COMMON CODE REFERENCE

JUNE 1984

Manual Name : Common Code Reference
Order Number: 29200-44
Issue date: June 1984

IMPORTANT NOTICE -- PLEASE READ

This is an interim version of the Common Code Reference manual. While it provides a comprehensive reference to the functions and procedures supported by the Common Code, it does not yet include examples for all of the functions and procedures. An enhanced version of the manual will be available in several months that will include many example programs to help you in implementing your own applications using the capabilities provided by the Common Code. To receive the enhanced version of the manual, please fill out the form below and mail it to:

GRiD Systems Corporation
Attn: Technical Marketing Services
2535 Garcia Avenue
Mountain View, CA. 94043

Since the entire manual will be reprinted and redistributed, this also gives you a chance to provide some valuable input into the next edition. Any comments, suggestions, or corrections that you can give us to improve the next edition are hereby solicited and will be incorporated if possible. Please use the space below for your comments.

****

Please put me on the distribution list for the final version of the Common Code Reference manual.

Name ________________________________

Address ____________________________City_____________State______Zip_____

Suggestions/Corrections/Comments:

TABLE OF CONTENTS

ABOUT THIS BOOK

CHAPTER 1: INTRODUCTION

The Common Code: A Common User Interface 1-1
String Manipulation 1-2
Menus and Forms 1-3
Field Formatting 1-3
Tabular Formatting 1-4
Hierarchy of the Common Code 1-4

CHAPTER 2: THE USER INTERFACE

The Operating Environment 2-1
CODE Commands 2-2
Messages 2-3
The Flow of Control 2-3
Common Commands 2-3
Menus and Forms 2-5
   Menus 2-5
   Example... 2-6
   Forms 2-6
   Example... 2-8
Formatting Information 2-10
   Fields 2-10
   Tables 2-11
Editing Information 2-11
   Cursor Control 2-11
   Selection of Text or Cells 2-12

CHAPTER 3: DATA TYPES

Standard Data Types 3-1
The Bytes Type 3-3
Common Code Data Types 3-3

CHAPTER 4: COMMON PROPERTIES

Interchange Files 4-1
Data Records 4-2
Properties and Options 4-2
Common Properties Records 4-3
   Author Record 4-4
Font Properties and Print Options Record 4-5
   Font Record 4-5
   Print Options Record 4-6
   Text Heading Record 4-7
Field Characteristics Records 4-7
   Standard Field Record 4-9
   Column Field Record 4-10
   Row Field Record 4-11
   Column Width Record 4-12
   Row Height Record 4-13
   Cell Field Record 4-13
   Table Size Record 4-14
Application Properties Records 4-15
Properties Records Common Code Routines 4-16

CHAPTER 5: STRINGS

Data Structures 5-1
The String Routines 5-2
   Allocating and Deallocating Strings 5-2
   Comparing Strings 5-3
   Modifying Strings 5-3
   Converting String Types 5-4
   Miscellaneous String Routines 5-5

CHAPTER 6: COMMANDS, MESSAGES AND PROMPTS

Commands 6-1
Messages and Prompts 6-2
Data Structures 6-2
Message and Prompt Routines 6-3

CHAPTER 7: BYTES

Byte Routines 7-1

CHAPTER 8: DATA DRIVEN MENUS/FORMS

Overview 8-1
Data Driven Menus 8-2
   Menu Data Structures and Types 8-3
   Data Driven Menu Routines 8-4

Data Driven Menu Example 8-4
Data Driven Forms 8-6
Forms Data Structures and Types 8-8
Data Driven Forms Routines 8-12
Data Driven Forms Example 8-13
The File Form 8-16
Pathname Defaults 8-18
File Form Constants and Data Types 8-19
File Form Messages 8-22
Typical FileFormConfirmed Settings 8-22
Exchanging Applications 8-23

CHAPTER 9: FONTS

Font-Related Routines 9-1

CHAPTER 10: FIELDS

Constants 10-1
Data Structures 10-2
Field Routines 10-5

CHAPTER 11: TABLES

Constants 11-1
Data Structures 11-2
Table Routines 11-8
Allocating and Disposing Tables 11-8
Editing Tables 11-8
Specifying Cells 11-9
Drawing a Table 11-9
Inverting a Table 11-10
Highlighting a Table or Cell 11-10
Scrolling 11-10
Coordinating Text and Cell Selections 11-11

CHAPTER 12. COMMON CODE FUNCTIONS AND PROCEDURES

AppendAnyChar 12-2
AppendChar.. 12-2
AppendString.. 12-3
AuthorOfThisFile.. 12-4
CmdErase 12-5
CmdMediaUsage... 12-6
CmdProperties... 12-7
CompareBytes.. 12-8
CompareStrings.. 12-9
ConCatLIts.. 12-10
ConCatStrings... 12-10
CopyOfString.. 12-11

CopyString.. 12-11
DataFormConfirmed... 12-12
DataMenuConfirmed... 12-13
DeleteBytes... 12-14
DeleteFromString.. 12-14
EqualStrings.. 12-15
ExactCopyOfString... 12-15
ExpandHeading... 12-
FFExecuteCommand.. 12-16
FileFormConfirmed... 12-17
FinalizePropertiesLength.................................... 12-19
FindByte.. 12-20
FldDimHighlightField.. 12-21
FldDrawCursor... 12-21
FldDrawField.. 12-22
FldDrawFieldChars... 12-22
FldEditField.. 12-23
FldEraseCursor.. 12-25
FldFormatLine... 12-27
FldHilightField... 12-28
FldInsertInField.. 12-28
FldInvertChar... 12-28
FldReadKey.. 12-29
FldSetCursor.. 12-30
FldSetPos... 12-30
FldStartKeys.. 12-30
FontCount... 12-31
FontGetN 12-31
FontNthName... 12-31
FontSetName... 12-32
FontSetNth.. 12-32
FreeString.. 12-33
FreeStringsInDataForm....................................... 12-33
GetNextRecord... 12-34
GetVersionString.. 12-35
InsertBytes... 12-36
InsertCharInString.. 12-37
InsertInString.. 12-37
IntegerToString... 12-37
MoveBytes... 12-38
MoveReverseBytes.. 12-38
MsgClearMessage... 12-39
MsgClearPrompt.. 12-39
MsgExit... 12-40
MsgInit... 12-41
MsgInitialUsage... 12-43
MsgShowDecoded.. 12-43
MsgShowError.. 12-44
MsgShowMessage.. 12-45
MsgShowPrompt... 12-46
MsgStackMessage... 12-47
MsgStackPrompt.. 12-48
NewString... 12-49

NewStringLit.. 12-49
RealToString.. 12-50
SetBytes.. 12-51
SetPrefix... 12-51
SkipProperties...................................... 12-52
StringOfFormItem.................................... 12-52
StringToInteger..................................... 12-53
StringToReal.. 12-53
SubProperty... 12-54
SubStringLit.. 12-55
TblAddCol... 12-57
TblAddRows..
TblCellOnScreen..................................... 12-58
TblChangeFields..................................... 12-58
TblConfirmSelection................................. 12-59
TblDimHighlightCell................................. 12-59
TblDisposeCol.......................................
TblDisposeScreen.................................... 12-60
TblDisposeTable..................................... 12-60
TblDrawGrid... 12-61
TblDrawTable.. 12-61
TblEditTable.. 12-62
TblEqualCells....................................... 12-64
TblEscapeMode....................................... 12-64
TblFieldOfCellID.................................... 12-65
TblFieldOfColRow.................................... 12-65
TblFindBounds....................................... 12-65
TblGetSelectedCellIDs............................... 12-66
TblHighlightCell.................................... 12-67
TblHilightTable..................................... 12-67
TblInitTable.. 12-68
TblInvertRange...................................... 12-70
TblIinvertSpan...................................... 12-70
TblNewCol...
TblNewScreen.. 12-70
TblScroll... 12-71
TblScrollAdjustCellID............................... 12-72
TblSetCurrentCell................................... 12-72
TblSetVisible....................................... 12-73
TblSetVisibleRect...................................
TblStartSelection................................... 12-73
TblUnhilightTable................................... 12-74
TblUpdateRect....................................... 12-74
TimeToString.. 12-75
TranslateHeading.................................... 12-76
UnDoDataForm.. 12-77
UpperCase... 12-78
WMax.. 12-
WMin.. 12-

APPENDIX A. INCLUDE FILES

APPENDIX B. LISTINGS OF DATA DRIVEN MENU/FORM EXAMPLES

APPENDIX C. MENUS & FORMS: ANOTHER METHOD

Data Structures C-1
Menu and Form Routines C-4
MenuInit C-5
FormInit C-6
MenuFormConfirmed C-8
NilChoiceProc C-13
NilChoiceInfo C-13
NilItemString C-13
NilIScrollKey C-14
MenuFormDispose C-14

ABOUT THIS BOOK

This reference manual describes the Common Code, a package of PASCAL functions and procedures for formatting information and displaying it on the screen. It also contains routines for manipulating character strings and bytes of memory.

This book relies upon the information contained in other Compass Computer Manuals. The code in this manual conforms to the PASCAL language as it is described in the PASCAL-86 User's Guide. It assumes an operating system environment as defined by the GRiD Operating System (GRiD-OS) Reference manual. The methods for linking and loading the routines are described in the Program Development Guide.

If you plan to use the Common Code to write programs that conform to the software requirements of GRiD Systems Corporation, be sure to see the GRiD Mangement Tools Reference manual. It shows GRiD's programming conventions in practice.

The first two chapters of this manual introduce the Common Code and GRiD's programming conventions for menus, forms, and tables. Chapters 3 and 4 describe data structures used by Common Code routines and the characteristics of data files used by GRiD applications. Chapters 3 to 11 provide a summary of the procedures: related procedures are grouped together, allowing you to obtain an overview of the routines available. Chapter 12 lists all of the Common Code procedures in alphabetical order and provides a detailed description of each procedure. Use this chapter as the standard reference section once you are generally familiar with the routines that are available.

Appendix A lists the include files for the Common Code routines. Appendix B lists example programs for data driven forms and menus and appendix C describes an alternate method for developing menus and forms.

Additional appendices providing more examples of the use of Common Code routines will be available in several months. To obtain a new version of this manual which includes up-to-date examples, send in the "Important Notice" at the front of this manual.

CHAPTER 1: INTRODUCTION

This chapter introduces the Common Code package, and shows how the Common Code routines relate to GRiD's user interface and application programs. The Common Code comprises a number of subroutine packages, and this chapter explains their relation to one another.

THE COMMON CODE: A COMMON USER INTERFACE

GRiD applications have been designed for similarity in appearance and operation. This resulted from a conscious decision by GRiD to produce software that is easy to use and to learn to use. When applications have similar or identical commands, the user has to learn the commands only once -- from then on, all new learning is based upon what the user is already familiar with.

This approach is the basis of the GRiD user interface. The consistent interface is the common feature that makes a data base as easy to use as a text editor.

With this design in mind, it was logical for GRiD to put these common features into a single software package -- the Common Code. The Common Code provides the data structures and procedures for maintaining a consistent user interface among a variety of applications. It provides the mechanisms to implement the "ring" around the applications.

Figure 1-1 shows the major packages of the Common Code. The character string package and window graphics package form the basis upon which all else is defined.

> NOTE: The window graphics package was originally part of the Common Code but is now part of the GRiD Operating System (GRiD-OS). Refer to the GRiD-OS Reference manual for a description of window graphic routines.

Fields, tables, and menus/forms provide sophisticated mechanisms for displaying and formatting text on the screen. The features of each package are discussed below.

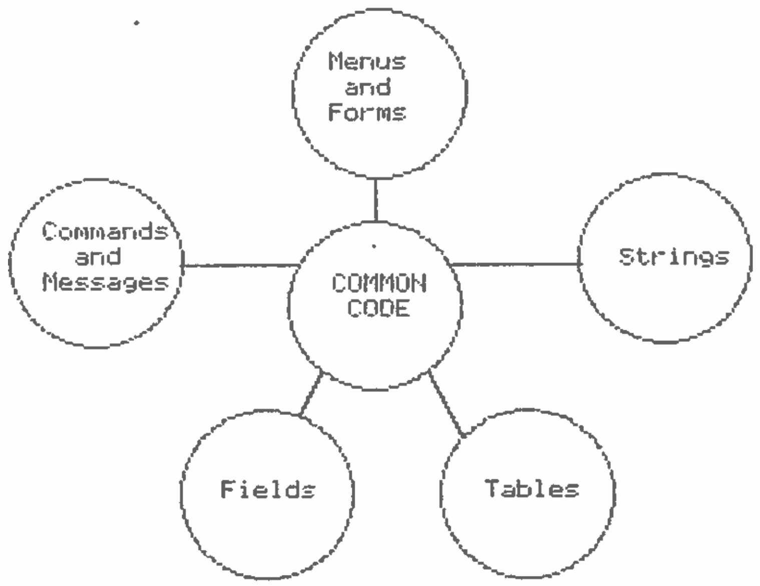


Figure 1-1. Major Packages of the Common Code

STRING MANIPULATION

The string package consists of routines for processing text characters. The routines offer greater flexibility than many comparable string packages.

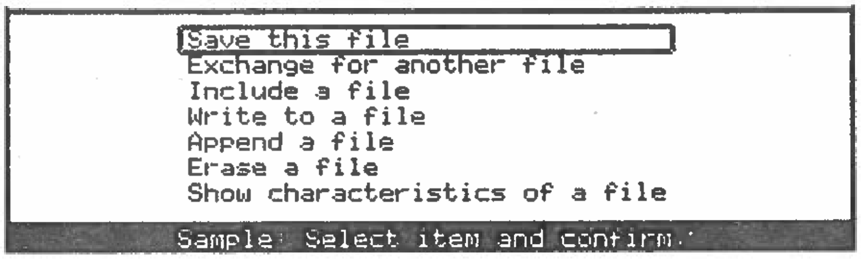
It features:

- o Strings up to 65535 characters long
- o A full range of string functions and numerical conversions
- o Literal strings
- o Dynamic allocation/deallocation of strings to save memory space

MENUS AND FORMS

With these routines, you can send and receive data from the user by means of menus and forms. The user can type input or choose a predefined item from a list without unnecessary typing.

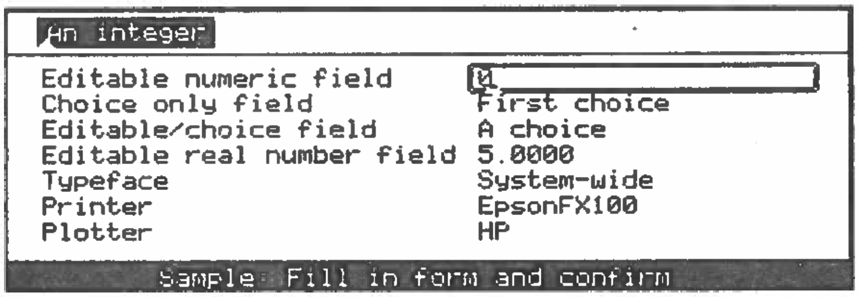
The user can confirm only one item on a menu. With a form, the user can set several items at once, either by choosing a setting or typing a new one.



Save this file
Exchange for another file
Include a file
Write to a file
Append a file
Erase a file
Show characteristics of a file

Sample: Select item and confirm.

Sample Menu



An integer

Editable numeric field	0
Choice only field	First choice
Editable/choice field	A choice
Editable real number field	5.0000
Typeface	System-wide
Printer	EpsonFX100
Plotter	HP

Sample: Fill in form and confirm

Sample Form

The menu and form package has these features:

- o Movement and editing within the menu or form is controlled completely by the Common Code
- o Scrolling is enabled automatically for large menus and forms
- o Menus and forms are allocated dynamically

FIELD FORMATTING

The field package lets you format text characters within a rectangular box (called a field) and display it on the screen. Each rectangular field can be formatted and displayed separately from any others.

Specifically, it includes routines to

- Display a blinking, triangular text cursor
- Facilitate cursor movement with arrow keys, BACKSPACE, and CODE-BACKSPACE (erases previous word)
- Outline and highlight individual fields
- Left-align, right-align, or center text within a field
- Format multiple lines in a field, with word-wrapping
- Specify individual fields as user-editable or display-only
- Allocate fields dynamically

TABULAR FORMATTING

The table routines let you group several fields into a table and manipulate them together. With these routines, numerical and character data are formatted so that users can examine and edit them easily.

The table code has these features:

- All field formatting functions are available for every data cell in the table
- A text cursor and cell outline can be moved from cell to cell to show the field being edited
- Text and cells can be duplicated, erased, moved, and inserted with built-in functions
- The user can select portions of the table as operands for commands
- Selections are highlighted automatically
- Automatic scrolling is available
- Tables are allocated dynamically

HIERARCHY OF THE COMMON CODE

The Common Code packages are designed to build upon one another. Figure 1-2 shows the hierarchy of these packages.

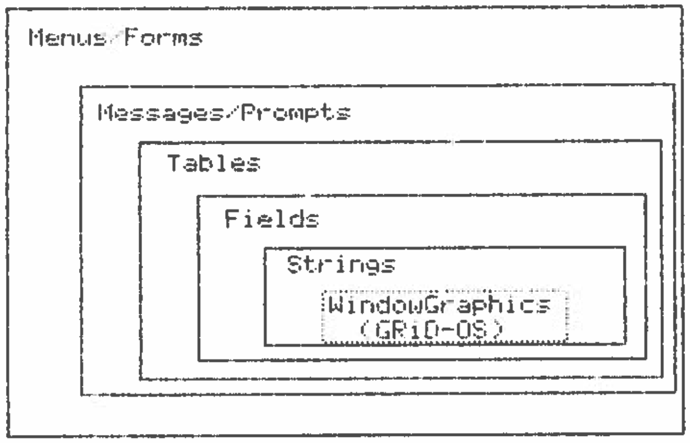


Figure 1-2. Hierarchy of the Common Code

Each outer level depends upon the data structures and procedures of an inner level. They have been left separate so that you can program at the level of detail and sophistication required by each application. The Common Code can do as little as displaying a single bit on the screen, or as much as managing, displaying, and updating several data structures in real time. When reading this manual, keep this structure in mind. The sequence in which the routines are presented in the following chapters do not, however, strictly follow this hierarchical approach. Instead, we present the basic structures and routines that you need to implement commands, messages, prompts, menus, and forms. This lets you begin using the most widely used capabilities provided by Common Code without becoming totally familiar with some of the underlying structures. We discuss fields and tables last because, although they profide the underlying structure for menus and forms, you usually need not use them directly unless you are working on a cell-based application such as a spreadsheet.

CHAPTER 2: THE USER INTERFACE

This chapter describes the major features of the user interface for GRiD software products. It provides the necessary background for using the Common Code to construct user interfaces that are compatible with GRiD software products. The terminology introduced in this chapter is used throughout the manual.

Many of the capabilities provided by the Common Code can be utilized without understanding some of the underlying or auxiliary capabilities. For example, you can easily implement menus and forms just like those used in GRiD applications without delving into the complexities of cells and fields. For this reason, the chapters in this book present a few basic routines, such as string handling routines, that are needed to use menus and forms, but save explanations of cells and fields until the later chapters. This approach should let you begin using some of the powerful features of the Common Code immediately. When you feel comfortable with these capabilities, such as messages, commands, menus, and forms, you can begin exploring the more complex functions and procedures provided by the Common Code.

THE OPERATING ENVIRONMENT

The Common Code is designed to operate specifically with the GRiD Operating System (GRiD-OS), a multiprocessing system.

- o It is designed to be reentrant so that several concurrent processes can use it at once.

- o Different processes are assigned different areas, or windows, on the screen to operate. Each window represents a separate process. The Common Code controls the display within each window. Hence, all data and graphics are displayed relative to a window, and never defined upon the absolute size of the screen.

- All data structures are allocated and deallocated dynamically to save memory space for other applications to run. This is the rationale behind the extensive use of pointers in the field, table, and menu/form packages.

CODE COMMANDS

The user causes the computer to perform an action by pressing a CODE key command, such as CODE-D for Duplicate.

The CODE key is a modifier key, like CTRL or SHIFT. The CODE command characters are not displayed on the screen; they are carried out directly. GRiD chose the CODE key for commands so that CTRL and SHIFT would be left free for commands to existing terminal emulators and timesharing systems.

Figure 2-1 shows the syntax of a typical command.

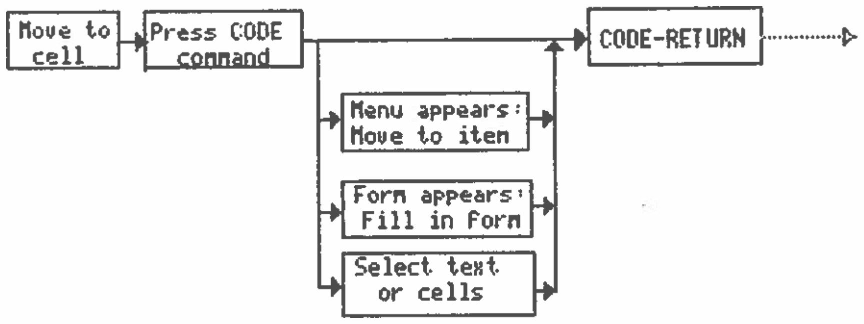


Figure 2-1. Syntax of a Typical Command

The user moves the cursor and outline to the text or cell to be acted on by the command. After pressing the CODE key command, the user faces one of these options, depending on the command:

- To move to an item on a menu.
- To fill in a form.
- To select additional text or cells by pressing arrow keys.

These three options are described later. The user then presses CODE-RETURN to confirm the menu item, form, or selection. If the command requires additional parameters, it will display additional menus, forms, or messages requesting a selection.

MESSAGES

Messages and prompts to the user are displayed in lines at the bottom of the screen. The messages or prompts are highlighted within a display-only field that the user cannot move to. They are centered within the field.

Messages should be programmed to disappear upon the next keystroke. Command prompts should remain displayed until the user confirms or cancels the command. Messages and prompts have this appearance:

<Command name>: <Prompt>

Duplicate: Make a selection and confirm

Properties: Fill in form and confirm

Transfer: Select item and confirm

THE FLOW OF CONTROL

GRiD applications are designed to be "modeless". That is, the user does not have to press special keys to enter Edit Mode, Command Mode, Retrieve Mode, etc. In any GRiD application, the user simply types text (without having to enter an Edit Mode) or presses a command key.

The user can always terminate a command at any point. At each step in a command, whether confirming a menu item, filling in a form, or making a selection, the user can press ESC and the command is aborted. The user can then type text or issue any command.

Pressing a command key during another command preempts the pending command and starts a new one. By pressing one command key, the user can stop any command and begin any other one.

COMMON COMMANDS

The real advantage of the leveraged learning interface is that the commands in different applications have similar names and syntax. Table 2-1 below shows these similar commands. Many of these are available as functions in the Common Code.

KEY	COMMAND	GRiDPlot	GRiDFile	GRiDWrite	GRiDPlan	GRiDManager
CODE-A	ACCESS	X	X	X	X	X
CODE-B	BEGIN	X	X	X	X	
CODE-C	COLUMN	X	X		X	
CODE-D	DUPLICATE	X	X	X	X	X
CODE-E	ERASE	X	X	X	X	X
CODE-F	FIND		X	X		
CODE-H	HEADINGS	X			X	
CODE-I	INSERT	X	X		X	
CODE-J	JUMP			X	X	
CODE-M	MOVE	X	X	X	X	X
CODE-O	OPTIONS	X	X	X	X	X
CODE-P	PROPERTIES	X	X		X	
CODE-Q	QUIT	X	X	X	X	
CODE-R	ROW	X	X		X	
CODE-S	SUBSTITUTE		X	X		
CODE-T	TRANSFER	X	X	X	X	X
CODE-U	USAGE	X	X	X	X	X
CODE-W	WILDCARD		X			X
CODE-ESC	CANCEL	X	X	X	X	X
CODE-RETURN	CONFIRM	X	X	X	X	X
CODE-?	HELP	X	X	X	X	X

Table 2-1. Common Commands

The arrow keys are standard across applications too. See Appendix D for a discussion of the keys and the Common Code procedures to control them.

MENUS AND FORMS

Commands can request data from the user by presenting a menu or a form. With a menu, the user selects a single value as input to the Compass. Forms allow the user to give the computer several values at once.

Menus

See Figure 2-2 for a sample GRiD menu. All GRiD menus resemble this one.

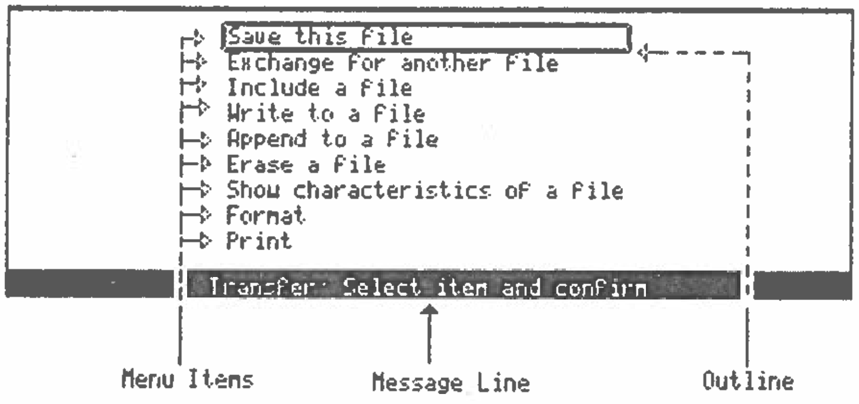


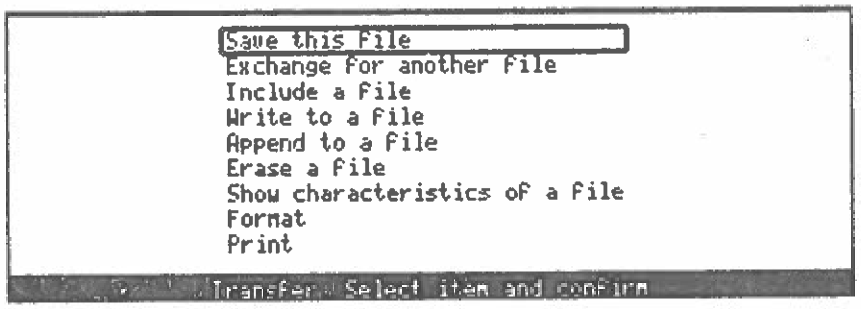
Figure 2-2. A Sample GRiD Menu

A menu consists of:

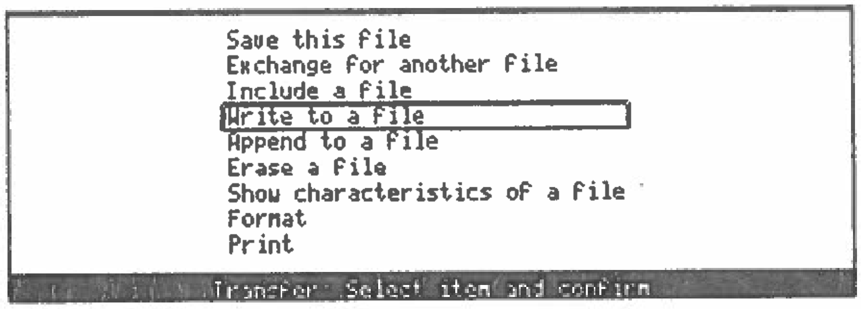
Menu items	A vertical list of objects or operations, such as commands, file titles, or storage media. By confirming an item, the user tells the Compass to operate with that item instead of the others. The items are display-only fields that the user cannot modify.
Outline	A moving indicator that rests upon the current item. A triangular cursor never appears within this outline, because text can never be typed into a menu.
Message Line	An informational message instructing the user as to what action to take with the menu.

Example

Within a word processing program, the user presses CODE-T to transfer a file to a storage medium. The menu shown below appears:


Save this File
Exchange for another File
Include a File
Write to a File
Append to a File
Erase a File
Show characteristics of a File
Format
Print
Transfer: Select item and confirm

The user presses RETURN three times to move the outline to the item titled Write to a file. Users can move the outline as much as they like before confirming an item.


Save this File
Exchange for another File
Include a File
Write to a File
Append to a File
Erase a File
Show characteristics of a File
Format
Print
Transfer: Select item and confirm

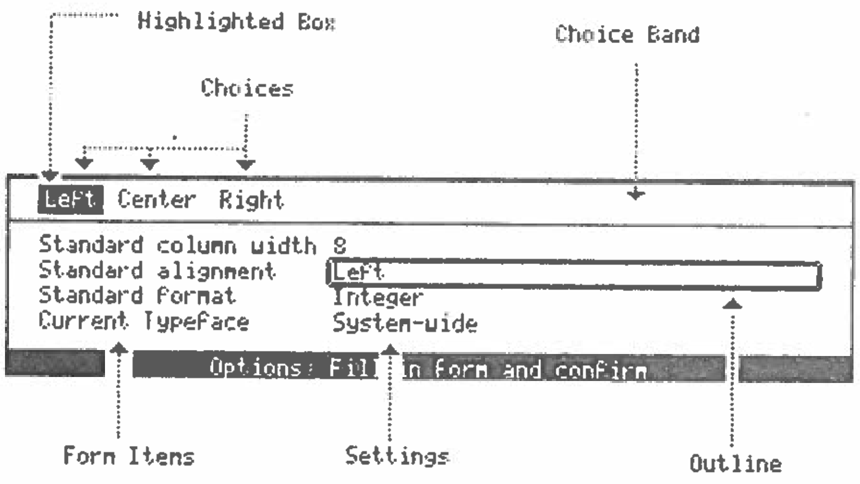
The user presses CODE-RETURN. The word processing program finds out which item was confirmed and performs the operation. Only one item can be confirmed at a time.

If the user presses ESC or another command key, the menu disappears and no operation is performed.

Forms

See Figure 2-3 for a representative GRiD form. Most GRiD forms resemble this one. Forms are different from menus, for these reasons:

o Most forms let users change the settings of several items. Menus let them confirm only one item.
o Users can type their own settings. Many forms do not limit them to predefined choices .
o When users press CODE-RETURN, they confirm the settings of all the form items, not just the outlined setting.



Left Center Right

Standard column width	8
Standard alignment	Left
Standard format	Integer
Current Typeface	System-wide

Options: Fill in form and confirm

Figure 2-3. A Sample GRiD Form

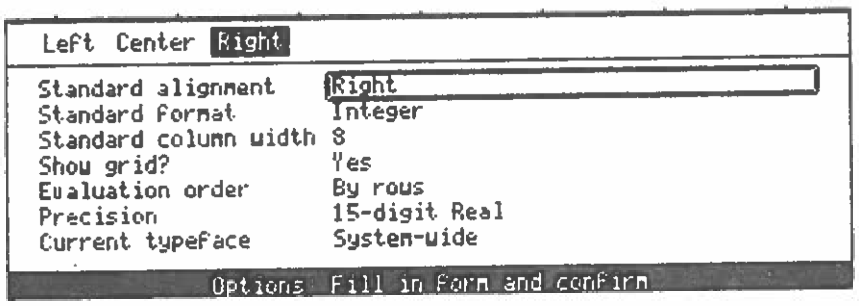
A form consists of:

Form items	Labels which identify the data to be modified. Each item has a setting associated with it. These are display-only fields that the user cannot move into.
Settings	The actual values that the user types or chooses from the choice band. Application programs read these values and operate on them. These are editable, choice, or editable-choice fields, depending on the application.
Outline	A moving indicator that surrounds the setting currently being modified. RETURN moves it down and SHIFT-UpArrow moves it up. If the outlined setting contains a blinking cursor, users can type their own value for that setting.
Choice band	Located at the top of the form, it contains the choices associated with an item. As the user moves from item to item, the choices in the choice band change. The choices appear either horizontally or vertically. (Forms without choices do not have a choice band.)
Choices	Predefined values for a setting, which appear in the choice band. The highlighted choice appears within the outline automatically. It is the value for the outlined setting. The choices are display-only fields.
Highlighted box	Indicates the choice that appears in the outlined setting.

Pressing the arrow keys moves it among choices.

