GRiDTask User's Guide

November 1985

COPYRIGHT (C) 1985 GRiD Systems Corporation
2535 Garcia Avenue
Mountain View, CA 94043
(415) 961-4800

Manual Name: GRiDTask User's Guide
Order Number (Manual): 021250-44
Order Number (GRiDTask Software): 021250
Issue date: November 1985

The information in this document is subject to change without notice.

NEITHER GRiD SYSTEMS CORPORATION NOR THIS DOCUMENT MAKES ANY EXPRESSED OR IMPLIED WARRANTY, INCLUDING, BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, OR FITNESS FOR A PARTICULAR PURPOSE. GRiD Systems Corporation makes no representation as to the accuracy or adequacy of this document. GRiD Systems Corporation has no obligation to update or keep current the information contained in this document.

UNDER NO CIRCUMSTANCES WILL GRiD SYSTEMS CORPORATION BE LIABLE FOR ANY LOSS OR OTHER DAMAGES ARISING OUT OF THE USE OF THIS MANUAL.

GRiDTask Manual
Table of Contents

Appendices

Chapter 1

Overview

1.1 What Is GRiDTask ? GRiDTask is an interpreted programming language: it is used to create custom software systems built around GRiD Management Tools or other GRiD-OS applications. GRiDTask customized applications enhance the power of GRiD software by presenting an interface in the context of the user's business. GRiDTask applications can take advantage of every capability of GRiD's Management Tools software. At the same time, the user interface can be in familiar terms and so simple that the user may need very little training to quickly learn the system.

In addition, GRiDTask can be used to automate repetitive tasks such as accessing a mainframe, downloading data, and displaying the data in a graph. With a custom GRiDTask application, the entire sequence can be started with a single keystroke. GRiDTask is also well-suited for creating tutorials and sales presentations.

Features of GRiDTask

o A GRiDTask application can control any software running under the GRiD-OS operating system, handling the interface to multiple applications. The interface can be designed so that the user merely makes selections in a menu or fills in a form and the GRiDTask application does the rest.

o GRiDTask displays messages, text, graphics, and GRiDPaint "canvas" images to supply information to the user. The user can make choices or supply information in GRiDTask menus, forms, or the familiar GRiD file form.

o GRiDTask controls windows: the application appears in one window while GRiDTask menus, forms, messages, and graphics appear in another window. The screen can be filled completely with either the GRiDTask window or the application window, or can be divided between the two windows.

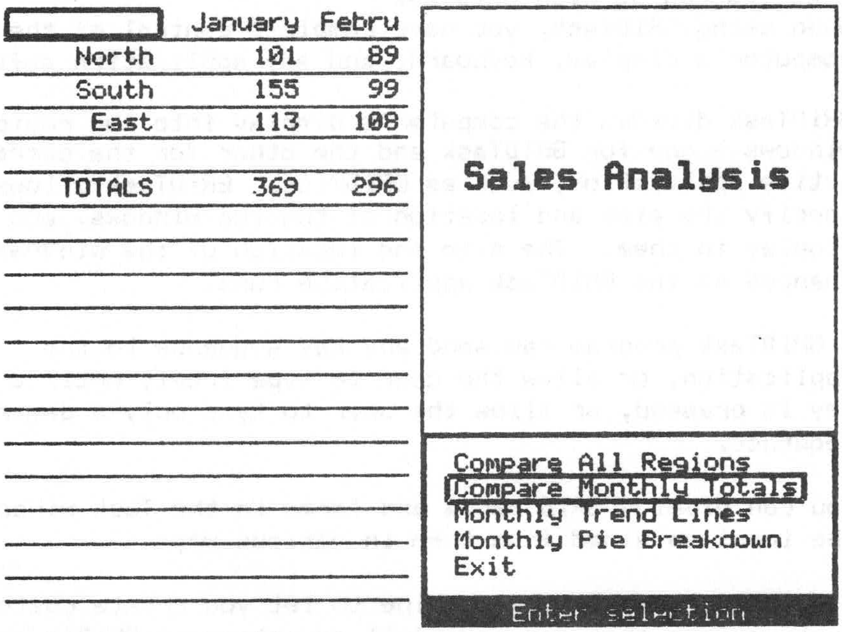
o GRiDTask supplies a set of flow control commands and a procedure definition capability. GRiDTask applications can be written in separate modules controlled from a main program. Modules are smaller and easier to develop and maintain, allowing new applications and enhancements to be developed quickly.

o GRiDTask provides sophisticated commands for manipulation of strings and real numbers.

o The GRiDTask language can be extended to perform special functions outside the normal scope of GRiD Management Tools. To do this, procedures written in languages such as Pascal or PLM can be "installed" as part of the GRiDTask language.

GRiDTask Example

This uses GRiDPlot to display sales data. Figure 1-1 shows the screen prior to a selection from the menu. Figure 1-2 displays the graph drawn after selecting "Compare Monthly Totals". After the selection is made, GRiDTask sends "keystrokes" to GRiDPlot to select rows and columns of data, set graph options, and draw the graph. With GRiDTask, the user does not need to understand all the commands within GRiDPlot to draw a useful graph.



	January	Febru
North	101	89
South	155	99
East	113	108
TOTALS	369	296

Figure 1-1. GRiDTask Application Using GRiDPlot

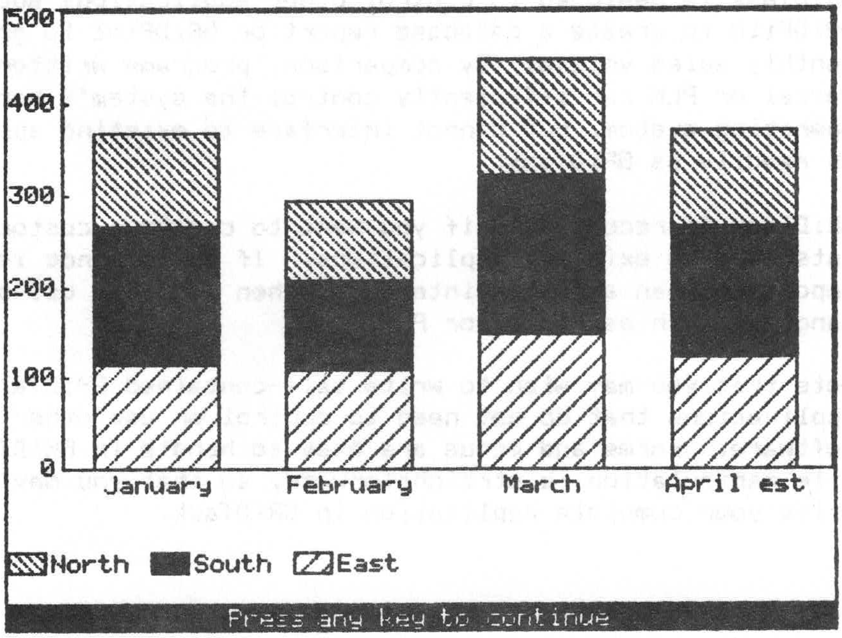


Figure 1-2. Graph Created by Selecting "Compare Monthly Totals"

What you can do with GRiDTask
When using GRiDTask, you have complete control of the computer's display, keyboard, and any application software.

GRiDTask divides the computer's display into two regions, or windows - one for GRiDTask and the other for the currently active application, such as GRiDPlot. GRiDTask allows you to specify the size and location of the two windows, and what to display in them. The size and location of the windows can be changed as the GRiDTask application runs.

A GRiDTask program can send any key sequence to the application, or allow the user to type freely until a specified key is pressed, or allow the user to type only a desired key sequence.

You can display GRiD menus and forms in the Task window and use the input obtained from them in various ways.

All these capabilities combine to let you create custom applications that can take full advantage of GRiD's software, yet require practically no training to use.

Using GRiDTask vs. Other Programming Languages
GRiDTask is designed to control other applications such as GRiDFile to create a database report or GRiDPlot to graph monthly sales volume. By comparison, programs written in Pascal or PLM can efficiently control the system's hardware and operating system, but cannot interface to existing applications as readily as GRiDTask.

GRiDTask is recommended if you want to create a custom interface to existing applications. If performance is more important than a custom interface, then consider using a language such as Pascal or PLM.

Note that you may wish to write self-contained GRiDTask applications that do not need to control or use other GRiD software. Forms and menus are easy to handle in GRiDTask, and file manipulation is straightforward, so that you may wish to write your complete application in GRiDTask.

1.2 Who Should Read This Manual

This manual is for anyone writing an application using GRiDTask. These include custom applications, tutorials, and presentations.

GRiDTask is a relatively easy programming language to learn and use. If you have had programming experience with high level languages such as Pascal or Fortran, you should have no difficulty learning GRiDTask. If you have had little or no programming experience then you may benefit from some assistance. This manual assumes familiarity with common programming concepts.

Anyone using GRiDTask should also be an accomplished user of GRiD software - GRiDWrite, GRiDPlot, GRiDPlan, etc.

1.3 Software and Hardware Required For Writing a GRiDTask Program

Software Required to Develop GRiDTask Applications
GRiDTask is an interpreted language. GRiDWrite is used to write the application, and the GRiDTask interpreter is used to run the application. GRiDTask files to be interpreted by GRiDTask have the kind "Task".

GRiDTask requires version 3.1.0 or later of GRiD software.

To write a GRiDTask application you need GRiDTask, GRiDWrite, and any other GRiD applications that you wish to control. If you want your program to display graphic images, you may need GRiDPaint to create or modify those images.

Requirements for controlled application programs An application must meet certain requirements in order to work properly with GRiDTask.

Specifically, the application should

- Run under the GRiD-OS operating system.
- Run independently of window size.
- Process the WindowUpdate key properly.

All GRiD software meets these requirements. Any user-written software meeting these criteria can also be run with GRiDTask.

Hardware Required
GRiDTask runs on any computer that supports the GRiD-OS operating system. GRiD computer models with larger RAM space - 512K - and/or ROM capability support more powerful GRiDTask applications.

GRiDTask uses a significant amount of RAM memory, so you should check that enough RAM is available for GRiDTask, the application, and any data files required. Currently GRiDTask requires a minimum of about 40K of RAM.

A GRiDTask program running on one computer will operate on any other computer running GRiD-OS. Note that there are two possible exceptions to this: 1) due to RAM requirements, some GRiDTask applications that run on a 512K RAM machine, for instance, may exceed memory on a 256K RAM machine, 2) different computer models have different screen sizes, so your GRiDTask programs may need some modifications to allow for this. GRiDTask programs can be written to adjust automatically for different screen sizes.

Chapter 2

GRiDTask Concepts

This chapter covers concepts necessary to use GRiDTask effectively. It will be very helpful to read this entire chapter before trying to design or implement a GRiDTask application.

2.1 How GRiDTask Interacts with Applications GRiDTask can manipulate other applications and execute commands within them. One application at a time can be controlled from GRiDTask by sending keystrokes to the application. For example, if you want to start GRiDFile from within a GRiDTask program, the GRiDTask program fills in the File Form with the name of a database file and provides a "confirm" keystroke. This is the standard way to start an application from GRiDTask - by filling in the File form, just as a user would. You may elect not to show the File Form on-screen, but your GRiDTask program must still fill in the File form and provide the "confirm".

To send keys to an application from a GRiDTask program, the ADDKEYS statement is used. The ADDKEYS statement is followed by a list of the keys that you want passed to the application.

A GRiDTask program can also extract information from an application. For example, the CELL$ function returns the value of the cell currently outlined in a table, such as in GRiDPlan.

2.2 Windows A "window" is an area of the screen that an application uses to display its output. GRiD-OS applications usually have one window which occupies the entire screen. GRiDTask applications can have two windows: the application window and the Task window. Window areas are specified within the GRiDTask program, and can be changed as the Task program runs.

The application window displays standard GRiD application software, such as File forms, GRiDPlan or GRiDMaster. The Task window may display a variety of items - menus, text, graphics, etc.: you control these in the GRiDTask program.

Three methods are used to separate the two windows.

- The Task window is surrounded by a one pixel wide frame. The application window is not surrounded by a frame.

- There is a 4 pixel gap between the two windows.

- The two windows can use different fonts.

The Task window
You can set the size and location of the Task window using the TASKWINDOW verb. The application window is automatically placed in

the largest space outside the Task window, whether that is above, below, or to the left or right of the Task window.

Note that either the Task window or the application window may use the entire available screen at a given time. In this case, the other window is not visible to the user and may still do everything that it normally does. This may be used to "hide" the application (or the Task program) from the user.

The contents of the Task window
A GRiDTask program can display the following items in the Task window.

- Menus
- Forms
- File forms
- Data-entry forms
- Text
- Messages
- Canvas images
- Graphics

The information obtained from the user via menus, forms, and File forms can be used as variables within the GRiDTask program.

Text displayed in the Task window can be in any font, or in multiple fonts at a given time. GRiDTask maintains an invisible cursor that marks the next position to display text.

You can display Canvas images created in GRiDPaint. These may include images created using Screenwatch and changed to Canvas files in GRiDPaint. Also, graphic images can be "drawn" in the Task window using Task verbs.

The Application Window
Within GRiDTask, the size of the application window is set to the remainder of the available screen after the Task window is set. The application display can be redrawn to fit the allotted area. Applications essentially run the same under GRiDTask as they do normally.

Note that even if the application is "hidden" (the entire window is the "Task window"), or only partially displayed, it still operates as it normally does.

Changing the size of windows
As the author of a GRiDTask program you need to keep track of the location and size of the two windows and what is being displayed in them. This requires knowledge of how an application reacts when its window has changed.

When an application starts running, one of its first actions is to "ask" GRiD-OS how big its window is. The application formats its output for proper display given these dimensions.

As an example of what this means, imagine that GRiDPlot's window uses all of the available screen. Then in your Task program, you reduce its window to the top half of the available screen, and the Task window displays text or graphics in the bottom half. At first, GRiDPlot is not aware that its window has been reduced to half its original size. It keeps running, and everything that would normally be visible in the bottom half of the window is cut-off. Pie charts become semi-circles, and most of the menus and forms are not visible at all. There is nothing wrong with this, but it may not be what you want. To correct this, your Task program can tell GRiDPlot to recheck the size of its window using the UPDATESCREEN verb.

The **UPDATESCREEN** verb
When used in a GRiDTask program, the **UPDATESCREEN** verb sends a special key value (WindowUpdateKey) to the application window. This key value tells the application program to recheck its window size.

2.3 Keys and the Keyboard

Just as you must keep track of where your windows are and what is being displayed in them, you must also keep track of the state of the keyboard.

Where do the keystrokes go?
Under normal operation, when an application is running without being controlled under GRiDTask, everything you type on the keyboard goes directly to the application. When an application is running under GRiDTask, either the GRiDTask window or the application window can receive keys.

When GRiDTask is running, it has control of the keyboard; anything you type will go to GRiDTask first. What GRiDTask does with the keys is controlled by your GRiDTask program. You can ignore the keys from the keyboard, you can use them to get input from the user, or you can pass them on to the application window.

Four ways to control the application
Since the application window is never connected directly to the keyboard, all of its "keys" come from the GRiDTask program. There are four ways that the GRiDTask program can send keys to the application.

o **PAUSE**
The PAUSE verb instructs GRiDTask to pass all keys typed on the keyboard directly to the application. In effect, PAUSE temporarily passes control of the application to the user. However, GRiDTask continues to watch the incoming keys. When a specified "termination" key is pressed, GRiDTask stops passing keys to the application window and begins to receive the keystrokes itself.

- **PASSKEYS**

  The PASSKEYS verb is a variation of the PAUSE verb. The PASSKEYS verb passes any of the keys in a selected group of keys to the application, until one of the termination keys is pressed. The keys to pass and the termination keys are specified with the PASSKEYS verb. (note that the PAUSE verb passes all keys until a termination key is pressed.) With the PASSKEYS verb, control of the application is "handed off" to the user until pressing a termination key, when control of the application is passed back to GRiDTask. Note that the passed keys can be limited so that certain functions normally available to a user of a GRiD application are blocked. Thus a user could be allowed to look around within a database file, but might be prevented from changing the data.

- **ADDKEYS**

  The ADDKEYS verb automatically passes a specified sequence of keys to the application window. The ADDKEYS verb does not interact with the physical keyboard at all. However, ADDKEYS gives a result similar to pressing the keys on a keyboard while running an application.

- **TESTKEYS**

  TESTKEYS is intended to be used in tutorials. The TESTKEYS verb functions similarly to the ADDKEYS verb except that it requires the user to press all the specified keys before they are passed to the application. The required keys are displayed highlighted in the Task window, and they un-highlight as they are pressed.

2.4 The File Form File forms operate somewhat differently when a GRiDTask program is running.

When any application tries to display a File form, GRiDTask prevents it from being displayed until a FILEFORM statement is executed in the GRiDTask program. A FILEFORM statement specifies the file name used to fill in the File form. This circumvents the normal method of filling in a File form by typing on the keyboard.

The advantage of this is that even if the position of a file within a subject changes, the File form will still be filled in correctly. Refer to Chapter 4 for more information on the FILEFORM verb.

Chapter 3

Language Constructs

3.1 Variables

Variable Types

GRiDTask supports two types of variables: Real numbers and Strings.

Real number variables have values with fifteen digits of precision. These values can be as large or small as numbers having exponents between -308 and +307. Note that when assigning values to GRiDTask real variables, numbers are written without using exponential notation. An example of a real variable value is:

```
RealVariable = 355.339664
```

Note that real variables can have integer values.

```
e.g.  RealVariable = 9
```

There are functions that can turn a Real variable value into an integer value. These are the ROUND and TRUNC (truncate) functions.

String variables represent a sequence of characters from 0 to 65,535 characters long. Each character in a string is associated with a value from 0 to 255. See Appendix C for a list of these values. There are several verbs within GRiDTask used to convert between string variable values and real variable values. See the descriptions of CHR$, STR$, and VAL in Chapter 4.

GRiDTask does not support arrays or any other types of structured variables. Note that you may use "installed" verbs that provide such capabilities. See Appendix H for more details.

GRiDTask allows real variables to be used in boolean expressions, as follows: a real number is converted to an integer. If the number is even, it is "false", and if it is odd it is "true". Boolean expressions can be used with IF statements to control the execution of GRiDTask statements. There are two built-in functions - the GRiDTask verbs "FALSE" and "TRUE" - which also can be used in boolean expressions. See Chapter 4 for more information.

Variable Names

Variable names can be any sequence of alpha and numeric characters. The first character of a variable name must be an alpha character (a letter). Characters after the first may be either alpha or numeric.

String variables must end with a dollar sign ($).

```
e.g. " StringX$ "
```

The following characters are not permitted in variable names:

= ( ) , ; : " + - * ^ / \ and "space"

You must also avoid using any of the reserved words listed in Appendix E when naming variables. If you use any of these words an error may occur.

Variable Declarations

GRiDTask does not require variable names to be explicitly declared. You can create a new variable by using it as the destination of an assignment statement. For example:

```
  topping$ = "pineapple slices"
```

This statement creates a new string variable topping$ and assigns an initial value to it - "pineapple slices". If topping$ has previously been assigned a value, that value is lost.

Variable Scope

Most variables in GRiDTask are defined globally. This includes branching to another program file with the TASK verb. Within a procedure you can declare variables that are local to the procedure. See Appendix H for more details.

