou de grupos de programas separadamente é a limitação das tabelas de constantes e de símbolos do Clipper. Quando compilando um programa, o Clipper estabelece duas tabelas: uma contendo as constantes utilizadas e outra contendo os símbolos. O número de constantes e de símbolos que podem ser colocados nestas tabelas obviamente não é infinito. Conseqüentemente, quando seus sistemas forem muito grandes, ao compilá-los de uma só vez em um único módulo-objeto, poderá ser apresentada uma mensagem indicando que uma das tabelas está cheia. Se o seu programa exceder 64 Kbytes para a tabela de constantes, será apresentada a seguinte mensagem de erro: "Too Many Constants". Da mesma forma, se for excedido o limite de 64 Kbytes da tabela de símbolos será apresentada a mensagem: "Too Many Symbols". Esta limitação do Clipper, entretanto, pode ser contornada compilando-se um ou um grupo programas separadamente, ao invés de compilá-los de uma única vez.

Para atender às necessidades expostas acima, o Clipper permite seja criado um arquivo-texto com a extensão ".CLP", que contém a lista dos nomes dos programas a serem compilados em um módulo-objeto separado.

Quando o Clipper for utilizado com um arquivo ".CLP", serão compilados **apenas** os programas fonte (.prg) que estiverem identificados neste arquivo. Mesmo que um programa incluído no arquivo ".CLP" chame outros programas, apenas ele será compilado; o Clipper não compilará os outros por ele referenciados.

Para criar um arquivo ".CLP" você pode utilizar o mesmo editor de textos que está sendo usado para criar os programas-fonte. O arquivo gerado deve possuir obrigatoriamente a extensão (.clp) e, em cada linha, o nome de um programa que deverá ser compilado, sem a extensão (.prg).

O exemplo a seguir ilustra um arquivo ".CLP", denominado "TESTE.clp", criado através do comando COPY CON. do DOS, que será utilizado para compilar apenas os programas GERAL.prg, ROTINAS.prg e HELP.prg, todos contidos no diretório \progs.

```
C:\clipper>COPY CON: TESTE.CLP
\progs\GERAL
\progs\ROTINAS
\progs\HELP
pressiona-se F6 e Enter para gravá-lo.

C:\clipper>TYPE TESTE.CLP
\progs\GERAL
\progs\ROTINAS
\progs\HELP
```

Para compilar os programas discriminados em um arquivo ".CLP", deve ser executado o seguinte comando no DOS:

C:\clipper>CLIPPER@<nome do arquivo ".CLP">

ou seja,

C:\clipper>CLIPPER@TESTE

O símbolo arrôba (@) indica para o Clipper que será utilizado um arquivo ".CLP" que contém os nomes dos programas a serem compilados.

O <nome do arquivo> define o nome do arquivo texto ".CLP" (previamente criado), contendo, em cada linha, o nome dos programas a serem compilados. Não deve ser incluída a extensão (.clp).

O comando do exemplo acima irá produzir, um único módulo-objeto, com o mesmo nome do arquivo ".CLP", ou seja TESTE.obj, contendo o código-objeto apenas dos programas nele discriminados (GERAL, ROTINAS e HELP).

Durante o processo de compilação através de um arquivo ".CLP" é normal que seja apresentada a mensagem "Cannot Open, Assumed External". Ela indica

apenas que o programa ou os programas que estão sendo compilados estão fazendo referência a outros que não foram incluídos na lista do arquivo ".CLP", sendo, portanto assumidos pelo Clipper como externos a esse grupo de programas. Posteriormente você deverá compilá-los, isoladamente ou em outros grupos, pois todos devem estar incluídos em módulos-objeto quando for feita a operação de link-edição para a geração do módulo executável.

O recurso de redirecionamento de saída do DOS também poderá ser utilizado com um arquivo ".CLP" para gravar as mensagens de compilação em um arquivo no disco. Este arquivo poderá ser posteriormente visto ou impresso através do comando TYPE do DOS, permitindo que você verifique com calma as eventuais mensagens de erros que foram detectados. Para criar um arquivo de saída, deve ser adicionado o sinal ">" e o nome do arquivo a ser gravado ao final da linha de comando do compilador, como ilustrado no exemplo a seguir:

```
C:\clipper>CLIPPER@TESTE >ERROS.TXT
```

O comando acima irá compilar somente os programas listados no arquivo "TESTE.clp" e gravará o arquivo "ERROS.txt", com as mensagens de saída do compilador. Este arquivo poderá ser então visto ou impresso através do comando TYPE do DOS.

8.2. ENCADEANDO (LINK-EDITANDO) OS PROGRAMAS

8.2.1. Utilizando o "Link-Editor"

O principal propósito de um "link-editor" ou encadeador é criar um arquivo ou módulo executável, a partir de um ou mais arquivos de código-objeto e arquivos de rotinas de biblioteca. Este arquivo poderá ser executado diretamente a partir do Sistema Operacional.

Os arquivos-objeto são identificados pela extensão ".OBJ", enquanto que os arquivos de bibliotecas, que contêm informações e rotinas internas usadas durante a execução do programa, são identificados pela extensão ".LIB". Os arquivos executáveis, gerados após a operação de encadeamento ("link-edição"), são identificados pela extensão ".EXE".

A versão Summer 85 do Clipper possui uma única biblioteca de rotinas internas, necessária à geração do módulo executável: a CLIPPER.lib. Nesta versão, portanto, apenas esta biblioteca deverá ser especificada ao link-editor.

As versões Winter 85 e Autumn 86 possuem três bibliotecas de rotinas internas: CLIPPER.lib, DBU.lib e MEMO.lib. De acordo com os comandos e funções que você estiver utilizando em seus programas, será necessário especificar ao link-editor a lista de bibliotecas a serem utilizadas. Verifique nos Caps. 5 e 6, no item "Biblioteca" de cada comando ou função, as bibliotecas necessárias à sua aplicação. Estas bibliotecas devem ser incluídas na lista de bibliotecas passada ao link-editor, como veremos mais adiante.

A versão Summer 87 do Clipper possui duas bibliotecas de rotinas internas, necessárias à geração do módulo executável: a CLIPPER.lib e a EXTEND.lib. Nesta versão, portanto, além da CLIPPER.lib, que é a biblioteca básica do Clipper, você deverá verificar se alguns dos comandos e funções utilizados em seus programas estão contidos na biblioteca EXTEND.lib. Caso estiverem, tanto a CLIPPER.lib como a EXTEND.lib devem ser especificadas na lista de bibliotecas passada ao link-editor. Verifique nos Caps. 5 e 6, no item "Biblioteca" de cada comando ou função, as bibliotecas necessárias à sua aplicação.

O link-editor do DOS ou o "Turbo-Link" da Borland podem ser utilizados para encadear os arquivos de módulos-objeto gerados pelo Clipper, desde que não seja necessária a criação de "overlays".

Quando uma aplicação for muito extensa poderá resultar em um código executável (arquivo .exe) muito grande que não poderá ser carregado de uma única vez na memória de um determinado equipamento. Para contornar isso, o arquivo executável poderá ser dividido em partes chamadas "overlays". O link-editor PLINK86 que acompanha o Clipper, permite a criação de "overlays", que são discutidos em mais detalhes na seção seguinte.

A operação de encadeamento através do "LINK.exe" (o link-editor do DOS) ou do "Tlink.exe" (o Turbo-Link da Borland) é muito mais rápida do que através do PLINK86. Portanto, se sua aplicação não exigir a criação de "overlays" é preferível utilizá-los ao invés do PLINK86.

O arquivo executável resultante, quer seja utilizado o link-editor do DOS, o Turbo-Link, o PLINK86 ou qualquer outro de sua preferência, será totalmente equivalente e irá "rodar" exatamente da mesma forma no seu computador.

8.2.1.1. Utilizando o "Link-Editor" do DOS

O link-editor do DOS encontra-se no arquivo LINK.exe, que normalmente acompanha todas as suas versões do DOS. Para executá-lo basta digitar o seguinte comando:

C:\clipper>LINK

Em seguida serão solicitados, através de mensagens, os nomes dos arquivos (com os respectivos diretórios onde se encontram) que participarão do processo, como detalhado a seguir:

Object Modules[.OBJ]:<nome>+<nome>+...+<nome>

Devem ser especificados os nomes dos arquivos objeto a serem link-editados. Cada nome deve estar ligado a outro através do sinal + (mais); não é necessário especificar a extensão (.obj). Pressione <Enter> ao final da lista.