Example

In a tabular worksheet program, a user presses CODE-O to adjust the worksheet's options. This form appears, with these initial values:

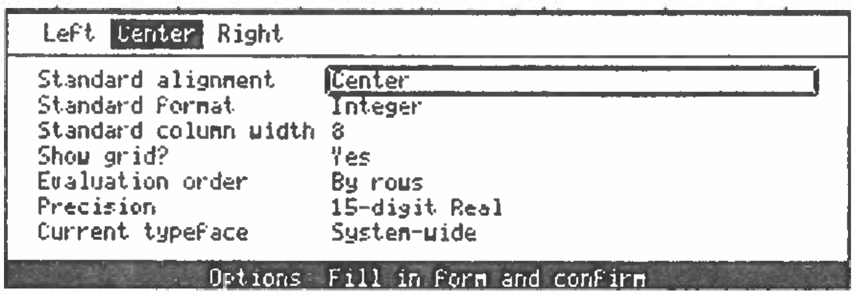


Standard alignment	Right
Standard format	Integer
Standard column width	8
Show grid?	Yes
Evaluation order	By rows
Precision	15-digit Real
Current typeface	System-wide

Options: Fill in form and confirm

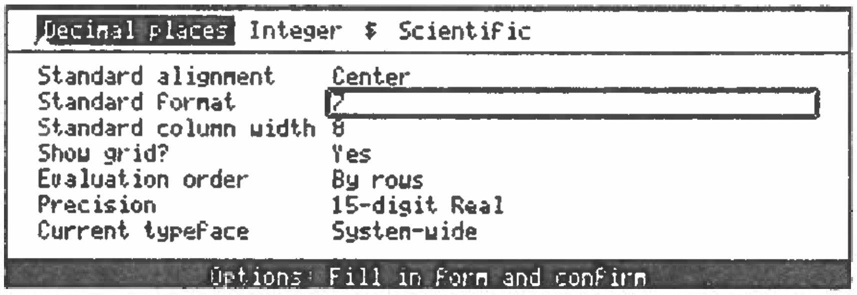
The outline surrounds the setting associated with the Standard Alignment item. This setting is a choice field. In the choice band, the highlighted box rests upon Right. Right also appears within the outline.

The user presses LeftArrow once, and the highlighted box moves to Center. The outlined setting now contains Center as well; the Common Code does this automatically.



```
Standard alignment    Center
Standard format       Integer
Standard column width 8
Show grid?            Yes
Evaluation order      By rows
Precision             15-digit Real
Current typeface      System-wide
```

The user presses RETURN to move the outline to another setting. New choices appear in the choice band. The user presses LeftArrow twice, and the highlighted box moves to Decimal Places. The setting is an editable-choice field, so a blinking cursor appears in the outline. The user types a number.

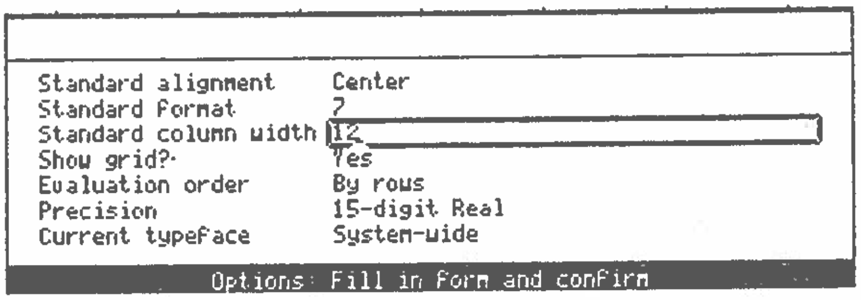

Decimal places Integer $ Scientific

```
Standard alignment    Center
Standard format       2
Standard column width 8
Show grid?            Yes
Evaluation order      By rows
Precision             15-digit Real
Current typeface      System-wide
```

Options: Fill in form and confirm

The user presses RETURN again. The choice band is now empty, but the blinking cursor appears within the outline. The setting is an editable field. The user presses BACKSPACE to erase the initial value, and types another value.

NOTE: Wherever the blinking cursor appears, the Common Code lets the user modify text using arrow keys, BACKSPACE, and CODE-BACKSPACE (erase previous word).


Standard alignment	Center
Standard Format	7
Standard column width	12
Show grid?	Yes
Evaluation order	By rows
Precision	15-digit Real
Current typeface	System-wide

Options: Fill in Form and confirm

Pressing CODE-RETURN now returns the cursor and outline to their previous context, the worksheet program. The worksheet program could then retrieve the new settings of the form and operate upon them. The form disappears.

If the form had appeared in the course of a command, the command would continue.

Pressing ESC or another CODE command would return the outline to the worksheet program (canceling any pending command), but the form would retain all its old settings.

FORMATTING INFORMATION

An essential function of the Common Code is to format information for display on the screen within an application window. It provides a sophisticated mechanism for putting raw text and data into formatted fields.

Fields

To the user, a field is a rectangular area on the screen that contains text or numeric values. It can be filled in by the user or the system.

To the programmer, a field is a data structure that contains a text string and formatting information for that text. The Common Code provides procedures for formatting the text and displaying the text on the screen.

The contents of a field can be left-aligned, right-aligned, or centered. Fields can contain more than one line of text. There are four types of fields, designed to protect data or enable the user to interact with it.

Editable	Editable fields allow the user to edit their values by typing, backspacing, or pressing arrow keys to move within the field.
Display-Only	The user cannot alter the values of these fields.
Choice	Choice fields can contain only settings from a predefined list. They are used only within forms, as described later.
Editable-Choice	Editable-choice fields can contain settings chosen from a

predefined list, or the user can edit their values by typing, backspacing, or pressing arrow keys. They occur only in forms, as described later.

Tables

Tables are collections of fields gathered together as a matrix. They are convenient for displaying large amounts of numerical data or for putting text into a tabular format.

Tables consist of editable fields, though the fields could be modified to become display-only in order to protect the field contents. Each field in a table is called a cell.

Tables are easier to use than individual fields. The Common Code has defined procedures for moving from cell to cell, and for controlling the cell that is to be edited. Automatic scrolling has been developed for tables, and several cell functions have been defined to operate upon selections of cells.

EDITING INFORMATION

The Common Code provides several mechanisms for editing information within a field: cursor control, code commands, menus and forms, text and cell selections, and screen messages.

Cursor Control

For editing within fields, the Common Code provides routines to generate a blinking triangular cursor, which is placed between character positions in a field.

For applications with more than one field on the display, the Common Code has routines to outline the field currently being edited. The table package has routines for moving both the cursor and the cell outline, and for keeping track of the "current cell".

The user moves the cursor and cell outline by pressing arrow keys. Appendix G lists these arrow keys along with the Common Code routines that control their operation.

The table package also has built-in functions for scrolling. If a user tries to move the cursor and cell outline outside of a scrolling boundary, the contents of the display will scroll. Tables can be constructed to display and scroll over large databases.

Selection of Text or Cells

Many commands require the user to select text or cells as operands. For example, the user selects some text within a cell to be erased, or a range of cells to be duplicated.

As the user presses arrow keys, CODE-B, CODE-C, or CODE-R, the selection is highlighted by the Common Code. The user can change the selection as much as desired before pressing CODE-RETURN to confirm the selected text or cells.

The selection area is always a rectangle no matter how the user moves the outline or cursor. The first selected cell and the current cell (i.e., with the outline) form two corners of this rectangle. The first selected cell is known as the "anchor," because the selection appears to be anchored to it.

CODE-B allows the user to restart the selection. When the selection is restarted, the original anchor is discarded and the outlined cell becomes the new anchor.

CHAPTER 3: DATA TYPES

This chapter defines the basic data types used in this manual. Every package, such as the table routines, has other data types that are defined specifically for it. The unique data types are defined in the same chapter where their corresponding routines are described.

STANDARD DATA TYPES

Table 3-1 lists the basic data types found in the Common Code.

Type	Description
Boolean	An ordinal type with two values, False (0) and True (1).
Integer	A simple ordinal type of two bytes in the range -32768 to 32767.
LongInt	A simple ordinal type of four bytes in the range of -2,147,483,647 to 2,147,483,646.
Word	A simple ordinal type of two bytes in the range of 0 to 65535.
Byte	An enumerated ordinal of the range 0..255. Not to be confused with the Bytes type, described below.
Char	A simple ordinal defined on the ASCII character set.
Real	A simple type defining single-precision floating point numbers with 24 bits of precision.
LongReal	A simple type defining floating-point numbers with 53 bits of precision.

Table 3-1. Standard Data Types

THE BYTES TYPE

A special data type has been defined to override PASCAL's rigorous type-checking. It is the Bytes type. Note that it is NOT the Byte (singular) type, which is defined to be 0..255. The Bytes type is not a part of standard PASCAL. The PASCAL-86 User's Guide describes it in more detail.

The Bytes type allows you to pass any type of data to a function or procedure. The passed data must match what the procedure expects to receive, of course.

The Bytes type is used to implement literals, such as literal character strings. For example,

```
     NewStringLit(VAR lit: Bytes): StringPtr;
```

accepts these literal inputs:

```
     VAR stringArray: ARRAY [1..80] OF Char;
         aString: StringPtr; (see Chapter 4)

     x := NewStringLit('abcdef');
     x := NewStringLit(stringArray);
     x := NewStringLit(aString^.chars[1]);
```

When passing a parameter of type Bytes, the procedure actually passes a pointer to the code itself. Hence, all Bytes parameters must be passed by reference rather than by name. The Bytes identifier can appear only in an external module's PUBLIC section: see the PASCAL manual for more details.

COMMON CODE DATA TYPES

This is a summary of the data types in the Common Code. The data types for each package are defined at the beginning of the appropriate chapter.

Common Properties

```
     AuthorType
     HeadingType
     FormFeedType
     ColumnType
     TypeSize
     PrintOptionsRecord
     CommonPropertiesRecord
     GeneralRecord
     GeneralRecordPointer
```

Data Driven Forms

SomeArrayOfBytes
PointerToSomeBytes
DataKindType
DataFormModeType
DataKindAliasType
DataRowType
DataFormType
DataMenuType

Fields

Alignment
FieldKind
FieldDescriptor
FieldPtr
FieldEditResult
SetType
CursorDescriptor

File Form

FFModeType
FFExchangeMode
FFExchangeResult
FFSaveResult
DefaultType

Menus and Forms(Not Data Driven)

MenuFormDescriptor
MenuFormPtr
ChoiceRequest
UpdateKind

Messages and Prompts

MessageStatus
MessagePtr

Strings

Literal
StringDescriptor
StringPtr

Tables

```
ColArray
ColPtr
ScreenArray
ScreenPtr
CellId
SelectionRangeKind
TableSelection
CellTable
TextCursor
CellTablePtr
```

CHAPTER 4: COMMON PROPERTIES AND OPTIONS

Much of the power of the GRiD Management Tools application programs derives from the ability of all the applications to operate on the same set of data files. Thus, a database file created in GRiDFile can be taken into the GRiDWrite application for use in a text file or taken into GRiDPlot to obtain a graphic display of data. The files can be freely exchanged between applications without reformatting the data.

This flexibility and power are achieved by ensuring that the data in all files is in a well-defined format regardless of which application created or most recently changed the file. The scheme used also allows applications to include additional (non-data) information in files to describe special attributes, or properties, of the file.

INTERCHANGE FILES

Files that conform to the format used by GRiD application programs are called "interchange files". These files can contain three different kinds of records; data records, common properties records, and application properties records. Data records contain the actual text and numerical values comprising the "meat" of the file. Common properties records describe such things as how data in the file is to displayed or printed by all applications programs. Application properties records have special meaning only within a particular application. There are three rules that must be observed with records in interchange files:

1. Data records can contain only printable characters, carriage return characters, line feed characters, and tab characters.
2. Properties records (both common and application) must begin with a non-printable character (FF, FE, and FD hexadecimal are currently assigned special significance).
3. Common properties records (if any) must be the first records in an interchange file. (Application properties and data records can occur

in any order after the common properties records.)

DATA RECORDS

Interchange file data records are stored in a tabular format: eeach record consists of a line of printable characters (text or numbers) terminated by a carriage-return line-feed pair and corresponds to one row of data in a table. Tab characters can also be intermixed with the printable characters in a record. In cell-based data files, such as worksheets, database, and graph files, a Tab character is used separate adjacent cells within a row.

Thus, the data records in a file consist only of printable characters, carriage return characters (OD hex), line feed characters (0A hex), and tab characters (09 hex). The following illustration shows the organization of a data record:

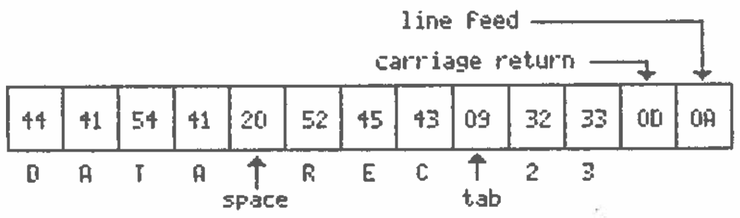


Figure 4-1. Interchange File Data Record Format

This illustration shows the contents of the record in hexadecimal representation and the ASCII equivalent of the record's contents below the figure. Each byte in the record contains a printable ASCII character or the tab character, and the carriage-return line-feed pair mark the end of the data record.

PROPERTIES AND OPTIONS

In order to tightly integrate GRiD applications, some attributes which define the display and printing of data files are stored within the files. These attributes ensure that a visual appearance of a file when it is displayed or printed will be the same regardless of which application is currently operating on the file.

These attributes can be set in various ways within an application. The Options (CODE-O) command is used to set display attributes that apply throughout an entire file, and the Properties (CODE-P) command is used to set attributes that apply only to a column, row, or range of cells. Attributes that apply when a file is being printed are set via an item ("Set Print Options") selected from the Print menu of the Transfer command.

COMMON PROPERTIES RECORDS

When an application reads in a data file, it can examine the first byte of the file to determine whether the file contains any common properties records. (If there are any common properties records, they must be the first records in the file, preceding any data records and application properties records.) If the first record is a common properties record, the first byte read will contain the "common properties flag" byte of FE (hexadecimal).

The format of common properties records can be illustrated as follows:

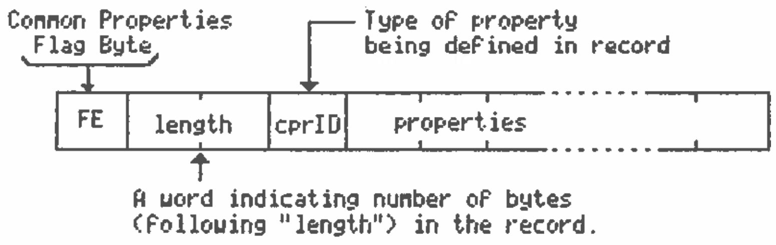

Common Properties Flag Byte
Type of property being defined in record
FE
length
cprID
properties
A word indicating number of bytes (following "length") in the record.

Figure 4-2. Common Properties Record Format

The two bytes (word) following the common properties flag byte (FE hex) define the number of bytes in the record (excluding the flag byte and the 2-byte length indicator). The common properties record identifier (cprID) byte (after the length word) defines which of the common properties this record describes. The cprID bytes currently defined are as follows:

cprID Byte		Property
Hex	ASCII	
61	a	Author (application) of this file
63	c	Column field properties
64	d	Standard field properties
66	f	Cell field properties
68	h	Text header properties
6C	l	Row height properties
6D	m	Print option properties
6E	n	Font properties
72	r	Row field properties
74	t	Table size properties
77	w	Column width properties

The cprID byte tells you which of the common properties recognized by GRiD applications is described in a record. If there are any common properties records in an interchange file, the first record of the file must contain the "author of this file" record.

The properties listed above can be grouped into three categories:

1. The Author ID property
2. Font properties and Print Options
   - o Font properties
   - o Print options
   - o Text header properties
3. Field Characteristics
   - o Standard field properties
   - o Column field and width properties
   - o Row field and height properties
   - o Cell field properties
   - o Table size properties

AUTHOR RECORD

This record MUST appear first in an interchange file if the file contains ANY common properties records. It tells you which application first wrote this file. The format for the Author record is as follows:

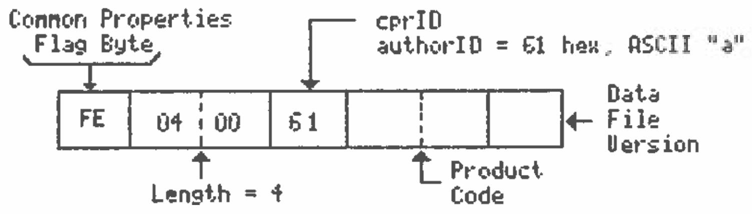


Figure 4-3. Author Record Format

As shown in this illustration, this record has four bytes following the length word. NOTE: Words are always stored with the low-order byte first. Thus, in the length word shown above, a length of four appears as 0400 (hex). The first byte after the length word indicates that the record is an authorID record, the next two bytes contain a product code identifying the application that created the data file, and the last byte is the data version (or compatibility level) of the data file. The product codes currently defined for GRiD applications are as follows:

Application	Product Code	
	Decimal	Hex
GRiDFile	21101	526D
GRiDPlan	21111	5277
GRiDPlot	21121	5281
GRiDWrite	21131	528B
GRiDTerm	21141	5295
GRiD3101	21151	529F
GRiDVT100	21191	52C7

The data version byte at the end of the authorID record defines the compatibility level of the data file. For example, database files created by 2.0 and 1.0 versions of GRiDFile would have this byte set to 01 (hex) while files created by the 3.0 version of GRiDFile have this byte set to 02. This indicates that database files created with older versions of GRiDFile are incompatible with 3.0 GRiDFile.

FONT PROPERTIES AND PRINT OPTIONS RECORDS

These three records define the font to use when displaying a data file, the options to use when printing a data file, and the text header (if any) to use when printing.

Font Record

GRiD application programs let the user select from a number of available fonts (or typefaces) to obtain the screen display of data most suitable for their needs or personal preference. (See Chapter 11 for a discussion of fonts.) The common properties Font record defines which font is to be used when displaying a data file. The format of the record is as follows:

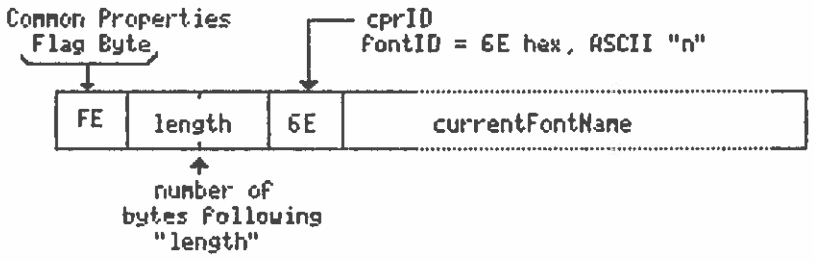

Common Properties Flag Byte
cprID
fontID = 6E hex, ASCII "n"
FE
length
6E
currentFontName
number of bytes following "length"

Figure 4-4. Font Record Format

The text string that is the name (title) of the current font follows the fontID byte. This is just the font name, for example "System-Wide" or "GRiD 64", not the pathname of the file.

Print Options Record

The Print Options record defines the way in which a file will be printed. The information in this record is originally obtained from a Print Options form which GRiD applications can display after a user has selected "Print" from the Transfer menu. The items on the Print Options form correspond to the entries in the Print Options record whose format is as follows:

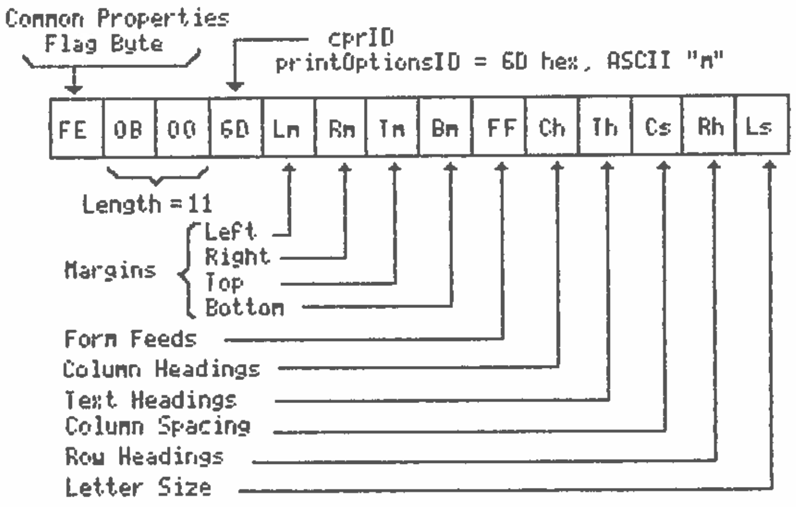


Figure 4-5. Print Options Record Format

The left and right margin bytes indicate the character positions where the first and last characters on each line are to be printed. The top and bottom margin bytes specify the first and last character lines on a page where data is to be printed. (The first possible line number is 1, and the first possible character position is 1.)

The Form Feeds byte specifies when form feeds are to be sent to the printer as follows:

1 = no form feeds
2 = form feeds before printing begins
3 = form feeds after printing is finished
4 = form feeds before and after printing

The column headings byte and the text headings byte specify when the column and text headings from a data file are to be printed as follows:

1 = print no headings
2 = print headings on first page only
3 = print headings on every page except first page
4 = print headings on every page

The contents of the column headings are generated within each application as appropriate and would be indicated in the data file by "private", or application-dependent, properties records (discussed later in this chapter). The Text Heading data is defined in the common properties record described in the next paragraph.

Text Heading Record

Text headings are independent of the application and they simply consist of a text string that can be printed (as specified by the column headings byte) centered at the top of pages. The format for the Text Heading properties record is as follows:

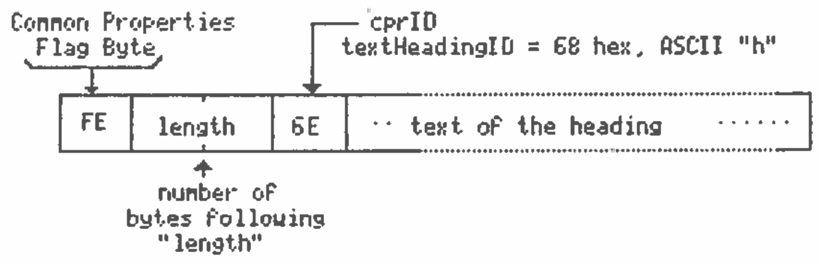

Common Properties Flag Byte
cprID
textHeadingID = 68 hex, ASCII "h"
FE
length
6E
·· text of the heading ······
number of bytes following "length"

Figure 4-6. Text Heading Record Format

FIELD CHARACTERISTICS RECORDS

These common properties records define the alignment and format of data displayed in columns, rows and cells, the width of columns, the height of rows, and the size of a table.

A single cell can have its properties defined in one of four ways:

- o By a cell field properties record (one or more cells selected after CODE-P)
- o By a column field properties record (an entire column is selected (CODE-C after CODE-P)
- o By a row field properties record (an entire row is selected (CODE-R after CODE-P)
- o By a standard field properties record (the Options command -- CODE-O)

Notice that a cell does not have to have its properties explicitly defined. Initially, a data file has all of its field (cell) characteristics defined by the standard field properties record; no additional field properties records are required until a user sets properties in an file (using the CODE-P command). At that point, you must create a properties record for the fields that were changed to be different than the standard settings. Depending on what fields the user selected, a cell, column, or row properties record will be required.

The approach used in defining properties minimizes the total number of field properties records that are required to fully describe the characteristics of a data file and thus keeps the file as small as possible. Obviously, it would be inefficient to define cell properties individually if all of the cells in a row or column have the same properties. Similarly, there is no need to define the properties of a column or row if they are they same as the standard properties.

To determine the properties of a cell, GRiD applications first look to see if the properties are defined in a cell field properties record. If not, then the column, row, and standard field records (in that order) are examined. The first of these records that contain a definition for the cell in question is the one that takes precedence.

Whenever a user sets properties for cells, columns, or rows, the application may have to alter previously set properties if they differ from the new ones. Therefore, existing properties records may have to be deleted or changed and new ones created. Since the search sequence defined for determining cell properties looks first to the cell, then the column, and thirdly to row, an application must examine field property records in that order to check for possible conflicts between new properties settings and previous ones.

For example, if you are changing the properties of a row, you would follow these steps:

1. Check for a cell properties record for the row being defined. Since cell properties records are defined on a per-row basis, and since the entire row is being defined, a cell properties reocrd for this row can be discarded.
2. Check for existing column properties records. If one or more column properties records conflict with the new row properties, then cell properties records must be defined for the intersections of the row and any conflicting columns.
3. Check existing row properties. If a field record already exists for this row, update it accordingly. If does not exist, create a new row field properties record.

NOTE: A special value of 255 decimal (FF hex) can be used in cell, column, and row field properties records to indicate that the "default" properties should be used. Default, in this context, means "do not use this properties byte; instead, look to the next level of field properties records for the properties." Thus, if you encountered the default byte (255) in a cell field properties record, you would then look to the column field record for properties.

Standard Field Record

The Standard Field properties record defines the standard settings for column width, format (decimal, integer, etc.), and alignment (left, center, right). These settings are defined by the user in GRiD applications with the Options (CODE-O) command. The format for the Standard Field properties record is as follows:

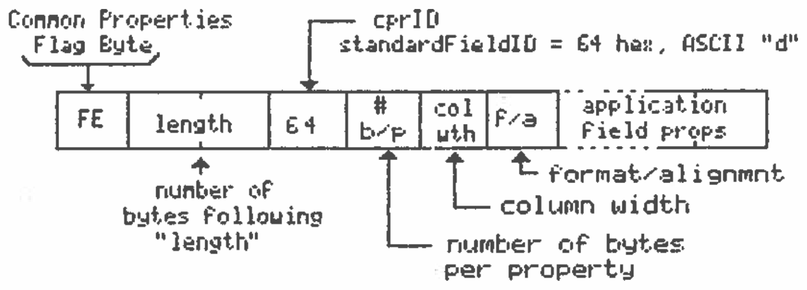

Common Properties Flag Byte
cprID
standardFieldID = 64 hex, ASCII "d"
FE
length
64
# b/p
col wth
f/a
application field props
number of bytes following "length"
format/alignmnt
column width
number of bytes per property

Figure 4-7. Standard Field Record Format

The byte following the standardFieldID, indicates the number of bytes that will be used to define each field property. Currently, most GRiD-developed applications use only a single byte to define the field properties of format and alignment. The bytes/property indicator, however, provides for using multiple bytes in the future to allow larger values or define additional field properties.

> NOTE: The bytes/properties value specified here applies not just for the standard field record, but also to the row field, column field, and cell field property records which will be described later in this chapter.

The next byte defines the standard width for columns in cell-based applications. (GRiD-developed applications currently use a standard width of 8 characters for columns.)

The format/alignment byte defines the standard arithmetic format (decimal, integer, etc) that will be used and the standard alignment (left, centered, right) that will be used for displaying data in cells. (GRiD-developed applications currently use a standard format of integer and a standard alignment of right-justified.) The 5 most-significant bits of this byte specify the format and the three least-significant bits specify the alignment. The contents and interpretation of this byte can be illustrated as follows:

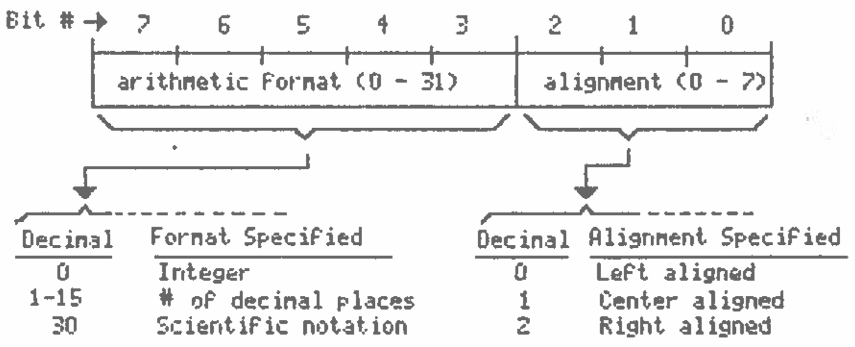
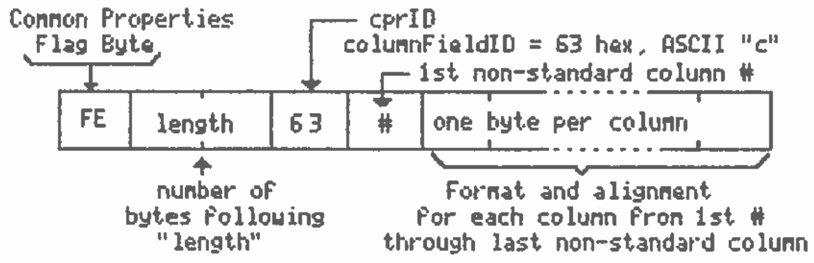


Figure 4-8. Format/Aignment Byte of Field Records

Column width and format/alignment are the only standard properties currently defined across all cell-based GRiD applications. However, each application can append additional standard properties bytes following the format/alignment byte. Thus, each application can establish any application-specific standard properties (Options) it might need. For example, GRiDPlan has options defining such things as evaluation order (row versus column), and whether or not to display a grid as a background for the worksheet.

Column Field Record

The Column Field properties record defines the settings for arithmetic format (decimal, integer, etc.), and alignment (left, center, right). This record overrides the settings of the Standard Field properties record and would be created as a result of the user setting column properties using the CODE-P command. The format for the Column Field property record is as follows:



Common Properties Flag Byte
cprID
columnFieldID = 63 hex, ASCII "c"
1st non-standard column #
FE
length
63
#
one byte per column
number of bytes following "length"
Format and alignment for each column from 1st # through last non-standard column

Figure 4-9. Column Field Record Format

The colFieldPropsID byte (63 hex) follows the common property flag and the length word. The next word indicates the first column in the file that is of a non-standard format or alignment. (NOTE: although a 16-bit word is provided to define the column number, no GRiD-developed applications currently allow more than 255 columns.) The subsequent bytes define the format and alignment of each of the succeeding columns. You must define the column format and alignment (even if they happen to be standard) of all columns until the last non-standard column has been defined. For example, if columns 2, 5, 7, and 9 (in a table that is 15 columns wide) are non-standard, the ColumnFIeld record must define the format/alignment of columns 2 through 9. Columns 1 and 10 - 15 will assume the standard format/alignment, but you must explicity define the standard format/alignment for columns 3, 6, and 8 in the Column Field record. You can specify standard format for a column in this record with a value of 31 (decimal) and standard alignment for a column in this record with a value of 6 (decimal). The following illustration shows the organization and interpretation of the format/alignment bytes in the Column Field record.

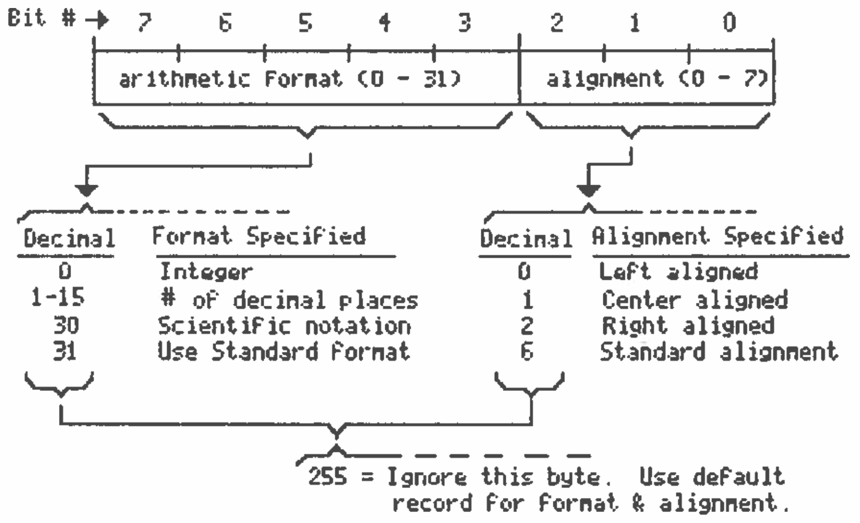


Figure 4-10. Column Field Record Format/Alignment Byte

This is the same as was shown earlier for the Standard Field properties record with the addition of the "Standard" and "Default" definitions. (Those definitions obviously had no meaning with the Standard Field properties record since that record itself defines the "standard" and it is also the last place you look for "default" definitions.)

Row Field Record

The Row Field properties record defines the settings for arithmetic format (decimal, integer, etc.), and alignment (left, center, right). This record overrides the settings of the Standard Field properties record and would be created as a result of the user setting row properties using the CODE-P command. The format for the Row Field property record is as follows:

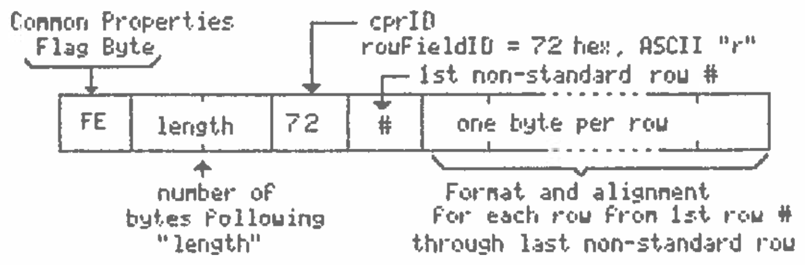

Common Properties Flag Byte
cprID
rowFieldID = 72 hex, ASCII "r"
1st non-standard row #
FE
length
72
#
one byte per row
number of bytes following "length"
Format and alignment for each row from 1st row # through last non-standard row

Figure 4-11. Row Field Record Format

As you can see, this record is nearly identical to the Column Field record described earlier. The only difference is the cprID byte value.