3.2 Constants

String Constants

String constants are any text enclosed in double quotes. Here are three valid string constants:

```
"I've been to the moon and back."
"This constant
 is on three
 lines."
"He asked ""Why?"" before I could find the door."
```

The second example illustrates that a string constant can include embedded carriage return-linefeeds (CR-LF). This makes it very important that you remember the second double quote to end the constant. The third example shows that to embed a double quote within a string constant, you use two consecutive double quotes.

Real Constants

Real constants are represented in base 10. They can only include numeric characters and the unary plus and minus sign. Here are three valid real constants:

```
30000            +45.4545            -7.8
```

3.3 Operators and Expressions

String Operators

Strings can be concatenated (added together) or compared. When comparing strings, upper and lower case are not significant. Here are the operators that can be used with string variables and constants, listed in order of precedence:

	Operation	Type of value returned
+	concatenation	string
=	equals	boolean (real)
<>	not equal to	boolean (real)
<	less than	boolean (real)
>	greater than	boolean (real)
<=	less than or equal to	boolean (real)
>=	greater than or equal	boolean (real)

Real Variable Operators

Real variables can be operated on by arithmetic operators and boolean operators. Here are the operators that can be used with real variables and constants, listed in order of precedence:

	Operation	Return
^	exponentiation	real
*	multiplication	real
/	division	real
+	addition	real
-	subtraction	real
MOD	modulo divide	real
=	equals	boolean (real)
<>	not equal to	boolean (real)
<	less than	boolean (real)
>	greater than	boolean (real)
<=	less than or equal to	boolean (real)
>=	greater than or equal	boolean (real)
NOT	not	boolean (real)
AND	and	boolean (real)
OR	or	boolean (real)
XOR	exclusive or	boolean (real)

Expressions

An expression can be used anywhere a string or real value is required. Expressions consist of constants, variables, functions, and operators. Use parentheses for clarity and to specify the order of evaluation. An example of an expression is as follows:

```
Peaches = (2 *  Grapes) + (Apples / 3)
```

3.4 Flow Control Constructs available for flow control within a GRiDTask program include the IF/ELSE/ENDIF and WHILE/WEND verbs.

IF/ELSE/ENDIF statements are used to execute groups of GRiDTask statements conditionally. WHILE/WEND statements provide looping control.

IF/THEN/ELSE blocks may be inserted into WHILE/WEND loops, and WHILE/WEND loops may be inserted into IF/THEN/ELSE blocks.

For more detail concerning IF/ELSE/ENDIF and WHILE/WEND verbs, refer to Chapter 4.

3.5 Branching

GRiDTask supports three different types of branching: procedures, TASK statements, and DO statements.

Procedures
You can define procedures that use parameters and local variables. When the procedure name (and parameters) is encountered in a GRiDTask statement, the procedure is executed. When the procedure execution is complete, GRiDTask returns to the place in the Task application from which the procedure was called.

Procedures provide an efficient way to rapidly execute a set of GRiDTask statements that may be used over and over within a Task application.

TASK Statements
The TASK verb may be used to execute a group of GRiDTask statements contained within a "module", or file that is separate from the main GRiDtask module. A TASK statement causes GRiDTask to execute the specified file, and then return to the statement following the TASK statement.

Typically, a module is a group of GRiDTask statements that have a common purpose and accomplish a given task. Using TASK modules, the GRiDTask programmer can divide the entire GRiDTask application into smaller and distinct operations. The GRiDTask application is thus easier and faster to write and debug.

DO Statements
A string containing a sequence of statements can be executed with a DO verb. This is similar to procedures, but there are several limitations. DO statements are used mostly in special circumstances such as self-modifying code. It is best to avoid the use of DO statements if possible.

Procedures, TASK, and DO are described in Chapter 4. See Appendix I for performance issues concerning these.

Chapter 4

Section One - GRiDTask VERBS

This chapter contains detailed descriptions of the verbs of the GRiDTask language, including functions, procedures and predefined variables. Note that this manual uses the terms "verbs" and "statements". GRiDTask verbs are the commands themselves, such as "PRINT", and a statement is one line in a GRiDTask application that uses a GRIDTask verb or verbs. The chapter is arranged alphabetically and there are some conventions used, as follows:

There is a group of GRiDTask verbs used to perform mathematical operations. These are placed in a section entitled "GRiDTask Real Number Functions" at the end of this chapter.

o GRiDTask verbs appear in capital letters.

e.g. APPENDFILE

o All variable names appear in lowercase letters. If a variable name consists of two or more words, then words after the first one may be capitalized.

e.g. apples itemNumber item$

Special Notes:

o You can continue a GRiDTask statement on a new line by entering an underscore character (_) as the last character in the line. (press RETURN to type the rest of the statement)

o Multiple GRiDTask statements can be placed on one line by separating them with a colon.

o You can place GRiDWrite text formatting commands (e.g., ^ep, ^nl, ^sl, etc.) in a GRiDTask program. GRiDTask ignores lines with a circumflex (^) as the first character.

o Many of the examples in this chapter are shown out of context. As such, they may not run exactly as shown. Also, some of the examples have not been tested.

o Appendix A is a quick-reference list of the verbs described in this chapter (4).

ADDKEYS

ADDKEYS "encodedKeyStr"

NOTES

ADDKEYS provides keystrokes to the application running in the application window. The result of ADDKEYS is similar to actually typing the keys on the keyboard. The parameter string "encodedKeyStr" is the sequence of keystrokes to be passed to the application window. Appendix B explains how keystrokes are encoded for placement in "encodedKeyStr".

ADDKEYS passes one key at a time to the application window, waiting for the previous key to be accepted before sending the next key. The rate at which the keystrokes are passed can be adjusted with the SPEED verb. The initial setting is full speed.

If the application window is trying to display a File form, but is waiting for the FILEFORM verb, then ADDKEYS terminates without passing any remaining keys to the application window.

EXAMPLES

```
FILEFORM "`Bubble Memory`Memos`Call Summary~Text~"
ADDKEYS "¦."              ; Confirm the File form
ADDKEYS "¦e¦V¦.¦t¦."
```

This example retrieves a text file, erases its contents, and then saves the file. The first ADDKEYS statement confirms the File form. The second ADDKEYS statement is equivalent to pressing:

```
     CODE-E                      " ¦e "
     CODE-SHIFT-DownArrow        " ¦V "
     Confirm                     " ¦. "
     CODE-T                      " ¦t "
     Confirm                     " ¦. "
```

To type the vertical bar character in GRiDWrite, press CODE-SHIFT-semicolon.

APPENDFILE

```
APPENDFILE addString$, pathname$
```

NOTES

APPENDFILE adds addString$ to the end of the file specified by pathname$. If the file does not already exist, GRiDTask creates a new one and writes addString$ into it.

Note that if you don't specify a Kind in pathname$, the Kind Text is assumed. If you do not specify a Device or Subject, the current Device and Subject of the last file accessed through GRiDTask or the application window is assumed.

APPENDFILE sets the ERRORCODE variable to the number of any error that occurred. If no error occurs, then the ERRORCODE variable is set to (0) zero.

EXAMPLE

```
newData$                = "A quick brown fox"
destination$            = "‘Hard Disk‘Forms‘Medical~Text~"
APPENDFILE newData$, destination$
```

In this example, the text "A quick brown fox" is appended to the end of text currently in the file "Medical~Text~" in the Subject "Forms" on the Device "Hard Disk".

ASC

```
num = ASC (anyString$)
```

NOTES

ASC returns the decimal ASCII value of the first character in the specified string.

Appendix C contains a table listing the ASCII values associated with each letter and key combination.

EXAMPLE

```
PROCEDURE DisplayAsciiValues theStr$
  i = 1
  WHILE i ≤ LEN(theStr$)
    PRINT STR$(ASC(MID$(theStr$,i,1))) + " "
    i = i + 1
  WEND
ENDP
```

This procedure prints the ASCII values for each character in the specified string.

BREAK

```
BREAK
```

NOTES

Executing a BREAK statement is the equivalent of pressing a break key (specified in a BREAKONKEY statement). The following actions take place:

- If a break key was earlier enabled by a BREAKONKEY verb, the break key is disabled.
- GRiDTask scans forward through the GRiDTask program searching for a BREAKRESET statement.
  - If a BREAKRESET statement is found, the code following the statement is executed.
  - If a BREAKRESET statement is not found in the program, GRiDTask halts execution.

Note that BREAK is independent of BREAKONKEY. BREAK executes whether a BREAKONKEY has been executed or not.

EXAMPLE

```
TASK "operation one~text~"
IF (ERRORCODE <> 0): BREAK: ENDIF
.....
.....
.....
TASK "operation two~text~"
IF (ERRORCODE <> 0): BREAK: ENDIF
.....
.....
.....
BREAKRESET
IF (ERRORCODE <> 0)
 TASK "ErrorHandler~Text~"
ENDIF
```

In this example, two operations are performed via TASK statements. If an error occurs during either of these operations, a BREAK statement is executed. The BREAK statement causes the program to immediately branch to an error handler module.

BREAKONKEY

BREAKONKEY key$

NOTES

BREAKONKEY specifies a keystroke that causes the following actions if pressed by the user:

- The break key is disabled.
- GRiDTask scans forward through the GRiDTask program searching for a BREAKRESET statement.
  - If a BREAKRESET statement is found, the statements following it are executed.
  - If a BREAKRESET statement is not found, GRiDTask halts execution.

key$ is an encoded key that triggers the break, and can be any valid key. See Appendix B for a description of how to encode keystrokes.

To cancel the break key, issue BREAKONKEY with a null parameter as shown below.

```
BREAKONKEY ""
```

EXAMPLES

```
BREAKONKEY "¦z"        ; At the beginning of the TASK application
breakKeyPressed = TRUE
.....
.....
.....
breakKeyPressed = FALSE
BREAKRESET
IF  breakKeyPressed  ; skip if breakKeyPressed = FALSE
                     ;   statement was encountered

         .....
         GRiDTask statements
         .....
ENDIF
```

In this case, if the user presses "¦z" then GRiDTask scans forward to BREAKRESET without executing the statement "breakKeyPressed = FALSE". Then breakKeyPressed is TRUE, so the statements after BREAKRESET are executed. However, if GRiDTask encounters BREAKRESET by sequential execution, then breakKeyPressed is FALSE and the statements following BREAKRESET are not executed.

BREAKRESET

BREAKRESET

NOTES

BREAKRESET marks the beginning of statements to be executed after a BREAK has been encountered, or if the user presses a break key (specified in a BREAKONKEY statement).

BREAKRESET disables any active break key (specified in a BREAKONKEY statement). Thus, after a BREAKRESET statement is executed, another BREAKONKEY statement must be executed to reactivate the break key. For convenience, an example from the BREAKONKEY verb description is printed here.

EXAMPLE

```
; Enable CODE-Z as the break key.
  BREAKONKEY "!z"
           .....
           GRiDTask statements
           .....
; GRiDTask statements to execute when "!z" is pressed
  BREAKRESET
           .....
           GRiDTask statements
           .....
```

In the above example, when the user presses CODE-z, GRIDTask scans forward to BREAKRESET, and the statements following it are executed. Note that these statements will be executed if GRiDTask encounters them directly, even if the user has not pressed the break key.

CELL$

contents$ = CELL$

NOTES

CELL$ is a string function which returns the contents of the current cell in the application window. The value returned is always a string, even if the cell is displaying a number.

The current cell is the cell containing the blinking cursor.

CELL$ is a method of retrieving information from a worksheet, database, or other table.

To retrieve the contents of a cell in a worksheet:

1. Retrieve the worksheet and GRiDPlan.
2. Move the cell outline to the desired cell.
3. Use CELL$ to return the contents of the cell.

CELL$ works in a GRiDPlan worksheet table, a GRiDFile database table, a GRiDPlot table, and any other application that displays a table, such as GRiDMaster. You can also use it to retrieve cell definitions from a worksheet by moving the cursor to the cell definition area.

CELL$ cannot be used to return the value of an item in a menu or form.

If you execute CELL$ when a table is not being displayed or when the cursor is not blinking in a cell, a GRiDTask error occurs.

See the next page for an example.

EXAMPLE - CELL$

```
;----------------------------------------
;  retrieve database file
;----------------------------------------
FILEFORM DEVICE$ + SUBJECT$ + "Employee Salaries~Database~"
ADDKEYS "¦."

;----------------------------------------
;  query database for a particular employee
;  the employee's name is in the variable "name$"
;----------------------------------------
ADDKEYS "¦fA = """ + name$ + """¦."

;----------------------------------------
;  extract salary from column C
;----------------------------------------
ADDKEYS "¦Q¦Q" ;two SHIFT right arrows
salary$ = CELL$

;----------------------------------------
;  display results
;----------------------------------------
IF LEN(salary$) = 0
  PRINT name$ + " doesn't work here."
ELSE
  PRINT name$ + " makes " + salary$ + " dollars per month."
ENDIF
```

This example illustrates how to use the CELL$ function to extract an employee's salary from a database. The following steps occur:

1) First, the Task program retrieves the database file.

2) Then the Task program does a query for a particular employee name. The program assumes that column A of the database contains employee names, and the name of the desired employee is already in a variable called "name$".

3) After completing the query, the program moves the cell outline to column C which contains the employee's salary.

4) The salary is retrieved with the CELL$ function, and displayed on the screen.

CENTER

```
CENTER "text"
```

NOTES

CENTER displays its string parameter in the Task window in the current font. The text is centered on the line where the GRIDTask invisible cursor is currently located. If the text does not fit on one line, it is displayed starting at the left edge of the Task window, and word-wraps on to the next line(s).

A CR-LF is passed at the end of the CENTER statement. This leaves the cursor at the left edge of the window, one line below the centered text.

The rate at which text is displayed by the CENTER verb is controlled by the SPEED verb. The initial setting is full speed.

EXAMPLES

```
CENTER "Welcome to the world of GRiD"
```

This greeting is centered on the line where the cursor is.

```
CENTER "Boy Scouts
+
Girl Scouts
+
4-H
-----------------
Community Health"
```

This example shows how the parameter string to CENTER can contain CR-LFs. This statement prints centered text on seven consecutive lines of the display as shown below.

```
    Boy Scouts
         +
   Girl Scouts
         +
        4-H
-----------------
 Community Health
```

CHANGEKIND$

newPathName$ = CHANGEKIND$ (pathName$, Kind$)

NOTES

CHANGEKIND$ is a string function requiring two string parameters. The first parameter is a file pathname and the second is a file Kind. CHANGEKIND$ creates a new string which is the same as PathName$ except with the new Kind.

EXAMPLE

```
;-----------------------------------------
; Reformat Historical Quotes Text File
;-----------------------------------------
textFile$ = GETFILE$("Select Text file to be reformatted")
FILEFORM "Historical Quotes~Reformat~00"
ADDKEYS "¦.¦t¦."
FILEFORM textFile$
ADDKEYS "¦."
graphFile$ = CHANGEKIND$(textFile$, "Graph")
FILEFORM graphFile$ + "21"       ; get new file and application
ADDKEYS "¦.¦."
```

This program reformats a text file of data that has been retrieved from a mainframe. It writes the reformatted data to a graph file.

1) It starts by asking the user to fill in a File form selecting the text file to be reformatted.

```
textFile$ = GETFILE$("Select Text file to be reformatted")
```

2) It then retrieves a Reformat file.

```
FILEFORM "Historical Quotes~Reformat~00"
ADDKEYS "¦.¦t¦."
```

3) It specifies the text file as the file to be reformatted.

```
FILEFORM textFile$
ADDKEYS "¦."
```

4) It specifies the output file as having the same name as the input text file except with a kind of "Graph". It writes the new graphfile, then brings it into GRiDPlot.

```
graphFile$ = CHANGEKIND$(textFile$, "Graph")
FILEFORM graphFile$ + "21"       ; get new file and application
ADDKEYS "¦.¦."
```

CHARWIDTH

```
width = CHARWIDTH
```

NOTES

CHARWIDTH is an integer-value function which returns the width of the current font in the Task window. The width is measured in pixels.

EXAMPLE

```
PRINT "This is your first message"
DELAY 2
PRINT "Your first message used 26 characters,"
PRINT "so it is " + STR$(26 * CHARWIDTH) + " pixels wide"
```

In this example, the width of the first message (printed in the current font) is calculated using CHARWIDTH.

CHR$

```
stringX$ = CHR$ (num)
```

NOTES

The CHR$ function converts a number (0-255) to a one-character string.

Appendix C contains a table showing the numbers associated with each key or key combination (one-character strings) that can be pressed on the keyboard.

EXAMPLES

Example 1:

```
stringX$ = "A"
stringX$ = CHR$(65)
```

These two statements are equivalent. They both create a one character string containing the letter "A".

Example 2:

```
codeZ$ = CHR$(250)
WHILE CONCHARIN$ <> codeZ$
WEND
```

This WHILE/WEND loop continues until a CODE-Z is pressed on the keyboard.

```
CLEARMSG
```

NOTES

CLEARMSG removes any messages currently showing in the Task window. It does not affect the application window.

EXAMPLE

```
; Beginning of Task program
PROCEDURE PauseForKey
  STACKMSG "Press any key to continue"
  PAUSE ""
  CLEARMSG
ENDP
.....
; Later in the program
PauseForKey
.....
```

When the procedure "PauseForKey" is executed, it displays a message and then waits until any key is pressed. It then removes the message.

CLEARSCREEN

CLEARSCREEN

NOTES

CLEARSCREEN erases the entire Task window, including any messages. It positions the cursor four pixels below and four pixels to the right of the upper left corner of the Task window.

EXAMPLE

```
CURSOR 10,150
PRINT "Welcome to the World of"
PRINT "  Portable Computers   "
PAUSE ""
CLEARSCREEN
PRINT " Welcome Back"
```

These statements position the cursor 150 pixels down from the top and 10 pixels from the left edge of the screen. Then the words "Welcome to the World of" and "Portable Computers" are printed starting at this point, and GRiDTask waits until the user presses a key (PAUSE). Then the entire Task window is cleared (CLEARSCREEN), and the message "Welcome Back" is printed at the top of the screen.

COMMANDLINE

```
COMMANDLINE command$, time
```

NOTES

COMMANDLINE executes the command specified in command$. These are Command Line Interpreter (CLI) commands. "time" tells GRiDTask the number of seconds to continue executing the command. Time may be meaningful for some commands and not for others, as discussed below.

Although the COMMANDLINE statement can execute any program, its primary purpose is to execute the following types:

o Utilities such as LADT, ACTIVATE, and COPY, which have no user interface.

o Visual graphic programs, such as SPIRAL, CLOCK, and VECTORS. These are popular in presentations.

Note that programs run with COMMANDLINE execute and display within the Task window. COMMANDLINE parameters are specified as follows:

command$ — This contains the CLI command and its parameters, where applicable. The format and rules for specifying these commands are given in Appendix C of the GRiD Program Development Guide. Note that if the Device, Subject, or Title names contain blanks, the pathname must be enclosed in single quotes, as shown below.

```
COMMANDLINE "’ ‘Hard Disk‘Demos‘Vectors’" , 5
```

time — Time is used for programs such as SPIRAL and VECTOR, which run continuously until CODE-ESC is pressed. Time indicates the number of seconds the program is to run. At the end of the alloted time, GRiDTask automatically halts the program by passing it a CODE-ESC sequence.