Run File [.EXE]:<nome do arquivo executável>

Deve ser especificado o nome a ser dado para o arquivo executável (.exe) que será gerado como resultado da link-edição. Se o nome do arquivo executável for o mesmo do primeiro módulo-objeto, apenas pressiona-se <Enter>.

List File [NUL.MAP]:<nome do arquivo>

Pode ser especificado o nome do arquivo "mapa de memória", extensão (.map) a ser gerado. Este arquivo documenta o processo de link-edição, mostrando todas as alocações da memória. Caso não seja necessária a geração do arquivo de mapa apenas pressione <Enter>.

Libraries [.LIB]:<nome>+<nome>+...+<nome>

Devem ser especificados os nomes dos arquivos de biblioteca, com extensão (.lib), a serem utilizados na link-edição. Cada nome como CLIPPER, DBU, MEMO ou EXTEND, deve estar ligado a outro através do sinal + (mais); não é necessário especificar a extensão (.lib). Pressione <Enter> ao final da lista.

Após a especificação de todos os arquivos envolvidos na link-edição o processo será iniciado e, caso não ocorram erros, será gerado o arquivo executável com o nome especificado. Se ocorrem erros, mensagens apropriadas serão apresentadas e o arquivo executável não será gerado. Para maiores detalhes sobre a utilização do LINK.exe, refira-se ao manual do DOS.

8.2.1.2. Utilizando o Turbo-Link da Borland

O Turbo-Link ou "TLINK" da Borland é um link-editor **extremamente** rápido, que pode ser utilizado para link-editar suas aplicações desenvolvidas em Clipper, desde que elas não necessitem de "overlays".

O Turbo-Link normalmente está contido no arquivo "TLINK.EXE". Para utilizá-lo basta especificar os arquivos que participarão do processo de link-edição através da seguinte sintaxe:

C>TLINK <arqobj1+arqobj2+...+arqobjn>,<arqexe>,<arqmap>,
<arqlib1+arqlib2+...+arqlibn>

Onde:

<arqobj1+arqobj2+...+arqobjn>: são os nomes dos módulos-objeto que deverão ser link-editados. Cada nome deve ser ligado a outro através do sinal de mais (+). Não é necessário especificar a extensão (.obj).

<arqexe>: é o nome do arquivo executável a ser gerado. Não é necessário especificar a extensão (.exe).

<arqmap>: é o nome do arquivo de documentação que traz o mapa de alocação na memória dos segmentos do módulo executável. Não é necessário especificar a extensão (.map).

<arqlib1+arqlib2+...+arqlibn>: são os nomes das bibliotecas que deverão ser utilizadas na link-edição. Cada nome deve ser ligado a outro através do sinal de mais (+). Não é necessário especificar a extensão (.lib).

Por exemplo, para link-editar os módulos-objeto PROG1.obj, PROG2.obj, PROG3.obj, gerando o módulo executável PROG.exe e o mapa de alocação de memória PROG.map, onde são utilizadas as bibliotecas CLIPPER.lib e EXTEND.lib (versão Summer 87), utiliza-se o seguinte comando:

C<TLINK PROG1+PROG2+PROG3,PROG,PROG,CLIPPER+EXTEND

Além da forma padrão descrita acima, o "TLINK" oferece uma opção bastante interessante para facilitar o processo. Pode ser especificado um arquivo-texto (padrão ASCII) que contenha em suas linhas o nome dos arquivos que participarão do processo. Assim, ao invés de digitá-los a cada vez que for necessário refazer a link-edição, bastará informar ao TLINK o nome deste arquivo através da sintaxe abaixo:

C>TLINK@<nome do arquivo>

O símbolo arrôba (@), indica para o TLINK que deverá ser utilizado um arquivo-texto, cujo nome deve ser especificado em <nome do arquivo>, para a obtenção dos nomes dos arquivos que participarão do processo de link-edição.

Para link-editar os arquivos do exemplo anterior deve ser criado um arquivo-texto, através do seu editor preferido ou do comando COPY CON: do DOS, chamado, por exemplo, "LINKA.lnk". Este arquivo deverá especificar em cada linha e nesta ordem os nomes dos arquivos-objeto, do arquivo executável, do arquivo de mapa e das bibliotecas a serem utilizadas. Caso haja necessidade de utilizar-se mais de uma linha para especificar um dos tipos de arquivos, basta terminar a linha anterior com o sinal de mais (+) e continuar a especificação dos arquivos na linha imediatamente seguinte. O exemplo anterior usaria o seguinte arquivo:

Arquivo "LINKA.lnk" para o exemplo anterior:

```
PROG1+PROG2+PROG3
PROG
PROG
CLIPPER+EXTEND
```

Para executar o TLINK, utilizando o arquivo LINKA.lnk deve ser fornecido o seguinte comando:

C>TLINK@LINKA

Utilizando-se arquivos-texto deste tipo, você terá muito mais facilidade para link-editar suas aplicações através do TLINK.

Outras opções do Tlink, que podem ser fornecidas ao final da linha de comando:

/m => gera o mapa de alocações com segmentos públicos

/x => não gera o arquivo de mapeamento

/i => inicializa todos os segmentos

/l => inclui a numeração das linhas dos programas fonte no programa executável

/s => mapeamento detalhado dos segmentos

/n => não utiliza as bibliotecas padrão

/d => avisa se forem encontrados símbolos duplicados nas bibliotecas utilizadas

/c => trata letras maiúsculas e minúsculas como sendo caracteres diferentes nos símbolos

Para maiores detalhes sobre a utilização do Turbo-Link consulte o seu manual.

8.2.1.3. Utilizando o PLINK86-Plus

A versão do PLINK86-Plus que acompanha o Clipper é específica para compilar módulos-objeto criados pelo compilador do Clipper. Por esse motivo, o primeiro módulo-objeto a ser encadeado deve ter sido obrigatoriamente gerado pelo Clipper, caso contrário será retornada uma mensagem de erro indicando que não se trata de arquivo do Clipper. Os outros módulos-objeto que vierem a seguir poderão ter sido gerados por outros compiladores através de outras linguagens.

O PLINK86 é em geral mais lento que outros link-editores, por esse motivo recomendamos utilizá-lo apenas quando houver a necessidade de serem construídos "overlays".

Há três métodos através dos quais pode-se utilizar o PLINK86:

- Interativamente
- Através de uma linha de comando
- Com um arquivo tipo ".LNK".

Se ocorrer um erro durante o processo de link-edição, uma mensagem com o respectivo código será apresentada na tela. Explicações sucintas sobre o significado destas mensagens podem ser encontradas no Apêndice, seção 13.2 — Erros mais Comuns Durante a Link-Edição. Para maiores detalhes refira-se ao manual do Clipper.

8.2.1.3.1. O Método Interativo

Para utilizar o PLINK86 interativamente deve-se apenas digitar PLINK86 na linha de comando do DOS e teclar-se <Enter>. O link-editor será carregado e apresentará o sinal "=>". Após o sinal, pode ser digitado um único comando ou uma série de comandos na mesma linha. Cada vez que for pressionado <Enter>, a linha atual será completada e o sinal mudará para a seguinte. Para finalizar os comandos basta digitar ponto-e-vírgula (;) e pressionar <Enter> na última linha desejada.

No exemplo a seguir são fornecidos os comandos necessários para link-editar os módulos-objeto PRIMEIRO.obj, SEGUNDO.obj e TERCEIRO.obj, gerando o módulo executável TESTE.exe.

```
C:\clipper>PLINK86
=>FILE PRIMEIRO,SEGUNDO,TERCEIRO
=>OUTPUT TESTE
=>LIBRARY CLIPPER
=>;
```

Onde:

- **FILE** (ou apenas FI) é o comando que indica a lista de módulos-objeto que deverão ser link-editados. Cada nome deverá ser separado por vírgulas.
- **OUTPUT** (ou apenas OUT) é o comando que indica o nome do módulo executável que deverá ser gerado; no caso TESTE.exe.
- **LIBRARY** (ou apenas LIB) é o comando que indica a lista de bibliotecas onde se encontram as rotinas internas a serem utilizadas na link-edição. Cada nome deverá ser separado por vírgulas.
- O **ponto-e-vírgula** finaliza a entrada de informações e inicia o processo de link-edição.