Column Width Record

The standard column width for cell-based applications is specified by the Standard Field Record (described earlier). If one or more columns in a data file are set (via the Properties command in an application) to be different than the standard width, you must write a ColumnWidth record into the data file to define the non-standard columns. The format for the Column Width common property record is as follows:

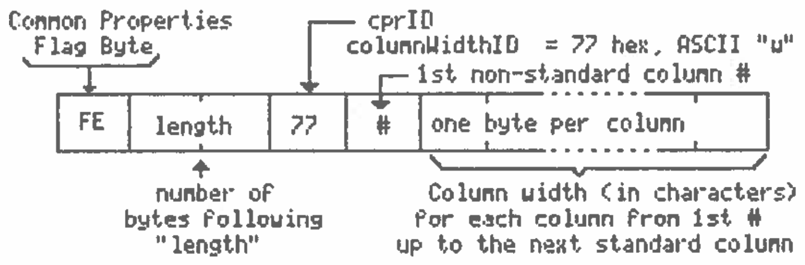

Common Properties Flag Byte
cprID
columnWidthID = 77 hex, ASCII "w"
1st non-standard column #
FE
length
77
#
one byte per column
number of bytes following "length"
Column width (in characters) for each column from 1st # up to the next standard column

Figure 4-12. Column Width Record Format

The colWidthID byte (77 hex) follows the familiar common property flag byte and the length word. The next word indicates the first column in the file that is of a non-standard width. (NOTE: although a 16-bit word is provided to define the column number, no GRiD-developed applications currently allow more than 255 columns.) The subsequent bytes define the width (in characters) of each of the succeeding columns. You must define the column widths (even if they happen to be standard width) of all columns until the last non-standard column has been defined. For example, if columns 2, 5, 7, and 9 (in a table that is 15 columns wide) are non-standard, the ColumnWidth record must define the widths of columns 2 through 9. Columns 1 and 10 - 15 will assume the standard column width, but you must explicity define the standard column width for columns 3, 6, and 8 in the ColumnWidth record. You can specify standard width for a column in this record with a byte of FF hex (255 decimal).

RowHeight Record

The standard row height for cell-based applications is one character line. This is also the only value used in GRiD application programs and none of them currently make any provision for setting a row height other than one character line high.

However, to provide for possible future enhancements, a common properties record is defined for row height. The format for this record is as follows:

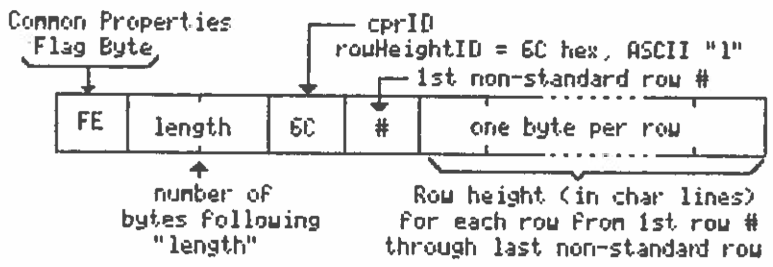

Common Properties Flag Byte
cprID
rowHeightID = 6C hex, ASCII "l"
1st non-standard row #
FE
length
6C
#
one byte per row
number of bytes following "length"
Row height (in char lines) for each row from 1st row # through last non-standard row

Figure 4-13. Row Height Record Format

As you can see, this record is almost identical to the ColumnWidth record; only the rowHeightID byte (6C hex) is different.

Cell Field Record

The Cell Field properties record defines the settings for arithmetic format (decimal, integer, etc.), and alignment (left, center, right) within individual cells. This record overrides the settings of the column or row properties records and would be created as a result of the user setting properties using the CODE-P command for a cell or range of cells. The format for the Cell Field property record is shown in the following illustration:

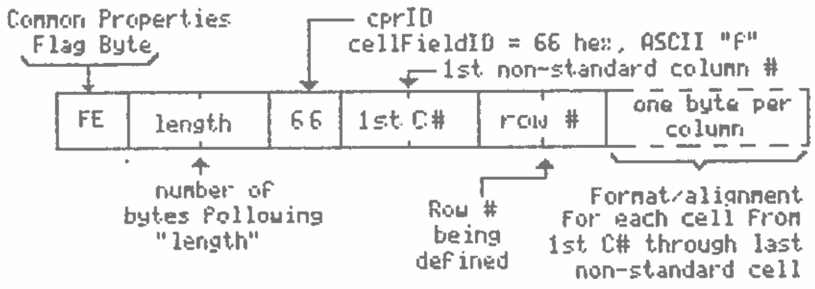

Common Properties Flag Byte
cprID
cellFieldID = 66 hex, ASCII "F"
1st non-standard column #
FE
length
66
1st C#
row #
one byte per column
number of bytes following "length"
Row # being defined
Format/alignment for each cell from 1st C# through last non-standard cell

Figure 4-14. Cell Field Record Format

The cellFieldPropsID byte (66 hex) follows the common property flag and the length word. The next word indicates the first column in the row that is of a non-standard format or alignment.

> (NOTE: although 16-bit words are provided to define the column number and the row number, no GRiD-developed applications currently allow more than 255 columns or rows.)

After specifying the column number, the next word specifies the row where format/alignment is being defined. The subsequent bytes define the format and alignment of each of the succeeding cells within that row. You must define the cell format and alignment (even if they happen to be standard) of all cells in that row until the last non-standard cell has been defined. For example, if columns 2, 5, 7, and 9 (in a table that is 15 columns wide) in row 3 are non-standard, the CellFIeld record must define the format/alignment of cells in columns 2 through 9 in row 3. Cells in columns 1 and 10 - 15 will assume the standard format/alignment, but you must explicity define the standard format/alignment for cells in columns 3, 6, and 8 in this Cell Field record. You can specify standard format in this record with a value of 31 (decimal) and standard alignment for a column in this record with a value of 6 (decimal). See the discussion of the Column Field record for a descripton of the format/alignment bytes.

Note that you must define a Cell Field record on a per-row basis. Each record indicates the row number where non-standard properties exist and also the first column within the row where non-standard properties begin.

TableSize Record

This common properties record is not currently used by any GRiD applications. It is intended to let you quickly discover the size of an entire table in a data file. However, since most GRiD-developed applications provide an Append command, the size of a data file and, thus its table, can be changed without ever bringing that file into memory. Nonetheless, the Table Size record is still defined and may be of future use. Its format is as follows:

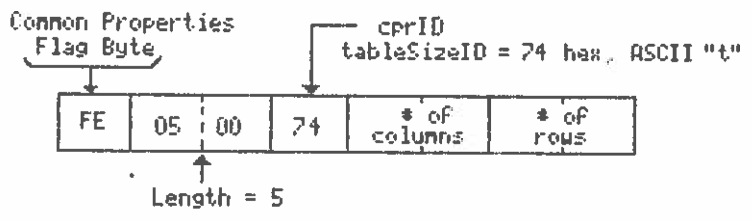


Figure 4-15. Table Size Record Format

APPLICATION PROPERTIES RECORDS

Application properties records can appear anywhere within a data file (except that they cannot appear until after any common properties records). They have predefined beginning identifiers but the actual contents of the records can be in any format and are interpreted within the context of that application.

The format for the type of application properties record currently used in most GRiD applications is as follows:

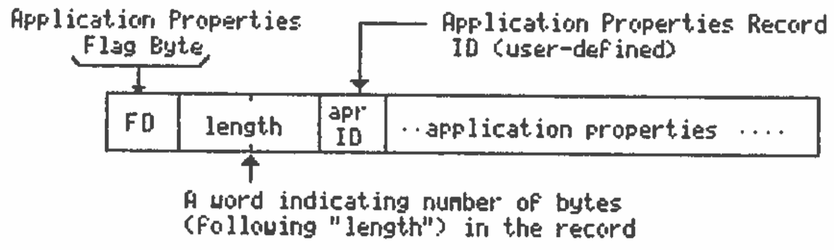

Application Properties Flag Byte
Application Properties Record ID (user-defined)
FD
length
apr ID
..application properties ....
A word indicating number of bytes (following "length") in the record

Figure 4-16. Application Properties Record Format (Binary)

These records begin with FD (hexadecimal) and identify an application properties record that consists of binary data. The word following the application properties flag byte defines the length of the record -- that is, the number of bytes following the "length" word.

Application properties records that begin with a byte of FF (hexadecimal) are expected to contain ASCII (textual) data and are terminated with a carriage-return line-feed pair (just like a data record). The format for this type of record can be illustrated as follows:

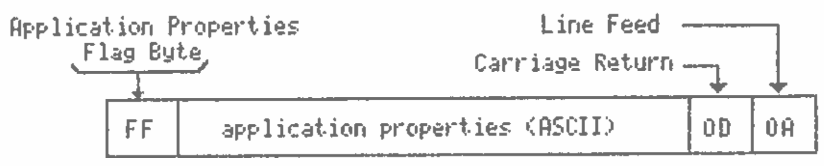

Application Properties Flag Byte
Line Feed
Carriage Return
FF
application properties (ASCII)
0D
0A

Figure 4-17. Application Properties Record Format (ASCII)

This type of properties record is currently used only in GRiDPlan to allow ASCII formulae describing cell definitions. Other GRiD applications use the binary format with its "length" word to store application properties records.

PROPERTIES RECORDS COMMON CODE ROUTINES

The Common Code package provides several routines to simplify the handling of properties records. The available calls and their purpose are as follows:

Common Code Call	**Purpose**
AuthorOfThisFile	Returns the author product code word and version byte from the first common properties record in a data file. You can then decide whether you need to look at application properties records in the file.
SkipProperties	Skips over all the common properties records in a data file.
GetNextRecord	Returns a pointer to the next record in a data file and also returns the length of the record. You can also specify that the call automatically skips over all application properties records in the file if you are not "author of this file".
FinalizePropertiesLength	Calculates the current length of all the common properties records in a data file and records that value in the file header for the file when it is written to a device. This value lets the SkipProperties call skip over the common properties when the file is subsequently read.

For a complete description of these calls, refer to Chapter 12.

CHAPTER 5: STRINGS

The Common Code string routines add ASCII character string manipulation to PASCAL. The strings are dynamic structures that can be allocated as needed while a program executes. They avoid PASCAL's rigorous type-checking, so their length does not need to be defined until they are referenced in a program.

DATA STRUCTURES

```
* TYPE StringDescriptor =
            RECORD
            len,max: Word;
            dummy: Byte;
            chars: ARRAY [1..65535] OF Char;
            END;
```

StringDescriptors have these elements:

len	The current length of the string. It may vary from 0 to max. Do NOT allow it to exceed string.max.
max	The maximum length of the string in characters. NEVER MODIFY MAX because the memory allocation routines refer to it when they create or dispose of the string. Also, do NOT refer to a character position beyond the max, because your doing so may destroy the memory space of other variables.
dummy	A dummy variable that was included so that PASCAL strings will be compatible with PL/M strings. The dummy enables the PL/M character array to be declared beginning with a 0, because PL/M arrays must begin at 0. Then, both PASCAL and PL/M can refer to all strings by beginning with string.chars[1].

chars	An array of characters, which is the actual string itself. Even though the string array has been declared 65535 characters long, you would never actually allocate a string this long. This "dummy length" enables the string package to allocate strings of different length during program execution. You can pass these strings as VAR parameters because they are not PACKED.

Note: These string structures must not be allocated with PASCAL allocation routines. You must create and dispose of them with the special procedures (NewString and NewStringLit) described below.

* **TYPE StringPtr** = ^StringDescriptor;

The StringPtr is the most common device for actually referring to a string or passing it as a parameter of a procedure. In fact, this pointer structure is what permits the string package to create and dispose of strings dynamically. Many common code and string routines require StringPtrs as their arguments.

* **TYPE Comparison** = (equal, less, greater);

A value of type Comparison will be returned from the string comparison routines (EqualStrings and CompareStrings). A result of less or greater means that the ASCII values of some characters in one string are numerically less or greater than those in another.

THE STRING ROUTINES

Common Code provides routines to allocate and deallocate strings, compare and modify strings, convert strings, and operate on substrings. In this chapter we will provide an overview of the routines available and give brief examples of their use. Complete descriptions of the individual routines are provided in the alphabetically ordered reference chapter -- Chapter 12.

Allocating and Deallocating Strings

Two Common Code functions are provided to allocate strings and a third is used to deallocate strings.

NewString	Allocates memory for a new string of a specified length and returns a string pointer to that area in memory.
NewStringLit	Takes a literal string, allocates memory for it, and returns a string pointer. The maximum length (max) and current length (len) of the new string is the length of the literal characters.
FreeString	Given the StringPtr to a string, FreeString will release the memory that the string occupied and return that memory to the PASCAL heap.

CAUTION: ONLY NewString and NewStringLit WILL PROPERLY ALLOCATE SPACE FOR STRINGS. NEVER call New(StringPtr) because New will allocate all 65535 bytes according to the declaration of String^.chars[1..65535] above.

When declaring your own static variables to deal with strings, you must declare them to be StringPtrs, NOT Strings. If you declare a static variable as type String, the compiler will try to allocate 65535 bytes for String^.chars[1..65535] according to the declaration of the String record. You should declare the variable to be of type StringPtr and then assign it with the value resulting from a call to NewString or NewStringLit.

Note: you must NEVER modify String^.max, because FreeString uses that number to determine how much memory to release to the heap. Other data values may be incorrectly released if String^.max is changed from its original value.

Comparing Strings

Two functions are provided to compare strings:

EqualStrings	Compares two strings character by character and returns True if they have the same characters and the same number of characters.
CompareStrings	Compares the ASCII values of two strings character by character, from left to right. Thus the greater string will be the one containing the first character with a higher ASCII value. If two strings match up exactly except that one string has additional characters, then the string with the extra characters will be the greater one.

Modifying Strings

Twelve different routines are provided by Common Code to give you great flexibility in modifying strings and manipulating:

CopyString	Copies the value of the source string to the dest string. Both source and destination must be allocated already.
CopyOfString	Creates a new string and copies the value of str to it.
AppendString	Concatenates a source string to the tail of a destination string. The source string remains unchanged.
AppendChar	Appends a single character to the tail of a destination string.
AppendAnyChar	Appends a single character to the tail of the string. It disposes of the original str and sets str to the newly created string.
Concat	Creates a new string containing str1 and str2 concatenated together.
ConcatStrings	Creates a new string containing str1 and str2 concatenated together and disposes of str1 and str2 after creating the new string.
ConcatLits	Creates a new string with the literals lit1 and lit2 concatenated together to let you concatenate string

	constants.
DeleteFromString	Deletes characters from a string and then joins the remaining characters together to close the gap.
InsertInString	Inserts a string into another string. The existing characters of str are moved aside to make room for the insertion.
InsertCharInString	Inserts a single character into a string beginning at a specified character position.
SubStringLit	Returns the Nth item (specified by count) from a literal that contains text items separated by delimiter characters.

Converting String Types

Five routines are provided to perform real number conversion, integer conversion, and convert lower case characters to uppercase.

UpperCase	Converts any lowercase alphabetic characters in the string to uppercase. It does not shift up numerals, punctuation, or special characters.
IntegerToString	Converts an integer between -32768 and 32767 inclusive into a string, then returns a stringPtr to the string value.
StringToInteger	Converts a string value into an integer. The string must represent an integer between -32768 and 32767 inclusive for the conversion to succeed. The variable "converted" indicates whether the conversion was successful or not.
RealToString	Converts a fifteen digit real number into a string variable. (The 8087 numeric processor uses fifteen and a half digits of precision; this routine returns fifteen digits, rounding off the half digit as necessary.)
StringToReal	Converts a string value into a real number. The variable "converted" indicates whether the conversion was successful or not. It will convert up to the first fifteen digits, and drop any extra digits without causing an error. If the conversion fails (from incorrect input, for example) the routine returns 0.

Note that the two real number routines produce real numbers of fifteen and a half digits. They cannot accomodate exponential notation, such as 6.03E+23. For details, see the Intel 8087 Floating Point Processor Manual or the Pascal Manual.

Miscellaneous String Routines

Five routines are provided to simplify handling of various other strings used throughout the system.

TimeToString	Converts time and date information from the OS to a string for easy use in an application.
SubProperty	Picks a name out of a string made up of names and special characters. The special characters are delimiters in the GRiD-OS file system.

SetPrefix	Used by the application to set the prefix subject.
GetVersionString	Returns a string containing a three numeral version number of a piece of software. Each process (application) can have a version number.
TranslateHeading	Translates the input into a centered output string for printing on an Epson printer. Special symbols are translated in the upper or lower case.

CHAPTER 6. COMMANDS, MESSAGES, AND PROMPTS

These Common Code routines simplify implementation of several commands that are used in all GRiD-developed applications and provide a standard, easy-to-use mechanism for displaying messages and prompts.

Commands

The four command-related routines provided by Common Code are listed in the table below. Two items that are standard on the Transfer menu supported in all application GRiD applications are "Erase a file" and "Show file characteristics" The first two calls listed in the table (CmdErase and CmdProperties) let applications share the code needed for these activities. Similarly, CmdMediaUsage and GetVersionString are supplied by Common Code since CODE-U and CODE-? are supported by all GRiD applications.

Routine	**Description**
CmdErase	Erases a file and displays appropriate prompts and messages. Used to implement "Erase a file" from the Transfer menu.
CmdProperties	Displays the properties of a specified file. Used to implement "Show file characteristics" from the Transfer menu.
CmdMediaUsage	Displays memory and media usage. Used to implement the CODE-U command.
GetVersionString	Returns a pointer to the version number string of the specified process. Used to display an application's version information as part of the CODE-? screen and when the application is first loaded.

MESSAGES AND PROMPTS

Messages and prompts are text fields that can be displayed anywhere in the window (full window width). The text is highlighted (inverse video) and its default position (which is the one typcally used by GRiD applications) is the bottom of the window. When a message or prompt is erased, it is the responsibility of the application to update the necessary rectangle. In GRiD applications, messages and prompts are handled in slightly different ways . The conventions used in GRiD applications are as follows:

> Messages -- A message should be displayed only until the next keystroke by the user of the application. For example, the user presses CODE-U to see media usage, the usage message appears, and remains on the screen until the user presses another character. The message should always be cleared BEFORE a character is processed.
>
> Prompts -- A prompt should be displayed only while an application is in command mode or responding to a MsgExit routine. The prompt persists until one of the following criteria is satisfied:
>
> 1. The user satisfies the condition of the prompt and presses CODE-RETURN.
> 2. The user presses ESC to escape the condition of the prompt.
> 3. The user presses another CODE key to preempt the current command.
>
> Prompts should be cleared after a character is processed unless the application is remaining in command mode. For example, if the users presses CODE-D to duplicate, the prompt "Duplicate: Make a selection and confirm" is displayed. This prompt should remain displayed while the user presses arrow keys to make a selection. The prompt should be cleared only when the user presses CODE-RETURN to confirm the command or when the user presses ESC or another CODE-key command to pre-empt the current command.

Note that these conventions are not enforced by the message and prompt routines; rather they are conventions observed by GRiD applications and which should be adhered to by other applications for the sake of consistency.

Data Structures

An application and the Common Code communicate message/prompt information by passing a pointer to a dynamic data structure of type MessageStatus. This variable must be declared in the application and initialized (dynamically allocated) by a call to FUNCTION MsgInit. (See MsgInit) The organization of the MessageStatus record is as follows:

```
TYPE MessageStatus =
       RECORD
       messageShowing: Boolean;
       stackSize : Byte;
       field: FieldPtr;
       rect: Rectangle;      {area to be updated}
       anythingShowing : Boolean;
       END;
```

```
     MessagePtr = ^MessageStatus;
```

messageShowing	A boolean that indicates if a message is currently displayed. If a prompt is showing, or if no message is showing, it is false. This field is NOT altered by the application. It is initialized by MsgInit and updated by the various message calls.
stackSize	Indicates the number of messages/prompts currently showing. This is NOT altered by the application. It is initialized by MsgInit and updated by the various message calls.
field	Pointer to the field descriptor record containing the text and location of the message.
rect	The rectangle that the application should update if the boolean result of one of the message FUNCTION calls is true. This value is initialized by MsgInit, updated by the various message calls, and read by the applications. It is not altered by the applications.
anythingShowing	Boolean field that is not used in the current version of the Common Code message module.

The organization of the field descriptor record pointed to by the *field* parameter is as follows:

```
     FieldDescriptor = RECORD
                         box: Rectangle;
                         text: StringPtr;
                         kind: FieldKind;
                       END;

     FieldPtr = ^FieldDescriptor;
```

Refer to Chapter 10 for a detailed description of this record.

Message and Prompt Routines

The ten message and prompt routines listed below are used to initialize, display, and clear messages and prompts. Messages and prompts can be displayed as the only message/prompt line in the window with any previous messages/prompts erased. Alternatively, they can be "stacked"; that is displayed above any previously displayed messges/prompts.

Routine	**Description**
MsgInit	Allocates and initializes a MessageStatus record and returns a pointer to the record.
MsgShowMessage	Displays a one-line message after erasing any previous message(s) or prompt(s). Returns a True boolean if the application should update the rectangle.

MsgStackMessage	Displays a message stacked on top of any currently displayed messages. Erases any previous prompt(s). Returns a True boolean if the application should update the rectangle.
MsgShowPrompt	Dispays a one-line prompt after erasing any previous prompt(s) or message(s). Returns a True boolean if the application should update the rectangle.
MsgStackPrompt	Displays a prompt stacked on top of any currently displayed prompt(s) or message(s). Returns a True boolean if the application should update the rectangle.
MsgShowError	Displays a one-line error message after erasing any previous message(s) or prompt(s) and then locks the keyboard for two seconds. Returns a True boolean if the application should update the rectangle.
MsgShowDecoded	Displays an error message specified by a GRiD-OS error code after erasing any previos message(s) or prompt(s) and then locks the keyboard for two seconds. Returns a True boolean if the application should update the rectangle.
MsgClearMessage	Clears any messages currently displayed but has no effect on currently displayed prompts. Returns a True boolean if the application should update the rectangle.
MsgClearPrompt	Clears any prompts currently displayed and any messaes that have prompts stacked on them. Has no effect on currently displayed messages if there are no stacked prompts. Returns a True boolean if the application should update the rectangle.
MsgExit	Displays one of three messages before exiting the application. Allows normal exit "Retrieving subjects: In progress", memory exhausted message "Out of memory: Confirm to exit", or reboot message "System Error: (code) Confirm to reinitialize system.

CHAPTER 7: BYTE MANIPULATION PROCEDURES

This chapter describes the byte manipulation procedures, which enable you to move, search for, and assign values to individual bytes in memory. They are the PASCAL equivalents to the PL/M String Manipulation Procedures (i.e. byte strings) as defined in the PL/M-86 User's Guide.

These routines provide a very rapid and efficient mechanism for updating the bit-mapped screen. By altering the memory allocated to the screen, the screen's display will change.

These routines use parameters of type Bytes to identify areas in memory. A segment of memory can be visualized as a one-dimensional array of bytes. The parameter of type Bytes acts as a pointer to the first element of the array of bytes.

BYTE ROUTINES

Routines are provided to move, find, compare, set, insert, and delete bytes.

WARNING: The two movement procedures can move up to 64K bytes in a segment at once. Memory areas are NOT PROTECTED by hardware. Use the "move" routines with care.

MoveBytes	Moves data from one location in memory to another.
MoveReverseBytes	Moves data from one location in memory to another starting from the end of the data rather than the beginning. This allows you to move bytes into a destination that overlaps the source location.
FindByte	Searches an array of bytes in memory for a given character, and returns its position in the array.
CompareBytes	Compares one memory area with another one to see whether they match. They must be the same length.
SetBytes	Sets every byte in the destination area to the same given

	value.
InsertBytes	Inserts bytes into a specified area of memory. The contents of the inserted bytes are undefined. This procedure is useful for inserting new elements into arrays, structures, strings, etc.
DeleteBytes	Deletes a given number of bytes from an area of memory; the remaining bytes are moved together to close up the resulting gap. This procedure is useful for removing elements from arrays, structures, strings, etc.

CHAPTER 8: MENUS AND FORMS

Commands can request data from the user by presenting a menu or a form. With a menu, the user selects a single value as input to the Compass. Forms allow the user to give the computer several values at once. This chapter describes the routines and techniques available to programmatically display menus and forms. The technique used is known at GRiD as "Data Driven Menus and Forms" because a predefined data structure determines both the appearance and contents of a menu or form.

> NOTE: This technique requires that you have both the Pascal and PLM compilers. There is another, older and more complicated, method of doing menus and forms that does not require the PLM compiler. This alternate technique is described in Appendix C.

A special form, the File form which is used throughout GRiD applications, is also described in this chapter.

OVERVIEW OF DATA DRIVEN MENUS AND FORMS

The basic concept behind using data driven menus and forms is simple. You need two modules. One module is written in PLM and contains data structures which specify what your forms and menus look like. The other module is written in Pascal and calls Common Code routines to display these menus and forms. When the Pascal routine calls Common Code to display a menu or form, it passes the name of the PLM data structure representing that menu or form as a parameter. Common Code takes care of the rest.

Why use PLM at all? Couldn't it all be written in Pascal? Yes it could, but Pascal does not let you declare variables with initial values and PLM does. If the data structures were declared in Pascal then extra code would be needed to initialize them before they could be used.

DATA DRIVEN MENUS

A menu is one of the simplest means of getting input from the user. GRiD menus eliminate unnecessary and repetitive typing. Instead of typing, users select standard inputs by pressing arrow keys and CODE-RETURN.

With Common Code routines, you can add GRiD menus to your programs very quickly. Once you're familiar with them, they will be easier to program than traditional methods of getting the user's input. And, they'll make your programs easier to understand and to use.

You don't have to understand the structure of the entire Common Code in order to program a menu. You can copy the example code given in this chapter, and modify it for your application.

Typical items for a menu include:

- o Parameters to a program. The program will take the selected parameter instead of any of the others.

- o Operations to be completed. The program completes the selected operation instead of any of the others.

The sample menu that we illustrate in this chapter is similar to the Transfer menu that is used throughout GRiD applications. This basic example is a building block that you can flesh out or modify to meet your needs. It displays seven items on the screen for the user to select. Each item represents an operation. When one of the items is selected and {confirmed,} the computer would perform the selected operation. (To keep the example simple, we will not show any of the code that would perform the selected operation.)

A GRiD menu returns information telling you which one of the possible menu items the user picked. Figure 8-1 shows a sample GRiD menu. All GRiD menus resemble this one.

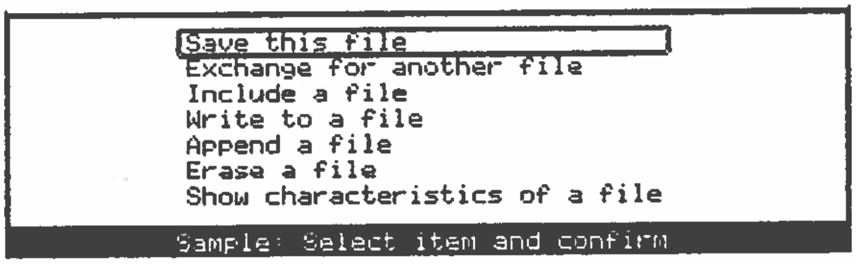

Save this file
Exchange for another file
Include a file
Write to a file
Append a file
Erase a file
Show characteristics of a file
Sample: Select item and confirm

Figure 8-1: A Sample Menu

A menu consists of:

Menu items   A vertical list of objects or operations, such as commands, file

	titles, or storage media. By confirming an item, the user tells the Compass to operate with that item instead of the others. The items are display-only fields that the user cannot modify.
Outline	A moving indicator that surrounds the current item. A triangular cursor never appears within this outline, because text can never be typed into a menu. You select an item by moving the outline to the desired item and confirming (pressing CODE-RETURN).
Message Line	An informational message instructing the user as to what action to take with the menu.

Menu Data Structures and Types

The DataMenuConfirmed routine that displays menus is given a pointer to a PLM data structure when the function is called. The pointer is defined as follows:

```
     SomeArrayOfBytes = ARRAY [1..1] OF CHAR;

     PointerToSomeBytes = ^SomeArrayOfBytes;

     DataMenuType = PointerToSomeBytes;
```

The array of bytes being pointed to is the data structure that is defined by the PLM module.

Figure 8-2 shows the PLM data structures that define the menu shown in Figure 8-1. The PLM data structures are almost self-explanatory. The first data declaration (the "sampleMenuTemplate") defines the constants which are the menu items to be displayed for the menu. A tilde (~) delimits each menu item and a vertical bar (produced by pressing Code-Shift-;) marks the end of the list of items. The second data declaration defines a pointer to the menu item data. Note that names for the two data items, "sampleMenuTemplate" and "theSampleMenu" are user defined and that the name "sampleMenuTemplate" appears in two places. When you change one then you must change the other.

```
     /*************** Sample menu ***************/

     DCL sampleMenuTemplate (*) BYTE PUBLIC DATA
         ('Save this file~',
          'Exchange for another file~',
          'Include a file~',
          'Write to a file~',
          'Append a file~',
          'Erase a file~',
          'Show characteristics of a file~¦');

     DCL theSampleMenu PTR PUBLIC DATA (@sampleMenuTemplate);
```

Figure 8-2: PLM Data Structures for Sample Menu

Data Driven Menu Routines

One Common Code function (DataMenuConfirmed) does all the work to implement data driven menus. The DataMenuConfirmed function displays the menu you have defined in the PLM data structure. It also lets the user select an item by pressing arrow keys. Note that DataMenuConfirmed cannot act upon the user's selection until after the user has pressed CODE-RETURN. Pressing CODE-RETURN is the only way to indicate that the outlined item in the menu should be used for processing.

The DataMenuConfirmed function definition is as follows:

```
     FUNCTION DataMenuConfirmed (dataMenu : DataMenuType;
                                 msgStatus : MessagePtr;
                                 msg : StringPtr;
                             VAR rect : Rectangle;
                                 keyProcess : WORD;
                             VAR selection : INTEGER;
                             VAR ch : CHAR) : BOOLEAN;
```

When the function is called, it is given a pointer to the PLM data structure defining the menu, and pointers to your message area and the prompt to be displayed with the menu. It returns the selected item and a Boolean indicating whether the menu was confirmed. Refer to Chapter 12 for a complete description of the parameters for DataMenuConfirmed.

Data Driven Menu Example

Figure 8-3 Shows the Pascal procedure that displays the menu shown in Figure 8-1. The Common Code function call "DataMenuConfirmed" displays the menu.

```
{-------------------------------------------------------------------------}
PROCEDURE SampleMenu;
VAR str: StringPtr;
    rect: Rectangle;
    itemSelected: Integer;
    confirmed: Boolean;
BEGIN
  str       := ConcatLits (TransferMsg, SelectMsg);
  rect      := windowRect;
  confirmed := DataMenuConfirmed
                (theSampleMenu,
                 msg,
                 str,
                 rect,
                 cursor.keyProcess,
                 itemSelected,
                 ch);

  IF confirmed THEN
    BEGIN
      CASE itemSelected OF
        1: ;  { do appropriate action for "save"}
        2: ;  { do appropriate action for "exchange"}
        3: ;  { do appropriate action for "include"}
        4: ;  { do appropriate action for "write"}
        5: ;  { do appropriate action for "append"}
        6: ;  { do appropriate action for "erase"}
        7: ;  { do appropriate action for "show characteristics"}
        8: ;  { do appropriate action for "print"}
        OTHERWISE;
      END;
    END;
END;
```

Figure 8-3: Pascal Procedure to Display Sample Menu

Appendix B contains complete source listings and the link command file for a progran that displays the sample menu.

If the user did not press CODE-RETURN to leave the menu, the variable ch contains the character which the user pressed to exit the menu instead. Any character except CODE-RETURN or Arrow keys will cause the menu to be exited. In this example, DataMenuConfirmed does not do anything with ch, but it returns the value of the variable "confirmed," so that the calling procedure will know whether an item on the menu was confirmed or not.

DATA DRIVEN FORMS

Forms are similar to menus in that they capture information specified by the user but they are different from menus in the following ways:

- o Forms can let users change the settings of several items. Menus let them confirm only one item.
- o Users can type their own settings. Forms do not have to limit them to predefined choices .
- o When users press CODE-RETURN, they confirm the settings of all the form items, not just the currently outlined setting.

Forms enable users to change the settings of many parameters with very little typing. They replace the tedious practice of stepping through a list of parameters and requesting settings (and corrections) from the user.

- o When a parameter's value must come from a predefined set, forms can force the user to choose from correct settings only. The user avoids the frustration of typing in a value, only to have the computer reject it.

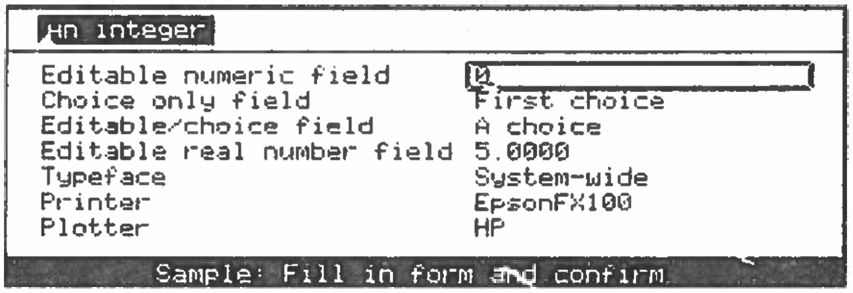
- o Forms present all the parameters at once. The user decides which parameters to change first.