Specify 0 for programs such as COPY and ACTIVATE that halt automatically after they execute. In all cases, GRiDTask continues after the program stops running.

EXAMPLES

```
TASKWINDOW 0,0,-1,-1
COMMANDLINE "Vectors", 4
PRINT "Let’s look at some other capabilities"
```

This causes Vectors to run in the Task window. After four seconds, Vectors is ended and GRiDTask continues with the next Task statement.

```
COMMANDLINE "Activate modem", 0
```

This activates the modem. GRiDTask continues with the next Task statement after completing the "Activate modem" command.

Comment

```
; comment text ...
```

NOTES

Any text following a semicolon is considered to be a comment and will not be executed. The comment extends to the end of the current line. Comments can be on lines of their own as well as following a statement. Blank lines are considered to be comments.

A semicolon embedded in a string constant is not considered a comment.

EXAMPLE

```
;---------------------------------------
; Ask them if they want a printed copy
;---------------------------------------
PRINT "Confirm to get a printed copy of this report"

IF ConCharIn$ = CHR$(141)    ; check for a Confirm
  ADDKEYS "¦t¦V¦.¦."
ENDIF
```

This example prints a text file if the user confirms. It contains several comments and a blank line. The first three lines in the example are comment lines, and there is a comment at the end of the "IF ConcharIn$ =..." line.

CONCHARIN$

```
ch$ = CONCHARIN$
```

NOTES

CONCHARIN$ (CONsole CHARacter INput) is a string function that waits until a key is pressed on the keyboard, and then returns that key as a one-character string.

INKEY$ is a similar function but does not wait for a key to be pressed.

EXAMPLE

```
CENTER "Please press the A or B key"
ch$ = CONCHARIN$   ;wait until the user presses a key

IF (ch$ = "A")
  PRINT "Thanks for the A"
ELSE
IF (ch$ = "B")
  PRINT "Thanks for the B"
ELSE
  PRINT "You weren't listening!"
ENDIF:ENDIF
```

This program prompts the user to press "A" or "B". It then executes CONCHARIN$, which returns the next key pressed. If the user presses upper or lower case "A", then one message is displayed. If the user presses upper or lower case "B", then another message is displayed. If neither "A" nor "B" is pressed, then a third message is displayed.

COPYFILE

COPYFILE sourcePath$, destinationPath$

NOTES

COPYFILE duplicates the file with the path name sourcePath$ to a file with the path name destinationPath$. The source file remains intact. If the destination file doesn't exist, it is created. If the destination file already exists, then its data is overwritten.

Note that if no Kind is specified in either pathname, then the Kind Text is assumed. If no Subject or Device is specified, then the Device and Subject are assumed to be the same as the Device and Subject of the last file accessed through GRiDTask or the application window.

COPYFILE sets the ERRORCODE variable to the number of any error that occurred. If no error occurs, then the ERRORCODE variable is set to (0) zero.

EXAMPLE

```
old$ = "Medical~Text~"
new$ = "'Hard Disk'Forms'Dental~Text~"
COPYFILE old$, new$
```

This copies the contents of Medical~Text~ in the current Device and Subject to the file Dental~Text~ in the Subject Forms on the Device Hard Disk.

CURSOR

```
CURSOR  x, y
```

NOTES

The CURSOR verb repositions the invisible cursor. When GRiDTask displays text with the PRINT and CENTER verbs, it uses an invisible cursor to determine where to start displaying the text.

The two parameters represent the new x and y coordinates of the cursor. Location 0,0 represents the upper-left corner of the Task window.

EXAMPLE

```
spacing = 15                    ; pixel distance between text lines
CENTER "Today's menu"
CURSOR 0, CURY + spacing
CENTER "Lasagna and salad"
CURSOR 0, CURY + spacing        ; CURSOR statement #2
CENTER "Lamb curry"
CURSOR 0, CURY + spacing
CENTER "Fried Chicken"
```

This program prints four lines in the center of the Task window. Each line is separated by a 15-pixel gap. By changing the variable "spacing", you can easily change the spacing of all the menu items.

Note that after CURSOR statement #2, the cursor has been moved down four times - two times by the CENTER statements and a total of 30 pixels by the two CURSOR statements.

CURX & CURY

```
distanceX = CURX
distanceY = CURY
```

NOTES

CURX and CURY are functions which return the current location of the cursor in pixel coordinates. CURX is the horizontal distance from the left edge of the Task window to the cursor, and CURY is the vertical distance from the upper edge of the Task window to the cursor.

EXAMPLES

```
.....
CURSOR  CURX - 10, CURY + 15
.....
```

This statement moves the cursor 10 pixels to the left and 15 pixels down from its current position, whatever that is. If the cursor starting position is beyond the edge of the Task window, then printed text may not appear.

DATE$

```
today$ = DATE$
```

NOTES

DATE$ is a string function that returns the date.

The date string is formatted as mm/dd/yy (month/day/year).

If the date is not correct, use GRiDManager to set the correct time and date.

EXAMPLE

```
ADDKEYS "Today is " + DATE$ + "."
```

If GRiDWrite is running in the application window, then this statement types a line containing the date into the text file. The line of text might look like this:

```
  Today is 05/15/85.
```

DELAY

DELAY seconds

NOTES

DELAY causes the GRiDTask program to wait for the number of seconds specified.

The maximum delay is 65,535 seconds (about 18 hours). The minimum delay is 0 (no delay).

DELAY is often used in presentations.

EXAMPLE

```
CLEARSCREEN
PRINT "Benefits of Grapefruit Juice vs. Orange Juice
"
DELAY 2
PRINT "      *  More nutritious
"
DELAY 2
PRINT "      *  Less expensive
"
DELAY 2
PRINT "      *  Less sticky"
DELAY 10
```

In this program example, DELAY statements are used between the bulleted items. This causes the Task program to wait two seconds before displaying the next item. This gives the viewer time to read each item before displaying the next one.

DEVICE$

```
dev$ = DEVICE$
```

NOTES

DEVICE$ is a string function which returns the current Device. This string includes leading and trailing backquotes. The current Device (and Subject) are the same as the Device (and Subject) of the last file accessed by GRiDTask or the application window.

DEVICE$ is used to make Task programs run independently of the Device and Subject where the Task program is stored. Because the current Device and Subject may change as the GRiDTask program executes, you may want to save the values (corresponding to the Device and Subject of the Task program) using DEVICE$ and SUBJECT$ statements at the beginning of the GRiDTask program.

EXAMPLES

```
; This is the beginning of the Task program
TaskDevice$  = DEVICE$
TaskSubject$ = SUBJECT$
.....
.....
; This is later in the Task program
PAINT 10,10, TaskDevice$ + TaskSubject$ + "Dali~Canvas~"
```

Here, DEVICE$ and SUBJECT$ are executed at the beginning of the Task program. At this time, the Device and Subject are the same as those of the Task program. Since the Canvas image is known to be in the same Device and Subject, the string variables TaskDevice$ and TaskSubject$ can be used to access it. If the current Device is the Hard Disk, then TaskDevice$ contains the string "`Hard Disk`".

```
FILEFORM DEVICE$ + "Programs`GridWrite~Run Text~"
ADDKEYS "!."
```

This instructs GRiDTask to look on the current Device for GRiDWrite.

DIRECTORY$

list$ = DIRECTORY$ (mode, path$, match$, delimiter$, sortOrder)

NOTES

DIRECTORY$ is a string function which returns a list of either Devices, Subjects, Titles, or Titles with Kinds. Each item in list$ is separated by the string specified in delimiter$.

DIRECTORY$ parameters are as follows:

mode is an integer value - either one, two, three, or four. Mode specifies the information to be returned as follows:

Mode	**Items Returned**
1	List of Devices
2	List of Subjects
3	List of Titles
4	List of Titles with Kinds (Title~Kind~)

path$ specifies either the Device, or the Device and Subject from which list$ is determined. path$ depends on mode as follows:

Mode	**Path**	
1 Devices	Null	("")
2 Subject	Device only	(e.g., "‘Hard Disk‘")
3 Titles	Device and Subject	(e.g., "‘Bubble Memory‘memos‘")
4 Titles and Kinds	Device and Subject	(e.g., "‘Bubble Memory‘memos‘")

match$ is a string of one or more characters that describes either the Devices, Subjects, or Titles to be listed in the directory. Wildcard characters can be used to request all files with a common feature, such as all Titles in a given Subject. To specify wildcard characters, use an assignment statement such as: match$ = "S" + CHR$(247)

247 is the decimal equivalent of the wildcard character (CODE-w). See Appendix B for a table of character equivalents.

delimiter$ is a string of one or more characters that DIRECTORY$ uses to separate directory items in list$.

sortOrder specifies the order in which the directory entries are to appear in list$.

1 specifies ascending order

2 specifies descending order

EXAMPLE - DIRECTORY$

```
; Set parameter constants for DIRECTORY$
mode            = 3
path$           = "`Bubble Memory`Programs`"
match$          = CHR$(247)                     ; Wildcard = match all
files
delimiter$      = "¦"
sortOrder       = 1

list$ = DIRECTORY$ (mode, path$, match$, delimiter$, sortOrder)
```

In the above example, a list of all the Titles in the Subject Programs in the Bubble memory is returned.

Setting mode = 3 requests a list of Titles. Since the mode = 3, path$ must specify the Device and Subject in which GRiDTask should look for Titles. Since match$ is the wild-card character, all Titles in path$ are returned. The Titles are separated from each other with "¦", the character specified by delimiter$. Finally, the list of Titles is returned in ascending order.

DO

```
DO code$
```

NOTES

DO executes one or more GRiDTask statements in the string parameter code$. If code$ does not contain valid statements, then a GRiDTask error occurs. Note that procedures can be used in GRIDTask, and are superior to DO statements in most cases. See the description of procedures for more information. It is recommended to use DO statements only in special situations where procedures cannot be used.

EXAMPLE

```
code$ = " PRINT ""What is your sign?"" "
DO code$
```

In this example, the message "What is your sign?" is printed on the screen.

DOFORM$

formCopy$ = DOFORM$ (msg$, form$, choiceLines)

NOTES

DOFORM$ is a string function that displays a standard GRiD form in the Task window. DOFORM$ is used with the FORMCHOICE and FORMCHOICE$ functions to obtain information from the user.

"msg$" is a string to be displayed as a message at the bottom of the form.

"form$" is a string containing all the information needed to display the form. "form$" contains the following items.

Item names which appear on the left side of the form.

Choices for each item.

Current settings for each item.

"choiceLines" is an optional parameter. If it is omitted, then DOFORM$ displays choices in a horizontal "choice band". If choiceLines is used, it specifies the number of lines to use in displaying the choices: choices are then displayed in a vertical column.

The "form$" string uses four special characters:

tildes	~
vertical bars	¦
"at" signs	@
carats	^

Tildes separate choices from item names and from other choices. Vertical bars indicate the end of a list of choices. For example, the string -

```
lunchForm$ = "Sandwich~Ham~Cheese~BLT¦Drink~Milk~Soda~Juice¦"
```

- displays a form with two items. The name of the first item is "Sandwich" and the name of the second item is "Drink". The choices for the "Sandwich" item are "Ham", "Cheese", and "BLT". The choices for the "Drink" item are "Milk", "Soda", and "Juice".

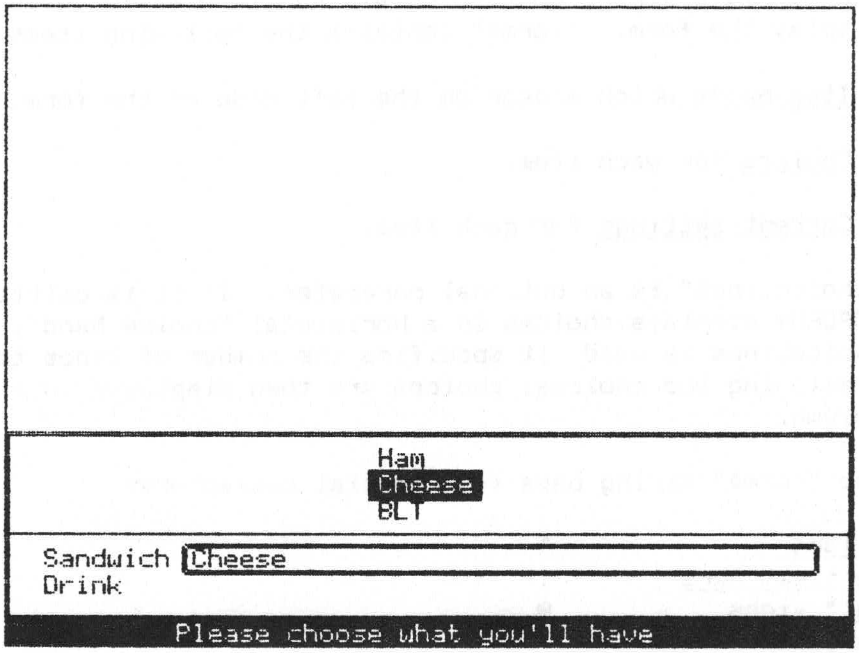
Placing an "at" sign (@) before a choice causes the choice to appear as the initial setting for its item. Placing a carat (^) at the beginning of a list of choices makes the item editable.

For example, the string

```
lunchForm$ = "Sandwich~Ham~@Cheese~BLT¦Drink~^~Milk~Soda~Juice¦"
```

displays the same two-item form discussed above. The "at" sign (@) causes "Cheese" to be the initial choice for the first item. The carat (^) makes the second item - "Drink" - an editable-choice item. See Figure DOFORM$-1 for an illustration of the form displayed when the DOFORM$ example program is executed.

Figure DOFORM$-1. An Example Form - The Lunch Menu



Sandwich Cheese
Drink

Please choose what you'll have

DOFORM$ returns a copy of the form string parameter modified to show the new settings of the form. Specifically, DOFORM$ inserts "at" signs (@) in the appropriate locations, and if an editable item has been filled in, DOFORM$ inserts the user-specified text after the carat (^). The FORMCHOICE and FORMCHOICE$ functions easily extract the information from the string returned by DOFORM$.

A form does not have to be confirmed. If the form is not confirmed, then DOFORM$ returns the form string (second parameter) unaltered.

The DOFORM$ verb sets the LASTKEY$ variable to whatever key terminated the form. You must check the LASTKEY$ variable to determine if the form was confirmed.

EXAMPLE - DOFORM$

```
TASKWINDOW 0,0,-1,-1
; Display the form
confirm$ = CHR$(141)
msg$         = "Please choose what you'll have"
lunchForm$ = "Sandwich~Ham~@Cheese~BLT¦Drink~^~Milk~Soda~Juice¦"
lunch$ = DOFORM$ (msg$, lunchForm$, 3)

IF LASTKEY$ = confirm$
 ; Extracting information from the form
 sandwich = FORMCHOICE  (lunch$, 1)
 drink$   = FORMCHOICE$ (lunch$, 2)

 ; Instructions to the sandwich chef
 IF sandwich = 1
  PRINT "Slice the ham"
 ELSE
 IF sandwich = 2
  PRINT "Slice the mozzarella"
 ELSE
 IF sandwich = 3
  PRINT "Cook the bacon"
 ENDIF:ENDIF:ENDIF

 ; Responses to the choice of drink
 IF drink$ = "Milk"
  PRINT "That'll be 50 cents please"
 ELSE
 IF drink$ = "Soda"
  PRINT "We have cola and orange soda"
 ELSE
 IF drink$ = "Apple" OR drink$ = "Grape"
  PRINT "Yes, we have " + drink$
 ELSE
  PRINT "We're out of that juice"
 ENDIF:ENDIF:ENDIF
ENDIF
```

This Procedure displays a form asking the user to specify choices for lunch, including a sandwich type and a juice type. It then extracts this information and gives instructions to the chef on preparing the sandwich, and responses the waiter might give the customer based on the desired drink.

The first section of the procedure - "Display the form" - defines and displays this two-item form.

The second section of the procedure - "Extracting information from the form" - uses the FORMCHOICE and FORMCHOICE$ functions to read the values that are in the form.

The third and fourth sections of the procedure - "Instructions to the sandwich chef" and "Responses to the choice of drink" - provide actions that might be taken as a result of different answers given by the user in the form.

DOMENU

```
choice = DOMENU (msg$, item$)
```

NOTES

DOMENU is an integer function that displays a GRiD menu in the Task window.

The first parameter is a string that is displayed as a message below the menu.

The second parameter is the list of items to appear in the menu. Each item is separated by a vertical bar (¦). To enter a vertical bar press CODE and SHIFT and ; together.

DOMENU returns an integer indicating which item was selected. If the first item is selected, DOMENU returns 1. If the second item is selected, DOMENU returns 2, etc.

The DOMENU verb sets the LASTKEY$ variable to whatever key terminated the menu. A menu does not have to be Confirmed. You must check the LASTKEY$ variable to see if the menu was Confirmed.

See the next page for an example.

EXAMPLE - DOMENU

```
CLEARSCREEN
CENTER "Automated Sales Analysis Program"
;-------------------------------------------------------
;                  display main menu
;-------------------------------------------------------
confirm$    =  CHR$(141)
msg$        = "Select activity and Confirm"
salesMenu$ = "Graph regional data¦Mail call summary¦Create
Margin report¦Exit"

WHILE TRUE
  LastKey$ = ""
  WHILE LastKey$ <> confirm$   ; force them to Confirm
    choice = DOMENU (msg$, salesMenu$)
  WEND
;-------------------------------------------------------
;              take appropriate action
;-------------------------------------------------------
  IF choice = 1
    TASK "RegionGraphs~Text~"
  ELSE
  IF choice = 2
    TASK "CallSummary~Text~"
  ELSE
  IF choice = 3
    TASK "MarginReport~Text~"
  ELSE
  IF choice = 4
    STOP
  ENDIF:ENDIF:ENDIF:ENDIF
WEND
```

This example displays a menu with four items.

The WHILE/WEND loop around the DOMENU statement forces the user to Confirm the menu.

If the user selects the fourth item on the menu, the program stops running. If they select any other item, a specified Task file is executed.

ELSE

```
ELSE
```

NOTES

ELSE is used with IF and ENDIF to conditionally control the execution of a block of GRiDTask statements. The statements following ELSE are executed when the condition in the corresponding IF statement evaluated false.

When using an ELSE verb, it must be the only word on the line.

See IF/ELSE/ENDIF for more information.

EXAMPLE

The example from the "FALSE" verb is repeated here for convenience.

```
IF temperature >= 70
   WARM = TRUE
ELSE
   WARM = FALSE
ENDIF
......
......
IF WARM
    TASK "‘w‘GoToBeach‘SantaCruz~Text~"
ELSE
    TASK "‘w‘GoSkiing‘LakeTahoe~Text~"
ENDIF
```

In this example, if the temperature is 70 or above, then WARM is true, and the module "GoToBeach‘SantaCruz~Text~" is executed. If the temperature is < 70, then WARM is false, and the module "GoSkiing‘LakeTahoe~Text~" is executed.

ENDIF

ENDIF

NOTES

ENDIF is used with an IF statement. ENDIF marks the end of the IF block of GRidTask statements.

See IF/ELSE/ENDIF for more information.