Atenção: Todos os comandos do PLINK86 deve ser escritos obrigatoriamente em letras maiúsculas.

8.2.1.3.2. O Método da Linha de Comando

Através deste método, em uma única linha de comando do DOS é possível especificar ao PLINK86 todos os arquivos que deverão participar do processo. A sintaxe que deverá ser utilizada é a seguinte:

C>PLINK86 FI <[d:\dir\]arq1.obj>,<arq2.obj>...,<arqn.obj>
LIB <[d:\dir\]bibli1.lib>...,<biblin.lib>

Onde:

PLINK86 carrega o link-editor;

FI ou FILE, especifica os módulos-objeto a serem encadeados; caso não estejam no mesmo diretório do PLINK86, d: indica o drive e \dir\ indica o diretório onde se localizam.
<arq1.obj>, <arq2.obj>...,<arqn.obj> são os nomes dos arquivos-objeto. Não é necessário especificar a extensão (.obj), pois ela será assumida.
LIB ou LIBRARY, especifica das bibliotecas de rotinas internas a serem utilizadas. Se este comando não for especificado, por definição serão utilizadas as bibliotecas CLIPPER.lib (do Clipper) e OVERLAY.lib (do PLINK86, no caso de utilização de overlays). Caso as bibliotecas não estejam no mesmo diretório do PLINK86, d: indica o drive e \dir\ o diretório onde se localizam.
<bibli1.lib>, <bibli2.lib>...,<bibli.lib> são os nomes dos arquivos que contêm as rotinas de biblioteca. Não é necessário especificar a extensão (.lib), pois ela é assumida.

O arquivo executável gerado será identificado pelo nome do primeiro módulo-objeto especificado e receberá a extensão (.exe). Por exemplo, o comando abaixo produz o módulo executável "TESTE", a partir dos módulos-objeto "TESTE.obj" "TESTE1.obj", "TESTE2.obj" e "TESTE3.obj" e da biblioteca CLIPPER.lib.

C:\clipper>PLINK86 FI TEST,TESTE1,TESTE2,TESTE3, LIB CLIPPER

8.2.1.3.3. O Método do Arquivo .LNK

O terceiro método permite que seja criado um arquivo-texto com a extensão (.lnk) que conterá os comandos e arquivos a serem passados ao PLINK86 através da seguinte sintaxe:

C:\clipper>PLINK86 @<nome do arquivo>

O símbolo arrôba (@) indica ao PLINK86 que deverá ser utilizado o arquivo-texto especificado, para a obtenção dos comandos e arquivos a serem utilizados no processo de link-edição.

O <nome do arquivo> define o arquivo-texto, criado com o editor de textos de sua preferência, que contêm os comandos e arquivos a serem passados para o PLINK86. A extensão (.lnk) será assumida, a menos que seja especificada outra extensão.

O exemplo a seguir demonstra a utilização do arquivo "LINKA.lnk" para a geração do arquivo executável "ESTOQUE.exe", a partir dos módulos-objeto EST.obj, EST1.obj, EST2.obj, EST3.obj e EST4.obj.

C:\clipper>PLINK86@LINKA

O arquivo-texto LINKA.lnk contêm as seguintes linhas:

```
FILE EST,EST1,EST2,EST3,EST4
OUTPUT ESTOQUE
LIBRARY CLIPPER,EXTEND
VERBOSE
```

Onde:

FILE ou apenas FI, especifica os módulos-objeto, separados por vírgulas, a serem encadeados; se necessário devem ser especificados o drive e o diretório onde se localizam. A extensão (.obj) é desnecessária.

OUTPUT ou apenas OUT, especifica o nome do arquivo executável a ser gravado. A extensão (.exe) é desnecessária.

LIBRARY ou apenas LIB, especifica as bibliotecas de rotinas a serem utilizadas (por exemplo CLIPPER.lib, DBU.lib, MEMO.lib ou EXTEND.lib); se necessário deve ser especificado o drive e o diretório onde se localizam. Não é necessário especificar a extensão (.lib), pois ela já é subentendida. Se este comando não for especificado, por definição serão utilizadas as bibliotecas CLIPPER.lib (básica do Clipper) e OVERLAY.lib (básica ao PLINK86), a serem localizadas no diretório atual.

VERBOSE é um comando que determina a apresentação na tela de mensagens relativas às operações efetuadas pelo PLINK86 durante o processo de link-edição. Este comando é útil para verificar as fases da link-edição, contudo torna a sua execução um pouco mais demorada.

Os arquivos ".LNK" podem ser utilizados juntamente com outros comandos do PLINK86. Por exemplo:

C>PLINK86@TESTE2 VERBOSE OUTPUT SISTEMA

inicia o processo de link-edição utilizando o arquivo TESTE2.lnk, invoca o comando VERBOSE para a apresentação de mensagens e define pelo comando OUTPUT o nome do arquivo executável a ser gerado.

Quaisquer comandos do PLINK86 podem ser utilizados em qualquer um dos três métodos apresentados. Entretanto, quando as aplicações a serem link-editadas forem complexas e possuírem "overlays" (discutidos na próxima seção), o uso de um arquivo ".LNK", apresentado pelo último método é recomendado.

8.2.1.3.4. Utilizando Arquivos Lote ("Batch") e Documentando o Processo

O uso de arquivos de processamento em lote, tipo ".BAT", é recomendado para agilizar o processo de compilação e "link-edição". Por exemplo, para compilar e link-editar o sistema "PROG.prg", pode ser utilizado o arquivo "CLIPA.bat":

```
ECHO OFF
CLS
CLIPPER %1 >%1.TXT
IF NOT ERRORLEVEL 1 PLINK86 FI %1
CLS
```

Para executá-lo digita-se apenas:

```
C:\clipper>CLIPA PROG
```

Ao compilar um programa, quando erros são encontrados, o Clipper reage de duas formas:

- Apresenta mensagens na tela, indicando o tipo de erro encontrado;
- Retorna para o DOS o código de erro 1.

Este código de erro pode ser detectado pelo DOS através do comando "IF NOT ERRORLEVEL", que pode ser utilizado, como no exemplo acima, para evitar que um programa contendo erros seja link-editado. Logo, o programa somente será link-editado se nenhum erro for encontrado durante o processo de compilação, sendo então gerado o arquivo executável "PROG.exe".

Também é possível documentar o processo de link-edição utilizando o recurso de redirecionamento de saída do DOS. Através da especificação do símbolo ">" as mensagens normalmente apresentadas na tela podem ser gravadas em um arquivo do disco. Por exemplo:

C:\clipper>PLINK86 FI PROG >LINKDOC.TXT

ou

C:\clipper>PLINK86@LINKA.LNK >LINKDOC.TXT

Nestes exemplos será gravado o arquivo LINKDOC.txt, que conterá todas as mensagens que o PLINK86 envia para a tela. Você poderá posteriormente vê-lo ou imprimi-lo através do comando TYPE do DOS.

8.2.1.3.5. O Resultado do "link-edição"

Ao final do processo de link-edição através do PLINK86, se este for bem sucedido, será apresentada uma mensagem na tela indicando o nome do arquivo executável gerado e a quantidade de Kbytes de memória RAM que será necessária para sua execução, como exemplificado abaixo:

PROG.EXE (255 K)

A quantidade de Kbytes apresentada não corresponde ao tamanho do arquivo executável no disco (que em geral é maior, particularmente quando são usados "overlays"), mas sim ao montante de memória que será ocupada durante sua execução. Além dessa quantidade de memória, devem ser adicionados de 64 a 100 Kbytes necessários à manipulação de variáveis e áreas de trabalho dos arquivos. O resultado será a quantidade mínima de memória requerida para executar o programa.

8.2.2. Link-Editando com os Utilitários do Clipper

Dependendo da versão do Clipper que você estiver utilizando, há alguns utilitários que poderão ser link-editados com os seus programas:

- O "Debugador do Clipper" (todas as versões)
- As Funções Complementares (Winter 85 e Autumn 86)
- O "Gerador de Índices NDX" (Summer 87)

8.2.2.1. O "Debugador do Clipper"

O "Debugador do Clipper" é um dos utilitários mais importantes, e está presente em todas as versões. Este utilitário facilita a depuração de erros através do fornecimento de várias informações durante a execução dos programas, principalmente quando há uma situação de erro.