- o Corrections are simple: the user returns to the item to be corrected, and changes the setting displayed. No time is wasted by advancing past correct values to find an incorrect one.

From a programmer's point of view, a form is a matrix of fields that has at least two columns:

Item column	A column of fields that identify the data being changed. They are not used in processing the form's settings. The user cannot move the outline into the item column or change the items there.
Setting column	A column of fields that the user can modify. The data values in the setting column indicate an initial setting, or some choice or typed input by the user. The form stores these modified settings in its own data structure. The settings can be kept stored in the form or they can be copied into other variables in your program.

Figure 8-4 shows a sample form.


An integer

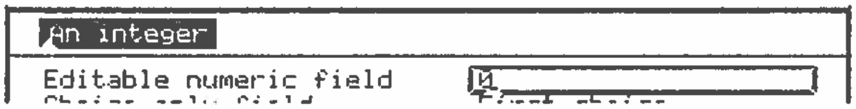
Editable numeric field	0
Choice only field	First choice
Editable/choice field	A choice
Editable real number field	5.0000
Typeface	System-wide
Printer	EpsonFX100
Plotter	HP

Sample: Fill in form and confirm

Figure 8-4: A Sample Form

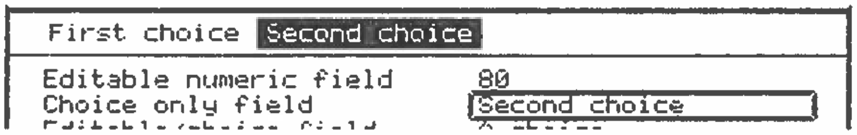
This form illustrates all the currently defined items for GRiD forms.

- o The first item is an editable integer-number with no choices. The user is expected to enter an integer value to set such things as document width, column width, and so on.


An integer

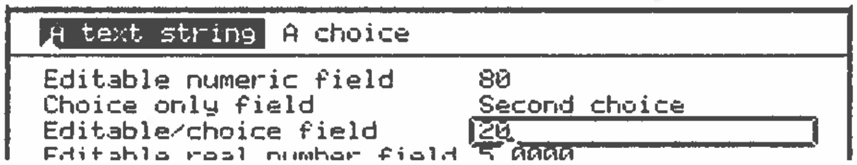
Editable numeric field 0

- o The second item is choice only. The user selects one of the displayed choices such as "Display headings" or "Don't display headings".


First choice Second choice

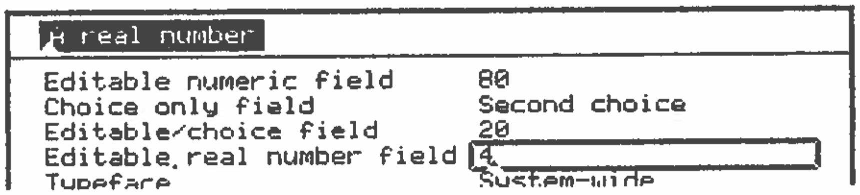
Editable numeric field	80
Choice only field	Second choice

- o The third item is an editable string with choices. The user can either choose one of the available choices or enter a string to specify a choice not offered. For example, the user might select the choice to display text using window width or can specify the line width to be used for display of text.


A text string A choice

Editable numeric field	80
Choice only field	Second choice
Editable/choice field	20

- o The fourth item is an editable real-number with no choices. The user is expected to enter an real number to set such things as precision. Form items that are editable real numbers will be displayed with four digits after the decimal point. If the user enters a number without decimal places, the system still treats it as a real number and supplies the four decimal digits.

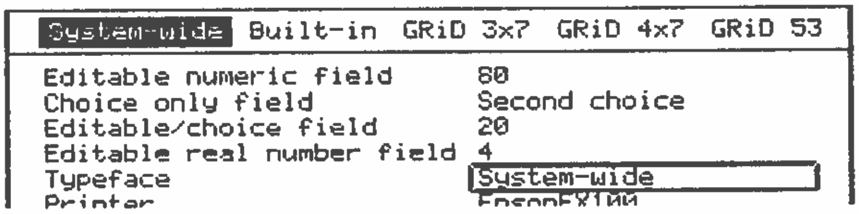


```
A real number

Editable numeric field      80
Choice only field           Second choice
Editable/choice field       20
Editable real number field  4
Typeface                    System-wide
```

If you want some other format, you can dislay an editable string and do the cnversion yourself.

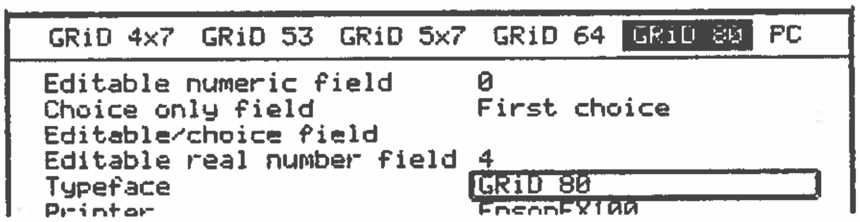
o The fifth, sixth, and seventh items are special kinds of choice-only items. They display the available font, printer, and plotter files in the choice band.



```
System-wide  Built-in  GRiD 3x7  GRiD 4x7  GRiD 53

Editable numeric field      80
Choice only field           Second choice
Editable/choice field       20
Editable real number field  4
Typeface                    System-wide
Printer                     EpsonFX100
```

These items automatically display all of the files with a Kind of "Font", "Printer", or "Plotter" that are in the Programs directories when the system was booted. The user can scroll to the desired choice displayed in the choice band. Notice that if all the choices do not fit in the choice band, the additional choices are automatically scrolled into view.



```
GRiD 4x7  GRiD 53  GRiD 5x7  GRiD 64  GRiD 80  PC

Editable numeric field      0
Choice only field           First choice
Editable/choice field
Editable real number field  4
Typeface                    GRiD 80
Printer                     EpsonFX100
```

Forms Data Structures and Types

Several Pascal data structures and types are used with data driven forms.

```
* TYPE SomeArrayOfBytes = ARRAY [1..1] OF CHAR;

       PointerToSomeBytes = ^SomeArrayOfBytes;
```

This pointer is used to keep track of different forms when you have a number of forms at once. It points to an array that contains the definitions of the labels and choices of the form and which also holds the form's data. (The labels and choices are initialized using the information defined in the PLM data structure.)

```
     * TYPE DataKindType =
                    (stringKind,
                     numberKind,
                  .  choiceOnlyKind,
                     fontKind,
                     realNumberKind,
                     printerKind,
                     plotterKind);
```

This enumerated type defines all of the different kinds of choice items you can use with forms.

```
     * TYPE DataFormModeType =
                        (normalDataForm,
                         initOnlyDataForm,
                         runOnlyDataForm);
```

You can specify three different kinds of forms: normal, initialize only, or run only. When you specify a "normal" data form, the form is immediately displayed by DataFormConfirmed and the results are stored in the form when it is confirmed. You would use the "initOnlyDataForm" and "runOnlyDataForm" types for more advanced programming techniques. If you specify "initOnlyDataForm", the form will not be displayed. A common use of this mode is to discover the rectangle that will be available for your form before you actually display the form. If you call DataFormConfirmed with "runOnlyDataFrom", the form is dispayed but it is not initialized. Therefore, if you use this mode, you must always call the routine after you have called it with the "initOnly" mode. When you call the routine in the "normal" mode, the form is both "initialized" and "run".

```
     * TYPE DataKindAliasType = RECORD
                           CASE INTEGER OF
                               1 : (string : StringPtr);
                               2 : (number : INTEGER);
                               3 : (realNumber : REAL);
                           END;
```

This record defines the three different kinds of data choices that can be used in forms.

```
     * TYPE DataRowType = RECORD
                    changed : BOOLEAN;
                    rowKind : DataKindType;
                    currentChoice : INTEGER;
                    tempChoice : INTEGER;
                    theData : DataKindAliasType;
                    tempData : DataKindAliasType;
                  END;
```

The DataRowType record is the structure where the choice data for each row (item) on the form is stored. The contents of this record are as follows:

changed	A Boolean that is set true if the choice for this row was changed from its previous setting. You can use this parameter as a "dirty" bit to determine if you need to examine the record.
rowKind	Specifies one of the seven kinds of possible choices.
currentChoice	An integer identifying the current choice for an item. On entry, determines which choice will be displayed as current choice for each item. On return, contains the choice that was confirmed. NOTE: You should always set the current choice even if the field consists only of an editable field (set currentChoice = 1). Otherwise, the "changed" Boolean will not be set correctly.
tempChoice	An internal variable used by the function itself. Should never be changed.
theData	The actual numeric or string data that is the current choice.
tempData	An internal variable used by the function itself. Should never be changed.

The last data type is the form itself.

```
     * TYPE DataFormType = RECORD
                  form : PointerToSomeBytes;
                  numItems : INTEGER;
                  labelsAndChoices : PointerToSomeBytes;
                  choiceLines : INTEGER;
                  rows : ARRAY [1..1] OF DataRowType;
                 END;
```

The DataFormType record defines the location of the form and its appearance, and contains the form's data. The contents of this record are as follows:

form	A pointer used internally by the DataFormConfirmed function. Initialized to NUL. Should not be changed by the user.
numItems	The number of items on the form.
labelsAndChoices	A pointer to the PLM data structure that contains the item labels and choices to be displayed on the form.
choiceLines	An integer that determines how many vertical lines will be used to display choices. A value of one displays all choices on a single line and scrolling is horizontal (as shown in the sample forms). Values greater than one display choices vertically within the number of "choiceLines" specified and scrolling is vertical.
rows	An array of DataRowType records holding the data that is in the form.

Figure 8-5 shows the PLM data structures required to display the form shown in Figure 8-4. This example structure can be used as a template for creating your own forms. The items that need to be modified when creating a new form are marked with numbers. These are described on the following page.

```
/*************** Sample form **************/          1

DCL sampleFormItemCount LIT '7'                            2
DCL sampleFormRowSize    LIT '98'  /* 14 times item count */    3

DCL sampleFormLabelsAndChoices (*) BYTE DATA
    ('#Editable numeric field^An integer^!',
     '?Choice only field^First choice^Second choice^!',
     '$Editable/choice field^A text string^A choice^!',        4
     '.Editable real number field^A real number^!',
     '&Typeface^!',
     '+Printer^!',
     '=Plotter^!');

DCL theSampleForm STRUCTURE
    (form PTR,
    numItems INTEGER,
    labelsAndChoices PTR,
    choiceLines INTEGER,
    rows (sampleFormRowSize) BYTE)

    PUBLIC DATA

    (nullPtr,                          /* form        */
    sampleFormItemCount,               /* numItems    */
    @sampleFormLabelsAndChoices,       /* items       */
    1);                                /* choiceLines */
         5
END;
```

Figure 8-5: PLM Data Structures for Sample Form

① The data structures for each form must have a unique name. Thus, if you were defining a "properties" form, you would change all occurences of the phrase "sample" to "properties" throughout this PLM data structure.

② This is the number of items in the form (seven, in our sample).

③ This number MUST be fourteen (14) times the number of items in the form. (It is used to allocate space for data.)

④ This is where you specify how the form will look and the characteristics of each item. The first character in each line determines what kind of item this is according to the following convention:

- # - an editable integer-number item. If you want an editable number item to initially display blank, you must initialize the number to "-MaxInt". (e.g. theSampleForm.row[1].theData.number := - MaxInt;) See the example that follows for an illustration of this method.

- ? - a choice-only item
- $ - an editable string and choice item
- . - an editable real-number item
- & - a font item. A special "Choice only" item of font files.
- + - a printer item. A special "Choice only" item of printer device files.
- = - a plotter item. A special "Choice only" item of plotter device files.

Following this initial character is the name of the item exactly as it is to appear in the form. The item name is terminated by the tilde (~) delimiter character. Then the choice band items (if any) are separated by tildes and the entire item definition is ended with a vertical bar (|).

The third item in the example form is an editable choice. It contains both an editable string and a choice band. In editable choice item definitions, the editable field is whichever one you list (define) first. The fields that follow are choice fields.

⑤ This number determines the orientation of the choice band. The number 1 specifies a horizontal choice band as shown in the sample form. Numbers greater than one specify vertical choice bands of that height. If the choices don't fit within the choice band, they are scrolled automatically (either horizontally or vertically) by the DataFormConfirmed function.

Data Driven Forms Routines

There are five Common Code routines used in conjunction with data driven forms:

DataFormConfirmed	This function displays the defined form. It is similar to DataMenuConfirmed. Refer to Chapter 12 for a complete description of DataFormConfirmed parameters.
UndoDataForm	This procedure deallocates all the tables and internal structures associated with a form. Its second parameter is a boolean indicating whether you want it to erase the area occupied by the form. You should ALWAYS call this after displaying a form.
FreeStringsInDataForm	This procedure frees all the strings in a data form. You should only call this after you have copied any strings from the form into permanent variables.

IMPORTANT: There is no reason that you have to store the values of a form in separate variables. You can leave them in the form. This latter method is often easier. If you do leave the values in the form's data structure then you should never call "FreeStringsInDataForm".

ExactCopyOfString	This function makes an exact copy of an indicated string and returns a stringPtr to the copy. (The "CopyOfString" function described in Chapter 5 is similar, but returns a string with .len set equal to .max. The exact copy will have .len set to the current .len of the indicated string.
StringOfFormItem	This function returns a stringPtr to the text actually displayed in a form item. It would be useful if you wanted to know the text of a choice selection as opposed to the number of the choice.

Data Driven Forms Example

Figure 8-6 shows a Pascal procedure that displays the form we have been looking at. It has been marked into four areas which will be explained next.

```
{------------------------------------------------------------------------}

PROCEDURE SampleForm;
VAR itemSelected: Integer;
    confirmed:    Boolean;
    rect:         Rectangle;
    str:          StringPtr;
BEGIN
  str := ConcatLits (OptionsMsg, FillInFormMsg);

  { Copy variables into Data driven forms structure. The variables can be kept
    permanently in this structure. }
  WITH theSampleForm DO
    BEGIN
      rows[1].theData.number     := theNumber;
      rows[1].currentChoice      := 1;
      rows[2].currentChoice      := curChoice2;
      rows[3].theData.string     := ExactCopyOfString (theString);    1
      rows[3].currentChoice      := curChoice3;
      rows[4].theData.realNumber := theRealNumber;
      rows[4].currentChoice      := 1;
      rows[5].currentChoice  := curFont;
      rows[6].currentChoice  := curPrinter;
      rows[7].currentChoice  := curPlotter;
    END;
  rect := windowRect;

  confirmed := DataFormConfirmed
                (theSampleForm,
                 normalDataForm,
                 msg,                 2
                 str,
                 rect,
                 cursor.keyProcess,
                 ch);

  IF confirmed THEN
    WITH theSampleForm DO
      BEGIN
        theNumber  := rows[1].theData.number;
        curChoice2 := rows[2].currentChoice;
        IF rows[3].currentChoice = 1 THEN
          theString  := ExactCopyOfString (rows[3].theData.string);
        curChoice3 := rows[3].currentChoice;                             3
        curChoice4 := rows[4].currentChoice
        curFont    := rows[5].currentChoice;
        curPrinter := rows[6].currentChoice;
        curPlotter := rows[7].currentChoice;
        FontSetNth (curFont, code);
      END;

  FreeStringsInDataForm (theSampleform);
  UndoDataForm (theSampleForm, TRUE);         4

END;
```

Figure 8-6: Pascal procedure for displaying form

① The values of the items in a form must be stored between each instance of the form. In this example these values are stored in separate data items. Before displaying the form the stored values are copied into the form's data structure. That is what is happening here.

The items on the right of the assignment statements are variables where the settings of the form are stored. The "rows" on the left side of the assignment are record items in "theSampleForm" record. (See the description of DataFormType earlier in this chapter).

The third item in the form is an editable string. We must make a copy of the string because the form will be disposed of later and we don't want to lose the original string.

② DataFormConfirmed displays the form. It is similar to DataMenuConfirmed. Refer to Chapter 12 for a complete description of DataFormConfirmed parameters.

③ If the form was confirmed then we want to copy the new values back into the permanent variables. This is done here. The new font is also loaded with the call to "FontSetNth".

④ "FreeStringsInDataForm" does just that. You should only call this after you have copied any strings from the form into permanent variables. IMPORTANT: There is no reason that you have to store the values of a form in separate variables. You can leave them in the form. This latter method is often easier. If you do leave the values in the form's data structure then you should never call "FreeStringsInDataForm".

"UndoDataForm" deallocates all the tables and internal structures associated with a form. Its second parameter is a boolean indicating whether you want it to erase the area occupied by the form. You should ALWAYS call this after displaying a form.

Appendix B contains complete source listings and a link file for a program that displays this form.

Example Notes:

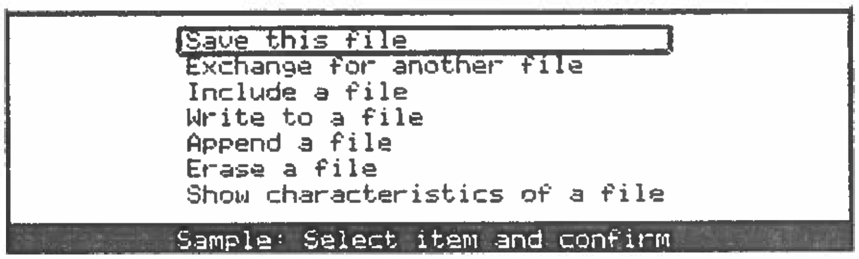
You can reduce the size of your program and make it more readable by placing a procedure shell around the calls to DataMenuConfirmed and DataFormConfirmed. Only three parameters would then be required to display a menu and two for a form. (the menu or form's data structure, a stringPtr for the message, and the itemSelected for a menu) The other parameters used with DataFormConfirmed and DataMenuConfirmed are usually global variables.

There are two ways of storing the data obtained from a form. The method shown in Figure 8-6 transfers the data into variables which are separate from the form. However, you could leave the data in the form. The latter case will probably create less code.

If you leave the data in the form then you shouldn't call "FreeStringsInDataForm". You should always call "UndoDataForm" because it frees all the internal tables associated with a form.

THE FILE FORM

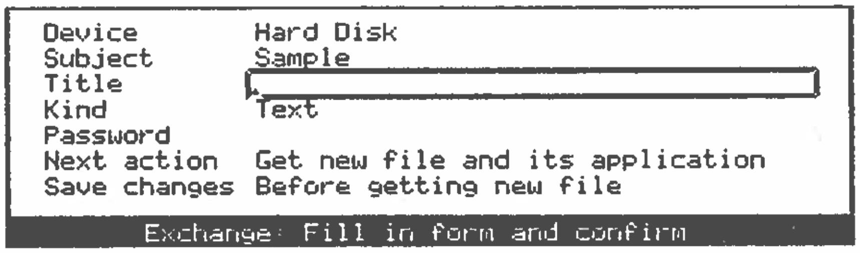
The File form is a special form that is used throughout GRiD applications to simplify implementation of the Transfer command (CODE-T). Figure 8-7 shows a typical Transfer menu.



```
Save this file
Exchange for another file
Include a file
Write to a file
Append a file
Erase a file
Show characteristics of a file
Sample: Select item and confirm
```

Figure 8-7. A Typical Transfer Menu

When the user selects and confirms an item from the Transfer menu, a File form similar to the one shown in Figure 8-8 can be displayed by calling the Common Code function FileFormConfirmed.

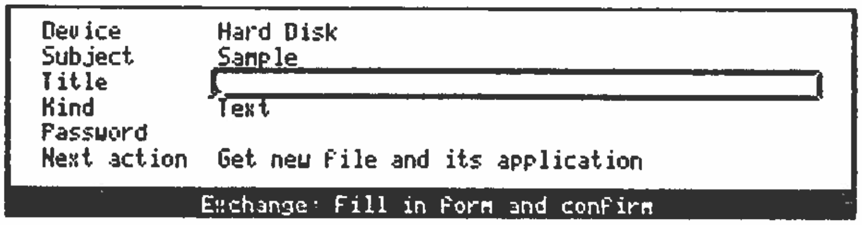


```
Device         Hard Disk
Subject        Sample
Title
Kind           Text
Password
Next action    Get new file and its application
Save changes   Before getting new file
Exchange: Fill in form and confirm
```

Figure 8-8. A Typical File Form

The FileFormConfirmed function is similar to the DataFormConfirmed function in several ways. Both functions display a form, handle movement of the selection outline when the user presses the arrow keys, and return with the selected items when the form is confirmed. With FileFormConfirmed, however, the items on the form are pre-defined instead of being defined by the programmer.

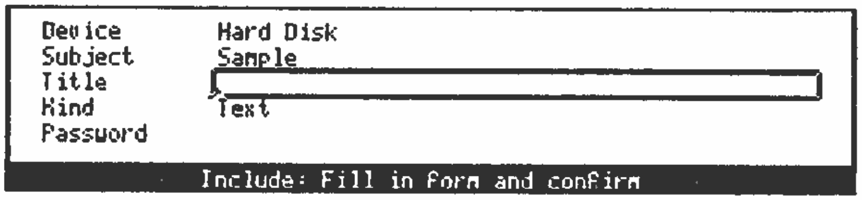
You can vary the content of the form slightly depending on the activity that is being initiated. For example, the last item on the form ("Save changes") need not be displayed unless you are leaving the current file and it has been altered since the last time its contents were saved.



Device	Hard Disk
Subject	Sample
Title	
Kind	Text
Password	
Next action	Get new file and its application

Exchange: Fill in form and confirm

Similarly, the second-to-last item ("Next action") need not be displayed for such operations as "Erase a file" or "Include a file" when you are not leaving the current file and "following" to another file and/or application.

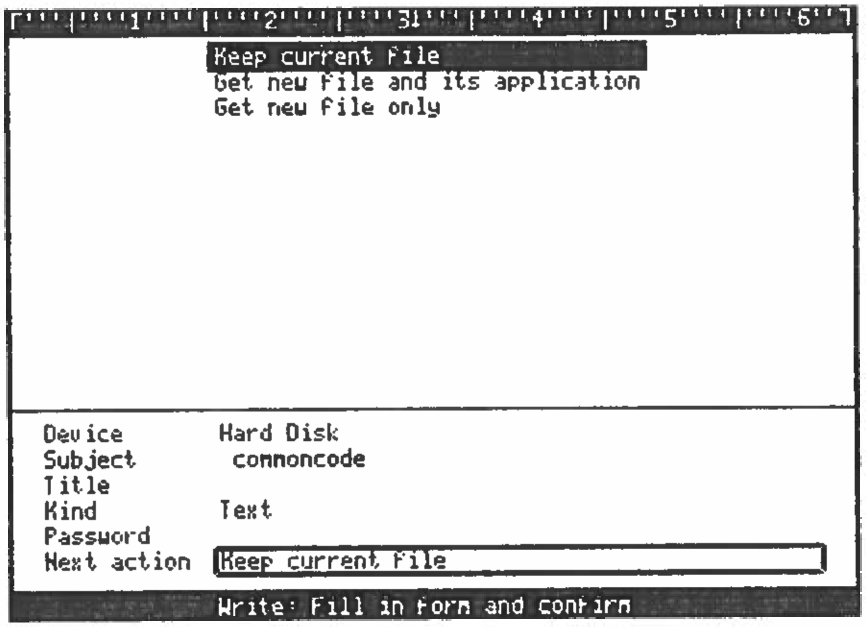


Device	Hard Disk
Subject	Sample
Title	
Kind	Text
Password	

Include: Fill in form and confirm

The FileFormConfirmed function lets you specify when these two items should or should not be displayed.

When the "Next action" item is displayed, two different combinations of choices can be displayed for the "Next action" item. If the Transfer operation being initiated is one where the user is given the choice of "following" to the selected file and immediately beginning work on that new file, then you would display the following three choices for "Next action":



Keep current file
Get new file and its application
Get new file only

Device	Hard Disk
Subject	commoncode
Title	
Kind	Text
Password	
Next action	Keep current file

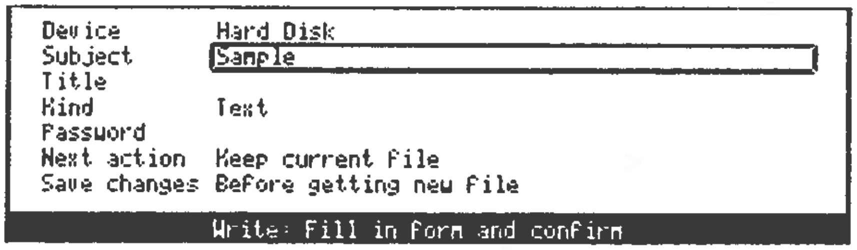
Write: Fill in form and confirm

These are the choices that GRiD applications display for "Write to a file" and "Append to a file". If the Transfer operation is one such as "Exchange for another file" that precludes keeping the current file, then the first choice ("Keep current file") for "Next action" can be suppressed.

When you call the FileFormConfirmed function you can set the initial choices that are to be displayed for each item. For example, for "Append to a file" operations, GRiD applications initially set the "Next action" item in the File form to the "Keep current file" choice since it is assumed that this will be the choice must frequently used. When the form is confirmed, the function returns the actual choices selected by the user.

Pathname Defaults

When the File form is displayed, you can specify which parts of the pathname (Device, Subject, Title, and Kind) are to be left blank and which parts should be defaulted (either to explicitly defined settings or to the current prefix). GRiD applications typically default the device and subject to the current prefix, leave the Title blank (don't default), and default the Kind to the same kind as the current file. For example, if the current prefix were 'Hard Disk'Sample', GRiDWrite would let the Device and Subject default to the prefix, and would default Kind to Text; Title would be left blank.



```
Device        Hard Disk
Subject       Sample
Title
Kind          Text
Password
Next action   Keep current file
Save changes  Before getting new file
              Write: Fill in form and confirm
```

Notice that the selection outline and cursor are positioned at the Subject item. The FileFormConfirmed function also lets you specify the pathname item on the form where the selection outline and cursor should originally be positioned when the form is displayed.

We will summarize the typical settings and values used by GRiD applications when displaying the File form after we have described the data types used in conjunction with FileFormConfirmed.

File Form Constants and Data Types

Seven constants are defined for the FileFormConfirmed function:

```
DevicePart   =1;
SubjctPart   =2;
TitlePart    =3;
KindPart     =4;
PasswordPart =5;
ExchangePart =6;
SavePart     =7;
```

These constants correspond to the seven items that can be displayed on the File form. The ExchangePart corresponds to the "Next action" item and the SavePart corresponds to the "Save changes" item.

Six data types are used with FileFormConfirmed.

```
*TYPE FFModeType =(FFGet, FFPut)
```

This enumerated type is used in combination with other conditions to determine what message (if any) the function should display below the File form. We will describe the messages and the circumstances when they are displayed after this discussion of data types. NOTE: The nomeclature of "Get" and "Put" is a carryover from an earlier, more primitive Common Code function and, while not very apropos in this context, they have been preserved for historical reasons.

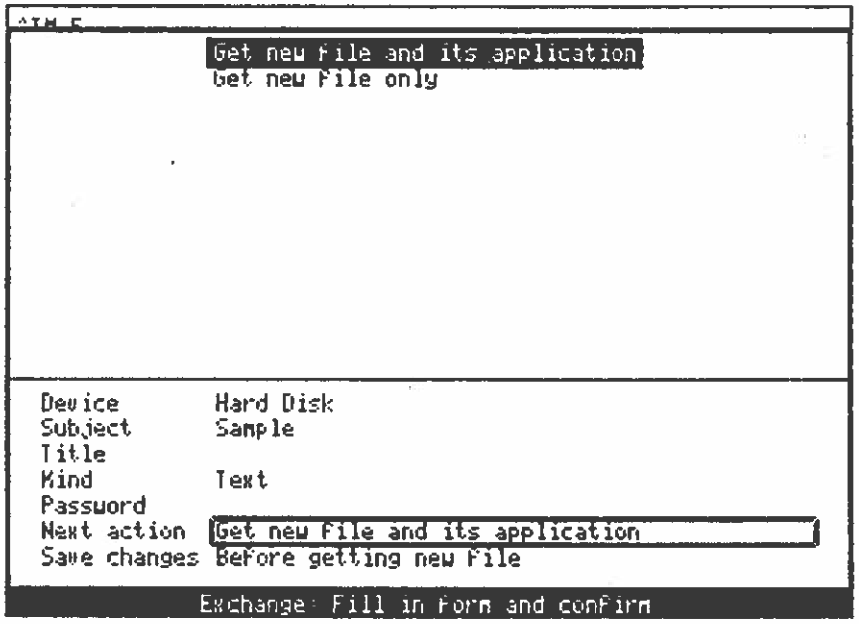
*TYPE FFExchangeMode = (FFNoExchangeOrSave, FFExchange, FFExchangeAndSave);

FFExchangeMode determines whether the form displays the "Next action" and "Save changes" items as follows:

FFNoExchangeOrSave	Display neither "Next action" nor "Save changes". Typically used for such activities as "Erase a file" and "Show file characteristics" when it is apparent that the user is remaining within the current file and application.
FFExchange	Display "Next action" only. Typically used for such activities as "Append to a file" or "Write to a file" when the user may be leaving the current file or application but there have been no changes made to the current file since the last time it was saved.
FFExchangeAndSave	Display "Next action" and "Save changes". Typically used for such activities as "Append to a file" or "Write to a file" when the user may be leaving the current file or application and there have been changes made to the current file since the last time it was saved.

*TYPE FFExchangeResult = (FFDontExchange, FFExchangeFiles, FFExchangeApplications); (PETER -- HOW ABOUT RENAMING THESE TO NextAction??)

FFExchangeResult specifies what the initial or default choice for the "Next action" item should be and indicates which one of the choices was selected and confirmed by the user.
If the initial value in FFExchangeResult is either FFExchangeFiles or FFExchangeApplication, then the "Keep current file" choice for "Next action" is not allowed and will not be displayed as a choice in the form. For example, if the activity being initiated is "Exchange for another file", GRiD applications initially set FFExchangeResult to FFExchangeApplications so that the form appears as follows:



```
                 Get new file and its application
                 Get new file only

  Device         Hard Disk
  Subject        Sample
  Title
  Kind           Text
  Password
  Next action    Get new file and its application
  Save changes   Before getting new file
              Exchange: Fill in form and confirm
```

The logic here is that if the user is obviously going to be exchanging either the current file or application then you do not want to present the meaningless choice of "Keep current file". In situations such as "Append to a file" or write to a file", however, where the user may want keep the current file, you must set FFExchangeResult to FFDontExchange in order to display the "Keep current file" choice.

```
     *TYPE FFSaveResult = (FFSaveFile, FFDontSaveFile); PETER - HOWABOUT
     FFSaveChanges??)
```

FFSaveResult specifies what the initial or default choice for the "Save changes" item should be and indicates which one of the choices was selected and confirmed by the user.

```
     *TYPE FFDefaultType = (FFDefaultThis, FFDefaultThisStartHere,
     FFDontDefaultThis, FFDontDefaultThisStartHere);
```

FFDefaultType specifies which of the File form pathname items (Device, Subject, Title, Kind) should initially be left blank and which should be supplied from the default values contained in the pathName parameter of the FileFormConfirmed function. If It can also specify the initial location of the selection outline and cursor on one of these four items. If you specify "DontDefaultIf", that part of the pathname is left blank. If you specify "Default" for Device or Subject and do not supply a default setting, the current prefix for these items is used. As mentioned before, GRiD applications use the prefix default for Device and Subject and supply a Kind default that is the same as that of the file currently being operated on. The Title is usually left blank (no default).

```
     *TYPE FFDefaultTypeRec = ARRAY[DevicePart..KindPart] OF FFDefaultType;
```

This array contains elements for each part of the pathname. Each element specifies whether to use that part of the pathname as a default in the form.

File Form Messages

The FileFormConfirmed function can generate six different messages to prompt the user when the File form is confirmed. These messages ask the user to verify that a new file is to be created or that an existing file is to be overwritten or notify the user that the form has been incorrectly filled out. These messages are displayed in the same area as the user-supplied message and are automatically removed and replaced with the user-supplied message when the user presses any key (except ESC or CODE-RETURN). The messages and the situations when they are generated are as follows:

"Confirm to overwrite old file"
- o fileMode parameter for FileFormConfirmed is not "old file"
- o AND FFMode = FFPut
- o AND the specified file already exists

"Confirm to create new file"
- o file or subject does not exist
- o AND FFMode = FFGet
- o AND fileMode parameter for FileFormConfirmed = "update file"
- o AND FFExchangeResult is not FFDontExchange

"All items except password must be filled in"
- o any item except password in the form is blank when the form is confirmed

"Wildcards not allowed here"
- o the wildcard character (CODE-W) is entered in an item

"Use GRiDManager to assign passwords"
- o file or subject does not exist
- o AND Password item is filled in

"DEL CTRL ~ ` | not allowed here
- o if any of these characters are entered in the form

Typical FileFormConfirmed Settings

Table 8-1 illustrates how a typical GRiD application sets the various parameters when calling FileFormConfirmed. You will notice that the majority of the parameter settings are the same regardless of which Transfer command activitiy is being initiated.