EXAMPLE

The example from the "CONCHARIN$" verb is repeated here for convenience.

```
CENTER "Please press the A or B key"
ch$ = CONCHARIN$   ;wait until the user presses a key

IF (ch$ = "A")
  PRINT "Thanks for the A"
ELSE
IF (ch$ = "B")
  PRINT "Thanks for the B"
ELSE
  PRINT "You weren't listening!"
ENDIF:ENDIF
```

This program prompts the user to press "A" or "B". It then executes CONCHARIN$, which returns the next key pressed. If the user presses upper or lower case "A", then one message is displayed. If the user presses upper or lower case "B", then another message is displayed. If neither "A" nor "B" is pressed, then a third message is displayed.

ENDP

```
ENDP
```

NOTES

ENDP is used to mark the end of a procedure. It is the last statement in the body of a procedure.

EXAMPLE

```
; These statements are in module 1
PROCEDURE SEARCH param1, param2
  statement1
  statement2
  .....
ENDP
GRiDTask statement 1
GRiDTask statement 2
.......


; These statements are in Module 2
GRiDTask statement 3
SEARCH  param1, param2
GRiDTask statement 4
.....
```

In this example, ENDP marks the physical end of procedure SEARCH. Note that after finishing the procedure SEARCH, GRiDTask continues execution with GRiDTask statement 4, not GRiDTask statement 1.

ERASEBOX

ERASEBOX topleftX, topleftY, widthX, heightY

NOTES

ERASEBOX erases a rectangle within the Task window. The first two parameters specify the top-left corner of the rectangle and the third and fourth parameters indicate the extent or size of the rectangle. "widthX" indicates the extent in the x, or horizontal direction, and "heightY" indicates the extent in the y, or vertical direction. All the parameters are in pixels. Using -1 for widthX or heightY extends the specified rectangle to the edge of the Task window.

EXAMPLE

```
ERASEBOX WINDOWWIDTH/2 , 0 , -1 , -1
```

This statement erases the right half of the Task window, if the Task window is currently set to the entire available screen.

ERASEFILE

ERASEFILE pathname$

NOTES

ERASEFILE deletes the file with the path name pathname$.

Note that if you do not specify a Kind item in pathname$, the Kind Text is assumed. If you do not specify a Subject or Device, the Device and Subject of the last file accessed through GRiDTask or the application window is assumed (these are termed the "current" Device and Subject).

ERASEFILE sets the ERRORCODE variable to the number of any error that occurred. If no error occurs, then the ERRORCODE variable is set to (0) zero.

EXAMPLES

```
ERASEFILE "CarParts~Database~"
```

This causes the file with Title "CarParts" and Kind "Database" in the current Device and Subject to be erased.

```
ERASEFILE "‘Bubble‘Junkers‘CarParts~Database~"
```

This causes the file with Title "CarParts" and Kind "Database" in the Device "Bubble memory" and Subject "Junkers" to be erased.

ERRORCODE

ErrorNum = ERRORCODE
ERRORCODE = Num

NOTES

ERRORCODE is a predefined number variable. Whenever an application looks up an error message in the file "@SystemErrors", the ErrorCode variable is set to the number of the error. If the file "@SystemErrors" is not present, ERRORCODE still gets set correctly.

The ERRORCODE variable is implemented to allow sophisticated error recovery while running a GRiDTask program. With good design, you can minimize the possibility of an application encountering an error, but in some cases it cannot be avoided. As an example, poor phone lines or incorrect phone numbers may interfere with communications products such as GRiDTerm.

By using the ERRORCODE variable, not only can you take appropriate action based on a detected error, but you can also display detailed instructions explaining what went wrong, and suggesting possible solutions.

Because ERRORCODE is a variable, you can set its value as well as test for its value. You should set ERRORCODE = 0 (no error) before performing an operation in which you suspect an error may occur.

The ERRORCODE variable is also set by the TASK verbs APPENDFILE, COPYFILE, ERASEFILE, READFILE, and WRITEFILE.

EXAMPLE

```
; Print the file
ERRORCODE = 0
ADDKEYS "¦t¦V¦.¦."
; Check for print error
IF ERRORCODE <> 0
  TASK "ErrorHandler"
ENDIF
```

This example prints a file and then checks to see if there were any errors. If an error was encountered it branches to another file which can display help information or take other appropriate actions. See the description of the TASK verb for another example using the ERRORCODE variable.

ERRORSTR$

```
err$ = ERRORSTR$(errorNum)
```

NOTES

ERRORSTR$ is a string function. It returns an error message string corresponding to errorNum in the file "@SystemErrors~Text~".

The "@SystemErrors~Text~" file must be in an accessible "Programs" Subject, otherwise ERRORSTR$ returns a string containing just the error number.

ERRORSTR$ does not affect the ERRORCODE variable.

EXAMPLE

```
IF ERRORCODE <> 0
  STACKMSG ERRORSTR$ (ERRORCODE)
ENDIF
```

This statement displays a message of the last error that occurred in the applications window.

FALSE

```
variable = FALSE
```

NOTES

The function FALSE returns the value 0 (the function TRUE returns the value -1). FALSE and TRUE can be used in boolean expressions.

In boolean expressions an even number is false, and an odd number is true (the low-order bit is used to determine boolean values - an odd number has a low-order bit = 1, and an even number has a low-order bit = 0.

EXAMPLE

```
IF temperature >= 70
   WARM = TRUE
ELSE
   WARM = FALSE
ENDIF
......
......
IF WARM
    TASK "`w`GoToBeach`SantaCruz~Text~"
ELSE
    TASK "`w`GoSkiing`LakeTahoe~Text~"
ENDIF
```

In this example, if the temperature is 70 or above, then WARM is true, and the module "GoToBeach`SantaCruz~Text~" is executed. If the temperature is < 70, then WARM is false, and the module "GoSkiing`LakeTahoe~Text~" is executed.

FILEFORM

FILEFORM "pathname"

NOTES

The FILEFORM verb specifies a pathname to be set in the next File form that appears in the application window.

The format of the FILEFORM parameter is very important. It should be a complete pathname, followed by two extra characters. Here is the format:

```
FILEFORM "`device`subject`title~kind~10"
```

The last two characters determine the settings of the two optional items of the File form: "Next action" and "Save changes".

Their interpretation is shown below.

Next Action	0 -	Keep current file
	1 -	Get new file only
	2 -	Get new file and its application
Save Changes	0 -	Before getting new file
	1 -	No

If the last character of the pathname parameter is a tilde (~), then FILEFORM assumes that you did not append these last two characters and appends two zeros for you.

If the first character of the pathname parameter is not a back quote (`), then FILEFORM assumes that you did not specify a Device and Subject. It searches for the file in the current Device and Subject. If the file isn't found, GRiDTask searches in the current Device and Programs Subject. If it isn't found in either place, an error occurs. As a rule, you should specify the complete pathname of the file, if possible. Searching for a file can take several seconds.

The FILEFORM verb does not cause the File form to appear. It only specifies the default settings for the next File form.

You must perform an ADDKEYS "¦." to confirm the File form once it is displayed. If a second confirm is required to overwrite an existing file or create a new one, then this must also be passed with the ADDKEYS verb.

EXAMPLES - FILEFORM

```
FILEFORM "`Hard Disk`Programs`GRiDWrite~Run Text~10"
```

This statement completely specifies the contents of the next File form. It does not check that GRiDWrite or even the Hard Disk is available. If the next File form has the "Next Action" and "Save Changes" items they will be set to "Get new file only" and "Before getting new file" respectively.

If GRiDWrite was not on the Hard Disk, the File form would say "Confirm to create new file", a condition you probably didn't anticipate. If a Hard Disk was not attached, the application window would show a system error message.

```
FILEFORM "`Hard Disk`Programs`GRiDWrite~Run Text~"
```

This statement is similar to the first. However, because the last character is a tilde, two zeros will be appended to the filename string.

```
FILEFORM "GRiDWrite~Run Text~"
```

This statement's parameter string does not contain a Device and Subject. FILEFORM searches for GRiDWrite in the current Programs Subject. If GRiDWrite isn't found in the Programs Subject, GRiDTask searches in the current Subject. If GRiDWrite is not found there you get an error.

```
FILEFORM DEVICE$ + SUBJECT$ + "Service Revenue~Graph~21"
```

The parameter in this FILEFORM statement is a complete pathname. The DEVICE$ and SUBJECT$ functions provide the Device and Subject portions of the pathname. If you removed the DEVICE$ and SUBJECT$ functions, the FILEFORM statement would still find the file "Service Revenue", but it might take longer.

FINDTITLE$

path$ = FINDTITLE$ ("Title~Kind~")

NOTES

FINDTITLE$ is a string function which returns a complete pathname of a file for which you know only the Title and Kind. If a file with this Title and Kind cannot be found, FINDTITLE$ returns a zero length string.

FINDTITLE$ is useful for making a program independent of the Device and Subject in which it is running. It is also useful for determining if a particular file exists prior to retrieving it.

The parameter in the FINDTITLE$ statement is a string containing the Title and Kind portions of a file name. FINDTITLE$ returns the complete pathname where the file is found.

FINDTITLE$ searches for this file in the Subject "Programs" in all the available Devices. If it is not found in Programs, then the current Subject is searched. If the file is not found in the current Subject, then FINDTITLE$ returns a zero length string.

Note: With some versions of GRiD-OS, FINDTITLE$ will display a blinking message, prompting you to insert the diskette containing the title.

See DEVICE$ and SUBJECT$ for more information on making GRiDTask programs independent of Subjects and Devices.

EXAMPLE

```
;-----------------------------------------------
; Make sure floppy with GRiDTerm is in disk drive
;-----------------------------------------------
done = FALSE
WHILE NOT done
  temp$ = FINDTITLE$ ("GRiDTerm~Run Terminal~")
  IF LEN(temp$) <> 0
    done = TRUE
  ELSE
    CLEARSCREEN
    PRINT "GRiDTerm is not available"
    PRINT "Insert the disk labeled ""DataComm"""
    PRINT "Press any key when ready to continue"
    PAUSE ""
  ENDIF
```

```
WEND
```

This program checks to see if GRiDTerm is currently available. If it can't find GRiDTerm it prompts the user to insert a diskette labeled "DataComm" and then tries again. This program does not proceed until GRiDTerm is available.

FONT

FONT "pathname"

NOTES

FONT specifies the font to be used while displaying text in the Task window. The parameter string is the name of the font file you want to become the current font.

The Task window always starts out in the Built-In font. You can subsequently load four additional fonts into memory and quickly change between them. If you exceed this limit (four loaded plus the Built-In), an error occurs.

The first time you specify a particular font there will be a delay while that font is loaded into memory. Fonts remain in memory until GRiDTask exits or until you execute a FREEFONT statement. Thus, changing to a font already in memory is fast.

If the pathname parameter has no Kind, then the Kind "Font" is assumed. If no Device or Subject is specified, FONT looks in all the available Programs Subjects. If the file is not found in Programs, then the current Subject is searched. If it is still not found, an error occurs.

EXAMPLES

```
FONT "ASCIIModern~Font~"
PRINT "This is AsciiModern Text"
DELAY 2
FONT "ASCIITimesRoman~Font~"
PRINT "This is AsciiTimesRoman Text"
```

This example loads the ASCIIModern font and prints a message using this font. After a two second delay, the ASCIITimesRoman font is loaded and prints another message using this new font. The Task application can now switch between these two fonts quickly, or can load one or two more fonts.

It is important that you use identical pathname parameters each time you change to a particular font. The first time you load a font the pathname parameter is stored. On subsequent FONT statements the new pathname is compared to the pathnames of the fonts that are already loaded. This determines if a font with the new pathname has already been loaded, or if it needs to be loaded. For example, the following two statements will mistakenly load the same font twice.

```
    FONT "GRiD 80"

    FONT "GRiD 80~Font~"
```

FORMCHOICE

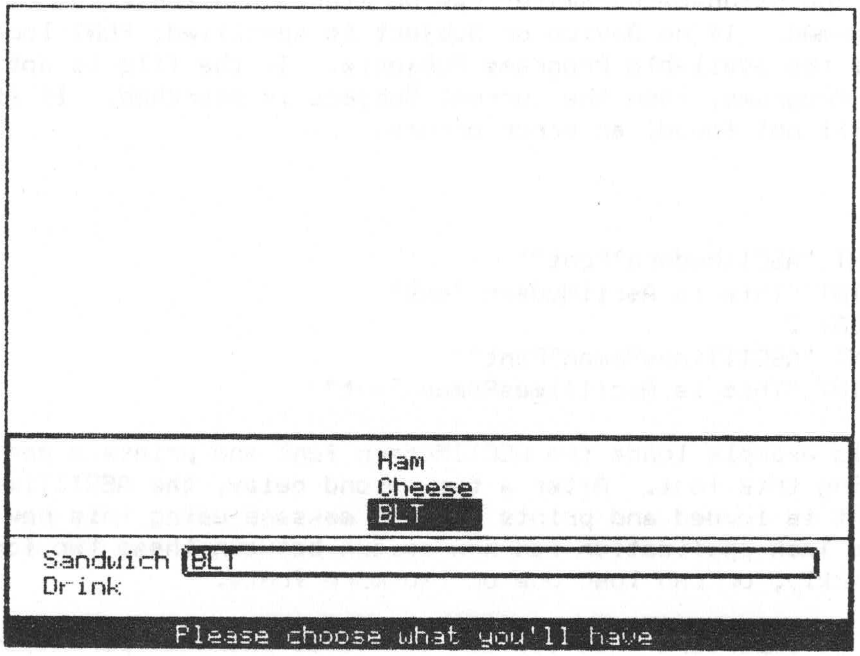
```
number = FORMCHOICE (form$, itemNum)
```

NOTES

FORMCHOICE is an integer-value function used with DOFORM$ to retrieve information from a user with GRiD forms.

The first parameter supplied to FORMCHOICE is the form string returned by DOFORM$ (see DOFORM$ for the format of this string). The second parameter - itemNum - is an integer which specifies the item of interest in the form.

FORMCHOICE returns the choice setting for the specified item. See Figure FORMCHOICE-1 for an example of a form.



Sandwich BLT
Drink

Please choose what you'll have

Figure FORMCHOICE-1. The Lunch Menu

If the user selects the third choice (BLT) for the first item (sandwich types), FORMCHOICE returns the number 3.

See the description for FORMCHOICE$, a related string function. FORMCHOICE$ extracts the text from a form setting.

For convenience, the example for DOFORM$ is repeated below.

EXAMPLE - FORMCHOICE

```
TASKWINDOW 0,0,-1,-1
; Display the form
msg$        = "Please choose what you'll have"
lunchForm$ = "Sandwich~Ham~@Cheese~BLT¦Drink~^~Milk~Soda~Juice¦"
lunch$ = DOFORM$ (msg$, lunchForm$, 3)

; Extracting information from the form
sandwich = FORMCHOICE  (lunch$, 1)
drink$   = FORMCHOICE$ (lunch$, 2)

; Instructions to the sandwich chef
IF sandwich = 1
 PRINT "Slice the ham"
ELSE
IF sandwich = 2
 PRINT "Slice the mozzarella"
ELSE
IF sandwich = 3
 PRINT "Cook the bacon"
ENDIF:ENDIF:ENDIF

; Responses to the choice of drink
IF drink$ = "Milk"
 PRINT "That'll be 50 cents please"
ELSE
IF drink$ = "Soda"
 PRINT "We have cola and orange soda"
ELSE
IF drink$ = "Apple" OR drink$ = "Grape"
 PRINT "Yes, we have " + drink$
ELSE
 PRINT "We're out of that juice"
ENDIF:ENDIF:ENDIF
```

This Procedure displays a form asking the user to specify choices for lunch, including a sandwich type and a juice type. It then extracts this information and gives instructions to the chef on preparing the sandwich, and responses the waiter might give the customer based on the desired drink.

The first section of the procedure - "Display the form" - defines and displays this two-item form.

The second section of the procedure - "Extracting information from the form" - uses the FORMCHOICE and FORMCHOICE$ functions to read the values that are in the form.

The third and fourth sections of the procedure - "Instructions to the sandwich chef" and "Responses to the choice of drink" -

provide actions that might be taken as a result of different answers given by the user in the form.

FORMCHOICE$

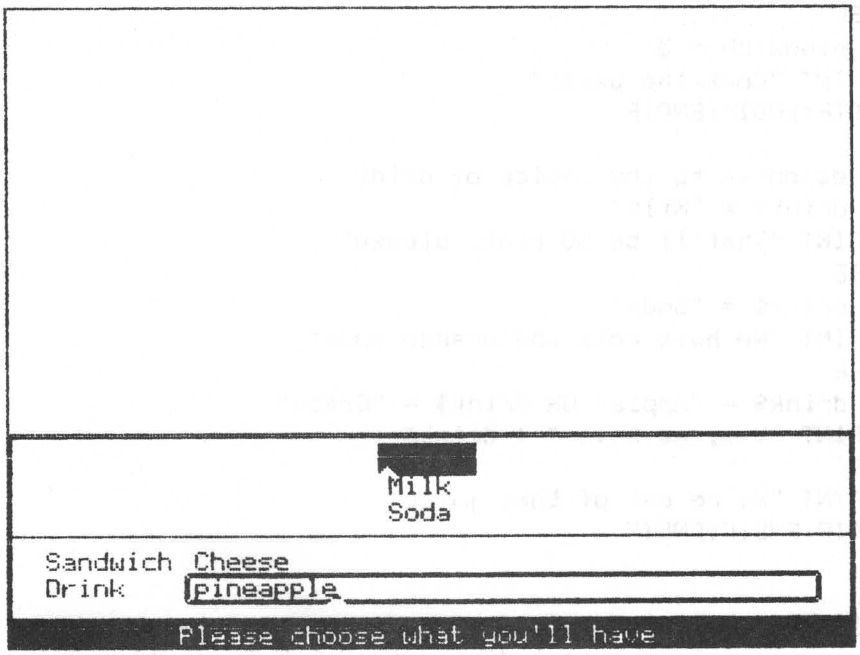
choice$ = FORMCHOICE$ (form$, itemNum)

NOTES

FORMCHOICE$ is a string function used with DOFORM$ to retrieve information from a user via GRiD forms.

The first parameter supplied to FORMCHOICE$ is the form string returned by DOFORM$ (see DOFORM$ for the format of this string). The second parameter - itemNum - is an integer specifying the item of interest in the form.

FORMCHOICE$ returns the setting for the specified item. If the item is an editable item, FORMCHOICE$ returns what the user typed. If the item is a choice item, FORMCHOICE$ returns the text of the choice. See Figure FORMCHOICE$-1 for an example of a form.



Sandwich Cheese
Drink pineapple

Please choose what you'll have

Figure FORMCHOICE$-1. The Lunch Menu

In this figure, the user has typed "pineapple" in the form. When the following statement is executed after the form is filled in and confirmed -

```
choice$ = FORMCHOICE$ (form$, 2)
```

- then choice$ is equal to "pineapple".

Note that FORMCHOICE, a related integer function, returns an integer value indicating which choice was selected.