Para incluir o "Debugador" em seus programas executáveis será apenas necessário incluir o arquivo-objeto DEBUG.obj que acompanha o Clipper na lista de módulos-objeto passada ao link-editor.

O exemplo a seguir utiliza o PLINK86 para link-editar os módulos-objeto PROG.obj e PROG2.obj, que correspondem à aplicação desenvolvida, e o DEBUG.obj, que é o módulo-objeto que contêm o "Debugador". Este deverá sempre ser incluído na lista após os módulos-objeto de sua aplicação.

C:\clipper>PLINK86 FI PROG,PROG2,DEBUG

Através do comando acima será gerado o módulo executável PROG.exe que conterá o debugador. Para invocá-lo durante a execução do PROG.exe tecla-se <Alt-D> simultaneamente. Para maiores detalhes sobre a utilização do debugador veja a seção 9.2 — O Debugador do Clipper.

8.2.2.2. As Funções Complementares

Com o objetivo de manter a máxima compatibilidade possível com o dBASE, ao ser lançado o dBASE III Plus, foram incluídas, de forma provisória, nas versões Winter 85 e Autumn 86, uma série de funções complementares. Estas funções estão contidas em três arquivos-objeto:

- **EXTENDDB.obj**: que contêm funções escritas em Clipper através do comando FUNCTION;
- **EXTENDC.obj**: que contêm funções escritas em linguagem "C" e
- **EXTENDA.obj**: que contêm funções escritas em Assembler.

Se sua aplicação estiver utilizando funções que estão contidas em um destes módulos-objeto, o mesmo deverá ser incluído na lista de módulos-objeto passada ao link-editor. Verifique no Cap. 6, no item "Biblioteca" de cada função, se será necessário incluir um destes módulos na sua aplicação. Inclua-os juntamente com os módulos-objeto gerados a partir dos seus programas-fonte.

Por exemplo, para link-editar os módulos-objeto PROG.obj, PROG1.obj e PROG3.obj, juntamente com os módulos EXTENDDB.obj e EXTENDC.obj através do PLINK86 pode ser utilizado o seguinte comando:

C:\clipper>PLINK86 FI PROG,PROG2,PROG3,EXTEDDDB,EXTENDC

Através dele será gerado o arquivo executável PROG.exe que conterá todas as funções complementares definidas nos módulos EXTENDDB.obj e EXTENDC.obj, tornando-as disponíveis para a aplicação desenvolvida.

Se seus programas fazem uso de uma das funções contidas nestes módulos e o mesmo não for link-editado juntamente com os módulos-objeto de sua aplicação, será apresentada uma mensagem de erro pelo link-editor, acusando símbolo indefinido ("Undefined Symbol"). Esta mensagem indica que a função requerida não foi definida em nenhum módulo-objeto ou biblioteca passados ao link-editor.

Na versão Summer 87, a maioria das funções complementares foi incluída nas bibliotecas do Clipper. Neste caso você não deve link-editar estes módulos-objeto com os módulos de sua aplicação.

Para mais informações consulte também o Cap. 6 — Funções do Clipper.

8.2.2.3. O Gerador de Índices NDX

A versão Summer 87 vem acompanhada com um módulo-objeto denominado "NDX.obj". Link-editando este módulo juntamente com os módulos-objeto de sua aplicação o Clipper permitirá automaticamente que sejam criados e utilizados índices do tipo ".NDX", compatíveis com os do dBASE III Plus.

Uma vez link-editado o módulo NDX.obj com sua aplicação, o programa executável gerado passará a trabalhar normalmente com os índices (.ndx). Não é possível, contudo, utilizar simultaneamente índices dos dois tipos: (.ntx) do Clipper e (.ndx) do dBASE.

Por exemplo, para link-editar os módulos-objeto PROG.obj, PROG1.obj e PROG3.obj, juntamente com o módulo NDX.obj através do PLINK86 pode ser utilizado o seguinte comando:

```
C:\clipper>PLINK86 FI PROG,PROG2,PROG3,NDX
```

Será gerado o arquivo executável PROG.exe que utilizará índices compatíveis com os do dBASE III/Plus, ao invés dos índices (.ntx), normais do Clipper e que são incompatíveis com os do dBASE.

Apesar do Clipper agora permitir a utilização de índices compatíveis com os do dBASE III é preferível utilizar os índices (.ntx), que trabalham mais de acordo com a arquitetura do Clipper. Com eles o processamento é muito mais rápido e eficiente.

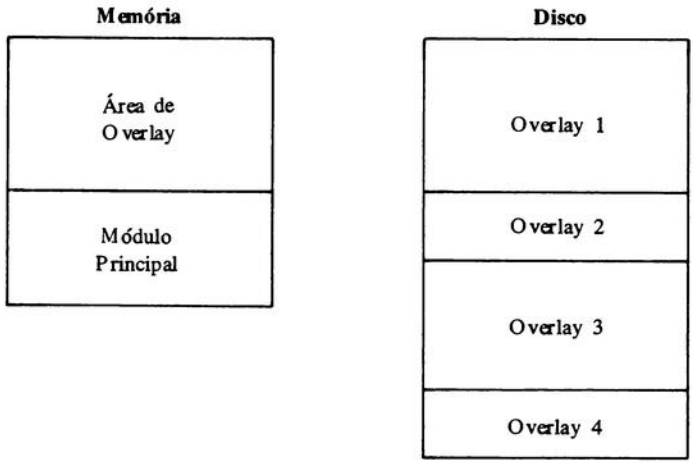
8.2.3. Construindo "OVERLAYS"

8.2.3.1. O que são "Overlays"

A construção de "overlays" é uma técnica que permite que programas grandes possam ser executados utilizando-se pouca quantidade de memória. Portanto, construindo "overlays" poderão ser executados programas muito maiores do que a memória disponível no seu computador poderia suportar.

"Overlays" são partes ou segmentos do seu programa aplicativo que não irão permanecer na memória RAM do computador até que sua execução seja solicitada. A parte principal de um programa de aplicação, ou o módulo principal, estará sempre residente na memória durante a execução. Os overlays ficam armazenados no disco até que o módulo principal os "chame" para serem executados. A porção da memória que os overlays dividem entre si é chamada de "Área de Overlay" e terá o seu tamanho determinado pelo maior dos overlays.

Um overlay será carregado para a "Área de Overlay" sempre que uma rotina nele contida for solicitada pelo módulo principal. Se na Área de Overlay já existir um outro overlay carregado anteriormente, este será sobreposto pelo atual. Assim, sempre que um novo overlay for carregado, irá sobrepor o anterior na mesma área da memória. É exatamente esta troca de overlays na mesma área de memória que permite economizar espaço. O esquema a seguir ilustra como os overlays são alocados na memória.



Suponha que o Overlay 1, que é o maior deles e que portanto determina o tamanho da Área de Overlay na memória, seja carregado na memória e executado. Em seguida o controle retorna para o Módulo Principal que chama o Overlay 4, que não está atualmente na memória. O Overlay 4 é, então, lido do disco e carregado na memória na mesma Área de Overlay, sobrepondo-se ao Overlay 1, para então ser executado. O processo se repete cada vez que um novo overlay é solicitado, havendo, portanto, uma troca constante de overlays entre a Área de Overlay e o disco.

Um programa que contenha um único Overlay não faz sentido, pois não reduzirá em nada a quantidade de memória necessária para sua execução; a Área de Overlay terá exatamente o seu tamanho. O uso eficiente da memória é conseguido quando vários Overlays dividem o mesmo espaço da memória em diferentes instantes da execução do sistema ou programa aplicativo.

A menos que você esteja certo que seu programa aplicativo deva possuir overlays, tente inicialmente link-editá-lo e executá-lo como um único módulo. O uso de overlays, quando a quantidade de memória exigida pelo módulo executável não exceder a capacidade de memória do seu computador, irá apenas tornar mais lenta a execução do sistema, devido à constante necessidade de leitura e carga dos overlays do disco para a memória.