Table 8-1. Typical parameter settings for FileFormConfirmed

	FFMode	Default				attach	file	access	Exchange	FFExchangeResult
		Dev	Subj	Title	Kind	Mode	Mode		Mode	(Input)
Save	FFPut	Yes	Yes	No	Yes	True	Update	Update	NoEx/NoS	Don't Exchange
Exchange	FFGet	Yes	Yes	No	Yes	True	Update	Read	ExAndSave	Don't Exchange
Include	FFGet	Yes	Yes	No	Yes	True	Old	Read	NoEx/NoS	Don't Exchange
Write	FFPut	Yes	Yes	No	Yes	True	New	Update	ExAndSave	Don't Exchange
Append	FFGet	Yes	Yes	No	Yes	True	Update	Update	ExAndSave	Don't Exchange
Erase	FFGet	Yes	Yes	No	Yes	True	Old	Update	NoEx/NoS	Don't Exchange
Characteristics	FFGet	Yes	Yes	No	Yes	False	Old	Update	NoEx/NoS	Don't Exchange

Exchanging Applications

If the File form is confirmed and the user has specified a "Next action" of "Get new file and its application", the function FFExecuteCommand is used to get the new application and file. This function requires only the file name (as returned by the FileFormConfirmed function) as its input. It passes this file name to the system Executive which retrieves the appropriate application to work with that file. For example, if you pass a file name with a Kind of ~Text~ to FFExecuteCommand, the Executive looks for an application program with a Kind of ~Run Text~ and loads that program and the specified text file into memory.

CHAPTER 9: FONTS

The Common Code package provides several routines to simplify the use of multiple fonts (also know as typefaces) within applications. The font routines let you specify (either by name or index number) the font that is to be loaded into memory and used by an application as the current font. Routines are also available to let you obtain the name or index number of the current font and to determine how many fonts are currently available in the system.

The font-related routines are as follows:

Call	Purpose
FontCount	Returns the number of fonts available in the system.
FontSetNth	Sets the font specified by its index number (Nth) as the current font and loads it into memory.
FontSetName	Sets the font specified by name as the current font and loads it into memory.
FontNthName	Returns the name of the font specified by its index number (Nth).
FontGetN	Returns the index number (N) of the font specified by its name.

Refer to Chapter 12 for detailed descriptions of each of these calls.

During the boot procedure executed by the Compass, Common Code builds a list of such things as printers, plotters, and fonts that are available in the system. This information is made available to application programs so that it can be displayed in forms, such as an option form. (See Chapter 8 for a discussion of data-driven forms.) After such a form has been confirmed, the DataFormConfirmed procedure returns an integer that indicates which font has been chosen as the current font. This integer can then be used by FontSetNth to load that font into memory. For example, the following example function (FontChanged) compares two variables (tempCurFont and curFont) to see if the present font (curFont) is the same as the value returned from

DataMenuConfirmed (tempCurFont). If it is not, FontSetNth is used to load the new font into memory.

```
FUNCTION FontChanged : BOOLEAN;
VAR curFont, tempCurFont : INTEGER;
BEGIN
   FontChanged := FALSE;
   IF temCurFont <> curFont THEN
     BEGIN
        FontSetNth (tempCurFont, code);
        IF code <> okCode THEN
          DisplayError (code)
        ELSE
          BEGIN
            curFont := tempCurFont;
            FontChanged := TRUE;
          END;
      END;
END;
```

FontNthName can also use the index integer returned by DataFormConfirmed to obtain the name of the current font. An application might need the name of the current font so it cab write this name into the Common Properties record associated with the current data file. The following example procedure (WriteFontName) obtains the name of the current font using FontNthName and then writes that name to the common properties record of the data file. (See Chapter 4 for a discussion of Common Properties.)

```
PROCEDURE WriteFontName;
VAR curFont: INTEGER;
    font: StringPtr;
BEGIN
  font := FontNthName (curFont);
  IF font <> NIL THEN
    BEGIN
    WriteByte (commonPropsByte);
    WriteWord (font^.len + 1); (record length)
    WriteByte (fontPropsID);
    FOR i := 1 TO font^.len DO
      WriteByte (ORD(font^.chars[i]));
    FreeString (font);
    END;
END;
```

Similarly, when an application first reads in a data file, it can examine the Common Properties record of the file to obtain the name of the current font for that file. FontSetName can then be used to load that font into memory if it is not the font currently being used. An application can obtain the index number associated with a font name by using the FontGetN call. For example, the following example procedure (ParseFontName) reads the name of the current font for a data file from the common properties record, then sets that font as the current font using FontSetName, and then obtains the index number associated with that font using FontGetN.

```
PROCEDURE ParseFontName;
VAR ch: INTEGER;
    font: StringPtr;
    code: WORD;
BEGIN
  font := NewString (maxFontLength);
  font^.len := Min (maxFontLength, pRecord^.commonProps.length-1;
  FOR ch := 1 TO font^.len DO
    font^.chars[ch] := pRecord^.commonProps.textString[ch];
  FontSetName (font, code);
  IF code = okCode THEN
    BEGIN
      InitFont;  {Initialize VARs based on font size}
      curFont := FontGetN (font);
    END;
  FreeString (font);
END;
```

CHAPTER 10: FIELDS

This chapter describes the constants and data structures used with fields and the routines available to display and edit individual fields.

To the user, a field is a rectangular area on the screen that contains text or numeric values. It can be filled in by the user or the system.

To the programmer, a field is a data structure that contains a text string and formatting information for that text. The Common Code provides procedures for formatting the text and displaying the text on the screen.

The contents of a field can be left-aligned, right-aligned, or centered. Fields can contain more than one line of text. There are four types of fields, designed to protect data or enable the user to interact with it.

Editable	Editable fields allow the user to edit their values by typing, backspacing, or pressing arrow keys to move within the field.
Display-Only	The user cannot alter the values of these fields.
Choice	Choice fields can contain only settings from a predefined list. They are used only within forms, as described later.
Editable-Choice	Editable-choice fields can contain settings chosen from a predefined list, or the user can edit their values by typing, backspacing, or pressing arrow keys. They occur only in forms, as described later.

CONSTANTS

The following constant defines the character location of a field, in pixels:

```
CONST bottomMargin (=  1) distance from the lower
                          field boundary to the
                          one-pixel line beneath
                          the descenders
```

The other parameters that determine character location are handled by the Window-related calls such as charHeight, baseLine, lineHeight, and rightMargin. Refer to the GRiD-OS manual for illustrations of these parameters.

DATA STRUCTURES

*** TYPE Alignment** = (leftAlign, centerAlign, rightAlign);

Alignment controls whether the contents of a field are left-justified, right-justified, or centered with regard to the field boundaries. All lines in a multi-line field share the same alignment, though each line is aligned separately.

```
* TYPE FieldKind =
        PACKED RECORD
        editable, choice, editableChoice, numeric: Boolean;
        align: Alignment
        END;
```

FieldKind specifies whether a field can be edited by the user and whether the user can use choice arrows to obtain the field's value. It includes specifications for formatting numbers and aligning text.

WARNING: editableChoice is a special status bit kept by the field package. Do NOT modify it. You specify editable-choice fields by setting the variables editable and choice to True.

*** TYPE FieldDescriptor** =

```
RECORD
box: Rectangle;
text: StringPtr;
kind: FieldKind;
END;
```

The FieldDescriptor is the fundamental structure that describes a field and its data.

box — Defines the size of the field and its display location within the window.

text — A pointer to the GRiD string variable of the field's text.

kind — Determines whether the field is an editable field, a choice field, both, or neither, according to these combinations:

editable	choice	Resulting field
False	False	A display-only field. It can be displayed, but not edited. For example, an item on a menu.
False	True	A choice field. The user can choose its setting from among several predefined options. Only the displayed options can be chosen, because the field cannot be edited.
True	False	An editable field. It may be edited with the CODE, arrow, and BACK SPACE keys, including CODE-BACKSPACE (erase previous word).
True	True	An editable-choice field. Users can choose among several predefined field values that are displayed, or they can write and edit their own values in the field.

*** TYPE FieldPtr** = ^FieldDescriptor;

FieldPtr pointers will enable you to keep track of FieldDescriptors directly, and thus, fields and their contents.

*** TYPE FieldEditResult** = (ignored, processed, outOfField, bufferFull, fieldFull, escaped, ok);

Many routines return the FieldEditResult after performing their functions. An outOfField condition signifies that the user tried to move outside the current field. The FieldEditResult should be used to verify successful display of a character and to control movement between fields. See the FldEditField routine in Chapter 12 for a desciription of the interpretation of these results. below.

```
* TYPE CursorDescriptor =
                   RECORD
                   field: FieldPtr;
                   pos: Word;
                   place: Point;
                   on: Boolean;
                   keyProcess: Word;
                END;
```

The CursorDescriptor record stores the logical position of the cursor position where text may be inserted or deleted in a field.

field	Points to the field descriptor of the current field to be edited.
pos	The character offset within the field where the next character should be inserted.
place	The x,y window-relative pixel coordinate of the tip of the cursor icon.
on	Controls the cursor blinking and its on/off status. FldSetCursor initializes the cursor to the off state.
keyProcess	contains the process identification number (PID) of the cursor's process.

NOTE: Do not set the place, on, or keyProcess elements; the interface routines set them directly.

FIELD ROUTINES

Common Code provides routines to position and draw the cursor, edit and display fields, move fields and format multi-line fields. The available routines are summarized below. Refer to the alphabetically ordered function and procedure descriptions in Chapter 12 for complete details on each of the routines.

FldStartKeys	"Initializes" the cursor by starting a process to control the cursor. It puts the PID of the process into cursor.keyProcess.
FldSetCursor	Sets the cursor at the last character position in a specified field. The procedure does not alter the display.
FldSetPos	Sets the x-y pixel coordinate for the place element of the cursor. An application can set the cursor to any character position in the field. The display is unchanged.
FldDrawCursor	Makes the cursor visible and sets the cursor blink count.
FldEraseCursor	Erases the cursor from the display without affecting its position in the field or in the window coordinates, and sets the blink count.
FldReadKey	Waits for an interrupt signifying that a key has been pressed. If no keys are pressed for a certain time interval, the function blinks the cursor, and then resumes waiting for a key to be pressed. If a key is pressed, then the function returns the character.
FldEditField	This all-purpose routine inserts values into the field's character string, performs various key functions, and updates both the display and the cursor.
FldInsertInField	Inserts a character in the field at the cursor's current character position, and verifies its insertion by returning True or False. It does not redraw the display on the screen. Most applications should call FldEditField instead.
FldDrawField	Erases the given field and then redisplays the field's text string. Call it when you need to display initial values or redraw the updated value of a single field. This procedure accomodates multi-line fields with word-wrapping, but does not word wrap the last line of the field.
FldDrawFieldChars	Draws the field's text string without erasing the field first. This procedure accomodates multi-line fields with word-wrapping, but does not word wrap the last line of the field.
FldInvertChar	Performs an exclusive OR operation with a field's screen display to change a character position to inverse-video.
FldHilightField	Draws an outline box around the field.
FldDimHilightField	Draws a one-pixel dashed outline box around the field, leaving a one-pixel space from the field boundary.
FldFormatLine	Examines the text of a FieldDescriptor and determines where the text should appear on each line of a multi-line field.

CHAPTER 11: TABLES

Common Code includes routines to initialize, edit, and display tables of fields, which contain string values. Tables are collections of fields gathered together as a matrix. They are convenient for displaying large amounts of numerical data or for putting text into a tabular format.

Tables consist of editable fields, though the fields could be modified to become display-only in order to protect the field contents. Each field in a table is called a cell.

Tables are easier to use than individual fields. The Common Code has defined procedures for moving from cell to cell, and for controlling the cell that is to be edited. Automatic scrolling has been developed for tables, and several cell functions have been defined to operate upon selections of cells.

CONSTANTS

The constant, nowhere (= 65535), is a possible value of anchor (below) to show that a selection has not been anchored.

These constants have Boolean values:

editableField	True
nonEditableField	False
allocText	True
dontAllocText	False
disposeText	True
dontDisposeText	False

DATA STRUCTURES

Figure 11-1, "Cell Table Pointer Structure," shows how the following data structures are related to one another.

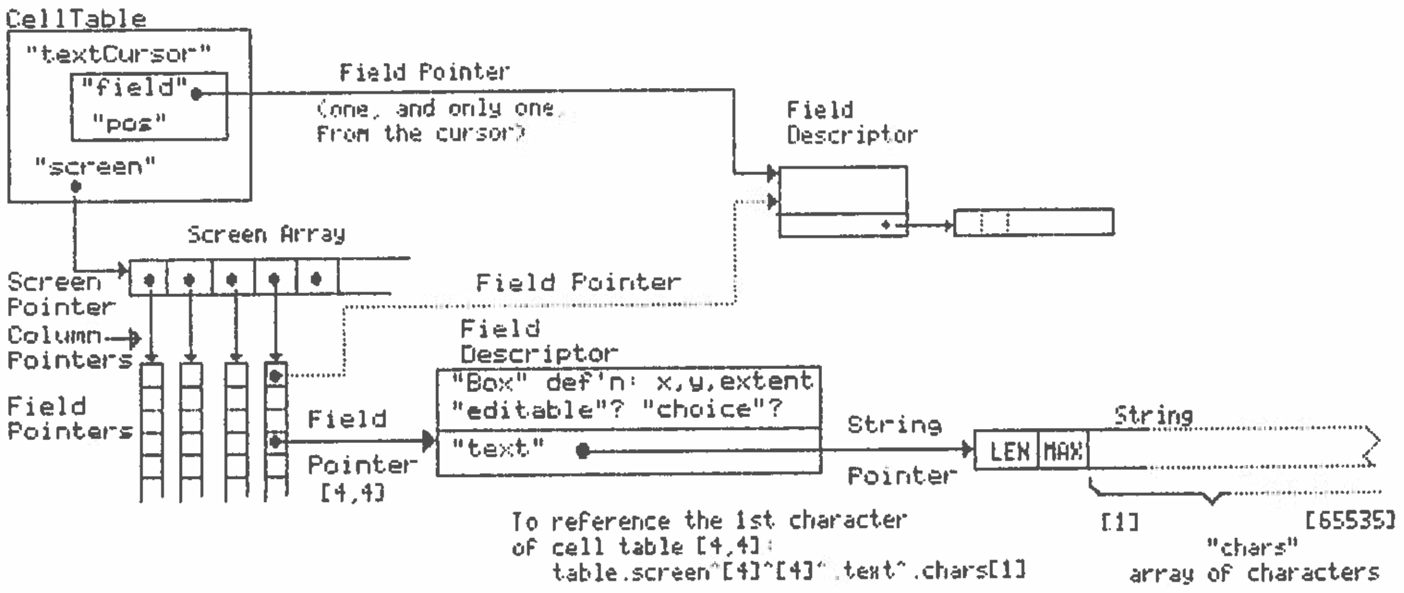


Figure 11-1. Cell Table Pointer Structure

*** TYPE ColArray** = ARRAY [1..2048] OF FieldPtr;

Each element of the ColArray points to a field pointer (and ultimately to a field and its value). Effectively, each column array is a column of a table, and elements of the column array refer to successive rows of the table. The fourth element of a ColArray would refer to the fourth row of the specified column in a table. The field pointers on the "successive rows" lead eventually to the values of the fields. The value 2048 is a dummy value -- the table package includes variable allocation routines as well.

*** TYPE ColPtr** = ^ColArray;

Points to a column array, and effectively to a column of the table of field values.

*** TYPE ScreenArray** = ARRAY [1..2048] OF ColPtr;

Each element of a ScreenArray points to a ColArray, so the third element of a screen array would refer to the third column of a table.

*** TYPE ScreenPtr** = ^ScreenArray;

A screen pointer refers to the screen array of a CellTable, i.e., to the table structure containing a matrix of fields.

*** TYPE CellId** = RECORD col,row: Integer END;

The CellId allows application programs to reference fields in a two-dimensional table. The CellIds in a CellTable structure are independent of window scrolling.

```
* TYPE SelectionRangeKind =
         (cellRange, textRange, rowRange, colRange);
```

SelectionRangeKind specifies whether the selection comprises the text within a cell, a group of cells, or several rows or columns of cells.

```
* TYPE TableSelection = RECORD
                        cell: CellId;
                        pos: Word;
                        rangeKind: SelectionRangeKind;
                        END;
```

The TableSelection structure contains an anchor selection, along with the kind of selection range. If rangeKind = textRange, then TableSelection.pos contains the anchoring character position within the text of one cell. If rangeKind is any of the other ranges, then TableSelection.cell contains the CellId of the anchoring cell.

```
* TYPE CellTable =
      RECORD
      colPerScreen,rowPerScreen: Integer;
      screen: ScreenPtr;
      movingCell, currentCell, scrollCell : CellId;
      visibleRect: Rectangle;
      constraint, visible:
         RECORD
         top, left, bottom, right: Integer;
         END;
      textCursor: Cursor;
      editMode: (normal, command);
      commandChar: Char;
      rangeKind: SelectionRangeKind;
      whichParameter: 0..10;
      hilightKind: (noHilight, dim, bright, splitHilight);
      gap: Point;
      anchor: TableSelection;
      sourceAnchor, sourceCurrent: TableSelection;
      commands: Keys;
      hilightOn, verticalGrid, horizontalGrid,
      frame, bottomFrame, rightFrame: Boolean;
      headingRows: Integer;
      headingCols: Integer;
      END;
```

The CellTable contains the information needed to keep track of a table of fields (cells), their cursor, and other display parameters such as highlighting, anchoring, and editing/command modes. CellId's begin at 1,1 unlike the screen and window pixel coordinates. Figure 11-21 shows the layout of fields within a cell table.

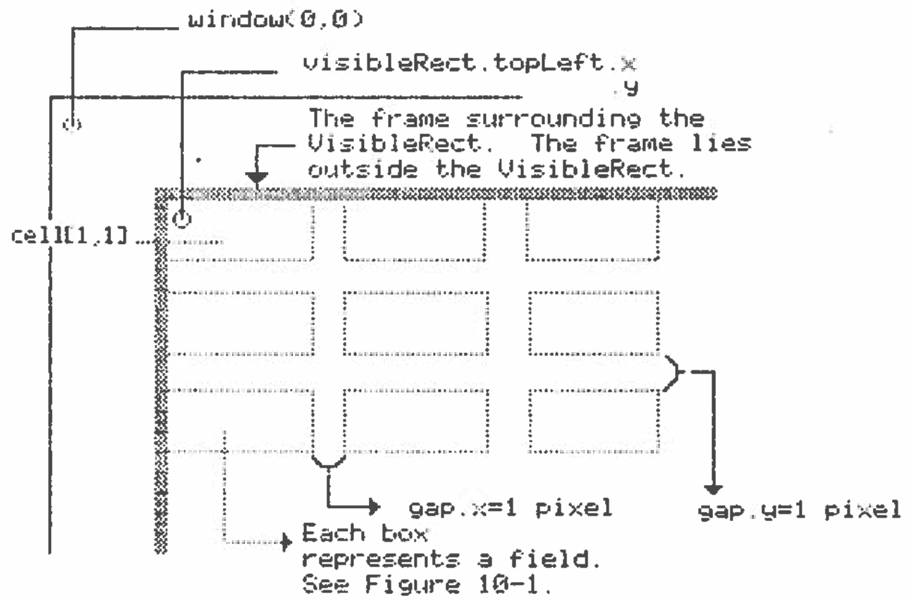

window(0,0)
visibleRect.topLeft.x
.y
The frame surrounding the VisibleRect. The frame lies outside the VisibleRect.
cell[1,1]
gap.x=1 pixel
gap.y=1 pixel
Each box represents a field. See Figure 10-1.

Figure 11-2. Fields in a Cell Table

colPerScreen rowPerScreen	Designate the number of columns and rows of fields that exist in a particular CellTable. These can be specified when initializing a CellTable. After they have been initialized, they can be changed using TblAddCol and TblAddRow. Only change them using these calls, however, because the dispose routines use them to determine how much memory to deallocate.
screen	A pointer to a screen array (and from there to a column array, then to a field designator, and finally to a field's text string).
movingCell	Useful in applications which require a second inverted outline to move while the ordinary outline remains in place. In selections, the anchor.cell outline remains at the anchor position, while the currentCell and movingCell move to make the cell selection. In the worksheet editor, the movingCell outline can be manipulated to generate cell references while the currentCell outline stays in place.
currentCell	The CellId of the cell which currently contains the cursor. This is a read only value. If you want to change the location of the cursor, use TblSetCurrentCell.
scrollCell	The absolute CellId of the top left cell in the CellTable displayed on the screen. In easy case scrolling, it is always (1,1). In difficult case scrolling, it is the visible top left cell, not the logical top left (the CellId of the logical top left cell is (1,1)). For example, after

	scrolling, scrollCell could equal (5,7) -- meaning that the absolute address of the top left cell on the display screen is (5,7).
visibleRect	A clipping rectangle defined in pixels based on the window size. It displays all or part of the cells defined to be "visible" (see below). The visibleRect is initialized to the rectangle between topLeftMargin (an input to TblInitTable) and windowExtent.
constraint	Defines the cells that the cursor can move into. It must be a rectangular area, defined in CellId coordinates.
visible	Specifies a rectangular area of all cells that are fully or partially visible on the display screen. It is defined in terms of CellIds, not pixel coordinates. The visibleRect clipping rectangle can clip these visible cells.
textCursor	A cursor associated with this CellTable's field values. It consists of the cursor defined in the field package.
editMode	Indicates whether a normal (text input) mode or a command mode is in effect.
commandChar	A character used to store command mode characters.
rangeKind	Indicates which type of range has been selected, whether text within a cell or a group, row, or column of cells.
whichParameter	Indicates the parameter the command requires the user to input next. If a command needs its third parameter, then whichParameter = 3.
hilightKind	Registers the type of highlighting, either none, bright, dim, or splitHilight. The splitHilight kind allows two cell outlines on the display at once, a bright outline and a dim one. Both should appear when the currentCell breaks away from the movingCell.
gap	The x and y pixel gap between fields in the table.
anchor	Consists of a rangeKind, a CellId, and a character position within a cell, so that anchoring may be based on groups of entire cells or on character positions within a cell.
sourceAnchor	In commands that require two selection areas, it contains the top left position of the first selection.
sourceCurrent	In commands that require two selection areas, it contains the bottom right position of the first selection.
commands	Set of Keys. The keynames in this set define the selection command keys that operate with a particular cell table. These correspond only to the commands that require a highlighted selection of text or cells.
hilightOn	Specifies whether the highlighting appears on the display. Highlighting includes the cursor, the cell outline, the split outline, the "additional" outlines (if any), and the inverse video highlighting for selections.
verticalGrid	Controls whether vertical lines are drawn to separate the fields.
horizontalGrid	Determines whether horizontal lines are drawn to separate the fields.
frame	Determines whether a one-pixel frame is drawn outside the visibleRect.
bottomFrame	Determines whether a one-pixel frame is drawn beneath the

	last row of cells. It appears only when the last row of cells is displayed on the screen.
rightFrame	Determines whether a one-pixel frame is drawn beside the last (rightmost) column of cells. It appears only when the last column of cells is displayed on the screen.
headingRows	An integer representing the number of rows, starting from the top of the table (not just the visible part of it) that can be displayed but into which the cursor and cell outline cannot move.
headingCols	An integer representing the number of columns, starting from the left of the table, that can be displayed but into which the cursor and cell outline cannot move. (Forms have headingCols = 1.)

```
* TYPE TableCommandResult = (tableCommandProcessed,
                             ourobouros);
```

The result specifies whether the command was successfully processed, or whether it tried unsuccessfully to write over its own operands in the table (the "ourobouros" result). (The ourobouros is a mythic symbol of infinity, a snake eating its own tail.)

TABLE ROUTINES

The paragraphs that follow summarize the routines available to work with tables. Complete descriptions of each of the routines are provided in the alphabetically ordered Chapter 12.

Allocating and Disposing Tables

The routines that follow allocate, initialize, and dispose of tables.

TblInitTable	Initializes and formats the CellTable it receives as an argument. Every cell within the initialized CellTable will be identical, with a uniform number of characters and lines in a field.
TblDisposeTable	Disposes of the specified cell table pointers and descriptors. You can specify whether it should dispose of the values of the fields in the table or retain them.
TblAddCol	Appends another column to the CellTable matrix. The appended columns may have a different field width (characters per line) from the columns of the table being appended.
TblNewScreen	Returns a pointer value to a screen array with the given number of columns, colCount.
TblDisposeScreen	Deallocates screen arrays that have been created by TblNewScreen. The number of columns to be disposed of must equal the number of columns that were allocated when the screen array was created.
TblDisposeCol	Deallocates column arrays that have been created by TblNewCol. The number of rows to be disposed of must equal the number that were allocated when the column array was created.

Editing Tables

The following routines are used to change the contents of a table and to allow movement of the field outline around the table.

TblEditTable	This all-purpose table routine inserts characters at the current field location and cursor position, performs various key functions, and redraws both the display and the cursor.
TblChangeFields	Given a table and a movement character, moves the field outline from cell to cell. (It moves the cursor, too, if the cursor actually was in the currentCell.)

Specifying Cells

The routines that follow simplify working with a cell or group of cells within a table.

TblSetCurrentCell	Sets CellTable.currentCell to the given column and row of the cellTable. This routine will change the position of the cursor and the highlighted cell. The display will change only when another procedure redraws the table, however.
TblFieldOfCellId	Converts a CellId into a FieldPtr reference, which makes table values easier to refer to and to change. It is useful when working with cell variables of type CellId, such as currentCell.
TblFindBounds	Calculates which cells lie within a rectangle that has been defined in the pixel coordinates of the display window. Given an area on the screen, it allows you to update only a portion of the table.
TblFieldOfColRow	Given a column and and a row of a cell table, it returns the pointer to the field.
TblEqualCells	Returns True if the given CellId's are equal.
TblCellOnScreen	Returns whether the cell is within CellTable.visible, i.e., whether it is to be displayed.

Drawing a Table

The routines that follow are used to actually display tables.

Application Note: To draw a newly initialized table, your application must call TblDrawTable (to draw the fields) and TblHilightTable (to draw the cursor and to outline the cursor's cell). Later, TblEditTable and TblChangeFields will update and redisplay the table when the application modifies it; they redraw the table, the cursor, the cell outline, and the range selection (if any).

TblDrawTable	Clears all fields from the screen and redisplays them with their current values, by calling FldDrawField for every field in the table. It overwrites the entire area defined by the visibleRect.
TblDrawGrid	Draws a frame around the visibleRect and grid lines between the fields of a table, if table.frame, table.verticalGrid, and table.horizontalGrid are True. If a variable is False, TblDrawGrid does not draw the graphics associated with it. It does not redraw the fields of the table. The frame and grid lines are one pixel wide.
TblUpdateRect	Updates the cells that lie within a rectangle defining a portion of the display window. Given an area on the screen, it allows you to update only a portion of the table. It is useful for redrawing the table after a message, a menu, or a form has been displayed.
TblSetVisible	Adjusts table.visible and table.constraint so that they lie within the table.visibleRect clipping rectangle. The procedure adjusts the top, bottom, left, and right of table.constraint as

well. Constraint is based upon the number of entire cells that can fit within visibleRect. TblSetVisible does not allow constraint to contain cells that appear only partially on the screen. This restriction ensures that the cursor and cell outline can move into entire cells only.

INVERTING A TABLE

The following routines are used to invert (display in reverse video) specified parts of a table.

TblInvertRange	Inverts the current selection range, either a range of cells or a range of text within a single field. A range is a rectangular span of cells that has been selected by the user. Nothing will happen if the procedure is called and no range has been selected.
TblInvertSpan	Given a span of cells, it inverts the displayed cell of each field within the span. Spans are rectangular areas defined by column and row parameters. TblInvertSpan will invert the additional selections when a user scrolls during a selection.

Highlighting a Table or Cell

The following routines are used to invert and highlight specified parts of a table.

NOTE: Call TblHilightTable and TblUnhighlightTable whenever moving from a table to a menu and back or when moving back and forth between windows.

TblHilightTable	Draws the cursor in the currentCell, inverts any selected range of cells, and highlights all cells in the table that require highlighting.
TblUnhilightTable	Given a cell table, it erases the cursor, uninverts any range of selected cells, and removes the highlighting from any highlighted cells. The cursor is erased graphically only, so you must reset it elsewhere with TblSetCursor.
TblHilightCell	Given a CellTable and a CellId, it draws the appropriate outline around a cell, based on the value of hilightKind.
TblDimHilightCell	Draws a dashed outline around a cell.

Scrolling

The table routines supports two types of scrolling, the easy case and the difficult case. The next version of this manual will provide up to date examples of how to program these two cases.

TblScroll	The easy case: It scrolls the view of the table in the direction indicated by ch (left arrow, right arrow, up

	arrow, or down arrow), and updates the display. It also updates visible and constraint so that they match the displayed area.
TblGetSelectedCellIds	The difficult case: locates the movingCell and anchor CellIds, rearranges them in ascending order, adjusts them from relative "unscrolled" CellIds to absolute "scrolled" CellIds, and returns them as "first" and "last" absolute (logical) coordinates.
TblScrollAdjustCellId	The difficult case: transforms an "unscrolled" CellId that is relative to the display screen into an absolute "scrolled" CellId.

Coordinating Text and Cell Selections

The following routines are used in conjuntion with commands such as Move and Erase to simplify highlighting and confirmation of selected areas of a table.

TblStartSelection	Puts the table into command mode and sets table.commandChar to ch. It works the same as if the ch character had been included in the set of keys (in table.commands) passed to TblInitTable, and then TblEditTable was called later with that character. In both cases, highlighting of selections is enabled.
TblConfirmSelection	Used to save the source selection range for commands that require two selection ranges, such as Move and Duplicate. The table code will leave the source selection highlighted while the user selects a destination range.
TblEscapeMode	Puts the table into the normal (non-command) state, un-inverts any cell or text selection ranges, but leaves the cursor and the highlighted cell on.

CHAPTER 12. COMMON CODE PROCEDURES AND FUNCTIONS

This chapter lists all of the procedures and functions provided by Common Code in alphabetical order. For discussions of concepts and interactions of these calls, refer to the appropriate chapter earlier in this manual. This chapter simply lists the calls in alphabetical order and provides a comprehensive description of each call for maximum ease-of-use for reference purposes.

AppendAnyChar

```
PROCEDURE AppendAnyChar(VAR str: StringPtr; ch: Char);
```

Purpose and Operation

This procedure appends a single character to the tail of the string:

```
str := str + ch
```

If the string would exceed its max length when the character was appended, then AppendAnyChar allocates a new string with a greater max length, copies str into it, and appends ch to it. It disposes of the original str and sets str to the newly created string.

AppendChar

```
PROCEDURE AppendChar(dest: StringPtr; ch: Char);
```

Purpose and Operation

This procedure appends a single character to the tail of the dest string:

```
dest := dest + ch
```

However, if the dest string would exceed its max length when the character was appended, then AppendChar will not append it, and it will not return an error message either.

AppendString

```
PROCEDURE AppendString(dest, source : StringPtr);
```

Purpose and Operation

AppendString concatenates the source string to the tail of the dest string, like so:

```
dest := dest + source
```

The source string remains unchanged. If the append operation would make dest too long (overflowing its max length), then the source string will be truncated to fit the available space.

AppendString

```
PROCEDURE AppendString(dest, source : StringPtr);
```

Purpose and Operation

AppendString concatenates the source string to the tail of the dest string, like so:

```
dest := dest + source
```

The source string remains unchanged. If the append operation would make dest too long (overflowing its max length), then the source string will be truncated to fit the available space.

AuthorOfThisFile

```
FUNCTION AuthorOfThisFile(conn: Word;
                          VAR authorProductCode: Word;
                          VAR versionOfThisFile: Byte;
                          VAR error: Integer ): AuthorType;
```

Purpose and Operation

Given the connection number of a file, this function, on return, provides the product code and version number from the file's authorID record. The function also returns an indication of whether the file is in the new (3.0) format, old (2.0) format, or in neither format.

Parameters

authorProductCode	The product code identifying the application that created this data file.
versionOfThisFile	A byte (from the authorID record) specifying compatibility level of the data file.

CmdErase

```
PROCEDURE CmdErase (conn : Word;
                    msg : MessagePtr;
               VAR  error : Word);
```

Purpose and Operation

This procedure displays the prompt "Confirm to erase file". If the user confirms the prompt, the routine erases the file specified by the connection number (conn) while displaying the message "Erasing file". When the file has been erased, the message "File erased" is displayed. If any key other than CODE-RETURN is pressed after the "Confirm to erase file" prompt, the message "No files erased" is displayed and no file is erased.