EXAMPLE - FORMCHOICE$

```
TASKWINDOW 0,0,-1,-1
; Display the form
msg$         = "Please choose what you'll have"
lunchForm$ = "Sandwich~Ham~@Cheese~BLT|Drink~^~Milk~Soda~Juice|"
lunch$ = DOFORM$ (msg$, lunchForm$, 3)

; Extracting information from the form
sandwich = FORMCHOICE  (lunch$, 1)
drink$   = FORMCHOICE$ (lunch$, 2)

; Instructions to the sandwich chef
IF sandwich = 1
 PRINT "Slice the ham"
ELSE
IF sandwich = 2
 PRINT "Slice the mozzarella"
ELSE
IF sandwich = 3
 PRINT "Cook the bacon"
ENDIF:ENDIF:ENDIF

; Responses to the choice of drink
IF drink$ = "Milk"
 PRINT "That'll be 50 cents please"
ELSE
IF drink$ = "Soda"
 PRINT "We have cola and orange soda"
ELSE
IF drink$ = "Apple" OR drink$ = "Grape"
 PRINT "Yes, we have " + drink$
ELSE
 PRINT "We're out of that juice"
ENDIF:ENDIF:ENDIF
```

This Procedure displays a form asking the user to specify choices for lunch, including a sandwich type and a juice type. It then extracts this information and gives instructions to the chef on preparing the sandwich, and responses the waiter might give the customer based on the desired drink.

The first section of the procedure - "Display the form" - defines and displays this two-item form.

The second section of the procedure - "Extracting information from the form" - uses the FORMCHOICE and FORMCHOICE$ functions to read the values that are in the form.

The third and fourth sections of the procedure - "Instructions to the sandwich chef" and "Responses to the choice of drink" - provide actions that might be taken as a result of different answers given by the user in the form.

FRAMEBOX

FRAMEBOX topleftX, topleftY, widthx, heightY

NOTES

FRAMEBOX draws a one-pixel wide frame around a rectangle within the Task window. The first two parameters specify the top-left corner of the rectangle, and the third and fourth parameters indicate the extent or size of the rectangle. "widthX" indicates the extent in the x, or horizontal direction, and "heightY" indicates the extent in the y, or vertical direction. All the parameters are in pixels.

EXAMPLE

```
FONT "GRiD 53~Font~"
CURSOR 65,65
PRINT "Up the Down Staircase"
FRAMEBOX 40,40,125,40
```

This example prints a message on the screen, then draws a one-pixel-wide box around it.

FREEFONT

FREEFONT "pathname"

NOTES

FREEFONT removes a currently loaded font from memory.

The pathname parameter should exactly match the parameter used with the FONT verb when the font was originally loaded.

If the specified font is not loaded, or if it is the current font, an error occurs.

There are two reasons you might want to free a font. If you want to use more than four loaded fonts, then you have to remove some font(s) to make room for new ones. Also you might want to free the RAM occupied by a font.

GRiDTask frees any loaded fonts when it stops executing.

EXAMPLE

```
FONT "ASCIITimesRoman~Font~"
CENTER "This is an example of ASCIITimesRoman text"
DELAY 2
FREEFONT "ASCIITimesRoman~Font~"
```

This example loads the ASCIITimesRoman font and displays a message in this font. After two seconds, the font is removed from memory. Note that the message remains on the screen until it is erased, even though the font is removed in the FREEFONT statement.

GETFILE$

pathName$ = GETFILE$ (msg$)

NOTES

GETFILE$ displays a File form within the Task window, and returns a string containing the pathname of the file selected by the user. The string parameter "msg$" is a message displayed with the File form.

GETFILE$ sets the LASTKEY$ variable to whatever key the user pressed to terminate the File form.

EXAMPLE

For convenience, the example for the CHANGEKIND$ verb has been reproduced here.

```
;----------------------------------------
; Reformat Historical Quotes Text File
;----------------------------------------
textFile$ = GETFILE$("Select Text file to be reformatted")
FILEFORM "Historical Quotes~Reformat~00"
ADDKEYS "¦.¦t¦."
FILEFORM textFile$
ADDKEYS "¦."
graphFile$ = CHANGEKIND$(textFile$, "Graph")
FILEFORM graphFile$ + "21"        ; get new file and application
ADDKEYS "¦.¦."
```

This program reformats a text file of data that has been retrieved from a mainframe. It writes the reformatted data to a graph file.

1) It starts by asking the user to fill in a File form selecting the text file to be reformatted.

```
textFile$ = GETFILE$("Select Text file to be reformatted")
```

2) It then retrieves a Reformat file.

```
FILEFORM "Historical Quotes~Reformat~00"
ADDKEYS "¦.¦t¦."
```

3) It specifies the text file as the file to be reformatted.

```
FILEFORM textFile$
ADDKEYS "¦."
```

4) It specifies the output file as having the same name as the input text file except with a kind of "Graph". It writes the new graphfile, then brings it into GRiDPlot.

```
graphFile$ = CHANGEKIND$(textFile$, "Graph")
FILEFORM graphFile$ + "21"       ; get new file and application
ADDKEYS "¦.¦."
```

IF
ELSE
ENDIF

```
IF <expression>
   statement(s)
ELSE
   statement(s)
ENDIF
```

NOTES

The IF, ELSE and ENDIF statements allow for the conditional execution of a sequence of statements. If the expression following the IF statement evaluates to TRUE, the statements between the IF and ELSE are executed, and the statements between the ELSE and ENDIF are not executed.

If the expression following the IF statement evaluates to FALSE, the statements between the IF and ELSE are not executed, and the statements between the ELSE and ENDIF are executed.

The ELSE statement is optional.

You can nest IF ELSE ENDIF statements to any depth. GRiDTask matches each ENDIF with the most recent IF. If you have unequal numbers of IF and ENDIF statements, an error occurs. The IF, ELSE, and ENDIF verbs cannot be on the same line with another statement.

See next page for example.

EXAMPLE

```
This is legal:          IF pizzaTopping = anchovies
                           PRINT "No Thank You"
                        ELSE
                          PRINT "I’ll take a slice"
                        ENDIF

This is illegal:        IF pizzaTopping = anchovies
                          PRINT "No Thank You"
                        ELSE PRINT "I’ll take a slice" ENDIF
```

The second example is illegal because the ELSE and ENDIF statements are on the same line as the second PRINT statement.

```
This is legal:          IF pizzaTopping = anchovies
                          PRINT "No Thank You"
                        ELSE:PRINT "I’ll take a slice" :ENDIF
```

This example is legal because colons separate the statements.

INKEY$

```
someKey$ = INKEY$
```

NOTES

INKEY$ is a string function. It returns a one-character string representing the last unprocessed character typed at the keyboard. If no character has been typed then INKEY$ does not wait and returns a zero length string.

CONCHARIN$ is a similar function but does wait for a key to be pressed.

EXAMPLES

```
;-----------------------------------------
; Retrieve the text file
;-----------------------------------------
FILEFORM "Sample~Text~"
ADDKEYS "¦."
;-----------------------------------------
; Print the file ten times or until
; a CODE-ESC is pressed
;-----------------------------------------
codeEsc$ = CHR$(155)
i = 1
WHILE (i <= 10) OR (INKEY$ <> codeEsc$)
  ADDKEYS "¦t¦V¦.¦.¦."   ; print entire file
  i = i + 1
WEND
```

This program prints a file ten times or until CODE-ESC is pressed. The WHILE statement uses the INKEY$ function to see what key, if any, has been pressed on the keyboard.

```
;-----------------------------------------
; Flush the keyboard queue
;-----------------------------------------
WHILE LEN(INKEY$) <> 0
WEND
```

This example empties the keyboard buffer. You might want to do this before executing a CONCHARIN$ if you suspect that some unwanted keys are still in the keyboard buffer.

INPUT$

value$ = INPUT$ (prompt$, length, height, initValue$)

NOTES

INPUT$ receives data input from the user. A prompt and a field in which to enter data are displayed. INPUT$ returns the contents of the field. The INPUT$ parameters are used as follows:

prompt$ appears to the left of the field. It begins at the current cursor location.

length specifies the number of characters in each line of the data field.

height specifies the number of lines in the data field. The first character of the field lies just to the right of prompt$.

initValue$ is the initial value that appears in the field. Specify a null value ("") to omit this feature.

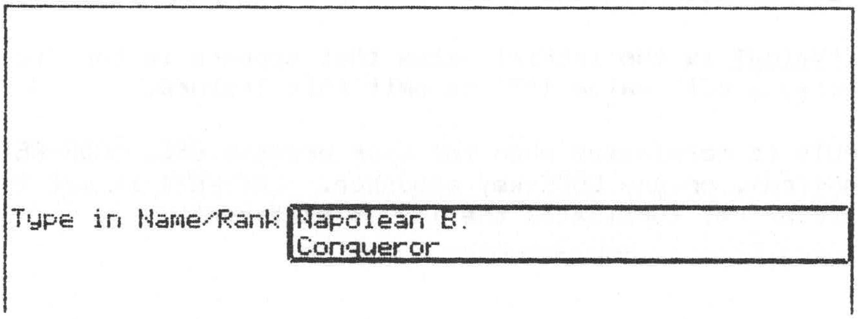
INPUT$ is terminated when the user presses ESC, CODE-RETURN (confirm), or any CODE-key sequence. LASTKEY$ is set to whatever key terminated the INPUT$ statement.

EXAMPLE

```
TASKWINDOW 0,0,-1,-1
CURSOR 2,75
prompt$          =  "Type in Name/Rank"
length           =   30
height           =   2
initValue$       =   "Napolean B.
Conqueror  "
value$ = INPUT$ (prompt$, length, height, initValue$)
```

In this example, "Type in Name/Rank" is displayed next to a data field. There are two lines available, each 30 characters long, for the user to type in a name and a rank. An initial answer (initValue$) is in the field.

Figure INPUT$-1 illustrates how this appears on the screen.



```
Type in Name/Rank Napolean B.
                  Conqueror
```

Figure INPUT$-1. The INPUT$ Field

INSTALL

```
INSTALL pathname$
```

NOTES

INSTALL lets you use custom GRiDTask verbs in your GRiDTask programs. You can program routines in Pascal, PLM, or other high-level languages and use these new functions to extend the GRiDTask language. An example is the set of library routines that display data-entry forms.

In the INSTALL statement, pathname$ specifies the file containing the library routines. Such files have the Kind "Library" and are placed in the Subject "Programs". A complete pathname (including the Device and Subject) is not required.

For example:

```
INSTALL "DataEntryForms~Library~"
```

Note that more than one library may be installed in a GRiDTask program. See Appendix H for information on the steps required to develop an INSTALL verb.

EXAMPLE

```
;----------------------------------------------------
;  This task program illustrates how to install
;  and use the sample user-written library -
;  ‘Programs‘Sample~Library~      - in GRiDTask.
;  The new functions are: CONCAT$, FLASH, MAX, and DIV
;----------------------------------------------------
TASKWINDOW 0,0,-1,-1
INSTALL DEVICE$ + "Programs‘Sample~Library~"
;-------------------------
str1$ = "One and "
str2$ = "two and ..."
PRINT CONCAT$ (str1$,str2$)
;-------------------------
FLASH: FLASH: FLASH
;-------------------------
PRINT "This is  MAX(4,5)"
PRINT STR$ (MAX(4,5))
;-------------------------
PRINT "This is  DIV (5,PI)"
PRINT STR$ (DIV (5,PI))
STACKMSG "Press any key to exit"
PAUSE ""
```

In this example, the user-written library "Sample~Library~" is installed. Then three functions in this library - CONCAT$, FLASH, and MAX are used like any standard GRiDTask verb. These functions are also in the example described in Appendix H.

INSTR

```
location = INSTR (start, source$, find$)
```

NOTES

INSTR finds the location of a string within another string. The third parameter is the string that you want to find. The second parameter is the string in which you are looking. The first parameter is the character location where you want to start the search. The INSTR function returns the character location where the string was found. INSTR returns 0 if the string isn't found.

EXAMPLE

```
source$  = "Once upon a time"
find$    = "a"
PRINT "The location of " + find$ + "is " + STR$(INSTR (1, source$,
find$))
```

This example prints the character location where "a" is found in "Once upon a time".

INVERTBOX

INVERTBOX topleftX, topleftY, widthX, heightY

NOTES

INVERTBOX inverts a rectangle within the Task window. The first two parameters specify the top-left corner of the rectangle and the third and fourth parameters indicate the extent or size of the rectangle. "widthX" indicates the extent in the x, or horizontal direction, and "heightY" indicates the extent in the y, or vertical direction. All the parameters are in pixels.

EXAMPLE

```
CURSOR 65,65
PRINT "Down the Up Staircase"
INVERTBOX 40,40,125,40
```

This example prints a message on the screen, then inverts a box containing the message. The message then appears in "inverse video".

INVERTLINE

```
INVERTLINE  x1, y1, x2, y2
```

NOTES

INVERTLINE inverts a line in the Task window. The four parameters specify the two end points of the line within the Task window. All the parameters are in pixels.

EXAMPLE

```
CURSOR 50,50
PRINT "This is the top portion"
CURSOR 50,150
PRINT "This is the bottom portion"
INVERTLINE 0,100,300,100
```

This program prints two messages, then inverts a horizontal line (300 pixels long) between the two messages. Note that if the inverted line crosses any pixels already "on", then those pixels will be turned off. Thus, a line drawn with INVERTLINE is solid only if all the pixels along the line were dark prior to the INVERTLINE statement.

ITEMCOUNT

```
numItems = ITEMCOUNT (list$, separater$)
```

NOTES

ITEMCOUNT is a function which returns the number of items in list$. The items in list$ are separated by the single character separater$.

EXAMPLE

```
list$          =  "Pomegranates*Kumquats*Rutabagas*Mangoes"
separater$     =  "*"

numItems = ITEMCOUNT (list$, separater$)
IF numItems > 3
  PRINT "Bring oxen"
ENDIF
```

In this example, 4 items (separated by the character specified in separater$) are counted in list$. The variable "numItems" gets the value 4, so the message is printed.

LASTKEY$

```
key$ = LASTKEY$
      or
LASTKEY$ = key$
```

NOTES

LASTKEY$ is a pre-defined string variable. It is set to the last key to terminate a form, menu or File form in the Task window. Menus and forms do not have to be confirmed. The LASTKEY$ variable tells you which key was pressed to terminate the menu or form.

Because LASTKEY$ is a variable, you can set its value as well as test for its value.

EXAMPLE

```
confirm$ = CHR$(141)
LASTKEY$ = ""
choice = DOMENU ("You may pick a color", "Blue!Red!Green")
IF (LASTKEY$ = confirm$)
 .....
 GRiDTask statements
 .....
ENDIF
```

In this example, a menu with three color choices is displayed. The user may select one and then press a key to terminate the menu. If the user presses CODE-RETURN, then the statements after IF (LASTKEY$ = confirm$) are executed.

LASTMESSAGE$

**message$** = LASTMESSAGE$

or

LASTMESSAGE$ = **message$**

NOTES

LASTMESSAGE$ is a pre-defined string variable. It is set to the last message displayed in the application window. Because LASTMESSAGE$ is a variable, you can set its value as well as test for its value.

Some messages - such as messages displayed when CODE-U is pressed - may not set LASTMESSAGE$. It is recommended to verify messages that you want to look at using LASTMESSAGE$.

EXAMPLE

```
FILEFORM "filename~Terminal~"
ADDKEYS "¦.¦a¦."
; press softkey 1 to sign-on to host
LASTMESSAGE$ = ""
ADDKEYS "¦1"
IF LASTMESSAGE$ <> ""    ; softkey terminated message
  ; take appropriate action
ENDIF
```

This example shows a case where LASTMESSAGE$ is the only method of determining successful completion of signing on to a host system. As part of the GRiDTerm terminal descriptor file, softkey #1 contains the log on sequence. LASTMESSAGE$ is checked to see if the softkey timed out.

LEN

```
num = LEN (stringX$)
```

NOTES

LEN returns the length of the specified string.

EXAMPLE

See the program examples for CELL$, FINDTITLE$, INKEY$. The LEN statement is used in all these examples. The example for the WHILE verb has been reproduced here for convenience.

```
found = 0
i     = 0
WHILE i < LEN(inputstringX$)
  i = i + 1
  IF MID$(inputstringX$, i, 1) = "?"
    found = found + 1
  ENDIF
WEND
PRINT "I found " + STR$(found) + " question marks!"
```

This example counts the number of question marks in a string (inputstringX$). It prints a message indicating how many were found.

LINEHEIGHT

```
height = LINEHEIGHT
```

NOTES

LINEHEIGHT is a function which returns the height of the current font in pixels. This is the font last set with the FONT verb.

EXAMPLE

```
TASKWINDOW     0,0,-1,-1
PRINT "You cannot fit more than "
PRINT  STR$(WINDOWHEIGHT/LINEHEIGHT) + " lines on-screen"
```

In this example, WINDOWHEIGHT gives the total height of the available screen. LINEHEIGHT gives the height of characters in the current font. The ratio of the two numbers gives the maximum number of lines that can be displayed on the screen.

LOCATE

```
LOCATE x , y
```

NOTES

The LOCATE verb is used to reposition the cursor, and is the same as the CURSOR verb with one exception:

The LOCATE x̲ and y̲ parameters are in units of characters.

The CURSOR x̲ and y̲ parameters are in units of pixels.

The x and y numbers directly specify the new cursor location.

EXAMPLE

```
CURSOR CHARWIDTH,0
PRINT "Fruit and cheese"
LOCATE 1 , 2
PRINT "Spaghetti and salad"
LOCATE 1 , 4
PRINT "Chips and chili"
```

This program prints three lines in the Task window. Each line begins the width of one character from the left edge of the Task window. The lines are separated by blank lines.

MEMORY

```
space = MEMORY
```

NOTES

MEMORY is a function that returns the number of free bytes of memory.

EXAMPLE

```
PRINT "The number of free bytes is  " + STR$(MEMORY)
DELAY 3
```

This program prints the amount of MEMORY space by converting the numeric value to a printable string.

MID$

```
portion$ = MID$ (wholeString$, start, length)
```

NOTES

MID$ is a string function which returns a portion of a specified string.

The first parameter is the string from which the portion is extracted. The second parameter is the character position at which to start the new portion string. The third parameter is the length of the portion string.

MID$ returns a zero-length string if length is zero, or if start is either zero or greater than the length of the string.

If start + length is greater than the length of the original string, then MID$ returns a string which only includes characters from start to the end of wholeString$.

EXAMPLE

```
found = 0
i     = 0
WHILE i < LEN(inputstringX$)
  i = i + 1
  IF MID$(inputstringX$, i, 1) = "?"
    found = found + 1
  ENDIF
WEND
PRINT "I found " + STR$(found) + " question marks!"
```

This example counts the number of question marks in a string (inputstringX$). It prints a message indicating how many were found.

PAINT

```
PAINT x, y, "pathname"
```

NOTES

PAINT displays a canvas image in the Task window. Canvas files can be created and modified in GRiDPaint.

The parameters indicate the name of the canvas file to be displayed and the pixel coordinates within the Task window where the top-left corner of the canvas image is to be placed.

Any portion of the canvas image extending beyond the edge of the window is clipped.

It is important that the image be created and saved using GRiDPaint, as a file with Kind "Canvas". Screenimage files do not work.

If the pathname has no Kind, then "Canvas" is assumed. If no Device or Subject is specified, then GRiDTask looks in the current Device and Subject.

EXAMPLE

```
TASKWINDOW 0,0,-1,-1
PAINT 10,10, "Renoir~Canvas~"
```

When GRiDTask executes this, the image with Title "Renoir" and Kind "Canvas" in the current Device and Subject is displayed on the screen. The upper left corner of the Canvas image is placed 10 pixels from the left edge of the Task window and 10 pixels down from the top edge of the Task window. GRiDTask displays as much of the image as there is room for.