Por outro lado, se o seu programa aplicativo exigir uma quantidade de memória superior à do equipamento onde deverá ser executado, a utilização de overlays praticamente será obrigatória, uma vez que é muito difícil reduzir o tamanho de uma aplicação, assumindo que nela foram utilizadas técnicas corretas de programação modularizada.

Desde que sua aplicação tenha sido construída modularmente, a utilização de overlays não requer modificações no código-fonte dos programas. A estrutura dos overlays é definida através dos comandos do link-editor PLINK86. Os overlays são depois automaticamente manipulados pelo programa durante sua execução. Se você decidir mudar a estrutura de seus overlays, apenas será necessário alterar os módulos-objeto compilados e os comandos do link-editor; o código-fonte dos programas permanecerá o mesmo.

Quando são desenvolvidos grandes sistemas aplicativos, a serem executados como um único módulo, deve haver uma considerável preocupação quanto ao projeto de sua estrutura lógica. A maioria das aplicações realiza determinadas operações que podem ser divididas em grupos logicamente independentes. Estes grupos geralmente podem ser armazenados em arquivos separados que não requerem ou requerem a mínima interação entre si. Este projeto é a base para a construção de uma estrutura eficiente de overlays.

O objetivo principal de um projeto de sistema que utilize overlays é criar overlays contendo grupos funcionalmente isolados de programas, que poderão ser executados independentemente uns dos outros. Estes grupos ou módulos podem conter um ou mais programas, procedures, funções-de-usuário ou telas de formatação (arquivos .prg e .fmt). Quanto mais modularizado for o projeto do sistema e quanto mais independentes forem os seus módulos, mais fácil se tornará a construção e a manutenção de overlays.

8.2.3.2. Construindo Overlays

Overlays são criados através de comandos especiais do PLINK86 durante o processo de link-edição, com a utilização de um arquivo do tipo ".LNK".

Para melhor explicar o processo de criação de overlays, apresentaremos um exemplo completo, no qual descreveremos todos os passos necessários para construí-los.

Considere inicialmente um sistema hipotético que possua os módulos descritos a seguir:

Programas-Fonte	**Módulos-Objeto**	**Overlays**
SIST.prg SISTPROC.prg HELP.prg	→ SISTEMA.obj	(Módulo Principal)

Programas-Fonte	Módulos-Objeto	Overlays
SIST1.prg SIST11.prg SIST12.prg SIST13.prg SIST14.prg SIST15.prg	→ SIST1.obj	→ SIST (Overlay 1)
SIST2.prg SIST21.prg SIST22.prg SIST23.prg SIST24.prg SIST25.prg	→ SIST2.obj	
UTIL.prg UTIL1.prg UTIL2.prg UTIL3.prg UTIL4.prg UTIL5.prg	→ UTIL.obj	→ UTIL (Overlay 2)
REL.prg REL1.prg REL2.prg REL3.prg REL4.prg REL5.prg	→ REL.obj	→ REL (Overlay 3)

Em seguida apresentaremos todos os passos necessários para a geração do módulo executável SISTEMA.exe com a estrutura de overlays ilustrada acima.

Operações para a construção módulo SISTEMA.exe:

1 – Construir o arquivo SISTEMA.clp para a compilação dos programas que farão parte do módulo principal: SIST.prg, gerenciador principal do sistema, SISTPROC.prg arquivo de rotinas (procedures) comuns a todos os programas do sistema, HELP.prg arquivo de "help" do sistema. Este arquivo deverá conter as seguintes linhas:

O arquivo "Sistema.clp"

SIST
SISTPROC
HELP

2 — Compilar os programas do módulo principal, para produzir o módulo objeto SISTEMA.obj, através do seguinte comando:

C:\clipper>CLIPPER @SISTEMA

3 — Compilar os programas SIST1.prg, SIST2.prg, UTIL.prg, REL.prg, produzindo-se os respectivos módulos-objeto SIST1.obj, SIST2.obj, UTIL.obj, REL.obj. Assume-se que os programas SIST11.prg, SIST12.prg, SIST13.prg etc. são chamados pelo programa SIST1.prg, o mesmo ocorrendo para o SIST2.prg, UTIL.prg e REL.prg. Para isso, utilizam-se os seguintes comandos:

C:\clipper>CLIPPER SIST1
C:\clipper>CLIPPER SIST2
C:\clipper>CLIPPER UTIL
C:\clipper>CLIPPER REL

4 — Construir o arquivo SISTEMA.lnk, a ser utilizado para a link-edição dos módulos-objeto gerados, com a construção automática dos overlays SIST, UTIL e REL, gerando um único módulo executável SISTEMA.exe. Discutiremos os comandos específicos para a construção dos overlays logo a seguir. O arquivo SISTEMA.lnk deverá conter as seguintes linhas:

Arquivo SISTEMA.lnk

```
FI SISTEMA
LIB CLIPPER,EXTEND,OVERLAY
DEBUG
OVERLAY PROG, $CONSTANTS (ou OVERLAY CODE)
BEGINAREA
    SECTION FILE SIST1,SIST2
    SECTION FILE UTIL
    SECTION FILE REL
ENDAREA
```

É importante notar que todos os comandos devem ser especificados em **letras maiúsculas**, caso contrário o PLINK86 não os reconhecerá.

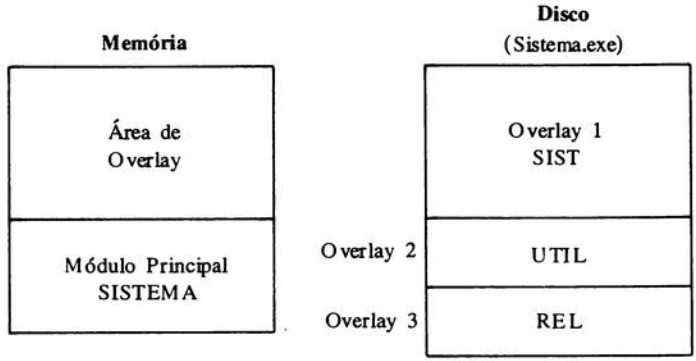
Atenção: Na versão Summer 87, o Clipper vem acompanhado da versão 2.24 do PLINK86 Plus. Neste caso o comando OVERLAY PROG, $CONSTANTS deve ser substituído por OVERLAY CODE.

5 — Link-editar os módulos do sistema utilizando o PLINK86 através do seguinte comando:

C:\clipper>PLINK86@SISTEMA

6 — Se nenhum erro for encontrado em cada uma das fases, será gerado o módulo executável SISTEMA.exe, que possuirá o módulo principal SISTEMA e três overlays: SIST, UTIL e REL, que se alternarão em uma única Área de Overlay na memória.

O esquema abaixo dá uma idéia de como a estrutura de overlays criada no exemplo anterior, através do arquivo SISTEMA.lnk, será alocada na memória e no disco do computador.



Quando um programa é link-editado, o PLINK86 separa as informações contidas nos arquivos de módulos-objeto em classes. A classe "data" (dados) contém informações constantes, como "strings" e nomes de variáveis. A classe "code" (código) contém realmente as instruções (código) dos programas. O Clipper normalmente aloca as constantes no módulo principal, contudo, para minimizar a utilização da memória, o comando **OVERLAY PROG, $CONSTANTS** (nas versões Winter 85 e Autumn 86) ou **OVERLAY CODE** (na versão Summer 87) é usado para alocar as informações da classe "data" nos overlays.

O comando **BEGINAREA** define o início da Área de Overlay, enquanto que o seu fim é indicado pelo comando **ENDAREA**. Os comandos **SECTION FILE**, existentes entre o **BEGINAREA** e o **ENDAREA**, automaticamente criam overlays relativos aos módulos-objeto especificados. Overlays assim definidos irão compartilhar um mesmo espaço da memória, a Área de Overlays.