Parameters

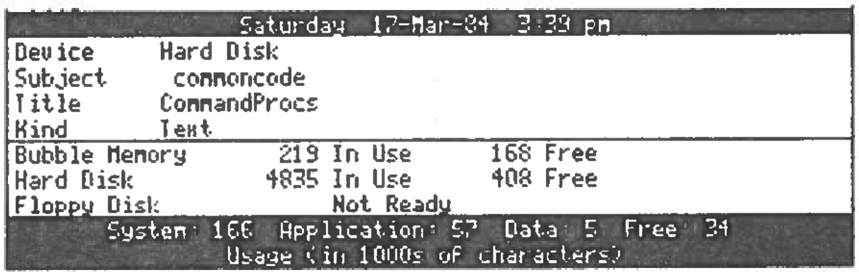
conn	The connection number of the file to be erased.
msg	A pointer to your message area.

CmdMediaUsage

```
PROCEDURE CmdMediaUsage (pathName : StringPtr;
                         initialUsage : LongInt;
                         msg : messagePtr;
                     VAR refresh : Rectangle;
                     VAR error : Word);
```

Purpose and Operation

This procedure implements the Usase (CODE-U) command that is supported in all GRiD applications. It displays the current usage of system memory and storage devices, and also the name of the current data file in the format shown below:



```
               Saturday  17-Mar-84  3:39 pm
Device     Hard Disk
Subject     commoncode
Title      CommandProcs
Kind       Text
Bubble Memory        219 In Use      168 Free
Hard Disk           4835 In Use      408 Free
Floppy Disk              Not Ready
        System: 166  Application: 57  Data: 5  Free: 34
               Usage (in 1000s of characters)
```

Parameters

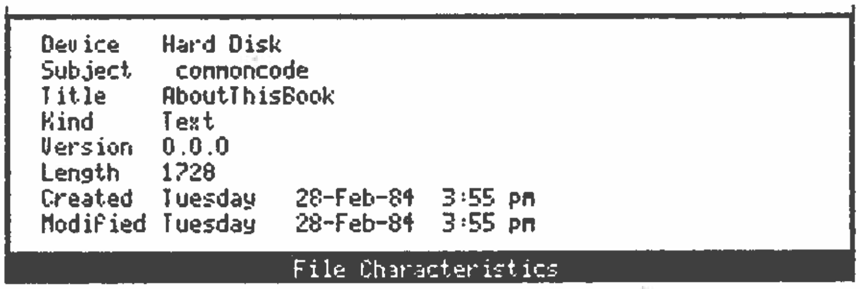
pathname	The pathname of the current data file.
initialUsage	The Long Integer returned by MsgInitialUsage which indicates the initial RAM usage before the data file was loaded into memory. (MsgInitialUsage should be called when you are initializing your application.)
msg	A pointer to the your message area.
refresh	A Rectangle that, on return, indicates what portion of the screen needs to be updated by the application.

CmdProperties

```
PROCEDURE CmdProperties (pathName : StringPtr;
                         msg : MessagePtr;
                     VAR refresh : Rectangle;
                  .  VAR error : Word);
```

Purpose and Operation

This procedure displays the properties (characteristics) of the file specified by the pathname parameter. It should be called when the user has selected "Show file characteristics" from the Transfer menu and has specified the desired file by confirming the resultant File form. An example of the resultant display is shown below:



```
Device   Hard Disk
Subject   conmoncode
Title    AboutThisBook
Kind     Text
Version  0.0.0
Length   1728
Created  Tuesday   28-Feb-84  3:55 pm
Modified Tuesday   28-Feb-84  3:55 pm
```

Parameters

pathname	A pointer to the pathname of the file whose characteristics are to be displayed.
msg	A pointer to the your message area.
refresh	A Rectangle that, on return, indicates what portion of the screen needs to be updated by the application.

CompareBytes

```
FUNCTION CompareBytes(VAR source1, source2: Bytes;
                      count: Word;
                      VAR index: Word): Boolean;
```

Purpose and Operation

Compares one memory area with another one to see whether they match. They must be the same length.

Parameters

source1	A pointer to the first location of the data.
source2	A pointer to the second location of the data.
count	The number of bytes to compare. This routine compares bytes in source1 to an equal number of bytes in source2.
index	An index into the source's memory area, it indicates the first position in the first memory area where the two memory areas did NOT match. If the two memory areas are identical, then index = FFFF. This value is returned by reference. NOTE: the index starts at 0 (not 1) in order to be compatible with PL/M. Hence, index = 0 represents the first position in the memory area.

Returns

CompareBytes returns a Boolean indicating whether or not the two memory areas matched. If CompareBytes returns True, then the two areas matched exactly. If CompareBytes returns False, then the index variable contains the first character position where the two areas did not match (all character positions before it did match).

CompareStrings

```
FUNCTION CompareStrings(str1,str2: StringPtr): Comparison;
```

Purpose and Operation

The function compares the ASCII values of two strings character by character, from left to right. Thus the greater string will be the one containing the first character with a higher ASCII value. If two strings match up exactly except that one string has additional characters, then the string with the extra characters will be the greater one.

NOTE: This routine does NOT discriminate between uppercase and lowercase. The string 'a red cat ran' is greater than 'A Red Cat'.

ConcatLits

```
FUNCTION ConcatLits(VAR lit1, lit2: Bytes): StringPtr;
```

Purpose and Operation

The function will create a new string with the literals lit1 and lit2 concatenated together. It allows you to concatenate string constants. The len and max of the created string is the sum of the lengths of the two literals.

Example

```
CONST x = 'In progress ';

MsgString := ConcatLits('Find: ', x);

MsgString^chars now equals 'Find: In progress'
```

ConcatStrings

```
FUNCTION ConcatStrings(str1,str2:StringPtr): StringPtr;
```

Purpose and Operation

The function will create a new string containing str1 and str2 concatenated together. The len and max of the created string is the sum of the current lengths of the two strings (not the sum of their max lengths). It disposes of str1 and str2 after creating the new string.

CopyOfString

```
FUNCTION CopyOfString(str: StringPtr): StringPtr;
```

Purpose and Operation

This function creates a new string and copies the value of str to it. The len and max of the new string are both equal to the len of str.

NOTE: some Common Code routines deallocate the strings they receive as arguments. If you don't want to have certain strings deallocated by them, make a copy of the string with this procedure.

CopyString

```
PROCEDURE CopyString(source, dest: StringPtr);
```

Purpose and Operation

This procedure copies the value of the source string to the dest string. Both source and dest must be allocated already. If the source string is longer than the dest string, then any extra characters will be truncated.

DataFormConfirmed

```
FUNCTION DataFormConfirmed (VAR dataForm : DataFormType;
                                dataFormMode : DataFormModeType;
                                msgStatus : MessagePtr;
                                msg : StringPtr;
                            VAR rect : Rectangle;
                                keyProcess : WORD;
                            VAR ch : CHAR) : BOOLEAN;
```

Purpose and Operation

This function displays the specified form and, when confirmed, returns with the choices the user selected for each item on the form. The function handles display of the form and responds to arrow keys to move the selection outline from choice to choice. The appearance of the form, definition of item types, and the choices that will be displayed must be defined in a PLM data structure. Refer to Chapter 8 for a description of the PLM data structure and the data types used with the form.

Parameters

dataForm	The form's PLM data structure.
dataFormMode	This is an enumerated type: "normalDataForm" initializes and displays the form, "initOnlyDataForm" just initializes the form, "runOnlyDataForm" displays an initialized form.
msgStatus	The MessagePtr you use for all activity with messages. If any messages are currently visible then the form will be displayed above them.
msg	A pointer to the string to be displayed as the prompt for the form. If some messages are already displayed this one will be stacked upon the others. Passing "NIL" for this parameter causes no additional messages to be displayed. This stringPtr is automatically deallocated. Note: This string is actually displayed as a prompt. You must call MsgClearPrompt to remove it.
rect	This rectangle defines what part of your window the form will be displayed in. Common Code will update this rectangle to reflect what part of the window was actually used. It will not include the area used by any messages.
keyProcess	This is the cursor process ID. Common Code requires this for menus, forms, and tables. It must be initialized with "FieldStartKeys" prior to use.
ch	This CHAR returns the key that was pressed last. You should look at this value only if the form was not confirmed.

DataMenuConfirmed

```
FUNCTION DataMenuConfirmed (dataMenu : DataMenuType;
                            msgStatus : MessagePtr;
                            msg : StringPtr;
                        VAR rect : Rectangle;
                            keyProcess : WORD;
                        VAR selection : INTEGER;
                        VAR ch : CHAR) : BOOLEAN;
```

Parameters

dataMenu	The menu (the name of the second data item defined in the PLM module).
msgStatus	The MessagePtr you use for all activity with messages. If any messages are currently visible then the menu will be displayed above them.
msg	A pointer to the string to be displayed as the prompt for the menu. If some messages are already displayed this one will be stacked upon the others. Passing "NIL" for this parameter causes no additional messages to be displayed. This stringPtr is automatically deallocated. Note: This string is actually displayed as a prompt. You must call MsgClearPrompt to remove it.
rect	This rectangle defines what part of your window the menu will be displayed in. Common Code will update this rectangle to reflect what part of the window was actually used. It will not include the area used by any messages.
keyProcess	This is the cursor process ID. Common Code requires this for menus, forms, and tables. It must be initialized with "FieldStartKeys" prior to use.
selection	This Integer returns which item was selected on the menu.
ch	This CHAR returns the key that was pressed last. You should look at this value only if the menu was not confirmed.

DeleteBytes

```
PROCEDURE DeleteBytes (VAR source, dest: Bytes;
                           sourceLen, pos, byteCount: Word);
```

Purpose and Operation

This procedure deletes a given number of bytes from an area of memory; the remaining bytes are moved together to close up the resulting gap. This procedure is useful for removing elements from arrays, structures, strings, etc.

Source and dest can refer to the same area of memory or to different areas.

Parameters

source	A pointer to an area of memory. DeleteBytes copies source into dest, removing a specified number of bytes, as shown below.
dest	A pointer to the resulting area of memory that contains the source area without the deleted bytes.
sourceLen	The length of the source area, in bytes.
pos	The position within the source area where DeleteBytes begins deleting bytes.
byteCount	The number of bytes to be deleted.

DeleteFromString

```
PROCEDURE DeleteFromString(str: StringPtr;
                            firstPos, lastPos: Integer);
```

Purpose and Operation

This procedure deletes characters from the string, starting at firstPos and ending at lastPos. The routine then joins the remaining characters together to close the gap. The max value of the string is unchanged.

EqualStrings

```
FUNCTION EqualStrings(str1,str2: StringPtr):Boolean;
```

Purpose and Operation

The routine compares two strings character by character and returns True if they have the same characters and the same number of characters.

NOTE: This routine does NOT discriminate between uppercase and lowercase. The string 'GRiD' is equal to 'grid'.

ExactCopyOfString

```
FUNCTION ExactCopyOfString (oldStr : StringPtr) : StringPtr;
```

Purpose and Operation

This function makes an exact copy of a specified string and returns a stringPtr to the copy. The exact copy will have .len set to the current .len (as opposed to .max) of the specified string. This function is often used in conjunction with data driven forms to obtain a copy of editable string choices which would otherwise be when the form is disposed of.

Parameters

oldStr	A pointer to the string that is to be copied.

Function Return

A pointer to the new string created by the copy operation.

FFExecuteCommand

```
FUNCTION FFExecuteCommand (filename: StringPtr) : WORD;
```

Purpose and Operation

When the File form is confirmed and the user has specified a "Next action" of "Get new file and its application", the function FFExecuteCommand is used to get the new application and file. This function requires only the file name (as returned by the FileFormConfirmed function) as its input. It passes this file name to the system Executive which retrieves the appropriate application to work with that file. For example, if you pass a file name with a Kind of ~Text~ to FFExecuteCommand, the Executive looks for an application program with a Kind of ~Run Text~ and loads that program and the specified text file into memory.

The calling program must check for an error return of "ok" from the function and then do an OsExit. The system Executive will not load the new application program and the specified file into memory until the current process has exited.

Parameters

fileName	A string pointer to the name of the file as returned by the FileFormConfirmed function.

Function Return

The function will return an error such as "File not found" if it cannot locate an application that matches the specified file's kind. If an appropriate application is found, the function returns "ok".

FileFormConfirmed

```
FUNCTION FileFormConfirmed (FFMode: FFModeType;
                                userPID: WORD;
                            VAR ch: CHAR;
                            VAR formRect: Rectangle;
                                prompt: StringPtr;
                            VAR pathName: StringPtr;
                                spare: StringPtr;
                            VAR defaultRec: FFDefaultTypeRec;
                                attachMode: BOOLEAN;
                                mode :BYTE;
                                access: BYTE;
                            VAR connection: WORD;
                                ExchangeMode: FFExchangeMode;
                            VAR ExchangeResult: FFExchangeResult;
                            VAR SaveResult: FFSaveResult) : Boolean;
```

Purpose and Operation

This function displays the File form, handles movement of the selection outline when the user presses the arrow keys, and returns with the selected items when the form is confirmed. The function also displays appropriate messages and prompts. The items that will be displayed in the form can be varied according to conditions established when the function is called. For a thorough discussion of the capabilities of this function, refer to Chapter 8.

Parameters

FFMode	An FFGet or FFPut. Usually set to FFGet except for "Write to a file". Determines which message will be displayed with the form.
userPID	The process ID of your keyboard process. Used by the function to read keystrokes.
ch	The last keystroke typed. You should need to look at this character only when the form is not confirmed.
formRect	This rectangle defines what part of your window the form will be displayed in. Returns the rectangle that your application should refresh.
prompt	A pointer to the string to be displayed as the prompt for the form. This stringPtr is automatically deallocated.
pathName	The pathName parts that the function should display if defaults are specified in the defaultRec. On return, it indicates the actual pathName that the user confirmed.
spare	Not currently used. Pass NIL to this parameter to enusre compatibility with future uses.
defaultRec	Defines which part(s) of the pathName parameter should be displayed initially in the form and which parts should be initially blank.
attachMode	Species whether the indicated file should be attached. You'll usually want to attach the file except for such operations as

	"Show file characteristics".
fileMode	The file mode for the attach such as update, old, new. (See OsAttach in the GRiD-OS Reference for a discussion of these modes.)
access	The access mode for the attach such as read only, write only, update (read/write). (See OsAttach in the GRiD-OS Reference for a discussion of these modes.)
connection	The connection number of the attached file returned by the function.
exchangeMode	Specifies whether to display the "Next action" (exchange) and/or "Save changes" items on the form.
ExchangeResult	On entry, speciifies which of the "Next action" choices should be in the selection outline, on return contains the choice that was confirmed.
SaveResult	On entry, specifies which of the "Save changes" choices should be in the selection outline, on return contains the choice that was confirmed.

FinalizePropertiesLength

```
PROCEDURE FinalizePropertiesLength(conn: Word;
                                VAR error: Integer);
```

Purpose and Operation

This procedure takes the current File Position (LongInt) and writes that value to the file's header when the file is written to a device. The procedure does not, itself, know the length of common properties. You should call this procedure immediately after you have written the last of your common properties records. After calling this procedure, you can begin writing data records and application properties records.

Parameters

conn	The connection number specifying the data file whose properties records length are being finalized.

FindByte

```
FUNCTION FindByte(VAR source: Bytes;
                  ByteToFind: Char;
                  count: Word;
                  VAR index: Word): Boolean;
```

Purpose and Operation

This function searches an array of bytes in memory for a given character, and returns its position in the array.

Parameters

source	A pointer to the location of the data to be examined.
ByteToFind	The character or byte to be compared with the memory area.
count	The length of the memory area, in bytes.
index	An index into the source's memory area, it indicates the position in the area where the memory and the character matched. This value is returned by reference. NOTE: the index starts at 0 (not 1) in order to be compatible with PL/M. Hence, index = 0 represents the first position in the memory area.

Returns

FindByte returns a Boolean indicating whether or not the byte was found in the memory area. If FindByte returns True, then the index variable contains the character position where the match was successful.

FldDimHilightField

```
PROCEDURE FldDimHilightField(VAR field: FieldDescriptor);
```

Purpose and Operation

This procedure draws a one-pixel dashed outline box around the field, leaving a one-pixel space from the field boundary.

FldDrawCursor

```
PROCEDURE FldDrawCursor(VAR cur: CursorDescriptor);
```

Purpose and Operation

This procedure makes the cursor visible and sets cur.on to True. It also sets the cursor blink count.

FldDrawField

```
PROCEDURE FldDrawField(VAR field: FieldDescriptor);
```

Purpose and Operation

This procedure erases the given field and then redisplays the field's text string. It clips the cursor and text to the field's rectangle -- when the field is full, any extra characters don't overwrite the adjacent cells. Call it when you need to display initial values or redraw the updated value of a single field. This procedure accomodates multi-line fields with word-wrapping, but does not word wrap the last line of the field.

If the fieldKind is numeric, then the field receives additional formatting. If a number is too wide to fit in the field's display area, then FldDrawField truncates any additional fractional digits without rounding. If the integer portion of the number is too large, then it displays asterisks in the field to indicate overflow. The actual contents of the field are not changed, however.

FldDrawFieldChars

```
PROCEDURE FldDrawFieldChars(VAR field: FieldDescriptor);
```

Purpose and Operation

Draws the field's text string without erasing the field first. It clips the cursor and text to the field's rectangle -- when the field is full, any extra characters don't overwrite the adjacent cells. This procedure accomodates multi-line fields with word-wrapping, but does not word wrap the last line of the field.

If you're redrawing many fields at once, it's faster to erase many fields at the same time, instead of erasing them individually, as FldDrawField does. The faster way is to erase with WinEraseWindow or WinEraseRectangle, and then redraw the fields with FldDrawFieldChars.

FldEditField

```
FUNCTION FldEditField(VAR cur: CursorDescriptor;
                          ch: Word): FieldEditResult;
```

Purpose and Operation

This all-purpose routine inserts values into the field's character string, performs various key functions, and updates both the display and the cursor. It recognizes BACKSPACE, CODE-BACKSPACE (erase previous word), SHIFT-CarriageReturn, and arrow keys. Pressing the RETURN key enters both a Carriage Return (CR) and Line Feed (LF). To enter only a CR, press CTRL-M.

Returns

After attempting to insert a character or perform a function, the procedure returns one of these values: (See Caution below.)

ok	The procedure successfully processed a character, such as an arrow key, but did not change the contents of the field.
processed	The procedure processed a character that changed the field's contents. This includes inserting, modifying, or deleting text characters in the field.
escaped	The user pressed ESC, and nothing was done to the contents of the field.
ignored	The procedure received a character that it did not know how to process. By testing for this result, you returns bufferFull. That is, if no text string has been allocated for a field, then that field's text buffer cannot accept text and therefore will appear to be full.
fieldFull	Though the text string has room for more characters, the additional character would have been displayed outside the field boundaries. This result restricts fields to their own boundaries, and prevents them from overwriting other fields.

If you want to pass a character of type Char to this routine, use the following example code. In the example, cursor is of type CursorDescriptor, oneCh is of type Char, and result is of type FieldEditResult:

```
result := FldEditField(cursor, Ord(oneCh));
```

If the user types a number that is too large to fit in a numeric field, the field now fills up with asterisks. Erasing part of the number or enlarging the field causes the asterisks to disappear.

The FieldEditResult of fieldFull can now be used to support multiline fields. The fieldFull result means that there was enough room in the field's text buffer to hold the new character, but not all the characters in the field can be displayed. (The character is inserted into the field anyway. The characters that are not shown are at the end of the buffer.) To display the hidden characters by changing the field into a multiline field, the program can change the size of the field's box and redraw it.

There are two ways to generate a fieldFull result.

- o Fill up the field with text. The first character that cannot be displayed will cause a fieldFull result.

- o Type SHIFT-RETURN on the last line of a field. After inserting the SHIFT-RETURN character into the field, FieldEditField returns fieldFull.

By checking for the fieldFull condition, the application program can then add another line of vertical space to the field. To add space to a single field, change the size of its box and redraw it.

Pressing the RETURN key by itself only inserts a Carriage Return - Line Feed pair into the field. You must press SHIFT-RETURN to begin a new line of a field. This distinction is necessary to maintain compatability with Compass Computer interchange files. SHIFT-RETURN enables you to define multiple lines in a field, which can then be combined as a single Compass file record.

CAUTION

FldEditField currently does not produce a FieldEditResult of "processed"; it returns the "ok" result under the conditions described for "ok" and "processed". It currently does not produce a FieldEditResult of "escaped"; it returns the "ignored" result under the conditions described for "ignored" and "escaped". (The "processed" and "escaped" values are still legal values for FieldEditResult. They are not currently returned as values.)

FldEraseCursor

```
PROCEDURE FldEraseCursor(VAR cur: CursorDescriptor);
```

Purpose and Operation

This procedure erases the cursor from the display without affecting its position in the field or in the window coordinates, and sets the blink count.

FldFormatLine

```
FUNCTION FldFormatLine (VAR field: FieldDescriptor;
                        charIndex: Word;
                        VAR limPos: Word;
                        VAR leftEdge: Integer): Boolean;
```

Purpose and Operation

FldFormatLine examines the text of a FieldDescriptor and determines where the text should appear on each line of a multi-line field. It does not display the field, however.

It performs word-wrapping automatically: if a word is too long to fit on a line, FldFormatLine does not include it in that line. The last line of the field is not word-wrapped, however. Note that all characters in a field are displayed, including spaces. If a space occurs in the field, it may be displayed as the first character of a line; that line will appear indented by a space.

This function, also interprets RETURN and SHIFT-RETURN separately. FldFormatLine formats Carriage Return and Line Feed characters just as it does any other characters, by inserting them into the line. If FldFormatLine encounters a SHIFT-RETURN character when formatting the line, it ends the line with that character.

Parameters

It takes these parameter variables:

field	The FieldDescriptor of the field being formatted.
charIndex	An index into the field's text string. It shows which character in the text string will become the first character of a line in the field. For example, if you call FldFormatLine three times with charIndex = 1, 6, and 11, then the three formatted lines in the field would begin with the first, sixth, and eleventh characters respectively from the text string.
limPos	An index into the field's text string that shows which character in the text string will begin the NEXT line. This variable is an OUTPUT from FldFormatLine.
leftEdge	The position of the left edge of the text, measured in pixels from the left boundary of the window. In conjunction with FieldDescriptor.box, it controls the alignment of text within the cell.

Returns

The output of FldFormatLine is a Boolean. If it is True, then the current formatted line contains the last of the text from the text string. If it is False, then there is still more text left in the text string to be formatted.

Example

To format a multi-line field, the application will need to call FldFormatLine repeatedly. The limPos that FldFormatLine calculates becomes the new charIndex when FldFormatLine is called again:

```
     FldFormatLine (  , charIndex, limPos,  )
                        = 1

     FldFormatLine (  , charIndex, limPos,  )


     FldFormatLine (  , charIndex, limPos,  )
```

The limPos variable is an output representing where the next line should start. When you call the procedure again, the old limPos should now become charIndex, which shows where the current line begins.

FldHilightField

```
PROCEDURE FldHilightField(VAR field: FieldDescriptor);
```

Purpose and Operation

This procedure draws an outline box around the field. The line of the box's outline is three pixels wide, and it lies one pixel away from the field's outer boundary. The cursor's three-pixel outline is generated automatically.

FldInsertInField

```
FUNCTION FldInsertInField(VAR cur: CursorDescriptor;
                          ch: Char): Boolean;
```

Purpose and Operation

This function inserts a character in the field at the cursor's current character position, and verifies its insertion by returning True or False. It does not redraw the display on the screen. Most applications should call FldEditField instead.

FldInvertChar

```
PROCEDURE FldInvertChar(field: FieldPtr; pos: Word);
```

Purpose and Operation

This routine performs an exclusive OR operation with a field's screen display to change a character position to inverse-video.

FldReadKey

```
FUNCTION FldReadKey(VAR cursor: CursorDescriptor): Word;
```

FldReadKey replaces two routines, ConKeyPressed and ConCharIn, that were used to busy-wait for input from the keyboard. Instead, this procedure leaves the processor free for other tasks while waiting for a key to be pressed.

The function waits for an interrupt signifying that a key has been pressed. If no keys are pressed for a certain time interval, the function blinks the cursor, and then resumes waiting for a key to be pressed. (Your application program will wait at the statement containing this function call.) If a key is pressed, then the function returns the character, and your application program can continue.

Note: Call FldStartKeys once when you initialize your program, before calling this function. When your program finishes, call OsDeleteProcess to delete the cursor process.

Status bits from the keyboard are returned in the high order byte of the word. The high-order byte of the word is defined as follows:

Bit	Abbreviation	Meaning
8	IBF	=1 if character available
9	OBF	=1 if latest command has not been processed
10	(not used)	(not used)
11	(not used)	(not used)
12	RPT	=1 if a repeated character
13	SHFT	=1 if a shifted character
14	CODE	=1 if a code character
15	CTRL	=1 if a control character

Use the example code below to ignore the status byte in your programs. In it, ch is of type Char, and cursor is of type CursorDescriptor.

```
ch := Chr(FldReadKey(cursor));
```

FldSetCursor

```
PROCEDURE FldSetCursor(VAR cur: CursorDescriptor;
                          field: FieldPtr);
```

Purpose and Operation

This procedure sets the cursor to the given field, at the last character position and sets Cur.on to False. The procedure does not alter the display.

FldSetPos

```
PROCEDURE FldSetPos(VAR cur: CursorDescriptor; pos: Integer);
```

Purpose and Operation

Given the character position of the cursor, it sets the x-y pixel coordinate for the place element of the cursor. An application can set the cursor to any character position in the field. The display is unchanged. Cur.on is set to False.

FldStartKeys

```
PROCEDURE FldStartKeys (VAR cursor: CursorDescriptor);
```

FldStartKeys starts a process to control the cursor. It puts the PID of the process into cursor.keyProcess. (You use it to "initialize" the cursor, in effect.)

FontCount

```
FUNCTION FontCount: Integer;
```

Purpose and Operation

This function returns an integer which indicates how many fonts are available in the system. Most applications will not need to use this function since, if they have the name of a font, they can directly obtain the index number of that font and need not scan through the list of fonts. However, if an application (for example GRiDManager) needs to maintain its own list of fonts, FontCount may be useful.

FontGetN

```
FUNCTION FontGetN (name: StringPtr): Integer;
```

Purpose and Operation

Given a pointer to the font name, this function returns an integer (index) that is associated with the current font. This value can then be used to correctly position the choice field highlight when an application displays a data driven form.

FontNthName

```
FUNCTION FontNthName(index: Integer): StringPtr;
```

Purpose and Operation

Given the font index number, this function returns a pointer to the string containing the name of a font. (If the index value is not in the list of fonts, the function returns NIL.) This function can thus be used by an application to obtain the name of the current font associated with a data file. The application could then, for example, write the name of that font to the common properties record of the file.

FontSetName

```
PROCEDURE FontSetName(name: StringPtr; VAR error: Word);
```

Purpose and Operation

This procedure causes the font specified by the name parameter to be loaded into memory. This font thus becomes the current font.

Possible Errors

All disk errors.
Out of memory.

FontSetNth

```
PROCEDURE FontSetNth(index: Integer; VAR error: Word);
```

Purpose and Operation

This procedure causes the font specified by the index parameter to be loaded into memory. This font thus becomes the current font.

Possible Errors

All disk errors.
Out of memory.

FreeString

```
PROCEDURE FreeString(VAR str: StringPtr);
```

Purpose and Operation

Given the StringPtr to a string, FreeString will release the memory that the string occupied and return that memory to the PASCAL heap.

Note: you must NEVER modify String^.max, because FreeString uses that number to determine how much memory to release to the heap. Other data values may be incorrectly released if String^.max is changed from its original value.

FreeStringsInDataForm

```
PROCEDURE FreeStringsInDataForm (VAR dataForm : DataFormType);
```

Purpose and Operation

This procedure frees all the strings in a data form. It should be used only if you do NOT store the values of a form in the form itself. If you store the values of a form in permanent variables, you can call this procedure after you have copied current form values into the variables.

Parameters

dataForm	The form whose strings are to be freed.

GetNextRecord

```
FUNCTION GetNextRecord (conn: Word;
                   VAR gRecord: GeneralRecordPointer;
                   VAR gRecordLength: Word;
                       thisIsTheAuthorCalling: Boolean;
                   VAR error: Integer): Boolean;
```

Purpose and Operation

Given a file's connection number, this function returns with a pointer to the next record from the data file and the length of that record. If you have specified that you are the author of this file, all records, including application (private) properties, will be retrieved using this call. If you have specified that you are not the author, application properties records are automatically skipped and only common properties records and data records will be retrieved. The procedure updates the current file position pointer so that it is pointing just beyound the end of the record just returned.

Parameters

conn	The connection number for the file whose records are being read.
gRecord	A pointer to the beginning of the next record from the data file. The procedure allocates a new record if gRecord is NIL or if the current length of the record is shorter than the next record in the file. (The format of GeneralRecord is shown below.)
thisIsTheAuthorCalling	A Boolean that, if set true, permits all records in the data file to be read. If set false, the function automatically skips application properties records and returns only common properties records and data records.

```
GeneralRecord = RECORD
  headerByte: Byte;
  CASE Integer OF
    0FFh: ( textString:   ARRAY [1..2] OF Char ); { ends with CRLF }
    0FEh: ( commonProps:  CommonPropertiesRecord );
    0FDh: ( userLength:  Word; userProps:  ARRAY [1..1] OF Byte );
    000h: ( length: Word; binaryProps: Byte)
  END;

GeneralRecordPointer = ^GeneralRecord;
```

Function Return

The procedure returns a Boolean that is True if the first byte of the record contains FE hex (a common properties record) or FD hex (an applications properties record). The Boolean will be True for application properties records only if you are the author of the file (thisIsTheAuthorCalling set True).

GetVersionString

```
FUNCTION GetVersionString (pID : Word) : StringPtr;
```

Purpose and Operation

This function returns a pointer to the string containing the version number and message for the file identified by the pID parameter. GRiD applications use this function to obtain their own version number for display when the application is first initialized and when the user presses CODE-?.

Parameters

pID	process ID for the current application. (Use DsWhoAmI to obtain your own pID.)

Returns

A pointer to a string containing a three numeral version number in the following format:

```
'Version x.y.z'
```

where x, y, and z are integers in the range [0..255]. Applications may assign significance to x, y, and z. If the software has not had the version set in the file descriptor by the version program, the string returned will be:

```
'Version 0.0.0'
```

InsertBytes

```
PROCEDURE InsertBytes (VAR source, dest: Bytes;
                       len, pos, byteCount: Word);
```

Purpose and Operation

This procedure inserts bytes into a specified area of memory. The contents of the inserted bytes are undefined. This procedure is useful for inserting new elements into arrays, structures, strings, etc.

Source and dest can refer to the same area of memory or to different areas.

Parameters

source	A pointer to an area of memory. InsertBytes copies source into dest, inserting a specified number of bytes, as shown below.
dest	A pointer to the resulting area of memory containing the source area and the inserted bytes.
len	The length of the source area, in bytes.
pos	The position within the source area where the insertion begins.
byteCount	The number of bytes to be inserted.

InsertCharInString

```
FUNCTION InsertCharInString(ch: Char;
                            str: StringPtr;
                            pos: Integer): Boolean;
```

Purpose and Operation

This function inserts a single character into a string. It inserts ch into str beginning at the character position given by pos.

If inserting the ch would make str longer than its max length, then InsertInString returns False, and nothing is inserted.

InsertInString

```
FUNCTION InsertInString(piece, str: StringPtr;
                        pos: Integer): Boolean;
```

Purpose and Operation

This function inserts a string into another string. It inserts piece into str beginning at the character position given by pos. The existing characters of str are moved aside to make room for the insertion.

If inserting the piece would make str longer than its max length, then InsertInString returns False, and nothing is inserted.

IntegerToString

```
FUNCTION IntegerToString(int: Integer): StringPtr;
```

Purpose and Operation

IntegerToString converts an integer between -32768 and 32767 inclusive into a string, then returns a stringPtr to the string value.

MoveBytes

```
PROCEDURE MoveBytes(VAR source: Bytes;
                    VAR dest: Bytes; length: Word);
```

Purpose and Operation

MoveBytes moves data from one location in memory to another.

Parameters

source	A pointer to the location of the data to be moved (i.e., to the first element of an array of bytes).
dest	A pointer to the new destination of the moved data (i.e., to an element of an array of bytes).
length	Specifies how many bytes are to be moved, from 0 to 65535.

MoveReverseBytes

```
PROCEDURE MoveReverseBytes(VAR source: Bytes;
                           VAR dest: Bytes; length: Word);
```

Purpose and Operation

MoveReverseBytes moves data from one location in memory to another. It moves the data starting from the end of the data rather than the beginning, as shown in the figure below. This allows you to move bytes into a destination that overlaps the source location.

Parameters

source	A pointer to the location of the data to be moved (i.e., to the first element of an array of bytes).
dest	A pointer to the new destination of the moved data (i.e., to an element of an array of bytes). (Note that this is the first element of the destination, not the last.)
length	Specifies how many bytes are to be moved, from 0 to 65535.