$PARSEONLY

```
$PARSEONLY
```

NOTES

$PARSEONLY causes a GRiDTask file to be checked for syntax errors without executing it.

$PARSEONLY is useful for finding syntax errors in a new GRIDTask program without taking time to actually run it. It also guarantees that every line is checked.

After creating a new program or sequence of code, place the $PARSEONLY statement at the beginning of the program and run it. The program won't execute, but every line will be checked for syntax errors.

To execute the code, remove the command $PARSEONLY.

EXAMPLE

```
$PARSEONLY
GRiDTask statement1
GRiDTask statement2
GRiDTask statement3
.....
```

When this program is executed in GRiDTask, the syntax of each statement is checked, but the statements are not actually executed. To run the program, remove the $PARSEONLY statement.

PASSKEYS

```
PASSKEYS keysToPass$, keysToTerminate$
```

NOTES

PASSKEYS allows you to specify keys that the user can send to the application (running in the application window) and keys that the user can press to terminate the PASSKEYS statement.

keysToPass$ defines the key sequences that, when pressed by the user, are passed to the application.

keysToTerminate$ defines the key sequences that the user can press to terminate PASSKEYS.

The variable LASTKEY$ is set to the key that terminates PASSKEYS. This termination key is not passed to the application.

You can execute a PASSKEYS statement at any time. If the application window is waiting for a FILEFORM statement when you execute a PASSKEYS statement, then the application window stops waiting, and a File form is displayed with its normal defaults. Similarly, if a File form is being displayed when the PASSKEYS statement ends, the File form stops being displayed and waits for the next FILEFORM statement.

See Appendix B for information on how to encode keystrokes. A null string for either parameter of PASSKEYS is considered as "all keys."

See the next page for examples.

EXAMPLES PASSKEYS

```
FILEFORM "`Hard Disk`Programs`GRiDWrite~Run Text~"
ADDKEYS "¦."

HalfWindow = WINDOWHEIGHT/2
TASKWINDOW 0,HalfWindow,-1,-1
UPDATESCREEN

PASSKEYS "","¦."
PRINT "Welcome back to GRiDTask"
```

In this example, GRiDWrite is loaded in the application window. Then the Task Window is set to the bottom half of the screen, and GRiDWrite updates its window to the top half of the screen. The user may type any keystrokes except CODE-RETURN, which is the termination key. This means, for example, that if the user presses CODE-t(ransfer), the CODE-t menu appears. But if the user presses CODE-RETURN to "Save a file", the CODE-RETURN terminates PASSKEYS and the file is not saved (GRiDWrite does not receive the CODE-RETURN key).

Of course, the termination key(s) could be different so that the user would have the full use of GRiDWrite, but still be able to terminate it when ready.

```
FILEFORM "`Hard Disk`Programs`GRiDWrite~Run Text~"
ADDKEYS "¦."

HalfWindow = WINDOWHEIGHT/2
TASKWINDOW 0,HalfWindow,-1,-1
UPDATESCREEN

keysToPass$ =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ¦.¦t¦d"
PASSKEYS keysToPass$, "¦z#*"
PRINT "Welcome back to GRiDTask"
```

In this example, GRiDWrite is loaded and the user may type any letter characters, as well as CODE-RETURN (confirm), CODE-t (transfer), and CODE-d (duplicate). Note that CODE-keys are specified with small letters (¦t).

When the user presses "¦z" or "#" or "*", PASSKEYS is terminated and GRiDTask continues execution with the PRINT statement.

PAUSE

PAUSE keysToTerminate$

NOTES

PAUSE allows all keys typed at the keyboard to go directly to the application window until the user presses one of a set of keys.

keysToTerminate$ defines the key sequences that, when pressed by the user, are passed to the application.

The variable LASTKEY$ is set to the key that terminates PAUSE. This termination key is not passed to the application. If keysToTerminate$ is zero length (PAUSE "") then PAUSE waits until any key is pressed.

While executing a PAUSE verb, File forms work "normally": they do not wait for a FILEFORM verb before appearing.

You can execute a PAUSE statement at any time. If the application window is waiting for a FILEFORM statement when you execute a PAUSE statement, then the application window stops waiting, and a File form is displayed with its normal defaults. Similarly, if a File form is being displayed when the PAUSE statement ends, the File form stops being displayed and waits for the next FILEFORM statement.

Note that the PAUSE verb gives the same result as

```
PASSKEYS "",keysToTerminate$
```

EXAMPLES

```
PAUSE ""
```

This statement waits for any key to be pressed. When the next key is pressed, the program continues. No keys are passed to the application.

```
PAUSE "!z"
```

This statement waits for a CODE-Z to be pressed. Any other keys pressed prior to CODE-Z are passed on to the application. This is a way of turning control of the system over to the user. Until the user presses CODE-Z, any keystrokes can be pressed.

PLAY

PLAY musicStr$

NOTES

The PLAY verb creates music (or other sounds) using a speaker built-in to your computer. The parameter musicStr$ specifies the musical notes, their durations, and their tempos.

The sound driver (Title Sound and Kind Device) must be in the Subject Programs for PLAY to execute. If the file is not present, PLAY statements are ignored.

Notes and Octaves

A..G	This specifies a note - A, B, C, D, E, F, or G. A pound sign (#) after the note specifies a sharp note and a minus sign (-) a flat note.
> A..G	This says to go up one octave and play the note A,B,C etc. The octave is raised each time ">" is executed, up to octave 6. If ">" is specified again, the note played is in Octave 6.
< A..G	This says to go down one octave and play the note A,B,C etc. The octave is lowered each time "<" is executed, down to octave 0. If "<" is specified again, the note played is in Octave 0.
O num	This specifies an octave. num is a number from 0 through 6 specifying one of the seven octaves available. The initial octave is 3.
N note#	This specifies a note by number. note# is a number from 1 to 84 specifying one of the 84 possible notes in the 7 available octaves. This is an alternative to specifying an octave character "<" or ">" and a note A,B,C,D,E,F, or G. To cause a rest, specify note# as 0.

Duration L duration

This specifies the duration of notes. duration is a number from 1 through 64 specifying the length of all the notes following it. The following tables show the results for different values of duration:

Length	Equivalent
L1	whole note
L2	half note
L3	one of a triplet of three half notes (1/3 of a 4-beat measure)
L4	quarter note
L5	one of a quintuplet (1/5 of a measure)
L6	one of a quarter note triplet
.	
.	
.	
L64	sixty-fourth note

You need not specify L if you want to change the length of one particular note. Instead, specify the note followed by the desired length. For example, A16 is equivalent to L16A. The initial value is L4.

MN

Music Normal. Each note plays 7/8 of the time as set by the Length subcommand (L).

ML

Music Legato. Each note plays the full period set by the Length subcommand (L).

MS

Music Staccato. Each note plays 3/4 of the time set by the Length subcommand (L).

Tempo P pause#

This specifies a pause. pause# specifies a pause ranging from 1 through 64. The length is determined in the same way as the Length command (L n).

.

This is a dotted note (specified as a period). When specified after a note, its length (as specified by the Length command) is multiplied by 3/2. When more than one dot is specified after the note, the length is determined by multiplying by 3/2 for each dot. For example, "A.." plays 9/4 as long as the value specified by L n; "A..." plays 27/8 as long. You can also specify dots after the Pause command (P) to control the length in the same way.

T Tnumber — Tempo. Sets the number of quarter notes in a minute, where Tnumber is a range from 32 through 255. The default is 120.

EXAMPLES - PLAY

Subcommand	Result
PLAY "<<CDE"	Goes down two octaves and plays the notes C, D, and E (do re mi)
PLAY ">FGA"	Goes up one octave and plays the notes F, G, and A (fa so la).
PLAY "B>C"	Plays the note B (la), then goes up one octave and plays the note C (do).

The following example plays the C major scale, ascending and descending, starting at Octave 3:

```
PLAY "O3  CDEFGAB>C<BAGFEDC"
```

PRINT

```
PRINT "text ..... text"
```

NOTES

PRINT displays the specified text within the Task window starting at the current cursor location and using the current font. If the text reaches the right edge of the window, it continues on the next line. Words are not broken at the edge of the window (word wrap).

After executing a PRINT statement, the cursor is positioned at the left edge of the window, one line below the last character displayed.

Text that continues below the bottom of the Task window is not displayed.

If you want to print more than one line at a time, you can let the text word wrap at the right edge of the window, or you can force a new line by embedding a CR-LF in the string. In either case, each new line is left justified to the same position as the first word in the string.

The rate at which text is displayed by the PRINT verb is controlled by the SPEED verb. The initial setting is full speed.

EXAMPLES

```
PRINT "
This is a narrow
paragraph.  It has
CR-LFs embedded
in it."
```

The above command contains CR-LFs in the string and prints out exactly as you see it. If the cursor is originally at 120,30, then each of the four lines begins at x position 120.

```
PRINT "The time is now " + TIME$ + ", do you know where you
are?"
```

This example illustrates the use of a string variable (TIME$) in a PRINT statement. It prints the current time along with the text.

PROCEDURE

```
PROCEDURE procedureName parameter(s)
   LOCALS variable(s)
   .....
   RETURN
   .....
   ENDP
```

NOTES

How to Define a Procedure
You can define procedures that use parameters and local variables. A procedure definition begins with the keyword PROCEDURE, followed by the name of the procedure and a list of parameters (optional) separated by commas.

```
PROCEDURE procedureName parameter1$, parameter2, ...
```

Like variables, if the name of a parameter ends with a '$' it is a string parameter; otherwise it is a number. There can be up to 255 parameters in each procedure.

Next, local variables may be defined.

```
PROCEDURE procedureName parameter1$, parameter2, ...
   LOCALS   var1, var2$, ...
```

Then the body of the procedure follows. The word ENDP is the last line in the procedure.

```
PROCEDURE procedureName parameter1$, parameter2, ...
   LOCALS   var1, var2$, ...
   .....
   statements
   .....
   RETURN
   .....
   statements
   .....
ENDP
```

Local variables
Local variables are optional. If used, they are declared following the procedure declaration. The keyword LOCALS should be followed by a list of local variable names. There is no limit to the number of local variables declared. Several LOCALS statements can be used if desired.

RETURN
RETURN(s) are optional. If used, they do not have to be at the end of the procedure. There may be more than one RETURN in a procedure. GRiDTask returns from the procedure to the statement following the procedure call when a RETURN is executed or when the last line in the procedure is executed.

ENDP
ENDP marks the last line of the procedure definition. GRiDTask returns from the procedure to the statement following the procedure call when ENDP is executed.

Where to Define a Procedure in the Task program
A procedure definition can be placed anywhere within a program. You can place the procedure definition at the beginning of a program, in the middle, or in another file that is accessed with the TASK command.

The only restrictions are:

1) A procedure's definition must be encountered before the procedure is invoked.

2) A procedure can not be defined within the body of another procedure.

No harm is done if a procedure definition is encountered more than once. However, if two different procedures are defined with the same name, an error occurs.

Executing the procedure in the Task program
Executing a procedure is the same as executing any GRiDTask statement. The number and type of parameters must be correct. When the procedure is invoked within the GRiDTask program, the following syntax is used.

```
.....
procedureName parameter1$, parameter2, ...
.....
```

Everytime a procedure is executed a new set of local variables is created, if used. Local string variables are initialized to zero length strings, and local number variables are initialized to 0.

Referencing local variables and parameters is identical to referencing global variables. Any new variables created while executing a procedure are global.

Note that the syntax of a procedure is checked when the procedure is executed, not when it is defined. See the next page for an example.

EXAMPLE - PROCEDURE

```
; =========  Start of procedure  =========
PROCEDURE PZ
LOCALS   character$
  character$     = INKEY$
  IF character$ <> pause$
      RETURN
  ELSE
     fontheight = LINEHEIGHT
     IF fontheight <> 8 ;  Change font if not currently GRiD 53
       FONT "GRiD 53~font~"
     ENDIF
     CLEARMSG
     STACKMSG "Pause: Press any key to continue"
     PAUSE ""
     CLEARMSG
     IF fontheight = 12
       FONT "Ascii9x12~Font~"
     ENDIF
     IF fontheight = 16
       FONT "Ascii12x16~Font~"
     ENDIF
  ENDIF
ENDP
; =========   End of procedure  =========
; Beginning of Task program
pause$     = CHR$(240)               ; Pause character (CODE-p)
.....
.....
; Later - in the main body of the program
TASKWINDOW 0,0,-1,-1
PZ
IF Choice = 0
  PZ
  FONT "Ascii9x12~Font~"
  TASK prefx$ + "DemoOne~Text~"
  FONT "Ascii12x16~Font~"
  PZ
ENDIF
PZ
.....
```

In this program, a procedure - PZ - has been defined to handle pauses. Each time PZ is encountered, the procedure is executed. If the user presses CODE-p (pause$) then the next time the procedure is executed a message is displayed, and the procedure waits until a key is pressed. The font height upon entering the procedure is calculated using the LINEHEIGHT verb. This allows the procedure to print messages in a known font (GRiD 53) but to return to the font in use when the procedure was called. Note that in this example, an indicator of the current font could have been stored in a parameter passed to the procedure.

READFILE$

```
contents$ = READFILE$ (pathname$)
```

NOTES

READFILE$ is a string function which returns the contents of the file specified by pathname$. The file specified by pathname$ remains unchanged.

Note that if you do not specify the Kind in pathname$, then the Kind Text is assumed. If you don't specify a Device or Subject, then the current Device and Subject of the last file accessed through GRiDTask or the application window are assumed.

The maximum length allowed for a string variable is 64K bytes, so that if you attempt to read a file larger than 64K bytes, a GRiDTask error occurs.

READFILE$ sets the ERRORCODE variable to the number of any error that occurred. If no error occurs, then the ERRORCODE variable is set to (0) zero.

EXAMPLE

```
pathname$    =  "‘Floppy Disk‘BaseballCards‘MickeyMantle~Text~"
statistics$ =        READFILE$ (pathname$)
```

The above example copies the contents of the file MickeyMantle into the string variable statistics$.

RETURN

RETURN

NOTES

RETURN is used within procedures. When executed, GRiDTask exits from the procedure, and returns to the GRiDTask statement following the procedure call.

RETURN(s) are optional. If used, there may be more than one RETURN within a procedure, and RETURN verbs may be placed anywhere within the procedure body. A RETURN is not needed at the physical end of a procedure.

EXAMPLE

See the section in Chapter 4 entitled "Procedures" for an example using RETURN.

SCROLL

SCROLL distance, speed

NOTES

SCROLL causes the entire Task window to scroll, or move up.

The first parameter is an integer indicating how many pixels to scroll the Task window. The second parameter is an integer indicating how fast to scroll the window.

The higher the number, the faster the window scrolls, according to the following rules: if the number is positive and greater than zero, then it represents the number of pixels the window moves in each step. If the number is negative, the window moves one pixel at a time, with an additional delay between each move. The additional delay is the absolute value of the number in milliseconds.

The proper speed is best determined by trial and error, since the speed of scrolling is affected by the size of the window. A good strategy is to first try a speed of one. If this is too slow, then increment the speed value by one until a satisfactory speed is reached. If a value of one is too fast then try -10, -20, -30 and so on until a slow enough value is reached.

EXAMPLE

```
TASKWINDOW 0,0,-1,-1
PAINT 10,10, "Renoir~Canvas~"  ; Canvas height = 150 pixels
DELAY 3
SCROLL 175,5
```

In this example, the image with Title "Renoir" and Kind "Canvas" in the current Device and Subject is displayed in the Task window. After a delay of three seconds, the image scrolls upward 5 pixels per step until it has moved a total of 175 pixels. Since the Canvas height is 150 pixels and it is originally 10 pixels below the top of the screen, the image completely disappears after scrolling up 160 pixels.

SCROLLBOX

SCROLLBOX topleftX, topleftY, widthX, heightY, "direction", distance, speed

NOTES

SCROLLBOX scrolls a rectangle within the Task window.

The first two parameters - topleftX and topleftY - specify the top-left corner of the rectangle. The third and fourth parameters - widthX and heightY - indicate the extent or size of the rectangle. "widthX" indicates the extent in the x, or horizontal, direction, and "heightY" indicates the extent in the y, or vertical, direction. The first four parameters are in pixels.

The "direction" parameter is a string indicating which direction the rectangle should scroll. The possible directions are "up", "down", "right", and "left". Only the first letter is significant, and upper or lower case does not matter.

The "distance" parameter is an integer indicating how many pixels to scroll the rectangle.

The "speed" parameter is an integer indicating how fast to scroll the rectangle. The higher the number, the faster the rectangle scrolls, according to the following rules: if the number is positive and greater than zero then it represents the number of pixels the rectangle moves in each step. If the number is negative, the rectangle moves one pixel at a time, with an additional delay between each move. The additional delay is the absolute value of the number in milliseconds.

The proper speed is best determined by trial and error, since the speed of scrolling is affected by both the size of the rectangle and the direction it is scrolled. A good strategy is to first try a speed of one. If this is too slow then increment the speed value by one until a satisfactory speed is reached. If a value of one is too fast then try -10, -20, -30 and so on until a slow enough value is reached.

EXAMPLE

```
PAINT 10,125, "Buffalo~Canvas~" ;Canvas = 100 pixels wide,50
high
DELAY 10
SCROLLBOX 10,125,100,50,"right",210,8
DELAY 10
```

In this example, the buffalo is "resting" on the left side of the Task window for ten seconds. Then it charges to the right across the Task window.

SPEED

SPEED "str"

NOTES

SPEED controls how fast the keys specified by the ADDKEYS verb are fed to the application. SPEED also controls how fast characters are displayed by the CENTER and PRINT verbs.

The parameter string can be "Fast", "Medium" or "Slow". "Fast" represents no delay between characters, "medium" is 0.2 seconds delay and "slow" is 0.5 seconds delay between characters.

The parameter string can also represent the number of milliseconds delay between characters. To specify a 0.1 second delay between characters you would use the following statement.

```
SPEED "100"
```

The initial setting is "Fast".

Note that some programs, such as terminal emulators connected to hosts, may not accept keys at the fast rate.

EXAMPLE

```
SPEED "Fast"
PRINT "I am an Olympic typist"
SPEED "Medium"
PRINT "I have pudgy fingers"
SPEED "Slow"
PRINT "I have boxing gloves on"
```

In this example, the first message is printed on the screen with no delay between characters. The second message is printed on-screen with a .2 second delay between characters, and the third message is printed on-screen with one-half second delay between characters.

STACKMSG

```
STACKMSG "messageText"
```

NOTES

STACKMSG displays messageText as a message at the bottom of the Task window. If any messages are already displayed, then the new message appears above them.

The message is displayed in the current font.

If you stack several messages on top of one another, you should use the same font for all the messages. (Messages using fonts with different character heights may not stack properly.)

NOTE: If the Task window is not high enough to show the message, then no part of the message is displayed.

EXAMPLE

```
STACKMSG "Press any key to continue"
PAUSE ""
```

The STACKMSG statement displays a message telling the user to press a key when ready to continue.

STACKSIZE

```
size = STACKSIZE
```

NOTES

STACKSIZE is a function which returns the number of bytes left on the CPU stack. STACKSIZE is normally used in special debugging situations.