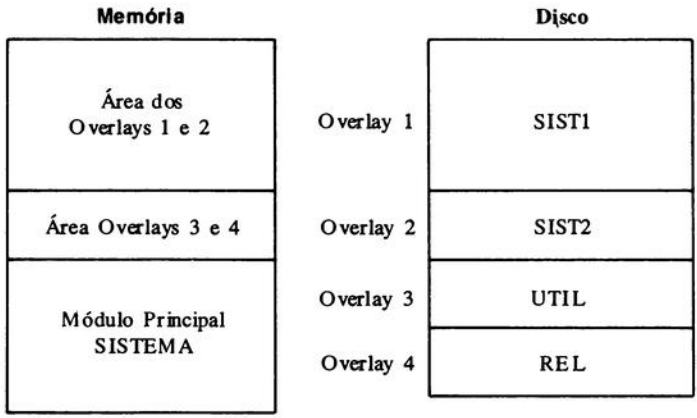
Quando o programa SISTEMA.exe é executado, o módulo principal é carregado na memória e o espaço requerido pelo maior overlay é reservado como Área de Overlay. No exemplo o overlay SIST, contendo os módulos-objeto SIST1.obj SISTE2.obj, determina o tamanho da Área de Overlay pois é o maior deles. Quando ele é carregado ocupa toda a Área de Overlay. Se outro overlay for carregado

irá sobrepor totalmente o anterior, mesmo que não ocupe toda a Área de Overlay. A área reservada que não for ocupada não será utilizada enquanto este overlay permanecer na memória.

Um programa contido em um overlay **jamais** deverá chamar outro programa contido em outro overlay que compartilhe a mesma Área de Overlay, pois ambos nunca poderão estar na memória simultaneamente. Neste caso somente poderão ser chamados programas ou rotinas contidos no mesmo overlay, no módulo principal ou em outras Áreas de Overlays, que, como veremos a seguir, também podem ser criadas.

A configuração básica da memória quando da utilização de overlays consiste de uma Área para o Módulo Principal e uma Área de Overlay, contudo também podem ser construídas múltiplas Áreas de Overlays, dependendo da necessidade de cada sistema aplicativo desenvolvido. O exemplo esquematizado na figura abaixo foi obtido através do seguinte arquivo ".LNK":

```
                FI SISTEMA
                LIB CLIPPER,OVERLAY,EXTEND
                DEBUG
                OVERLAY PROG, $CONSTANTS (ou OVERLAY CODE)
                BEGIN AREA
Primeira            SECTION FILE SIST1
  Área              SECTION FILE SIST2
                ENDAREA
                BEGIN AREA
Segunda             SECTION FILE UTIL
  Área              SECTION FILE REL
                ENDAREA
```



Note que, neste caso, a memória requerida para ambas as Áreas de Overlay é inferior à utilizada pela área única do exemplo anterior. Isto pode acontecer porque o maior overlay, SIST, composto pelos módulos-objeto SIST1 e SIST2 foi dividido em dois SIST1 e SIST2.

A utilização de múltiplas Áreas de Overlays pode também ser necessária quando um programa contido no overlay carregado na Área de Overlay 1 necessitar chamar outro programa contido em um overlay na Área de Overlay 2, e assim por diante.

É importante frisar que a utilização "ótima" da memória é conseguida quando todos os overlays de uma dada Área forem aproximadamente do mesmo tamanho. Lembre-se que o maior overlay de cada Área determinará a quantidade de memória a ela alocada durante a execução do sistema. Se os overlays possuírem tamanhos muito diferentes haverá desperdício na utilização da memória. Você poderá estimar grosseiramente o tamanho dos overlays através dos módulos-objeto que os compõem.

8.2.3.3. Tipos de Overlays

Utilizando-se o PLINK86 podem ser criados dois tipos de overlays: internos e externos. Ambos funcionam de forma idêntica, diferenciando-se apenas na forma como são armazenados no disco.

Overlays internos são armazenados em um único arquivo no disco, o mesmo arquivo executável do módulo principal, com a extensão ".EXE". Os overlays externos são armazenados em arquivos próprios, identificados pela extensão ".OVL".

Os overlays internos, que estivemos vendo até agora, são criados através do comando SECTION FILE do PLINK86. Eles são gerados e incluídos juntamente com o módulo principal dentro do mesmo arquivo .EXE, não sendo visíveis para o sistema operacional. Portanto o arquivo executável, neste caso, poderá possuir grandes dimensões, por exemplo 500 Kbytes, pois conterá o módulo principal e todos os overlays que compõem o sistema, não sendo portanto possível gravá-lo em um único disquete.

Já os overlays externos, sendo gravados em arquivos separados, que recebem a extensão ".OVL", não são incluídos no arquivo ".EXE" que contêm o módulo principal. Com isto, os arquivos ".OVL" podem ser transferidos para um disquete diferente daquele que contêm o módulo principal, sendo necessários apenas quando o programa estiver sendo executado. Durante a execução você poderá inicialmente colocar no drive o disquete que contêm o módulo principal, e assim que este for carregado, substituí-lo pelo disquete que contêm os overlays, solucionando o problema de acomodação do sistema, gravando-o em disquetes separados.

Deve ser observado, contudo, que a operação com overlays externos é mais lenta do que com overlays internos, pois será necessário "buscá-los" em arquivos diferentes, exigindo leituras do diretório.

Os overlays externos são gravados em arquivos separados através do comando **SECTION INTO ... FILE** do PLINK86, cuja sintaxe é a seguinte:

SECTION INTO <arquivo overlay> FILE <módulos-objeto>

Onde:

<arquivo overlay> é o nome do arquivo de overlay externo a ser criado.
<módulos-objeto> são os nomes (separados por vírgulas) dos módulos-objeto que deverão estar contidos neste overlay.

Para criá-los, pode ser utilizado o seguinte arquivo ".LNK":

```
FI SISTEMA
LIB CLIPPER,EXTEND,OVERLAY
OVERLAY PROG, $CONSTANTS (ou OVERLAY CODE)
BEGINAREA
    SECTION INTO SIST.OVL FILE SIST1,SIST2
    SECTION INTO UTIL.OVL FILE UTIL
    SECTION INTO REL.OVL FILE REL
ENDAREA
```

Neste exemplo serão produzidos quatro arquivos no disco:

– SISTEMA.exe	– que conterá o módulo principal.
– SIST.ovl	– que conterá o overlay do SIST1 e SIST2.
– UTIL.ovl	– que conterá o overlay do módulo UTIL.
– REL.ovl	– que conterá o overlay do módulo REL.

Para executar o programa aplicativo será inicialmente carregado o módulo principal SISTEMA, a seguir, conforme forem solicitados, serão procurados no disco e carregados na Área de Overlay os arquivos SIST.ovl, UTIL.ovl ou REL.ovl, que contêm os overlays do sistema.

Com a criação de "overlays" externos, tanto a limitação da capacidade da memória do equipamento utilizado, como a da capacidade dos disquetes que conterão a aplicação desenvolvida poderão ser contornadas.

8.3. EXECUTANDO OS PROGRAMAS

8.3.1. Configurando o DOS

As aplicações de Bancos de Dados, como tipicamente são as desenvolvidas em Clipper, envolvem, em geral, grande quantidade de arquivos e requerem uma utilização maior do disco do que outros tipos de aplicações. Para otimizar a utilização e operação do disco, o DOS permite que seja alterado o ambiente de trabalho padrão com o qual normalmente opera.

Para executar eficientemente um programa ou sistema desenvolvido em Clipper, dois parâmetros básicos do DOS devem ser reconfigurados:

- **FILES**: parâmetro que especifica ao DOS o número máximo de arquivos (de qualquer tipo) que podem ser abertos simultaneamente. O padrão do DOS é oito (8), ou seja, no máximo poderão ser abertos e manipulados simultaneamente até oito arquivos.

- **BUFFERS**: parâmetro que especifica o número de buffers (áreas da memória reservadas para a transição de dados) que devem ser utilizados durante as operações de leitura e gravação no disco. O padrão do DOS é dois (2), ou seja, apenas dois buffers de leitura e gravação no disco.

O número de arquivos abertos simultaneamente refere-se a arquivos de qualquer tipo (.dbf, .dbt, .ntx, .mem etc.) abertos em um dado momento pela aplicação desenvolvida, e dependerá das características de cada aplicação. Por exemplo, as linhas de comando abaixo abrem simultaneamente 10 arquivos (três de dados e sete de índice):

```
SELECT 1
USE Mala INDEX Mala1,Mala2,Mala3     && são abertos 4 arquivos
SELECT 2
USE Contatos INDEX Contato1          && são abertos 2 arquivos
SELECT 3
USE Clientes INDEX Cli1,Cli2,Cli3    && são abertos 4 arquivos
```

É comum, portanto, que as aplicações desenvolvidas requeiram a abertura de mais do que 8 arquivos; neste caso o parâmetro FILES do DOS deverá ser reconfigurado.