MsgClearMessage

```
FUNCTION MsgClearMessage(msg : MessagePtr) : Boolean
```

Purpose and Operation

Erases any messages currently displayed. This does not erase prompts. Visible prompts are not affected by this function.

If the return value of the function is true, the application must update the rectangle in the window indicated by *msg^.rect*.

MsgClearPrompt

```
FUNCTION MsgClearPrompt(msg : MessagePtr) : Boolean
```

Purpose and Operation

Erases any prompts currently displayed. Messages that have prompts stacked on them are also erased by this function, otherwise messages are not affected by this function.

If the return value of the function is true, the application must update the rectangle in the window indicated by *msg^.rect*.

MsgExit

```
PROCEDURE MsgExit(code : Word; msg : MessagePtr)
```

Purpose and Operation

This function is used before exiting an application. The code parameter will dictate which message will be displayed:

code	Message
0	Retrieving subjects: In progress
2	Out of memory Confirm to exit
other	System Error: [error code] Confirm to reinitialize system

If the system is ready to continue (code 0 or 2), MsgExit will call OsExit(0), otherwise it will reboot. Note that this procedure is actually a combination of message and prompts according to the guidelines layed down in Chapter 6. Code 0 simply displays a message and exits, ignoring anything the user may do at the keyboard. Codes 2 will automatically exit on CONFIRM but should remain in the application if any other key is pressed. Any "other" code indicates a catastrophic event and the exit will be performed on CONFIRM; any other key might leave the user in the application -- GRiD applications, however, typically exit regardless of what key is pressed at this point. (Here's your hat. There's the door.)

MsgInit

```
FUNCTION MsgInit : MessagePtr
```

Purpose and Operation

This function dynamically allocates a MessageStatus record and returns a pointer to it. All necessary fields of the record are initialized, including the location of the message. The *box* field in type FieldDescriptor is initialized to the bottom of the current window. This is the default position of all single message/prompts and the point at which stacking begins.

Each message or prompt you use must have a MessageStatus record. Therefore you must call this function before calling the functions which actually display the message or prompt.

The organization of the MessageStatus record is as follows:

```
TYPE MessageStatus =
       RECORD
       messageShowing: Boolean;
       stackSize : Byte;
       field: FieldPtr;
       rect: Rectangle;      {area to be updated}
       anythingShowing : Boolean;
       END;

    MessagePtr = ^MessageStatus;
```

messageShowing	A boolean that indicates if a message is currently displayed. If a prompt is showing, or if no message is showing, it is false. This field is NOT altered by the application. It is initialized by MsgInit and updated by the various message calls.
stackSize	Indicates the number of messages/prompts currently showing. This is NOT altered by the application. It is initialized by MsgInit and updated by the various message calls.
field	Pointer to the field descriptor record containing the text and location of the message.
rect	The rectangle that the application should update if the boolean result of one of the message FUNCTION calls is true. This value is initialized by MsgInit, updated by the various message calls, and read by the applications. It is not altered by the applications.
anythingShowing	Boolean field that is not used in the current version of the Common Code message module.

The organization of the field descriptor record pointed to by the *field* parameter is as follows:

```
    FieldDescriptor = RECORD
```

```
                         box: Rectangle;
                         text: StringPtr;
                         kind: FieldKind;
                       END;

FieldPtr = ^FieldDescriptor;

FieldDescriptor =
  RECORD
  box: Rectangle;
  text: StringPtr;
  kind: FieldKind;
  END;

FieldPtr = ^FieldDescriptor;
```

MsgInitialUsage

```
FUNCTION MsgInitialUsage: LongInt;
```

Purpose and Operation

When initializing your application, call MsgInitialUsage to find the amount of memory taken by the application code itself, without the user's workspace.

MsgShowDecoded

```
FUNCTION MsgShowDecoded(msg : MessagePtr; errorCode : Integer) : Boolean
```

Purpose and Operation

MsgShowDecoded takes an integer corresponding to an error message defined by the GRiD Operating System. It finds the text message corresponding to the error code, and displays it as a one line message. It erases any previous messages or prompts. It freezes the keyboard for two seconds, ignoring any input during that time.

If the return value of the function is true, the application must update the rectangle in the window indicated by *msg^.rect*.

For a list of error codes and their corresponding messages, see Appendix B of the GRiD-OS Reference Manual.

MsgShowError

```
FUNCTION MsgShowError(msg : MessagePtr; str : StringPtr) : Boolean
```

Purpose and Operation

This function erases the previous message or prompt (stack), then displays the given string as a one line message at the bottom of the window. Unlike the other display routines, it freezes the keyboard input for two seconds. Any characters entered during this period are ignored. The *msg* variable keeps track of the status of the message. MsgShowError disposes of the strimg it receives as input.

If the return value of the function is true, the application must update the rectangle in the window indicated by *msg^.rect*.

Parameters

msg	A pointer to the MessageStatus record for this message.
str	A pointer to the message text that is to be displayed.

Function Return

A boolean that, if true, indicates that the application must update the rectangle in the window indicated by *msg^.rect*.

MsgShowMessage

```
FUNCTION MsgShowMessage(msg : MessagePtr; str : StringPtr) : Boolean
```

Purpose and Operation

This function erases the previous message or prompt stack before displaying the given string as a one line prompt at the bottom of the window. The *msg* variable keeps track of the status of the message. MsgShowMessage disposes of the string it receives as input.

If the return value of the function is true, the application must update the rectangle in the window indicated by *msg^.rect*.

If the application changes the value of the *box* field of the FieldDescriptor (presumably to change the default position of the message), then all future stacking of messages will be in reference to this new position. MsgShowMessage only clears messages and prompts correctly if the default (base) position of the *box* is the bottom of the window. If the application alters the position of the *box*, it is responsible for clearing the messages and prompts with their respective clear functions.

Parameters

msg	A pointer to the MessageStatus record for this message.
str	A pointer to the message text that is to be displayed.

Function Return

A boolean that, if true, indicates that the application must update the rectangle in the window indicated by *msg^.rect*.

MsgShowPrompt

```
FUNCTION MsgShowPrompt(msg : MessagePtr; str : StringPtr) : Boolean
```

Purpose and Operation

This function erases the previous message(s) or prompt(s) before displaying the given string as a one line prompt at the bottom of the window. The *msg* variable keeps track of the status of the prompt. MsgShowPrompt disposes of the string it receives as input.

If the return value of the function is true, the application must update the rectangle in the window indicated by *msg^.rect*.

If the application changes the value of the *box* field of the FieldDescriptor (presumably to change the default position of the prompt), then all future stacking of messages will be in reference to this new position. MsgShowPrompt only clears messages and prompts correctly if the default (base) position of the *box* is the bottom of the window. If the application alters the position of the *box*, it is responsible for clearing the messages and prompts with their respective clear functions.

Parameters

msg	A pointer to the MessageStatus record for this prompt.
str	A pointer to the prompt text that is to be displayed.

Function Return

A boolean that, if true, indicates that the application must update the rectangle in the window indicated by *msg^.rect*.

MsgStackMessage

```
FUNCTION MsgStackMessage(msg : MessagePtr; str : StringPtr) : Boolean
```

Purpose and Operation

This function stacks a message on top of currently displayed messages. The *msg* variable keeps track of the status of the message. MsgStackMessage disposes of the string it receives as input.

If the return value of the function is true, the application must update the rectangle in the window indicated by *msg^.rect*.

Stacking a message on a prompt will first erase the prompt (stack of prompts) and then display the message at the bottom of the window.

Parameters

msg	A pointer to the MessageStatus record for this message.
str	A pointer to the message text that is to be displayed.

Function Return

A boolean that, if true, indicates that the application must update the rectangle in the window indicated by *msg^.rect*.

MsgStackPrompt

```
FUNCTION MsgStackPrompt(msg : MessagePtr; str : StringPtr) : Boolean
```

Purpose and Operation

This function stacks a prompt on top of currently displayed messages or prompts. The *msg* variable keeps track of the status of the prompt. MsgStackPrompt disposes of the string it receives as input.

If the return value of the function is true, the application must update the rectangle in the window indicated by *msg^.rect*.

Stacking a prompt on a message will place the prompt on top of the message. The resulting prompt-message block is considered a prompt for future message rules (See Chapter 6).

Parameters

msg	A pointer to the MessageStatus record for this prompt.
str	A pointer to the prompt text that is to be displayed.

Function Return

A boolean that, if true, indicates that the application must update the rectangle in the window indicated by *msg^.rect*.

NewString

```
FUNCTION NewString(maxLength: Word): StringPtr;
```

Purpose and Operation

NewString allocates memory for a new string of the given length and returns a string pointer to that area in memory. The maxLength that is passed as a parameter becomes the maximum length (max) of the new string, and the current length (len) is initialized to zero. If you try to refer to an element in the string beyond string^.chars[max] another variable's memory area may be damaged.

CAUTION: ONLY NewString and NewStringLit WILL PROPERLY ALLOCATE SPACE FOR THESE STRINGS. NEVER call New(StringPtr) because New will allocate all 65535 bytes according to the declaration of String.chars[1..65535] above.

When declaring your own static variables to deal with strings, you must declare them to be StringPtrs, NOT Strings. If you declare a static variable as type String, the compiler will try to allocate 65535 bytes for String.chars[1..65535] according to the declaration of the String record. You should declare the variable to be of type StringPtr and then assign it with the value resulting from a call to NewString or NewStringLit.

NewStringLit

```
FUNCTION NewStringLit(VAR lit: Bytes): StringPtr;
```

Purpose and Operation

NewStringLit takes a literal string, allocates memory for it, and returns a string pointer. The maximum length (max) and current length (len) of the new string is the length of the literal characters.

NOTE: The last character of lit must be a DEL character (CODE-SHIFT-hyphen). NewStringLit needs the DEL character to determine the length of the string.

Example

```
string := NewStringLit('The results are▮');
```

The len and max of string are both 15. String^.chars is equal to 'The results are'.

RealToString

```
FUNCTION RealToString(aReal: LongReal;
                      fracDigits: Integer): StringPtr;
```

Purpose and Operation

RealToString converts a fifteen digit real number into a string variable. (The 8087 numeric processor uses fifteen and a half digits of precision; this routine returns fifteen digits, rounding off the half digit as necessary.) Note that this routine produces real numbers of fifteen and a half digits. It cannot accomodate exponential notation, such as 6.03E+23. For details, see the Intel 8087 Floating Point Processor Manual or the Pascal Manual.

The variable fracDigits determines how many digits after the decimal point will be included in the string. Any extra digits beyond fracDigits will be rounded. The maximum value for fracDigits is 16 places. Setting fracDigits greater than 16 is not recommended. Setting fracDigits to -1 will cause the routine to strip trailing zeros and to return only the significant digits of the number. If fracDigits is zero, then the real number is rounded to an integer (without a decimal point) and converted to a string.

NOTE: If fracDigits = 15 and the integer portion of the number is greater than zero, then some of the numerals to the right of the decimal will be incorrect. For example:

```
0.123456789012345

1234.123456789012222
```

The first number is accurate, but the final four digits of the second number ("2222") are spurious.

Although this function takes a LongReal parameter, it can also convert numbers of type Real and LongInt. Real numbers can be used as parameters directly, because Pascal converts them to LongReals automatically.

RealToString is the only function that can be used to convert LongInt numbers to strings. (The LongInt type is not compatible with the IntegerToString function.) To convert a LongInt number to a string:

```
VAR a: LongInt;
    b: LongReal;
b := a;
resultstring = RealToString(b,0);
```

SetBytes

```
PROCEDURE SetBytes (value: Char;
                    VAR dest: Bytes; count: Word);
```

Purpose and Operation

This procedure sets every byte in the destination area to the same given value.

Parameters

value	The value that SetBytes will assign to every byte in the memory area.
dest	A pointer to the destination area in memory.
count	The length of the destination area, in bytes.

SetPrefix

```
PROCEDURE SetPrefix (subjectName: StringPtr);
```

Purpose and Operation

This is used by the application to set the prefix subject, ie. if the current prefix is `Hard disk`programs, then SetPrefix(NewStringLit('Incs ')) sets the prefix to `Hard disk`Incs.

SkipProperties

```
PROCEDURE (conn: Word;
       VAR error: Word);
```

Purpose and Operation

Given a connection number of a file, this procedure automatically skips over all common properties records in a data file and moves the current file position pointer. Thus, a subsequent OsRead or GetNextRecord call would begin with the first record following the common properties.

StringOfFormItem

```
FUNCTION StringOfFormItem (VAR dataForm : DataFormType;
                               row : INTEGER) : StringPtr;
```

Purpose and Operation

This function returns a stringPtr to the text actually displayed in a form item. It can be used if you need to know the text of a choice selection as opposed to the number of the choice.

Parameters

dataForm	The form containing the desired text.
row	The row within the form containing the text of the desired choice selection.

Function Return

A stringPtr to the text of specified the choice selection.

StringToInteger

```
FUNCTION StringToInteger(str: StringPtr;
                         VAR converted: Boolean): Integer;
```

Purpose and Operation

This function converts a string value into an integer. The string must represent an integer between -32768 and 32767 inclusive for the conversion to succeed. The variable "converted" indicates whether the conversion was successful or not.

StringToReal

```
FUNCTION StringToReal(str: StringPtr;
                      VAR converted: Boolean): LongReal;
```

Purpose and Operation

This function converts a string value into a real number. The variable "converted" indicates whether the conversion was successful or not. It will convert up to the first fifteen digits, and drop any extra digits without causing an error. If the conversion fails (from incorrect input, for example) the routine returns 0.

Note that this routine produces real numbers of fifteen and a half digits. It cannot accomodate exponential notation, such as 6.03E+23. For details, see the Intel 8087 Floating Point Processor Manual or the Pascal Manual.

SubProperty

```
FUNCTION SubProperty(Str : StringPtr;
                     index : Integer) : StringPtr;
```

Purpose and Operation

Picks a name out of a string made up of names and special characters. The special characters are delimiters in the GRiD-OS file system.

Parameters

Str	The string that usually represents a pathname.
Index	An index into the string that represents the significance of the desired token.

Returns

A substring of Str that was contained between delimiters. The legal delimiters are: ~,`,!, and 0.

Example

```
If Str = `W0`Programs`sample~text~  then
SubProperty(str, 1) returns 'W0'
SubProperty(str, 4) returns 'text'.

If Str = sample~text!GRiDiRG  then
SubProperty(str, 0) returns 'sample'
SubProperty(str, 2) returns 'GRiDiRG'
SubProperty(str, 3) returns NIL.
```

SubStringLit

```
FUNCTION SubStringLit(VAR lit: Bytes;
                          delim: Char;
                          count: Word): StringPtr;
```

Purpose and Operation

This function returns the Nth item (specified by count) from a literal containing text separated by delim characters.

It's useful for constructing the ItemStr or ChoiceStr functional parameters as defined for menus and forms. For creating menus and forms, no carriage returns or line feeds should be embedded in the literal.

Parameters

lit	A pointer to a literal. All items should be concatenated together into a single literal, with the items separated from one another by delim characters. A delim character must follow the last item.
delim	The character that separates the items in the literal. It can be the literal character surrounded by single quotes, such as ' ' or '/', or it can be the ASCII value, as in CHR(127). Most programmers use the DEL character as a delimiter -- '█' or CHR(127). While DEL characters appear as a solid square in the text editor, they do not appear on printouts.
count	An integer index to the substring of lit.

Returns

SubStringLit returns a StringPtr pointing to a copy of the substring. The substring does not include any DEL characters.

Example

```
If menuString = 'COPY█MOVE█DELETE█'
                     ^    ^      ^
                     |    |      |
                    DEL characters
```

then SubStringLit(menuString,'█',1) returns a StringPtr containing 'COPY'.

Other legal calls include:

```
     CONST x = 'a/b/c/'
     VAR A [1..40] of Char;
         substr: StringPtr;
```

```
substr := SubStringLit('ReadOWriteOAppendO', '■', 2);
substr := SubStringLit(x, '/', 2);
substr := SubStringLit(string^.chars[1], '/', 2);
substr := SubStringLit(A, CHR(137), 2);
```

TblAddCol

```
PROCEDURE TblAddCol(chPerLine,linesPerField:Integer;
                    VAR table: CellTable;
                    shouldAlloc : Boolean;
                    editable: Boolean);
```

Purpose and Operation

This procedure appends another column to the CellTable matrix. The appended columns may have a different field width (characters per line) from the columns of the table being appended. By appending columns of different widths, you are not limited to tables of one width, such as the ones produced by TblInitTable. The number of lines in each field should be the same, however.

You can specify whether the procedure should allocate memory space for the values of the appended fields or not. The appended fields can be editable or non-editable, as well: TblAddCol enables you to construct tables made up of both editable and non-editable fields. For example, questionnaire templates would include non-editable areas for the questions and editable areas for the responses. When you add a column, the constraint is widened to include it. It's best to modify the cursor's constraint area after adding columns.

TblCellOnScreen

```
FUNCTION TblCellOnScreen(VAR table: CellTable;
                             cell: CellId):
                             Boolean;
```

Purpose and Operation

This function returns whether the cell is within CellTable.visible, i.e., whether it is to be displayed. This routine has NO relation to CellTable.visibleRect, the table's clipping rectangle.

TblChangeFields

```
FUNCTION TblChangeFields(VAR table: CellTable;
                      ch: Char): Boolean;
```

Purpose and Operation

This procedure, given a table and a movement character, moves the field outline from cell to cell. (It moves the cursor, too, if the cursor actually was in the currentCell.) Call it when EditTable returns a result of outOfField, and include the same ch character that you called TblEditTable with. TblChangeFields will return False if it is unable to move in the indicated direction, meaning that scrolling is necessary.

TblConfirmSelection

```
PROCEDURE TblConfirmSelection(VAR table: CellTable);
```

Purpose and Operation

Call TblConfirmSelection to save the source selection range for commands that require two selection ranges, such as Move and Duplicate. The table code will leave the source selection highlighted while the user selects a destination range. It performs these functions:

- It copies the CellIds and cursor positions in table.textcursor, table.anchor, and table.currentCell into table.sourceAnchor and table.sourceCurrent. The sourceAnchor and sourceCurrent CellIds are "normalized" so that sourceAnchor is the top left cell of the selection range, and sourceCurrent is the bottom right cell of the range. Both cells are corrected for scrolling -- table.scrollCell is added to them. (This has the same result regardless of whether scrolling is easy or difficult case.)

- It sets {table.anchor.pos} to nowhere, which will keep that value until the user presses CODE-B to select a destination range later.

- The variable table.whichParameter is set to 2, indicating that the user must select another parameter (such as a character position or CellId) to complete the command.

TblDimHilightCell

```
PROCEDURE TblDimHilightCell(VAR table: CellTable;
                                cell: CellId);
```

Purpose and Operation

This procedure draws a dashed outline around a cell.

TblDisposeScreen

```
PROCEDURE TblDisposeScreen(screen: ScreenPtr;
                           colCount: Integer);
```

Purpose and Operation

TblDisposeScreen deallocates screen arrays that have been created by TblNewScreen. The variable colCount represents the number of columns to be disposed of; it must equal the number of columns that were allocated when the screen arrE TblDisposeCol(col: ColPtr; rowCount: Integer);

Purpose and Operation

The procedure deallocates column arrays that have been created by TblNewCol. The number of rows (rowCount) to be disposed of must equal the number that were allocated when the column array was created.

TblDisposeTable

```
PROCEDURE TblDisposeTable(VAR Table: CellTable;
                          shouldDispose: Boolean);
```

Purpose and Operation

This procedure disposes of the specified cell table. It can dispose of the values of the fields in the table or retain them, according to these values of shouldDispose:

shouldDispose =	operation
disposeText (True)	dispose of the values of the fields
dontDisposeText (False)	keep the values of the fields

When shouldDispose = dontDisposeText, the procedure disposes of the table pointers and the field descriptors, but retains the values of each field. You might want to retain these values when other pointers need the values.

TblDrawGrid

```
PROCEDURE TblDrawGrid (VAR table: CellTable);
```

Purpose and Operation

Draws a frame around the visibleRect and grid lines between the fields of a table, if table.frame, table.verticalGrid, and table.horizontalGrid are True. If a variable is False, TblDrawGrid does not draw the graphics associated with it. It does not redraw the fields of the table. The frame and grid lines are one pixel wide.

TblDrawTable

```
PROCEDURE TblDrawTable(VAR table: CellTable);
```

Purpose and Operation

The procedure clears all fields from the screen and redisplays them with their current values, by calling FldDrawField for every field in the table. It overwrites the entire area defined by the visibleRect.

If table.verticalGrid and CellTable.horizontalGrid have been set to True, then the routine will draw lines between the fields. If table.frame is True, then it will draw a one-pixel frame outside table.visibleRect. (The frame is not within the coordinates of table.visibleRect).

Application Note: To draw a newly initialized table, your application must call TblDrawTable (to draw the fields) and TblHilightTable (to draw the cursor and to outline the cursor's cell). Later, TblEditTable and TblChangeFields will update and redisplay the table when the application modifies it; they redraw the table, the cursor, the cell outline, and the range selection (if any).

TblEditTable

```
FUNCTION TblEditTable(VAR table: CellTable;
                      ch: Word): FieldEditResult;
```

Purpose and Operation

This all-purpose table routine inserts characters at the current field location and cursor position, performs various key functions, and redraws both the display and the cursor. Call it once for every character read from the keyboard. It does not redraw the entire table, but just the changed field.

The routine recognizes BACKSPACE, CODE-BACKSPACE (erase previous word), RETURN, and arrow keys.

If the value of the ch character belongs to the set of CellTable.commandKeys, the currentCell will be displayed in inverse video, and the selection mechanism will be turned on. That is, when the user types a selection command, the current cell is inverted to show that a selection has begun.

Returns

After attempting to insert a character or perform a function, TblEditTable returns one of these values:

ok	The procedure successfully processed a character, such as an arrow key, but did not change the contents of the cell.
processed	The procedure processed a character that changed the cell's contents. This includes inserting, modifying, or deleting text characters in the cell. <<DOES THIS HAPPEN YET? ALSO ESCAPED? 7/30 UPDATE SAYS NO>>
escaped	The user pressed ESC, and nothing was done to the contents of the cell. Any selections are cleared, and the table leaves command mode.
ignored	The procedure received a character that it did not know how to process. By testing for this result, you can allow terminal emulation characters (such as CTRL characters) to pass through the application to another application or processor.
outOfField	An attempt was made to move the cursor out of the cell. TblEditTable doesn't actually move the cursor out of the cell. Call TblChangeFields to do so.
bufferFull	The text string of the cell is full. It cannot contain any additional characters.
	Note: if the text pointer of a field descriptor is nil, then TblEditTable returns bufferFull as well. That is, if no text string has been allocated for a cell, then that cell's text

	buffer cannot accept text and will therefore appear to be full.
fieldFull	The text buffer is not full, but not all the characters in the cell can be displayed. The character is inserted into the cell anyway. A fieldFull result occurs when the user presses SHIFT-RETURN or types enough text to fill the displayed area of the cell. By checking for the fieldFull condition, the application can then add another line of vertical space to the cell. (To keep scrolling and selection consistent, you must add another line to every cell in the row using TblChangeRowHeight.)

TblEqualCells

```
FUNCTION TblEqualCells(cell1,cell2: CellId):Boolean;
```

Purpose and Operation

If the given CellId's are equal, TblEqualCells returns True.

TblEscapeMode

```
PROCEDURE TblEscapeMode (VAR table: CellTable);
```

Purpose and Operation

This procedure puts the table into the normal (non-command) state, un-inverts any cell or text selection ranges, but leaves the cursor and the highlighted cell on.

TblFieldOfCellId

```
FUNCTION TblFieldOfCellId(VAR table: CellTable;
                          cell: CellId): FieldPtr;
```

Purpose and Operation

This function converts a CellId into a FieldPtr reference, which makes table values easier to refer to and to change. It is useful when working with cell variables of type CellId, such as currentCell.

TblFieldOfColRow

```
FUNCTION TblFieldOfColRow(VAR table: CellTable;
                       col, row: Integer): FieldPtr;
```

Purpose and Operation

This procedure, given a column and and a row of a cell table, it returns the pointer to the field. These two references return the same pointer value:

```
TblFieldOfColRow(table1, 1, 2)
table1.screen^[1]^[2]
```

TblFindBounds

```
PROCEDURE TblFindBounds(VAR table: CellTable;
                        VAR rect: Rectangle;
                        VAR left, right,
                        top, bottom: Integer);
```

Purpose and Operation

This procedure calculates which cells lie within a rectangle that has been defined in the pixel coordinates of the display window. Given an area on the screen, it allows you to update only a portion of the table.

TblGetSelectedCellIds

```
PROCEDURE TblGetSelectedCellIds(VAR table:CellTable;
                                VAR first, last: CellId);
```

Purpose and Operation

This subroutine is used for the "difficult case" of scrolling. It locates the movingCell and anchor CellIds, rearranges them in ascending order, adjusts them from relative "unscrolled" CellIds to absolute "scrolled" CellIds, and returns them as "first" and "last" absolute (logical) coordinates.

When referring to a selection, call TblGetSelectedCellIds instead of referencing the anchor and movingCell directly. The movingCell could be located either before or after the anchor, but the variables "first" and "last" prevent any errors that could arise from this ambiguity.

TblHilightCell

```
PROCEDURE TblHilightCell(VAR table: CellTable;
                         cell: CellId);
```

Purpose and Operation

This procedure, given a CellTable and a CellId, it draws the appropriate outline around a cell, based on the value of hilightKind.

TblHilightTable

```
PROCEDURE TblHilightTable(VAR table: CellTable);
```

Purpose and Operation

This procedure draws the cursor in the currentCell, inverts any selected range of cells, and highlights all cells in the table that require highlighting.

NOTE: The variable table.hilightOn stores whether the highlighting is on or off. You can call TblHilightTable repeatedly, and the table will remain hilighted each time (rather that inverting from highlighted to unhighlighted). Do NOT erase the window while the table thinks its highlight is on, however.

TblInitTable

```
PROCEDURE TblInitTable(colPerScreen,rowPerScreen,
                   chPerLine,linesPerField: Integer;
                   topLeftMargin,
                   fieldGap: Point;
                   VAR table: CellTable;
                   shouldAlloc: Boolean;
                   editable: Boolean;
                   commands: keys);
```

Purpose and Operation

TblInitTable initializes and formats the CellTable it receives as an argument. Every cell within the initialized CellTable will be identical, with a uniform number of characters and lines in a field.

The procedure sets the current cell to column 1, row 1 of the CellTable, and initializes the cursor to that field, as well. It initializes the constraint and visible areas to the bounds of the entire table. It sets table.headingRows and table.headingCols both to zero.

Parameters

colPerScreen	The number of vertical columns per table.
rowPerScreen	The number of rows per table.
chPerLine	The maximum number of characters allowed in each line of a field.
linesPerField	The maximum number of lines allowed in each field -- especially useful for producing multi-line fields.
topLeftMargin	The top left margin of the top left cell. Its x component is the horizontal distance in pixels from the left window bound to the top left pixel position of the top left field. Its y component is the vertical distance from the top window bound down to the top left pixel position of the top left field.
fieldGap	The distance in pixels between fields, as they are displayed on the screen. The x component is the horizontal distance between field boundaries; the y component is the vertical distance.
table	The CellTable to be initialized by this procedure.
shouldAlloc	A parameter to the procedure which specifies whether memory space should be allocated for the field values. If shouldAlloc = allocText, the procedure will allocate the space; if shouldAlloc = dontAllocText, it sets the text pointer to nil and doesn't allocate any space. If you

	allocate the text after you've initialized the table, be sure to allocate strings with a maximum width no longer than the width of the columns in the table (chPerLine is the column width).
editable	A Boolean variable that determines whether or not all fields in the table can be edited by the user. The fields are allocated as editable or non-editable regardless of the value of shouldAlloc. You can modify the editable property of any individual field by changing table.screen^[column]^[row]^.kind.editable.
commands	The set of legal command Keys for this particular table. Names for the Keys in the set can be found in the Keys.inc.

TblInvertRange

```
PROCEDURE TblInvertRange(VAR table: CellTable);
```

Purpose and Operation

This procedure inverts the current selection range, either a range of cells or a range of text within a single field. A range is a rectangular span of cells that has been selected by the user. Nothing will happen if the procedure is called and no range has been selected. (To see whether a range has been selected, examine the variable table.anchor: if table.anchor.pos = nowhere, then no range has been selected.)

TblInvertSpan

```
PROCEDURE TblInvertSpan(VAR table: CellTable;
                        col1, col2, row1, row2: Integer);
```

Purpose and Operation

This procedure, given a span of cells, inverts the displayed cell of each field within the span. Spans are rectangular areas defined by column and row parameters. TblInvertSpan will invert the additional selections when a user scrolls during a selection.

TblNewScreen

```
FUNCTION TblNewScreen(colCount: Integer): ScreenPtr;
```

Purpose and Operation

This function returns a pointer value to a screen array with the given number of columns, colCount.

TblScroll

```
PROCEDURE TblScroll(VAR table: CellTable; ch: Char);
```

Purpose and Operation

This procedure is used for the "easy case" and scrolls the view of the table in the direction indicated by ch (left arrow, right arrow, up arrow, or down arrow), and updates the display. It also updates visible and constraint so that they match the displayed area. TblScroll uses bitblt software for rapid scrolling.

TblScrollAdjustCellId

```
PROCEDURE TblScrollAdjustCellId
                               (VAR table: CellTable;
                                VAR unscrolled,
                                    scrolled: CellId);
```

Purpose and Operation

This routine transforms an "unscrolled" CellId that is relative to the display screen into an absolute "scrolled" CellId, according to these formulas:

```
     scrolled.col = unscrolled.col + (scrollCell.col - 1)
     scrolled.row = unscrolled.row + (scrollCell.row - 1)
```

It returns the adjusted result in the variable "scrolled."

Example

When scrollCell = [3,4], the absolute CellId of the top left cell in the table display (screen^[1]^[1]) is col = 3 and row = 4. TblScrollCell would map unscrolled = [1,1] into scrolled = [3,4].

TblSetCurrentCell

```
PROCEDURE TblSetCurrentCell(VAR table:CellTable;
                               col, row: Integer);
```

Purpose and Operation

This procedure sets CellTable.currentCell to the given column and row of the cellTable. This routine will change the position of the cursor and the highlighted cell. The display will change only when another procedure redraws the table, however.

TblSetVisible

```
PROCEDURE TblSetVisible(VAR table: CellTable);
```

Purpose and Operation

This procedure adjusts table.visible and table.constraint so that they lie within the table.visibleRect clipping rectangle. Before you call TblSetVisible, set table.visible.top and table.visible.left to the top left CellId that you want to be visible on the screen. (They are both initialized to 1 by TblInitTable.) TblSetVisible calculates the other values based upon these two.

When TblSetVisible executes, it will adjust table.visible.right and table.visible.bottom so that visibleRect is filled with cells or portions of cells. Portions of cells can be visible, too; the cells are clipped at the boundary of the visibleRect clipping rectangle.

The procedure adjusts the top, bottom, left, and right of table.constraint as well. Constraint is based upon the number of entire cells that can fit within visibleRect. TblSetVisible does not allow constraint to contain cells that appear only partially on the screen. This restriction ensures that the cursor and cell outline can move into entire cells only.

Note that if table.visible overlaps table.headingCols or table.headingRows, the constraint area will be even smaller. Heading Columns or Rows can be visible, but the cursor and cell outline cannot move into them -- so the constraint area must be correspondingly smaller.

TblStartSelection

```
PROCEDURE TblStartSelection(VAR table: CellTable; ch: Char);
```

Purpose and Operation

This procedure puts the table into command mode and sets table.commandChar to ch. It works the same as if the ch character had been included in the set of keys (in table.commands) passed to TblInitTable, and then TblEditTable was called later with that character. In both cases, highlighting of selections is enabled.

TblUnhilightTable

```
PROCEDURE TblUnhilightTable(VAR table: CellTable);
```

Purpose and Operation

This procedure, given a cell table, erases the cursor, uninverts any range of selected cells, and removes the highlighting from any highlighted cells. The cursor is erased graphically only, so you must reset it elsewhere with TblSetCursor.

TblUnhilightTable also depends upon table.hilightOn for its status -- see the note under TblHilightTable.

TblUpdateRect

```
PROCEDURE TblUpdateRect(VAR table: CellTable;
                         VAR rect: Rectangle);
```

Purpose and Operation

This procedure updates the cells that lie within a rectangle defining a portion of the display window. Given an area on the screen, it allows you to update only a portion of the table. It is useful for redrawing the table after a message, a menu, or a form has been displayed.

TimeToString

```
FUNCTION TimeToString(format : Byte;
                         epoch : TimeType) : StringPtr;
```

Purpose and Operation

Converts time and date information from the OS to a string for easy use in an application.