EXAMPLE

```
PRINT "The number of bytes left is  " + STR$(STACKSIZE)
DELAY 3
```

This program prints the number of bytes left on the stack by converting the numeric value of STACKSIZE to a printable string.

STOP

STOP

NOTES

STOP causes GRiDTask to stop running. The application window is returned to its original size.

The other condition causing GRiDTask to stop running is when it reaches the end of its main program file.

EXAMPLE

```
mainMenu$ = "Status Reports¦Mail¦Exit"
msg$      = "Select activity and Confirm"

TASKWINDOW 0,0,-1,-1
WHILE TRUE
  choice = DOMENU (msg$, mainMenu$)
  IF choice = 1
    TASK "status"
  ELSE
  IF choice = 2
    TASK "mail"
  ELSE
  IF choice = 3
    STOP
  ENDIF: ENDIF: ENDIF
WEND
```

This example represents the main body of a Task program. It displays a menu with three items. If the third item, "Exit", is selected, GRiDTask executes a STOP statement and stops.

STR$

```
num1$ = STR$ (num)

         OR

num2$ = STR$ (num,precision)
```

NOTES

STR$ is a string function which converts a number to a string of decimal characters.

STR$ can have one or two parameters. The first parameter is the number to be converted. The optional second parameter indicates how many digits after the decimal point to display. If this second parameter is omitted, then STR$ returns a string containing the minimum number of characters required to precisely represent the number. See the examples.

To convert a string of digits to a decimal number, use the VAL function.

EXAMPLES

```
STR$ (9)          =>       "9"
STR$ (9/8)        =>       "1.125"
STR$ (9/8,0)      =>       "1"
STR$ (9/8,2)      =>       "1.13"
STR$ (9/8,6)      =>       "1.125000"
```

The result of each STR$(..) is shown above.

SUBJECT$

```
sub$ = SUBJECT$
```

NOTES

SUBJECT$ is a string function which returns the current Subject. This is the Subject of the last file accessed through GRiDTask or the application window. The trailing backquote is included as part of the string.

e.g. "Imported Beers`"

EXAMPLE

```
sub$ = SUBJECT$
```

This sets sub$ = to the current Subject.

SUBSTITUTE$

```
newStr$ = SUBSTITUTE$ (oldStr$, find$, replaceWith$)
```

NOTES

SUBSTITUTE$ is a string function. It replaces all instances of the string find$ in the string oldStr$ with the string replaceWith$. The new character string containing the substitution(s) is returned to newStr$. To find all occurrences of strings with a common attribute - such as any group of characters that start and end with "#" - you may include a wildcard character in find$.

```
e.g.    wild$ = CHR$(247)
        find$ = "#" + wild$ + "#"
```

The number 247 is the decimal value of the wildcard character.

EXAMPLE

```
oldStr$              =     "Mqrgqritq"
find$                =     "q"
replaceWith$         =     "a"

newStr$ = SUBSTITUTE$ (oldStr$, find$, replaceWith$)
PRINT  newStr$
```

In this example, GRiDTask searches through the string "Mqrgqritq" (oldStr$) for "q" (find$). Each "q" is replaced with "a" (replaceWith$). Finally, "Margarita" (newStr$) is printed.

SUBSTRING$

```
sub$ = SUBSTRING$ (source$, delimiter$, itemNumber)
```

NOTES

SUBSTRING$ is a string function which allows you to extract a portion of a string. Each word or group of characters to be extracted must be separated by a special character - delimiter$.

The following parameters are used in a SubString statement:

source$ contains the string of characters from which the items are extracted. source$ remains unchanged.

delimiter$ separates each item in source$.

itemNumber is a number giving the position of the item. The number 1 represents the first item, 2 the item after the first delimiter, 3 the item after the second delimiter, and so forth. If itemNumber is greater than the number of items present in source$, a zero-length string is returned.

EXAMPLE

```
source$        =   "frogs|toads|lizards|wombats"
delimiter$     =   "|"
itemNumber     =   3

PRINT SUBSTRING$ (source$, delimiter$, itemNumber)
```

In this example, GRiDTask prints the third item (lizards) in source$.

TASK

TASK "pathname"

NOTES

The TASK statement tells GRiDTask to jump to the beginning of the specified file for execution. These files are sequences of GRiDTask statements. The TASK statement acts like an Include statement in a compiled program. Execution returns to the main file when the end of the specified file is reached. This verb can be nested to any level.

If the pathname has no Kind, then the Kind "Task" is assumed. If no Device or Subject is specified, then GRiDTask looks in the current Device and Subject.

Note that files specified in TASK statements may have Kind "Text". Thus, they can be easily brought into GRiDWrite for modification, and can be executed directly using TASK statements.

The TASK verb provides an important set of advantages. It allows GRiDTask programs to be broken into many pieces. The advantages of this are :

1) Faster development time
2) Easier debugging
3) Easier maintenance

The TASK verb is somewhat slow due to the file I/O involved, so you shouldn't use it in a tight loop. It is not a general purpose procedure call. See "Procedures" in this chapter (4) for information on GRiDTask Procedures.

EXAMPLES

```
confirm$ = CHR$(141)
;------------------------------------------------
;   Mail menu
;------------------------------------------------
choice = DOMENU (msg$,"Send Mail!Retrieve Mail")
IF LastKey$ = confirm$
  IF (choice = 1)
     TASK "MailSend~Text~"
  ELSE
  IF (choice = 2)
    TASK "MailRetrieve~Text~"
  ENDIF:ENDIF
ENDIF
```

This example displays a menu showing two activities related to an electronic mail system. If the menu is Confirmed, then one of two files is executed. If the menu is not Confirmed, then the program continues on.

Because the two files "MailSend~Text~" and "MailRetrieve~Text~" are separate from this program, they can be developed and debugged independently, by someone else, at a later or earlier time. They can even be files created by GRiDRecord without alteration.

Also, this program executes much faster than a program where all the code is in one file. For example, suppose the statement "IF (choice = 1)" evaluated FALSE. GRiDTask would have to look at each subsequent line, searching for the corresponding ELSE. In this case, it only has to look at two lines before finding the ELSE. If the code in "MailSend~Text~" followed the IF statement, then GRiDTask would have to look at every line before finding the ELSE. This could take a long time.

"MailSend~Text~" and "MailRetrieve~Text~" can have any file Kind. Giving them a Kind of "Text" makes it very easy to edit them during a development phase.

```
ERROCODE = 0
;----------------------------------------
;        Log on to Dow Jones
;----------------------------------------
ADDKEYS "- - - - - - - -"
IF (ERRORCODE <> 0)
  TASK "ErrorHandler"
ELSE
;----------------------------------------
;     Retrieve and reformat data
;----------------------------------------
ADDKEYS "- - - - - - - -"
IF (ERRORCODE <> 0)
  TASK "ErrorHandler"
ELSE
;----------------------------------------
;          Print Report
;----------------------------------------
ADDKEYS "- - - - - - - -"
IF (ERRORCODE <> 0)
  TASK "ErrorHandler"
ENDIF:ENDIF:ENDIF
```

This program illustrates the use of the TASK verb to implement a simple procedure call. It also illustrates how you can handle errors in a complex, error-prone activity. If the ADDKEYS statements were completed this program could use GRiDTerm to retrieve data, reformat the data, and print it in a report. At three places in the program, the ERRORCODE variable is checked.

If an error occurs, the program branches to a file called "ErrorHandler~Task~". ErrorHandler acts like a procedure in that you can "call" it from different places. It also receives a pseudo-parameter, the global variable ERRORCODE. The ErrorHandler program could take appropriate action based on the error that occurred.

TASKWINDOW

TASKWINDOW topleftx, toplefty, widthX, heightY

NOTES

TASKWINDOW sets the size of the Task window. The application window defaults to the largest remaining rectangle outside the Task window. The parameters specify the top-left corner and size of the new Task window. The coordinates of the top-left point are in absolute screen coordinates.

Note that after changing the size of the Task window, you may want the application to update its display to fit the new application window. See the UPDATESCREEN verb.

The following occurs when you use the TASKWINDOW verb.

1. The contents of the current Task window are erased.
2. The Task window frame moves to its new location.
3. The new Task window is framed and erased.
4. The cursor is positioned in the upper left corner of the new window. (See CLEARSCREEN)
5. The application window is set to the largest available space outside the Task window. A four pixel gap is left between the two windows. The application window may be to the right, left, above, or below the Task window.

If the "widthX" and "heightY" parameters are -1, then the Task window automatically extends to the right and bottom edges of the screen. This saves the effort of calculating window extents, and reduces the changes necessary to make Task programs run on computers with different size screens.

Initially the Task window is set to a zero height line at the bottom of the screen, and the application window occupies the entire screen.

EXAMPLES

```
TASKWINDOW 0,0,-1,-1
```

This statement set the Task window to occupy the entire screen. The application window is set to 0,0,0,0 (a zero height rectangle at the top of the screen). An application running in the application window is not visible, but can still function.

```
TASKWINDOW 0, WINDOWHEIGHT/2 , -1, -1
```

This statement sets the Task window to the bottom half of the screen. The application window occupies the upper half of the screen.

```
TASKWINDOW WINDOWWIDTH/2 , 0 , -1 , -1
```

This statement sets the Task window to the right half of the screen. The application window occupies the left half of the screen.

```
TASKWINDOW 0, WINDOWHEIGHT , WINDOWWIDTH ,0
```

This statement sets the Task window to a zero height rectangle at the bottom of the screen. Anything displayed in the Task window is not visible. The application rectangle occupies the entire screen. (If either extent of the Task window is zero, then no gap is left between the two windows.)

```
TASKWINDOW 0,0,0,0
```

This statement sets the Task window to a point at the top-left corner of the screen. The application window is set to 0,0,0,0 (a zero height rectangle at the top of the screen). An application running in the application window is not visible, but can still function.

TESTKEYS

TESTKEYS "encodedKeyStr"

NOTES

TESTKEYS is specifically designed to be used in tutorials. Most users of GRiDTask will not have a need to use TESTKEYS.

TESTKEYS operates very similarly to the ADDKEYS verb. However, the keystrokes are first displayed as keycaps in the Task window at the current cursor location. As the user presses the correct keys, they are passed to the application and the keys displayed on the screen are un-highlighted. If an incorrect key is pressed, an appropriate message is displayed automatically. The size of the graphics used to display the keycaps on the screen are based on the current font.

After executing a TESTKEYS statement the cursor will be positioned one line below where it was at the start of the TESTKEYS statement.

If you use the TESTKEYS verb, you must make modifications to your font. Certain character values must display arrows (to represent the arrow keys). Two different character values must represent each arrow. The following table shows which characters must be modified.

Character value	Font display
11H, 0C4H	downArrow
12H, 0C5H	upArrow
13H, 0C6H	leftArrow
14H, 0C7H	rightArrow

EXAMPLE

```
TASKWINDOW 0,0,-1,-1
PRINT "The RETURN key moves the outline down to the
next item in the File form."
TESTKEYS "¦,¦,"
PRINT "That was great, press any key to continue."
PAUSE ""
```

In this example, the message tells the user to press RETURN. The TESTKEYS statement displays two keycaps. As the user presses, RETURN twice, the keycaps are unhighlighted. A message is displayed if the user does not press RETURN.

TIME$

```
clock$ = TIME$
```

NOTES

TIME$ is a string function that returns the current time.

```
The time string is formatted as   hh/mm/ss  a.m.
                             or   hh/mm/ss  p.m.
```

If the time is not correct, use GRiDManager to set the correct time and date.

EXAMPLE

```
PRINT "The time is now " + TIME$
  TASK  "SomeTimeConsumingTask~Text~"
PRINT "SomeTimeConsumingTask has just finished."
PRINT "The time is now " + TIME$
DELAY 2
```

This program illustrates how you could use the TIME$ verb to time how long it takes to execute a GRiDTask program called "SomeTimeConsumingTask~Text~".

TITLE

```
TITLE  "str"
```

NOTES

TITLE is specifically designed to be used in tutorials. Most users of GRiDTask will not have a need to use TITLE.

TITLE creates a title line in the Task window. The parameter string is displayed at the top-left corner of the window and a line is drawn underneath it extending all the way across the window.

The TITLE verb is intended to be used at the beginning of every page in a tutorial page sequence. After a TITLE statement, the cursor is positioned at the left edge of the window, just below the title line.

EXAMPLE

```
TITLE "Welcome to the world of GRiD"
```

This puts a title at the top of the TASKWINDOW.

TRUE

variable = TRUE

NOTES

The function TRUE returns the value -1, and the function FALSE returns the value 0. TRUE and FALSE can be used to set the values of boolean variables.

GRiDTask allows real variables to be used in boolean expressions. In this context an even number is false, and an odd number is true (the low-order bit is used to determine boolean values - an odd number has a low-order bit = 1, and an even number has a low-order bit = 0). These real variable values are not to be confused with the built-in functions TRUE and FALSE.

EXAMPLE

```
IF temperature >= 70
   WARM = TRUE
ELSE
   WARM = FALSE
ENDIF
......
......
IF WARM
    TASK  "`w`GoToBeach`SantaCruz~Text~"
ELSE
    TASK  "`w`GoSkiing`LakeTahoe~Text~"
ENDIF
```

In this example, if the temperature is 70 or above, then WARM is true, and the file "GoToBeach`SantaCruz~Text~" is executed. If the temperature is < 70, then WARM is FALSE, and the file "GoSkiing`LakeTahoe~Text~" is executed.

```
UPDATESCREEN
```

NOTES

UPDATESCREEN sends a WindowUpdate key to the application window. This tells the application to recheck its window dimensions and redraw its display accordingly.

In general you should use UPDATESCREEN after setting new window boundaries with the TASKWINDOW verb. However, in some cases this may not be necessary. See the example below.

If the application is waiting for the FILEFORM verb then UPDATESCREEN has no effect. Keys are not passed to the application window when the application is waiting for the FILEFORM verb.

Any menu or form being displayed by the application is removed when you execute the UPDATESCREEN verb.

EXAMPLE

```
TASKWINDOW 0,0,-1,-1          ; the Task window has the entire
screen
PRINT "Please wait...."
FILEFORM  "Customer~Database~00"
ADDKEYS "¦."                  ; confirm and wait for GRiDFile
TASKWINDOW 0,WINDOWHEIGHT/2,-1,-1
UPDATESCREEN
```

This program opens up the Task window to occupy the entire screen. It then retrieves a database and GRiDFile. Once the database has been retrieved the application window is opened to the top half of the screen and a WindowUpdate key is sent to GRiDFile. GRiDFile then displays properly within the new application window.

It was not necessary to use UPDATESCREEN after the first TASKWINDOW statement because the application window was not visible anyway.

VAL

```
num = VAL (stringX$)
```

NOTES

VAL converts a string to a real number. The VAL function accepts strings of digits. These strings may have digits following a decimal point. If the string doesn't convert, then it returns 0.

To convert an integer to a string use the STR$ function.

EXAMPLES

```
stringX$ = "12.55"
number1 = VAL (stringX$)
number2 = VAL ("12.55")
```

After these statements are executed, number1 and number2 have the same value, 12.55.

```
StringX$ = "500"
number3  = 2 * VAL(StringX$)
```

In this example, number3 gets the value 1000.

WEND

WEND

NOTES

WEND is used with WHILE verbs. WEND marks the end of a block of GRiDTask statements called a WHILE/WEND loop.

When using an WEND verb, it must be the only word on the line.

See WHILE/WEND for more information.

EXAMPLE

The example from the "WHILE/WEND" verb is repeated here for convenience.

```
found = 0
i     = 0
WHILE i < LEN(inputstringX$)
  i = i + 1
  IF MID$(inputstringX$, i, 1) = "?"
    found = found + 1
  ENDIF
WEND
PRINT "I found " + STR$(found) + " question marks!"
```

This example counts the number of question marks in a string (inputstringX$). It prints a message indicating how many were found.

WHILE
WEND

```
WHILE <expression with TRUE or FALSE value>
  .....
  GRiDTask statement(s)
  .....
WEND
```

NOTES

The WHILE and WEND statements create a program loop that continues to execute as long as the expression following the WHILE statement evaluates to true. If the expression evaluates to false, then execution continues with the first statement following the WEND statement.

You can nest WHILE / WEND statements to any depth. GRiDTask matches each WEND with the most recent WHILE. If you have unequal numbers of WHILE and WEND statements, an error occurs.

EXAMPLE

```
found = 0
i     = 0
WHILE i < LEN(inputstringX$)
  i = i + 1
  IF MID$(inputstringX$, i, 1) = "?"
    found = found + 1
  ENDIF
WEND
PRINT "I found " + STR$(found) + " question marks!"
```

This example counts the number of question marks in a string (inputstringX$). It prints a message indicating how many were found.

WINDOWHEIGHT

```
size  = WINDOWHEIGHT
```

NOTES

WINDOWHEIGHT is an integer function which returns the vertical height (in pixels) of the window available when the GRiDTask program began execution. WINDOWHEIGHT may be used to calculate reasonable proportions for the Task or application windows.

EXAMPLES

```
TASKWINDOW  0,WINDOWHEIGHT/2,-1,-1
```

In this example, the TASKWINDOW is set to the bottom half of the available screen, regardless of the available screen size.

WINDOWMOTION

```
WINDOWMOTION "ON" or "OFF"
```

NOTES

WINDOWMOTION determines whether the Task window frame is visible as it moves to its new position as specified in a TASKWINDOW statement.

You may specify either "ON" or "OFF" (the default is "ON"). When WINDOWMOTION is ON, the Task window frame is visible as it moves to its new position. When WINDOWMOTION is "OFF", the window frame appears in its new position without visible motion from its old position. With WINDOWMOTION "Off", the TASKWINDOW command executes more rapidly.

EXAMPLE

```
TASKWINDOW 100,100,-1,-1
DELAY 2
WINDOWMOTION "OFF"
TASKWINDOW 0,0,-1,-1
```

In this example, the Task window is set to the lower right portion of the screen. After two seconds, the Task window increases to the full screen size, but is not visible during that change.

WINDOWWIDTH

```
width = WINDOWWIDTH
```

NOTES

WINDOWWIDTH is an integer function which returns the horizontal width (in pixels) of the window available when the GRiDTask program begins execution. WINDOWWIDTH may be used to calculate reasonable proportions for the Task or application windows.

EXAMPLE

```
TASKWINDOW 0, 0, WINDOWWIDTH/3, -1
```

In this example, the TASKWINDOW is set to the left third portion of the screen regardless of the screen size.

WRITEFILE

```
WRITEFILE information$, pathname$
```

NOTES

WRITEFILE writes information$ to the destination file specified by pathname$. GRiDTask creates the destination file if it doesn't already exist. If it does exist, GRiDTask overwrites the data currently in the file.

Note that if you do not specify a Kind in pathname$, then the Kind Text is assumed. If you do not specify a Device or Subject, then the Device and Subject of the last file accessed through GRiDTask or the application window is assumed.

WRITEFILE stes the ERRORCODE variable to the number of any error that occurred. If no error occurs, then the ERRORCODE variable is set to (0) zero.

EXAMPLE

```
information$           =    "A slow rabbit"
pathname$              =    "QuickBrownFox~Text~"
WRITEFILE  information$ , pathname$
```

In this example, the file "QuickBrownFox~Text~" is created in the current Device and Subject and has the contents "A slow rabbit". If the file already existed, it would have the same contents, regardless of its original data.