Caso seja feita uma tentativa para abrir mais do que 8 arquivos sem a devida reconfiguração, será apresentada pelo DOS a mensagem de erro: "Too many files are Open", que no Clipper é codificada como erro 4 do DOS ("DOS error 4").

Um programa ou sistema compilado através do Clipper pode abrir até 15 arquivos simultaneamente (ou 255 se estiver sendo utilizado o DOS 3.3 e a versão Summer 87), o que sem dúvida supera o número de arquivos padrão que podem ser abertos pelo DOS.

Por outro lado, a quantidade de buffers a ser utilizada para leitura e gravação de dados no disco depende exclusivamente da arquitetura interna do Clipper. A Nantucket, após exaustivos testes, chegou à conclusão de que o número ideal de buffers a ser utilizado durante a execução das aplicações desenvolvidas em Clipper é oito (8).

A reconfiguração dos parâmetros FILES e BUFFERS é realizada através da criação de um arquivo denominado **CONFIG.sys**, específico para a alteração de parâmetros do DOS. Ao ser carregado (ou quando se executa um "boot"), o DOS procura no diretório raiz o arquivo CONFIG.sys. Quando este arquivo é encontrado, o Sistema Operacional é configurado de acordo com as definições nele contidas, não assumindo as padrão.

Para executar as aplicações desenvolvidas em Clipper o arquivo CONFIG.sys deverá ser gravado no diretório raiz (ou naquele a partir do qual o DOS é carregado) e conter as seguintes linhas:

Arquivo "CONFIG.sys"

```
FILES=20
BUFFERS=8
```

Sugere-se que seja utilizado 20 para o número de arquivos que podem ser abertos pelo DOS e 8 para o número de buffers de leitura e gravação no disco. Caso o arquivo CONFIG.sys já exista no "diretório de carga" do Sistema Operacional do seu disco (em geral a raiz), verifique se os valores especificados para os parâmetros FILES e BUFFERS estão de acordo com o recomendado. Caso não estejam, é possível executar os programas com outras configurações, mas sem dúvida haverá perda de performance. Aconselhamos que você utilize a configuração recomendada.

Nas versões do DOS anteriores à 3.3, o número máximo de arquivos que pode ser aberto simultaneamente é 20, sendo que 5 desses arquivos são de uso do próprio DOS, restando apenas 15 para uso efetivo. Portanto, neste caso, o número máximo de arquivos que podem ser abertos por uma aplicação escrita em Clipper é 15. Na versão 3.3 do DOS, e provavelmente nas posteriores, é possível abrir simultaneamente até 255 arquivos, neste caso, o número máximo de arquivos que podem ser abertos em uma aplicação é 255, desde que se esteja utilizando a versão Summer 87 do Clipper.

Consulte o manual do DOS para obter informações mais detalhadas a respeito dos parâmetros FILES e BUFFERS.

8.3.2. Utilização da Memória

O Clipper possui recursos que permitem que seja alocada e controlada a memória RAM disponível no equipamento durante a execução das aplicações desenvolvidas.

Ao ser executado um programa compilado pelo Clipper, além da área que o mesmo ocupará, a memória disponível será dividida e alocada para quatro propósitos:

- manipulação de variáveis de memória
- "buffers" para realizar indexações
- execução de outros programas através do comando RUN
- área livre para manipulação de dados

A distribuição da memória disponível a ser utilizada para estes quatro propósitos será automaticamente feita pelo Clipper, caso uma outra não seja definida através do comando SET do DOS.

Normalmente o Clipper irá alocar a memória disponível para a execução da aplicação de acordo com a seguinte ordem de prioridade:

1 — O módulo executável é carregado na memória, sob o DOS;
2 — Em seguida 24 Kbytes são reservados como área livre de trabalho;
3 — Do restante da memória disponível, 20% são alocados para a manipulação e controle de variáveis de memória, até um máximo de 44 Kbytes;
4 — Da memória disponível após o passo 3, 33% ou um mínimo de 16 Kbytes são alocados para permitir a execução de outros programas através do comando RUN e para funcionar como "buffers" na elaboração e manipulação de índices;
5 — O restante da memória disponível é adicionado aos 24 Kbytes reservados inicialmente para ser utilizado como um espaço de trabalho livre para o processamento em geral, que inclui por exemplo o armazenamento de variáveis caractere, armazenamento de dados em vetores, cópias temporárias de registros de arquivos etc.

De acordo com as regras acima, a memória mínima necessária para executar uma aplicação pode ser calculada pela seguinte fórmula:

Memória Mínima = Módulo Principal + 24 K + 16 K + Variáveis

onde:

Módulo Principal: é a quantidade de memória ocupada pelo Módulo Principal da aplicação mais as áreas de Overlay, se houverem. É a quantidade de memória necessária informada pelo PLINK86 após a link-edição.

24 K: é a quantidade de memória mínima a ser utilizada como área livre de trabalho.

16 K: é a quantidade de memória mínima a ser utilizada como buffer para a elaboração e manipulação de índices.

Variáveis: é a quantidade de memória necessária para o controle das variáveis, no máximo 44 Kbytes, pois para cada variável são reservados 22 bytes.

Se o equipamento possuir expansão de memória (além do padrão), o programa compilado em Clipper irá automaticamente utilizá-la, alocando até 1 Megabyte como buffer para índices, sendo que a alocação realizada no passo quatro (descrito acima) será utilizada exclusivamente para a execução de programas externos através do comando RUN.

A alocação de memória automaticamente realizada pelo Clipper é plenamente satisfatória para a maioria das aplicações. Há, contudo, algumas circunstâncias onde será interessante controlar "manualmente" a forma de alocação. As situações

mais comuns onde isso é necessário referem-se à redistribuição de memória para um propósito que seja mais intensamente usado do que outro ou à mudança intensional da performance de um determinado aspecto, de acordo com a característica da aplicação desenvolvida.

O Clipper oferece a possibilidade de controlar a alocação da memória através da utilização do comando SET do DOS, que permite modificar o ambiente operacional do equipamento.

A sintaxe a ser utilizada é a seguinte:

SET CLIPPER=[Vnnn] [;Rnnn] [;Ennn] [;Xnnn] [;Fnnn] [;Sn]

(não deve haver espaço entre CLIPPER e o sinal de igual)

onde:

nnn: representa o número de Kbytes destinado para um dos parâmetros que poderão ser reconfigurados.

V: representa o montante de memória a ser alocado para o controle de variáveis. Se não for especificado serão alocados automaticamente 20% da memória disponível, até um máximo de 44 Kbytes, pois o Clipper permite a criação de até 2.048 variáveis e cada uma requer 22 bytes onde são armazenados o seu nome, tipo e valor (ou localização na área de trabalho, se forem do tipo caractere). A presença ou ausência de expansão de memória não afeta este parâmetro, pois as variáveis de memória nunca são alocadas na expansão.

R: representa o montante de memória a ser alocado para buffers de indexação ou para a execução de programas externos através do comando RUN. Quando um programa externo está sendo executado será alocada nesta área tanta memória quanto necessária, e os buffers de indexação serão temporariamente desalocados.

E: indica o montante máximo da expansão de memória a ser utilizado. O Clipper utiliza a expansão de memória unicamente como buffer para as operações de elaboração e manutenção de índices, tornando-as muito mais rápidas. O número mínimo que poderá ser especificado é 16 Kbytes ("E016"). Se este parâmetro não for especificado e houver expansão de memória instalada, o Clipper irá automaticamente alocar toda a expansão disponível, até um máximo de 1 Megabyte como buffers para a elaboração e manutenção de índices. Neste caso a área "R" (descrita anteriormente) será alocada exclusivamente para a execução de programas externos.

Nota: a expansão de memória a que se refere este parâmetro é aquela que vai além da memória máxima normal do equipamento utilizado, normalmente conseguida através de placas especiais. Por exemplo, no caso dos equipamentos compatíveis com os IBM-PC/XT, é qualquer expansão além dos 640 Kbytes normais.