Parameters

format	Currently ignored; it will be used to arrange the string information differently.
epoch	Data that the OS returns from the system clock.

```
TYPE TimeType = RECORD
                  year : WORD;
                  month, day : BYTE;
                  hour, minute, second : BYTE;
                  tenthOfSec, dayOfWeek : BYTE;
                  dayOfYear : WORD;
                END;
```

Returns

The string is of the form: '20-Jan-83 11:00 am'. Other forms may be available at a later date.

TranslateHeading

```
FUNCTION TranslateHeading(inputStr : StringPtr;
                                width : Integer;
                              pageNum : Integer) : StringPtr;
```

Purpose and Operation

This routine translates the input into a centered output string for printing on an Epson printer. Special symbols are translated in the upper or lower case. The output string will be no wider than the width parameter, regardless of the number of symbols included or the length of the input string.

```
Special symbols: ^P   (page number)
        ^D   (date)
        ^T   (time)

 ie. a call to TranslateHeading with parameters:
     inputStr = 'Some file name   ^D   ^T  ^P'
     width    = 40
     pageNum  = 5
     will return a string looking like:

     'Some file name    5/09/83   8:33 am    5'
```

UnDoDataForm

```
PROCEDURE UnDoDataForm (VAR dataForm : DataFormType;
                            eraseDataForm : BOOLEAN);
```

Purpose and Operation

This procedure deallocates all the tables and internal structures associated with a data driven form. It does not, however, free the strings in the form (You must use FreeStringsInDataForm). UnDoDataForm should always be called after DataFormConfirmed.

Parameters

dataForm	The form whose tables and internal structures are to be deallocated.
eraseDataForm	This Boolean determines whether the form will be erased after it has been confirmed. If set True, the form is erased: this is the technique used by most GRiD applications. If special circumstances dictate, you can leave a form displayed after it has been confirmed by setting this parameter False.

UpperCase

```
FUNCTION UpperCase(ch: Char): Char;
```

Purpose and Operation

This function converts any lowercase alphabetic characters in the string to uppercase. It does not shift up numerals, punctuation, or special characters.

APPENDIX A: INCLUDE FILES

Include files are tools for the development environment. The content of each include file is a PUBLIC section of Pascal (or PLM) code that describes the interface to a corresponding Pascal (or PLM) module.

The structure of the files varies from that of the Pascal module for the sake of symbol table space in the compiler. Constants and types are usually included in one file, with functions and procedures in another. This allows easy reference for types that are defined in terms of other constants (parameters).

This structure makes the number of files necessary for successful compilation larger, but it saves on symbol space if the total number of included symbols is smaller in the end. This restriction on symbol space in the compiler has been improved with the latest release of the Intel compiler. The present file convention, however, will stand.

Lastly, the interface is purely for the use of the Pascal compiler. It should not be used as an External Reference Specification, or associated documentation. Writing programs that interface with external modules requires knowledge of the operations and their effects on the private data structures of a module. Much like a programming language is an implementation of a grammar, so include files are only a definition of an interface.

BEFORE COMPILING

To use Common Code routines, your source code must refer to the Common Code Include Files listed in Table A-1. This table lists all the include files for the Common Code -- the files that contain declarations of data types, functions, and procedures.

You include files with the $INCLUDE statement, as described in the PASCAL-86 User's Guide. The files must be available on-line during a compilation.

You do not have to include all of the files listed in Table A-1. Your source program generally needs to include only the procedures that it calls. For example, an application that uses only the window graphics routines and the string routines would include only WindowProcs.inc~Text~ and StringProcs.inc~Text~. (It would also need to include the data types defined in StringTypes.inc~Text~ and WindowTypes.inc~Text~.)

However, some packages refer to the data types defined in other packages. For instance, the Menu/Form routines need the data types defined in several other packages. Figure A-1 illustrates these dependencies. A package at one level requires all the data types defined on the level below it. For example, a program containing messages or prompts would require these include files:

```
MessageProcs.inc~Text~
MessageTypes.inc~Text~
FieldTypes.inc~Text~
StringTypes.inc~Text~
```

Routines	Include File Names
General	Common.inc~Text~
	Keys.inc~Text~
	Math.inc~Text~
String	StringTypes.inc~Text~
	StringProcs.inc~Text~
	RealStringProcs.inc~Text~
Field	FieldTypes.inc~Text~
	FieldProcs.inc~Text~
Table	TableInitTypes.inc~Text~
	TableInitProcs.inc~Text~
	TableEditTypes.inc~Text~
	TableEditProcs.inc~Text~
Data Driven Menu/Form	DataForms.Inc~Text~
File Form	FileFormProcs.Inc~Text~
	FileFormTypes.Inc~Text~
Menu/Form	MenuFormTypes.inc~Text~
	MenuFormProcs.inc~Text~
Common Properties	CommonPropsProcs.Inc~Text
	CommonPropsTypes.Inc~Text
Commands	CommandProcs.Inc~Text~
Message/Prompt	MessageTypes.inc~Text~
	MessageProcs.inc~Text~
Byte Manipulation	ByteProcs.inc~Text~
Fonts	FontProcs.Inc~Text~

Table I-1. Common Code Include Files

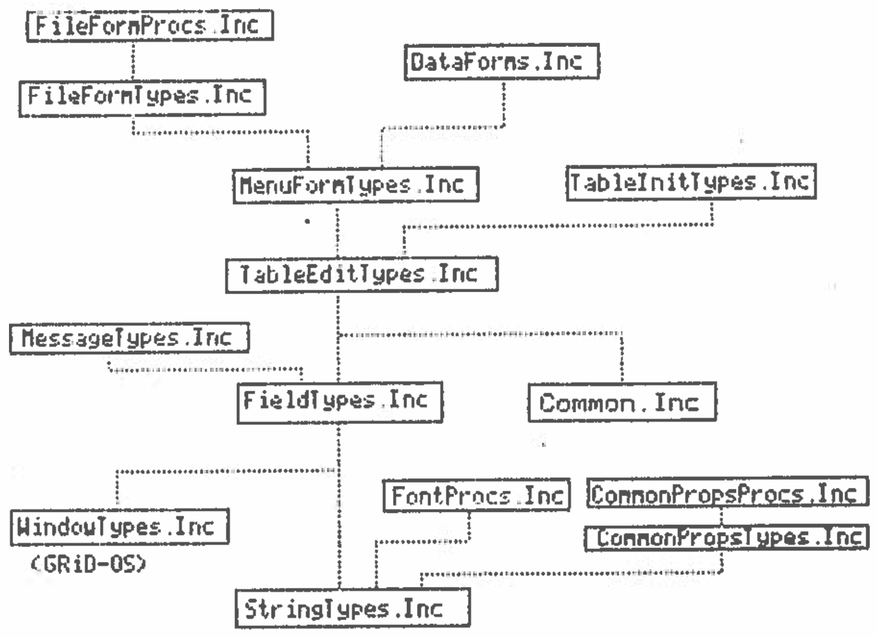


Figure A-1. Include File Hierarchy

APPENDIX B. LISTINGS OF DATA DRIVEN MENU/FORM EXAMPLES

This appendix lists the PLM and Pascal source modules and link command statements (as they might be entered with the GRiDDevelop "Link" token) used for the examples of data driven menus and forms described in Chapter 8. The files and link command for the menu example appear first, followed by those for the form example.

PLM MODULE FOR EXAMPLE "MENU"

```
$COMPACT NOLIST

MenuPLM: DO;

$INCLUDE ('w'Incs'PlmLits.Inc~Text~)

/*************** Sample menu ***************/

DCL sampleMenuTemplate (*) BYTE PUBLIC DATA
    ('Save this file~',
     'Exchange for another file~',
     'Include a file~',
     'Write.to a file~',
     'Append a file~',
     'Erase a file~',
     'Show characteristics of a file~!');

DCL theSampleMenu PTR PUBLIC DATA (@sampleMenuTemplate);

END;
```

SOURCE FILE FOR EXAMPLE PROGRAM "MENU"

```
$NOLIST COMPACT
MODULE  Main;
$INCLUDE ('w0'Incs'Common.Inc~text~)
$INCLUDE ('w0'Incs'ConPas.Inc~text~)

$INCLUDE ('w0'Incs'DataForms.Inc~text~)

$INCLUDE ('w0'Incs'FieldTypes.Inc~text~)
$INCLUDE ('w0'Incs'FieldProcs.Inc~text~)

$INCLUDE ('w0'Incs'MessageTypes.Inc~text~)
$INCLUDE ('w0'Incs'MessageProcs.Inc~text~)

$INCLUDE ('w0'Incs'StringTypes.Inc~text~)
$INCLUDE ('w0'Incs'StringProcs.Inc~text~)

$INCLUDE ('w0'Incs'WindowTypes.Inc~text~)
$INCLUDE ('w0'Incs'WindowProcs.Inc~text~)

$INCLUDE ('w0'Incs'OsPasTypes.Inc~text~)

PUBLIC MenuPLM;
  VAR theSampleMenu: DataMenuType;

PROGRAM Main;
CONST
  { miscellaneous strings }
      sampleMsg     = 'Sample: ';
      selectMsg       = ' Select item and confirm ';
VAR
    windowRect:   Rectangle;
    cursor:       CursorDescriptor;
    msg:          MessagePtr;
    ch:           CHAR;

{------------------------------------------------------------------------------
}
PROCEDURE InitDisplay;
VAR windowExtent: Point;
BEGIN
  ConDefCsr (FALSE);
  WinInitDefaultWindow;
  WinGetWindowExtent (windowExtent);
  FldStartKeys (cursor);
  msg          := MsgInit;

  WITH windowRect DO { entire window for use with menus and forms }
    BEGIN
```

```
      topLeft.x := 0;
      topLeft.y := 0;
      extent    := windowExtent;
    END;
END;

$EJ
{---------------------------------------------------------------------------
}
PROCEDURE SampleMenu;
VAR str: StringPtr;
    rect: Rectangle;
    itemSelected: INTEGER;
    confirmed: BOOLEAN;
BEGIN
  str       := ConcatLits (SampleMsg, SelectMsg);
  rect      := windowRect;
  confirmed := DataMenuConfirmed
                 (theSampleMenu,
                  msg,
                  str,
                  rect,
                  cursor.keyProcess,
                  itemSelected,
                  ch);

  IF confirmed THEN
    BEGIN
      CASE itemSelected OF
        1: ;  {do appropriate action for 'save'}
        2: ;  {do appropriate action for 'exchange'}
        3: ;  {do appropriate action for 'include'}
        4: ;  {do appropriate action for 'write'}
        5: ;  {do appropriate action for 'append'}
        6: ;  {do approptiate action for 'erase'}
        7: ;  {do approptiate action for 'erase'}
        8: ;  {do approptiate action for 'erase'}
        OTHERWISE;
      END;

    END;
END;

{--------------------------------------------------------------------------}
{                  THIS IS THE BEGINNING OF THE PROGRAM
}
{--------------------------------------------------------------------------}
BEGIN
  InitDisplay;
  SampleMenu;
END
.
```

PLM MODULE FOR EXAMPLE "FORM"

```
$COMPACT NOLIST

FormPLM: DO;

$INCLUDE ('w'Incs'PlmLits.Inc~Text~)

/*************** Sample form ***************/

DCL sampleFormItemCount LIT '7';
DCL sampleFormRowSize   LIT '98';  /* 14 times item count */

DCL sampleFormLabelsAndChoices (*) BYTE DATA
    ('#Editable numeric field~An integer~|',
     '?Choice only field~First choice~Second choice~|',
     '$Editable/choice field~A text string~A choice~|',
     '.Editable real number field~A real number~|',
     '&Typeface~|',
     '+Printer~|',
     '=Plotter~|');

DCL theSampleForm STRUCTURE
    (form PTR,
    numItems INTEGER,
    labelsAndChoices PTR,
    choiceLines INTEGER,
    rows (sampleFormRowSize) BYTE)

    PUBLIC DATA

    (nullPtr,                                    /* form         */
    sampleFormItemCount,                         /* numItems     */
    @sampleFormLabelsAndChoices,                 /* items        */
    1);                                          /* choiceLines  */

END;
```

LINK COMMAND FOR EXAMPLE PROGRAM "MENU"

```
:Link Menu:
link 'w'objs'Menu.PLM~Obj~, 'w'objs'Menu.Pas~Obj~,
'w'Libs'CompactSystemCalls~Lib~ TO Menu~Run~ NOPRINT Purge BIND Fastload
SS(STACK (+1000))
```

PASCAL LISTING FOR EXAMPLE PROGRAM "FORM"

```
$NOLIST COMPACT
MODULE  Main;

$INCLUDE ('w0'Incs'Common.Inc~text~)
$INCLUDE ('w0'Incs'ConPas.Inc~text~)

$INCLUDE ('w0'Incs'FontProcs.Inc~text~)

$INCLUDE ('w0'Incs'DataForms.Inc~text~)

$INCLUDE ('w0'Incs'FieldTypes.Inc~text~)
$INCLUDE ('w0'Incs'FieldProcs.Inc~text~)

$INCLUDE ('w0'Incs'MessageTypes.Inc~text~)
$INCLUDE ('w0'Incs'MessageProcs.Inc~text~)

$INCLUDE ('w0'Incs'StringTypes.Inc~text~)
$INCLUDE ('w0'Incs'StringProcs.Inc~text~)

$INCLUDE ('w0'Incs'WindowTypes.Inc~text~)
$INCLUDE ('w0'Incs'WindowProcs.Inc~text~)

$INCLUDE ('w0'Incs'OsPasTypes.Inc~text~)

PUBLIC FormPLM;
  VAR theSampleForm: DataFormType;

PROGRAM Main;
CONST
  { miscellaneous strings }
      sampleMsg      = 'Sample: ';
      fillInFormMsg   = ' Fill in form and confirm ';
      maxStringLength = 80;
VAR
    windowRect:   Rectangle;
    cursor:       CursorDescriptor;
    msg:          MessagePtr;
    ch:           CHAR;
    code:         WORD;

    theNumber:     INTEGER;
    curChoice2:    INTEGER;
    theString:     StringPtr;
    curChoice3:    INTEGER;
    theRealNumber: REAL;
    curChoice4:    INTEGER;
    curFont:       INTEGER;
    curPrinter:    INTEGER;
    curPlotter:    INTEGER;

{--------------------------------------------------------------------------------
}
PROCEDURE InitDisplay;
VAR windowExtent: Point;
BEGIN
  ConDefCsr (FALSE);
  WinInitDefaultWindow;
  WinGetWindowExtent (windowExtent);
  FldStartKeys (cursor);
  msg          := MsgInit;

  WITH windowRect DO { entire window for use with menus and forms }
    BEGIN
      topLeft.x := 0;
      topLeft.y := 0;
      extent    := windowExtent;
    END;
END;


{--------------------------------------------------------------------------------
}
PROCEDURE InitVars;
BEGIN
  theNumber     := 0;
  curChoice2    := 1;
  theString     := NewString (maxStringLength);
  curChoice3    := 2;
  theRealNumber := 5.0;
  curChoice4    := 1;
  curFont       := 1;
  curPrinter    := 2;
  curPlotter    := 2;

WITH theSampleForm DO
 BEGIN
  rows[1].theData.number := theNumber;
  rows[1].currentChoice  := 1;
  rows[2].currentChoice  := curChoice2;
  rows[3].theData.string := ExactCopyOfString (theString);
  rows[3].currentChoice  := curChoice3;
  rows[4].theData.realNumber := theRealNumber;
  rows[4].currentChoice  := curChoice4;
  rows[5].currentChoice  := curFont;
  rows[6].currentChoice  := curPrinter;
  rows[7].currentChoice  := curPlotter;

 END;
END;
```

```
$EJ
{-----------------------------------------------------------------------------
}
PROCEDURE SampleForm;
VAR itemSelected: INTEGER;
    confirmed:    BOOLEAN;
    rect:         Rectangle;
    str:          StringPtr;
BEGIN
  str := ConcatLits (SampleMsg, FillInFormMsg);

  rect := windowRect;

  confirmed := DataFormConfirmed
                (theSampleForm,
                 normalDataForm,
                 msg,
                 str,
                 rect,
                 cursor.keyProcess,
                 ch);

  IF confirmed THEN
    WITH theSampleForm DO
      BEGIN
        FontSetNth (theSampleForm.rows[5].currentchoice, code);
      END;

  UndoDataForm (theSampleForm, TRUE);

END;

{----------------------------------------------------------------------------}
{                   THIS IS THE BEGINNING OF THE PROGRAM
}
{----------------------------------------------------------------------------}
BEGIN
  InitDisplay;
  InitVars;
  SampleForm;
END
.
```

LINK COMMAND FOR EXAMPLE PROGRAM "FORM"

```
:Link Form:
link 'w'objs'Form.PLM~Obj~, 'w'objs'Form.Pas~Obj~,
'w'Libs'CompactSystemCalls~Lib~ TO Form~Run~ NOPRINT Purge BIND Fastload
SS(STACK (+1000))
```

APPENDIX C. MENUS & FORMS: ANOTHER METHOD

The data driven menus and forms techniques described in Chapter 8 greatly simplify implementation of these handy data gathering mechanisms. Previously, a much more complicated technique was used to implement the menus and forms. We will describe this earlier, more complicated technique for two reasons:

- o Some applications were developed before the availability of the data driven technique and associated calls. Users who are already using the earlier technique may find it impractical to convert existing programs to the new technique and may wish you use existing code to implement additional menus and forms.

- o There are some things that you can do with the older technique that are not supported by the data driven technique. For example, you cannot create "dynamic" menus and forms with the data driven technique. A dynamic menu or form is one where you can vary the appearance and contents of the form each time it is presented depending on what activities have taken place since it was last displayed. An example of a dynamic form is the File form. Refer to the description of the File form and the FileFormConfirmed function in Chapter 8 for an illustration of a dynamic form.

The calls described in this appendix create the data structures needed for menus and forms, and enable the user to add or modify their contents.

DATA STRUCTURES

Figure C-1, "Menu/Form Pointer Structure," illustrates how these data structures are related to one another and to the data structures in the chapter on cell tables.

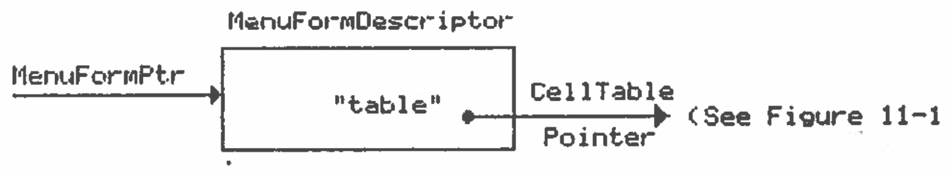


Figure C-1. Menu/Form Pointer Structure

```
* TYPE MenuFormDescriptor =
       RECORD
        table, choiceTable: CellTablePointer;
        obscuredRect,choiceRect: Rectangle;
        choiceLines: Integer;
       END;
```

This record specifies the cell table as either a menu or a form, and stores the appropriate parameters for them. Never alter any of the contents of the MenuForm Descriptor, except to read the CellTable pointer to reference the cell table settings directly.

*** TYPE CellTablePtr = ^CellTable**

This pointer lets you keep track of different cell tables.

*** TYPE MenuFormPtr = ^MenuFormDescriptor;**

This pointer enables you to keep track of different menus or forms at once.

```
* TYPE ChoiceRequest = (choiceCountRequest,
                        choiceCurrentRequest,
                        choiceSetCurrent,
                        choiceLeavingItem,
                        choiceEnteringItem);
```

The menu package uses a variable of this type to specify the choice fields of a form. A variable of this type can represent the different requests as follows:

Kind of request	Meaning
choiceCountRequest	Requests the total number of choices that the form should provide.
choiceCurrentRequest	Requests the string for the specified choice that is associated with a specified item.
choiceSetCurrent	Indicates the choice currently designated by the highlighted box.
choiceLeavingItem	Indicates that the user has moved the outline to a different item.
choiceEnteringItem	Indicates that the outline is about to move to a different item.

```
* TYPE UpdateKind = (dontUpdate, updateTop,
                     updateBottom, updateForward,
                     updateBackward);
```

This data type accompaniew the ScrollKey function used in MenuFormConfirmed. Do not use this data type or erase it from the include files.

MENU AND FORM ROUTINES

Four routines are provided to handle menus and forms. Two routines set up, or initialize, menus and forms, one routine handles the menu or form when it is confirmed, and one routine disposes of a menu or form. While only four routines are provided, several of these routines are quite complex and incorporate a number of subfunctions. Each of the routines is described in complete detail later in this appendix.

MenuInit	Creates and initializes a menu having a single column of choices. It requires the number of items in the menu, the string setting of each item on the menu, and the location on the window where the menu will be drawn.
FormInit	Creates and initializes a form with a non-editable column of items and column of choices, which may or may not be editable.
MenuFormConfirmed	This all-purpose function returns True when the user confirms a menu selection or new values on a form. It returns False if the user escapes out.
MenuFormDispose	Deallocates the menu or form pointed to by the MenuFormPtr. It disposes of everything, including all the text.

Dummy Functions

These functions should be passed as dummy functional parameters to MenuFormConfirmed whenever it controls a menu, or a form with no choice fields. They are never called.

NilChoiceProc	A dummy functional parameter, which should be passed in place of the ChoiceStr functional parameter to MenuFormConfirmed.
NilChoiceInfo	A dummy functional parameter, which should be passed in place of the ChoiceInfo functional parameter to MenuFormConfirmed.
NilItemStr	A dummy functional parameter, which should be passed in place of the ItemStr functional parameter to MenuFormConfirmed.
NilScrollKey	A dummy functional parameter, which should be passed in place of the ScrollKey functional parameter to MenuFormConfirmed.

MenuInit

```
FUNCTION MenuInit(usableRect: Rectangle;
                  itemCount: Integer;
                  FUNCTION ItemStr(index: Integer
                                  ): StringPtr
                 ): MenuFormPtr;
```

Purpose and Operation

This procedure creates and initializes a menu having a single column of choices. It requires the number of items in the menu, the string setting of each item on the menu, and the location on the window where the menu will be drawn. Vertical scrolling can occur when there are too many menu items to fit on the screen. Menus always occupy the full width of the window, but each menu item can take up one line only. At present, the function clips menus wider than the window because horizontal scrolling is not available yet.

Parameters

MenuInit requires these input and output parameters:

usableRect	The window-relative coordinates of the area that the menu can occupy. The usableRect is the maximum area of the window that the menu can take up.
itemCount	The total number of separate menu entries.
FUNCTION ItemStr	A function supplied by the application programmer. Its index parameter represents the nth item on the menu, and the ItemStr function must return the string (i.e., string pointer) corresponding to that nth element of the menu. For example, ItemStr(5) should return a pointer to the string of the fifth item on the menu. The strings returned may have different maximum widths, but strings wider than the window will be clipped. Since each item can take up only a single line, the strings cannot contain embedded carriage returns. The SubStringLit function works well here. WARNING: MenuInit will dispose of the string returned by ItemStr; if necessary, make sure that ItemStr returns only a copy of the string.

Returns

MenuInit returns a MenuFormPtr to the menu that it creates, but it does not draw the menu.

FormInit

```
FUNCTION FormInit
            (usableRect: Rectangle;
             itemCount,
             maxChPerLabel,
             choiceLines: Integer;
             FUNCTION
                ItemStr(col,row: Integer;
                        field: FieldPtr): StringPtr
             ): MenuFormPtr;
```

Purpose and Operation

This procedure creates and initializes a form with a non-editable column of items and column of choices, which may or may not be editable. It needs to know the number of items on the form, the characteristics and setting of each item on the form, and the location on the window where the form will be drawn. See Figure 8-1, "MenuForm Pointer Structure." FormInit returns a MenuFormPtr to the form that it creates, but does not draw the form.

Parameters

FormInit requires these input and output parameters:

usableRect	The window-relative coordinates of the area that the form can occupy. The usableRect is the maximum area of the window that the form can take up.
itemCount	The total number of separate entries, where an entry in a form comprises a item and its choice setting. There are (itemCount x 2) fields in a form.
maxChPerLabel	The maximum number of characters in each item, or equivalently, the width of the item column in characters. Note that the items cannot take up more than one line.
choiceLines	The maximum number of lines that the list of choice can occupy. FormInit interprets the value of choiceLines as follows:

= 0 A band of space for the choices is NOT allocated or displayed. You should set choiceLines = 0 when no value fields have choices, so that the empty choice space will not be displayed.

= 1 The choices will be displayed as a horizontal list on a band of space above the form. If all of the choices cannot be displayed on the screen at once, horizontal scrolling becomes available automatically.

> 1	If choiceLines is any positive integer greater than 1, FormInit will display the choices vertically, one choice on a line, up to the maximum number of lines specified by choiceLines. Each choice can take up one line only; the function clips any choice wider than the window. If there are too many choices to fit in the vertical area defined by choiceLines, then the function automatically enables the choices to scroll vertically.

FUNCTION ItemStr — A function supplied by the application. It should accept the column, row, and field pointer of a field in the form, and then return the string that is the item (if it is a item field) or the default value (if it is a setting field). (No multi-line fields or strings containing carriage returns are allowed.) FormInit also enables the user's ItemStr function to change the editable and choice properties of each field. ItemStr must accept these parameters:

col	The column number of the desired field.
row	The row number of the desired field.
field	The pointer to the field designated by col and row. (Since it's a pointer, the modified FldPtr is an implicit output of ItemStr, as well.) FormInit passes this pointer to the user's ItemStr function, allowing the application to modify the editable and choice properties of each field. ItemStr can modify these properties by altering field^.kind.editable and field^.kind.choice, which otherwise default to False (non-editable, non-choice) in FormInit. Note that it would be meaningless to define a item field as an editable or choice field, because the user should never be able to move into a item field, by definition.

The ItemStr function returns a StringPtr to the string of the desired field. It will return the string containing the item (if it is a item field) or the default value (if it is a setting field). (The string should not contain embedded carriage return or line feed characters.) If a setting field has no default setting, ItemStr should return an empty string with the maximum permissible length defined for that field.

MenuFormConfirmed

```
FUNCTION MenuFormConfirmed
         (menuForm: MenuFormPtr;
          keyProcess: Word;
          FUNCTION ItemStr(col, row: Integer;
                           field: FieldPtr;
                           ): StringPtr;
          FUNCTION ChoiceStr(col, row: Integer;
                             choice: Integer;
                             ): StringPtr;
          FUNCTION ChoiceInfo(col, row, choice: Integer;
                              request: ChoiceRequest
                              ): Integer;
          FUNCTION ScrollKey (ch: Char): UpdateKind;
          VAR selection: Integer;
          VAR ch: Char
          ): Boolean;
```

Purpose and Operation

This all-purpose function returns True when the user confirms a menu selection or new values on a form. It returns False if the user escapes out.

Parameters

MenuFormConfirmed requires these input and output parameters:

menuForm	A pointer to the menu or form to be edited.
FUNCTION ReadKey	A function supplied by the application programmer. It enables the programmer to specify any function that reads a character from the console or any other input device. The function should return data of type Char. For example, ConCharIn would be a typical function.
FUNCTION KeyPressed	A function supplied by the application programmer. The programmer can specify any function to detect whether a console key has been pressed (or, for other types of input devices, to detect whether the input buffer contains a character). The function must return a boolean value. ConKeyPressed is a typical function to supply here.
FUNCTION ItemStr	The same function that was passed to FormInit, above. It should accept the column, row, and field pointer of a field in the form, and then return the default values of the setting fields. It acts here only to supply the default settings of editable-choice fields.

NOTE: IF YOU ARE EDITING A MENU OR A FORM WITH NO

EDITABLE CHOICE FIELDS, YOU MUST SUBSTITUTE A NIL FUNCTION FOR THIS FUNCTIONAL PARAMETER. The Function NilItemStr, given below, should be included as a parameter here instead.

ItemStr has these parameters:

col	The number of the desired column.
row	The number of the desired row.
field	The pointer to the field designated by col and row.

The ItemStr function returns a StringPtr representing the default value to the string of the desired field. If the setting field has no default value, ItemStr should return an empty string with the maximum permissible length defined for that field.

FUNCTION ChoiceStr

A function supplied by the application programmer that returns a character string when it receives parameters that specify the column and row of the form, along with the choice.

NOTE: THE ChoiceStr FUNCTION PARAMETER IS NEEDED ONLY FOR FORMS WITH CHOICE FIELDS. IF YOU ARE EDITING A MENU OR A FORM WITH NO CHOICE FIELDS, YOU MUST SUBSTITUTE A NIL FUNCTION FOR THIS FUNCTION PARAMETER. The function NilChoiceProc, given below, should be included as a parameter here instead.

These are the parameters of ChoiceStr:

col	The column number of the desired field. NOTE: the ChoiceStr function will never be called for the item column (col = 1). As implemented now, it calls the function only with col = 2.
row	The row number of the desired field.
choice	The number of the choice for the given column and row for a setting field, if the field is a choice field. The program that calls ChoiceStr will never specify choice for a non-choice field.

Given the above references to the fields of a form, the ChoiceStr function should return pointers to these strings:

Reference to:	String returned:
item field	The item's string value
setting field--choice	The string value of the current choice
setting field--editable	The string value of the field, a user's typed response
setting field-- choice or editable	The string value of the current choice, which can include a user's typed response

FUNCTION ChoiceInfo

Written by the application progammer, the function must return information about the choice field of any column, row, or choice of a form.

NOTE: THE ChoiceInfo FUNCTION PARAMETER IS NEEDED ONLY FOR FORMS WITH CHOICE FIELDS. IF YOU ARE EDITING A MENU OR A FORM WITH NO CHOICE FIELDS, YOU MUST SUBSTITUTE A NIL FUNCTION FOR THIS FUNCTION PARAMETER. The function NilChoiceInfo, given below, should be included as a parameter here instead.

The ChoiceInfo function accepts these parameters:

col	The column number of the item which the outline is about to enter.
row	The row number of the item which the outline is about to enter.
choice	The number of the choice that the user made. It can be undefined or equal to zero when request = choiceCountRequest or choiceCurrentRequest.
request	These values of request determine what information is returned by ChoiceInfo: request = choiceCountRequest ChoiceInfo should return the total number of choices for the item of the form specified by "col" and "row".

request = choiceCurrentRequest
ChoiceInfo should return the number of the current choice for a given item. (The application is reading its choice value for the given item and passing it to the menu or form.)

request = choiceSetCurrent
The "choice" parameter represents the number of the choice designated by the highlighted box. The choice is associated with the item specified by "col" and "row". The application should assign the value of "choice" to its own variable representing the choice values. The functional value that ChoiceInfo returns is ignored.

request = choiceLeavingItem
The user has moved the outline from one item to another. The application should check the values of "col" and "row" to see which item the user has moved out of. This is useful if the application needs to perform numeric conversions or error checking before allowing the user to confirm the form.

request = choiceEnteringItem
Receiving this request means that the outline is about to move to a different item (even if the next item has no choices). This lets your application validate the setting of the current item before moving the outline to the next item.

FUNCTION ScrollKey	This function is intended to implement "difficult case" scrolling of choices in a form. However, the ScrollKey operation is not available for use currently. For the present, you must include the NilScrollKey function (described later under Dummy Functions) in all your calls to MenuFormConfirmed.
VAR selection	An output, it indicates which item the user selected from the menu. With a form, it returns the item (not the choice) that the user had moved to before confirming the form.
VAR ch	The character that removed the user from the menu or form. These characters include other CODE- commands, CONFIRM, ESC, and cancel (CODE-ESC). The application can thus respond differently when the user quits, aborts, escapes, etc.

Returns

MenuFormConfirmed returns a Boolean: it indicates that the user confirmed the menu or form and its values (= True) or that the user aborted or escaped the menu or form without changing any of the data values.

NilChoiceProc

```
          FUNCTION NilChoiceProc(col,row: Integer;
                                 choice: Integer): StringPtr;
```

Purpose and Operation

A dummy functional parameter, which should be passed in place of the ChoiceStr functional parameter to MenuFormConfirmed. It returns a StringPtr to nil.

NilChoiceInfo

```
FUNCTION NilChoiceInfo(col,row,choice: Integer;
                     request:ChoiceRequest): Integer;
```

Purpose and Operation

A dummy functional parameter, which should be passed in place of the ChoiceInfo functional parameter to MenuFormConfirmed. This function always returns a value of zero.

NilItemStr

```
FUNCTION NilItemStr(col,row,choice: Integer;
                 field: FieldPtr): StringPtr;
```

Purpose and Operation

A dummy functional parameter, which should be passed in place of the ItemStr functional parameter to MenuFormConfirmed. It returns a StringPtr to nil.

NilScrollKey

```
FUNCTION NilScrollKey(ch: Char): UpdateKind;
```

Purpose and Operation

A dummy functional parameter, which should be passed in place of the ScrollKey functional parameter to MenuFormConfirmed. This function always returns dontUpdate.

MenuFormDispose

```
FUNCTION MenuFormDispose(menuForm: MenuFormPtr):
                                         MenuFormPtr;
```

Purpose and Operation

This function deallocates the menu or form pointed to by the MenuFormPtr. It disposes of everything, including all the text. There is no dontDisposeText option for the text strings because each instance of a menu or form is unique. MenuFormDispose always returns nil so that the MenuFormPtr to the disposed menu or form can be assigned the value of nil.