X: indica o montante de memória que não deverá ser utilizado ou alocado pela aplicação desenvolvida em Clipper. Quando este parâmetro é utilizado o programa é executado e a memória é alocada como se o montante de memória por ele especificado não estivesse disponível. Por exemplo, se for especificado "X128" e o equipamento possuir 640 Kbytes, quando o programa for executado serão alocados apenas 512 K. Contudo, se algum programa externo for executado através do comando RUN, poderá utilizar a memória excluída pelo parâmetro X.
F: indica o número máximo de arquivos que poderão ser abertos simultaneamente. Este parâmetro deverá ser utilizado em conjunto com o arquivo CONFIG.sys, de configuração do DOS, através do parâmetro FILES (vide a seção anterior). O Clipper utilizará o menor dos dois parâmetros especificados para determinar o número máximo de arquivos que poderão ser abertos. As versões do DOS anteriores à 3.3 permitem apenas a abertura simultânea de 20 arquivos, cinco dos quais são utilizados pelo próprio DOS. A versão 3.3 permite a abertura de até 255 arquivos. Por exemplo:

```
CONFIG.sys      FILES=120
                BUFFERS=8

SET CLIPPER=F50
```

Neste caso o DOS 3.3 irá permitir que sejam abertos até 50 arquivos.

```
CONFIG.sys      FILES=20
                BUFFERS=8

SET CLIPPER=F50
```

Apenas poderão ser abertos simultaneamente 20 arquivos.
S: o parâmetro S elimina o "chuvisco" causado por algumas placas de vídeo gráficas durante a construção de telas. O valor 0 para o n é o padrão e construirá as telas mais rapidamente. O valor 1 irá resolver o problema do "chuvisco" se ele ocorre no seu equipamento, mas a apresentação das telas será mais lenta.

O comando "SET CLIPPLER=" deve ser emitido diretamente no sinal de pronto ("prompt") do DOS ou poderá ser incluído em um arquivo AUTOEXEC.bat para ser automaticamente executado. Ao ser carregado, o programa compilado pelo Clipper irá inicialmente verificar os parâmetros definidos acima foram alterados através do comando SET do DOS, e alocará o montante de memória correspondente aos valores estabelecidos para cada um deles.

Exemplo:

C>SET CLIPPER= V006;R064;E128;X128;F30

Neste exemplo hipotético serão alocados 6 Kbytes para o controle de variáveis ("V006"), o que permitirá a criação de até 256 variáveis, uma vez que cada variável utiliza 22 bytes (256 × 22=5.652 bytes). Serão alocados 64 Kbytes para a execução de programas externos através do comando RUN ("R064") e 128 Kbytes na expansão de memória como buffers de índices ("E128"). 128 Kbytes ficarão livres, não sendo alocados pelo programa ("X128"). Finalmente, poderão ser abertos até 30 arquivos ("F030"), desde que se esteja utilizando a versão Summer 87, o DOS 3.3 e que o arquivo CONFIG.sys, especifique um valor igual ou maior para o parâmetro FILES, por exemplo: FILES=50.

Cada um dos parâmetros do comando SET CLIPPER= deve ser utilizado com muito critério e apenas se necessário, caso contrário será preferível aceitar a alocação automática de memória efetuada pelo Clipper, perfeitamente satisfatória para a maioria dos equipamentos e aplicações. Se forem especificados parâmetros conflitantes ou se for alocada memória em excesso, o Clipper poderá reservar memória para algum propósito que não será efetivamente utilizado.

Caso o equipamento não possua a memória requerida será apresentada uma mensagem de erro — "System error — not enough memory", ou seja, "Erro do sistema — não há memória suficiente".

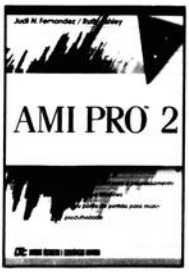
Índice Alfabético dos Comandos do Clipper

Índice Alfabético das Funções do Clipper

Índice Remissivo

Impressão e acabamento
(com filmes fornecidos):
EDITORA SANTUÁRIO
Fone (0125) 36-2140
APARECIDA - SP

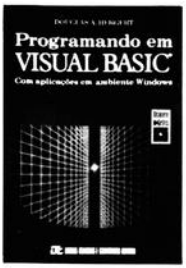
GALERIA DE SUCESSOS



AMI PRO 2
Judi N. Fernandez/Ruth Ashley
Esse livro conduzirá o usuário para o mundo do Ami Pro, mostrando como realizar as tarefas mais elementares até poder manipular as bases do trabalho.

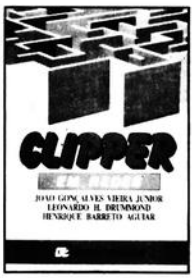
LOTUS 1-2-3 FOR WINDOWS
Peter Aitken
Esse livro apresenta a informação básica de que uma pessoa ocupada precisa para começar a usar o programa para tarefas reais.

TURBO PASCAL 6.0
Keith Weiskamp
Esse guia de auto-aprendizado lhe ensinará a escrever, compilar, executar e depurar programas Turbo Pascal.



PROGRAMANDO EM VISUAL BASIC
Douglas A. Hergert
Essa obra ensina como utilizar a linguagem Visual Basic e como criar programas com aplicações em Windows. (Disquete grátis).

APRENDA VOCÊ MESMO... CORELDRAW!
Paul Webster
Esse livro extraordinário permitirá que você se familiarize rapidamente com as ferramentas, menus, telas e os quadros de diálogo do CORELDRAW! Versão 2.0.



CLIPPER EM REDES
Leonardo Hees Drummond/João Gonçalves Vieira de Souza Jr./Henrique Barreto Aguiar
Nesse livro, são abordados os comandos e funções existentes no Clipper, mostrando através de exemplos como criar rotinas que facilitem o gerenciamento das informações em redes locais.

INTRODUÇÃO À ORGANIZAÇÃO DE COMPUTADORES
Mario A. Monteiro
Esse livro apresenta, em linguagem simples e direta, a organização e o modo de funcionamento básico dos computadores.

CLIPPER 5.0
Antonio Vidal
Escrito pelo maior especialista brasileiro na área. Está dividido em: Vol. 1 — Linguagem e comandos; Vol. 2 — Funções, compilação, linkedição e execução; Vol. 3 — Rede local, utilitários, aplicações e erros; Vol. 4 — Programação e arquitetura de sistemas. (Disquete grátis vols. 3 e 4)

ENTRE PARA NOSSA MALA DIRETA

Desejo receber, sem ônus, todo material promocional dos lançamentos da área de Informática.

Nome: .....

End. Res.: .....

CEP: ..... Cidade: ..... Estado: .....

Tel/DDD: ( ) ..... Profissão: .....

Empresa: .....

End. Com.: ..... CEP: .....

Cidade: ..... Estado: ..... Tel/DDD: ( ) .....

LTC — LIVROS TÉCNICOS E CIENTÍFICOS EDITORA LTDA.
Rua Vieira Bueno, 21 São Cristóvão – 20920 - 390 - Rio de Janeiro – RJ. Tel.: (021)580-9174

DOBRE

COLE
O SELO
AQUI

Editora
LIVROS TÉCNICOS
E CIENTÍFICOS
Rua Vieira Bueno, 21 — São Cristóvão
Caixa Postal 21162 — Rio de Janeiro/RJ

20920-390

DOBRE

CLIPPER®

VERSÃO SUMMER 87

Antonio Geraldo da Rocha Vidal

Vol.1

Obra destinada a usuários e profissionais de microinformática que já possuem alguma experiência no desenvolvimento de sistemas de informação com a utilização do software dBase III e que gostariam de implementá-los utilizando o poder do Clipper.

O objetivo é tentar fornecer aos que utilizam a linguagem de programação dBase III suficientes conhecimentos do Clipper, o compilador do dBase III, que permitirão a construção de sistemas mais eficientes, sofisticados e de alto nível profissional.

A obra está dividida em dois volumes. O volume 1 descreve os comandos, as funções, a compilação e a execução do Clipper. O volume 2 descreve os utilitários, a construção de um sistema e a rede local.

MAIS UM LANÇAMENTO
DA

LIVROS TÉCNICOS E CIENTÍFICOS EDITORA

ISBN 85-216-0639-7