Section Two — Mathematical Functions

This section contains descriptions of the GRiDTask verbs used to perform mathematical operations. The section is arranged alphabetically.

ACOS

ACOS(number)

NOTES

ACOS is a function which returns the arc-cosine of any number between -1 and +1 inclusive.

EXAMPLE

```
PRINT  STR$(ACOS(0.5),2)
PAUSE ""
```

In this example, the value 1.05 - the arc-cosine of 0.5 - is printed. The answer 1.05 is the decimal representation of the value PI/3 . Note that the second parameter of STR$ - 2 - specifies two decimal places.

ATN

ATN(number)

NOTES

ATN is a function which returns the arc-tangent of a number.

EXAMPLE

```
PRINT  STR$ (ATN(1))
```

In this example, 0.785.... - the arc-tangent of 1 - is printed. 0.785.... is the decimal representation of PI/4 . Note that the displayed precision of 0.785... was not set in the STR$ function.

COS

```
COS(angle)
```

NOTES

COS is a function which returns the cosine of an angle. Specify the angle in radians - a full circle, or 360^{o}, is 2 PI radians.

EXAMPLE

```
PRINT "COS(PI/4) = "  + STR$(COS(PI/4),3)
```

In this example, "COS(PI/4) = 0.707" is printed.

EXP

EXP(exponent)

NOTES

EXP is a function which returns the value e raised to the power specified by "exponent".

EXAMPLES

```
PRINT STR$(EXP(3),2)
```

In this example, e^3 (20.09) is printed.

```
PRINT STR$(EXP(LOG(10)*3))
```

In this example, e is raised to the power (LOG(10) * 3). This is the same as 10 raised to the power 3. Thus, 1000 is printed.

LOG

LOG(number)

NOTES

LOG is a function which returns the logarithm to the base e of "number".

EXAMPLE

```
x  =  LOG(10)
```

In this example, the variable x gets the value 2.3025...

LOG10

LOG10(number)

NOTES

LOG10 is a function which returns the logarithm to the base 10 of "number".

EXAMPLE

```
y = LOG10(100) + LOG10(10000)
```

In this example, y gets the value 2 + 4, or 6.

PI

PI

NOTES

PI provides the well-known value "pi", the ratio of the circumference of a circle to its radius.

EXAMPLE

```
PRINT "The area of a 12-inch pizza is"
PRINT STR$((PI * 6 * 6),2) + " square inches"
```

In this example, the area of a pizza is calculated using PI and the formula for the area of a circle.

RND

RND(1)

NOTES

RND is a function which returns a random number between 0 and 1. Any number can be placed in parentheses: the result is the same. Note that RND(0) gives the last random number generated.

EXAMPLE

```
; First flip
flip1 = RND(1)
IF flip1 < 0.5
  PRINT " You got heads, flip again "
ELSE
IF flip1 >= 0.5
  PRINT " You got tails, you're out "
ENDIF:ENDIF

IF flip1 <= 0.5  ; Allow a second flip, if first one = heads
  flip2 = RND(1)
  IF flip2 > 0.5
    PRINT " You got heads again, you win "
  ELSE
  IF flip2 <= 0.5
    PRINT " You got tails, pay to play again "
ENDIF:ENDIF:ENDIF
```

In this example, flip1 gets a random value between 0 and 1. Since half the numbers returned by RND are less than .5, we can call those outcomes "heads" and any values >= 0.5 "tails". If heads was flipped the first time, the player can flip again. A second "heads" wins the game. Note that for flip2, "heads" is any RND value > 0.5 .

ROUND

ROUND(number)

NOTES

ROUND is a function which rounds a real number to an integer.

EXAMPLE

```
numberpeople = 27
numbertables = ROUND (numberpeople/8) + 1
```

In this example, there are 27 people attending a banquet. Since 8 people can be seated at a table, the minimum number of tables required is 3 + 1, or 4.

SIN

```
SIN(angle )
```

NOTES

SIN is a function which returns the sine of an angle. Specify the angle in radians - a full circle, or 360°, is 2 PI radians.

EXAMPLE

```
PRINT "SIN(PI/3) = "  + STR$(SIN(PI/3),3)
```

In this example, the value 0.866 is printed.

SQR

```
SQR(number)
```

NOTES

SQR is a function which returns the square root of a number.

EXAMPLE

```
PRINT "The square root of 169 is " + STR$(SQR(169))
```

In this example, "The square root of 169 is 13" is printed.

TAN

TAN(number)

NOTES

TAN is a function which returns the tangent of a number.

EXAMPLE

```
PRINT "The tangent of PI/4 is " + STR$(TAN(PI/4),4)
```

In this example, "The tangent of PI/4 is 1.0000" is printed.

TRUNC

TRUNC(number)

NOTES

TRUNC is a function which truncates a number. The returned value is an integer, and its absolute value is less than or equal to the absolute value of the original number.

EXAMPLE

```
bl$ = "                           "
PRINT "TRUNC(10.7)                TRUNC(-10.7) "
PRINT STR$(TRUNC(10.7)) + bl$ +  STR$(TRUNC(-10.7))
```

In this example, the output looks as follows:

```
 TRUNC(10.7)                TRUNC(-10.7)
 10                           -10
```

Appendix A

GRiDTask Verb Summary

GENERAL PURPOSE VERBS

ADDKEYS	ADDKEYS "encodedKeyStr"
APPENDFILE	APPENDFILE addString$, pathname$
ASC	num = ASC (anyString$)
BREAK	BREAK
BREAKONKEY	BREAKONKEY key$
BREAKRESET	BREAKRESET
CELL$	contents$ = CELL$
CENTER	CENTER "text....."
CHANGEKIND$	newPathName$ = CHANGEKIND$ (pathName$, kind$)
CHARWIDTH	width = CHARWIDTH
CHR$	stringX$ = CHR$ (num)
CLEARMSG	CLEARMSG
CLEARSCREEN	CLEARSCREEN
COMMANDLINE	COMMANDLINE command$, secondsDelay
COMMENT	; Place text here
CONCHARIN$	ch$ = CONCHARIN$
COPYFILE	COPYFILE sourcePath$, destinationPath$
CURSOR	CURSOR x, y
CURX , CURY	CURSOR CURX + 5, CURY - 10
DATE$	today$ = DATE$
DELAY	DELAY seconds
DEVICE$	dev$ = DEVICE$
DIRECTORY$	list$ = DIRECTORY$ (mode, path$, match$, delimiter$, sortOrder)
DO	DO taskStatements$
DOFORM$	form$ = DOFORM$ (msg$,form$,numLines)
DOMENU	choice = DOMENU (msg$, item$)
ELSE	ELSE
ENDIF	ENDIF
ENDP	ENDP
ERASEBOX	ERASEBOX topleftX, topleftY, extentX, extentY
ERASEFILE	ERASEFILE pathname$
ERRORCODE	ERRORCODE = number or number = ERRORCODE
ERRORSTR$	err$ = ERRORSTR$(errorNum)
FALSE	variable = FALSE
FILEFORM	FILEFORM "pathname"
FINDTITLE$	path$ = FINDTITLE$ ("Title~Kind~")
FONT	FONT "fontPathName"
FORMCHOICE	number = FORMCHOICE (form$, itemNumber)
FORMCHOICE$	choice$ = FORMCHOICE$ (form$, itemNumber)
FRAMEBOX	FRAMEBOX topleftX, topleftY, extentX, extentY
FREEFONT	FREEFONT "fontPathName"
GETFILE$	pathName$ = GETFILE$ (msg$)

IF/ELSE/ENDIF	IF <exp> / stmts / ELSE / stmts / ENDIF
INKEY$	someKey$ = INKEY$
INPUT$	value$ = INPUT$(prompt$, length, height, initValue$)
INSTALL	INSTALL pathname$
INSTR	location = INSTR (start, source$, find$)
INVERTBOX	INVERTBOX topleftX, topleftY, extentX, extentY
INVERTLINE	INVERTLINE X1, Y1, X2, Y2
ITEMCOUNT	numItems = ITEMCOUNT (list$, separater$)
LASTKEY$	key$ = LASTKEY$ or LASTKEY$ = stringX$
LASTMESSAGE$	LASTMESSAGE$ = message$ or message$ = LASTMESSAGE$
LEN	num = LEN (stringX$)
LINEHEIGHT	height = LINEHEIGHT
LOCATE	LOCATE x, y
MEMORY	space = MEMORY
MID$	portion$ = MID$ (wholeString$, start, length)
PAINT	PAINT x, y, "pathname"
$PARSEONLY	$PARSEONLY
PASSKEYS	PASSKEYS keysToPass$, keysToTerminate$
PAUSE	PAUSE keysToTerminate$
PLAY	PLAY musicStr$
PRINT	PRINT "text...."
PROCEDURE	PROCEDURE procedureName param1, param2$, ...
READFILE$	contents$ = READFILE$ (pathname$)
RETURN	RETURN
SCROLL	SCROLL distance, speed
SCROLLBOX	SCROLLBOX topleftX, topleftY, extentx, extenty, "direction", distance, speed
SPEED	SPEED "str"
STACKMSG	STACKMSG "messageText"
STACKSIZE	stackSpace = STACKSIZE
STOP	STOP
STR$	num1$ = STR$(num) or num2$ = STR$(num,precision)
SUBJECT$	sub$ = SUBJECT$
SUBSTITUTE$	newStr$ = SUBSTITUTE$ (oldStr$, find$, replaceWith$)
SUBSTRING$	sub$ = SUBSTRING$ (source$, delimiter$, itemNumber)
TASK	TASK "pathname"
TASKWINDOW	TASKWINDOW topleftX, topleftY, extentx, extenty
TESTKEYS	TESTKEYS "encodedKeyStr"
TIME$	clock$ = TIME$
TITLE	TITLE "title text..."
TRUE	variable = TRUE
UPDATESCREEN	UPDATESCREEN
VAL	num = VAL (stringX$)
WEND	WEND
WHILE	WHILE <exp> / stmts / WEND
WINDOWHEIGHT	size = WINDOWHEIGHT
WINDOWMOTION	WINDOWMOTION "ON" or "OFF"

WINDOWWIDTH	width = WINDOWWIDTH
WRITEFILE	WRITEFILE information$, pathname$

MATHEMATICAL FUNCTIONS

ACOS	Arc Cosine	ACOS(number)
ATN	Arc Tangent	ATN(number)
COS	Cosine	COS(angle)
EXP	Exponential	EXP(exponent)
LOG	Natural Logarithm	LOG(number)
LOG10	Base 10 Logarithm	LOG10(number)
PI	The constant Pi	PI
RND	Random number	RND(1)
ROUND	Round to an integer	ROUND(number)
SIN	Sine	SIN(angle)
SQR	Square Root	SQR(number)
TAN	Tangent	TAN(number)
TRUNC	Truncate to an integer	TRUNC(number)

VERBS INSTALLED IN DataEntryForms LIBRARY (SEE APPENDIX I)

DISPOSEFORM	DISPOSEFORM formNum
EDITFORM$	key$ = EDITFORM$ (formNum, topLeftX, topLeftY, widthX, heightY, mode)
FORMINIT	formNum = FORMINIT (formStr$)
FORMINITFROMFILE	formNum = FORMINITFROMFILE (pathName$)
GETALLFIELDS$	values$ = GETALLFIELDS$ (formNum, delimiter$)
GETCURRENTFIELD	currentField = GETCURRENTFIELD (formNum)
GETFIELDVALUE$	value$ = GETFIELDVALUE$ (formNum, currentField)
INDEXFROMNAME	fieldIndex = INDEXFROMNAME (formNum, name$)
NAMEFROMINDEX$	name$ = NAMEFROMINDEX$ (formNum, currentField)
PARSEFORM$	parsedSpec$ = PARESEFORM$ (fileSpec$)
PRINTFORM	error = PRINTFORM (formNum, printMode, destination$,topMargin, bottomMargin,leftMargin, printSize, formFeed)
SETALLFIELDS	SETALLFIELDS formNum, values$, delimiter$
SETCURRENTFIELD	SETCURRENTFIELD formNum, currentField
SETFIELDVALUE	SETFIELDVALUE formNum, fieldIndex, newValue$

Appendix B

Encoded Keystroke Chart

ENCODING KEYSTROKES

The chart on the next page shows how keystrokes are represented in a string when used with the ADDKEYS, PAUSE, PASSKEYS, BREAKONKEY and TESTKEYS verbs. The general scheme is as follows:

- Alpha, numeric, and punctuation characters require no special encoding.

- CODE keys are represented by a vertical bar followed by a lowercase character. The lowercase character is the key that is pressed with the CODE key. For example:

  ¦e represents CODE-E
  ¦p represents CODE-P

- ARROW keys are represented by a vertical bar followed by an uppercase character. Use the chart on the next page to determine which uppercase character to use. For example:

  ¦E represents UpArrow
  ¦X represents CODE-SHIFT-LeftArrow

- CTRL keys are represented by a back slash followed by a lowercase character. The lowercase character is the key that is pressed with the CTRL key. For example:

  \s represents CTRL-S

  There are some special symbols shown in the chart. They can be typed as follows:

TO GET	PRESS THESE KEYS
¦	CODE and SHIFT and ;
\	CODE and SHIFT and '
"ESCAPE"	CTRL and ESC

Encoded Key String	**Keyboard Function**
||	|
\\\\	\\
|,	RETURN
|.	CODE-RETURN (Confirm)
	ESC
|	CODE-ESC
\\key	CTRL-key
|a - |z	CODE-A through CODE-Z
|0 - |9	CODE-0 through CODE-9
|)	CODE-SHIFT-0
|!	CODE-SHIFT-1
|@	CODE-SHIFT-2
|#	CODE-SHIFT-3
|$	CODE-SHIFT-4
|%	CODE-SHIFT-5
|^	CODE-SHIFT-6
|&	CODE-SHIFT-7
|*	CODE-SHIFT-8
|(	CODE-SHIFT-9
|+	CODE +
|-	CODE -
|=	CODE =
|?	CODE ?
|D	DownArrow
|E	UpArrow
|F	LeftArrow
|G	RightArrow
|N	SHIFT-DownArrow
|O	SHIFT-UpArrow
|P	SHIFT-LeftArrow
|Q	SHIFT-RightArrow
|R	CODE-DownArrow
|S	CODE-UpArrow
|T	CODE-LeftArrow
|U	CODE-RightArrow
|V	CODE-SHIFT-DownArrow
|W	CODE-SHIFT-UpArrow
|X	CODE-SHIFT-LeftArrow
|Y	CODE-SHIFT-RightArrow

Appendix C

Key Value Chart

The chart on the next page shows the numbers which correspond to each combination of keys you can press on the keyboard.

As an example, suppose you want to read a key from the keyboard and check if it's a CODE-P. "CODE-P" is 240 in decimal according to the chart. You can use this number as follows:

```
ch$ = ConCharIn$
IF ch$ = CHR$(240)
  TASK "Properties~Task~"
ENDIF
```

Note that all number constants in GRiDTask must be in decimal.

Key	UnSHIFTed	SHIFT	CODE	CODE SHIFT	CTRL	SHIFT CTRL	CODE CTRL	CODE SHIFT CTRL
'	39 (')	34 (")	96 (`)	92 (\)	7 (BEL)	2 (STX)	0 (NUL)	28 (FS)
,	44 (,)	60 (<)	91 ([)	123 ({)	12 (FF)	28 (FS)	27 (ESC)	27 (ESC)
-	45 (-)	95 (_)	173	127 DEL	13 (CR)	31 (US)	141	31 (US)
.	46 (.)	62 (>)	93 (])	125 (})	14 (SO)	30 (RS)	29 (GS)	29 (GS)
/	47 (/)	63 (?)	191	191	15 (SI)	31 (US)	159	159
0	48 (0)	41 ())	176	169	16 (DLE)	9 (HT)	144	137
1	49 (1)	33 (!)	177	161	17 (DC1)	1 (SOH)	145	129
2	50 (2)	64 (@)	178	192	18 (DC2)	0 (NUL)	146	128
3	51 (3)	35 (#)	179	163	19 (DC3)	3 (ETX)	147	131
4	52 (4)	36 ($)	180	164	20 (DC4)	4 (EOT)	148	132
5	53 (5)	37 (%)	181	165	21 (NAK)	5 (ENQ)	149	133
6	54 (6)	94 (^)	182	222	22 (SYN)	3 (RS)0	150	158
7	55 (7)	38 (&)	183	166	23 (ETB)	6 (ACK)	151	134
8	56 (8)	42 (*)	184	170	24 (CAN)	10 (LF)	152	138
9	57 (9)	40 (()	185	168	25 (EM)	8 (BS)	153	136
;	59 (;)	58 (:)	126 (~)	124 (¦)	27 (ESC)	26 (SUB)	30 (RS)	28 (FS)
=	61 (=)	43 (+)	189	171	29 (GS)	11 (VT)	157	139
A	97 (a)	65 (A)	225	225	1 (SOH)	1 (SOH)	129	129
B	98 (b)	66 (B)	226	226	2 (STX)	2 (STX)	130	130
C	99 (c)	67 (C)	227	227	3 (ETX)	3 (ETX)	131	131
D	100 (d)	68 (D)	228	228	4 (EOT)	4 (EOT)	132	132
E	101 (e)	69 (E)	229	229	5 (ENQ)	5 (ENQ)	133	133
F	102 (f)	70 (F)	230	230	6 (ACK)	6 (ACK)	134	134
G	103 (g)	71 (G)	231	231	7 (BEL)	7 (BEL)	135	135
H	104 (h)	72 (H)	232	232	8 (BS)	8 (BS)	136	136
I	105 (i)	73 (I)	233	233	9 (HT)	9 (HT)	137	137
J	106 (j)	74 (J)	234	234	10 (LF)0	10 (LF)	138	138
K	107 (k)	75 (K)	235	235	11 (VT)	11 (VT)	139	139
L	108 (l)	76 (L)	236	236	12 (FF)	12 (FF)	140	140
M	109 (m)	77 (M)	237	237	13 (CR)	13 (CR)	141	141
N	110 (n)	78 (N)	238	238	14 (SO)	14 (SO)	142	142
O	111 (o)	79 (O)	239	239	15 (SI)	15 (SI)	143	143
P	112 (p)	80 (P)	240	240	16 (DLE)	16 (DLE)	144	144
Q	113 (q)	81 (Q)	241	241	17 (DC1)	17 (DC1)	145	145
R	114 (r)	82 (R)	242	242	18 (DC2)	18 (DC2)	146	146
S	115 (s)	83 (S)	243	243	19 (DC3)	19 (DC3)	147	147
T	116 (t)	84 (T)	244	244	20 (DC4)	20 (DC4)	148	148
U	117 (u)	85 (U)	245	245	21 (NAK)	21 (NAK)	149	149
V	118 (v)	86 (V)	246	246	22 (SYN)	22 (SYN)	150	150
W	119 (w)	87 (W)	247	247	23 (ETB)	23 (ETB)	151	151
X	120 (x)	88 (X)	248	248	24 (CAN)	24 (CAN)	152	152
Y	121 (y)	89 (Y)	249	249	25 (EM)	25 (EM)	153	153
Z	122 (z)	90 (Z)	250	250	26 (SUB)	26 (SUB)	154	154
BACKSPACE	8	200	136	138	8 (BS)	136	136	138
RETURN	13	205	141	140	13 (CR)	141	141	140
Down-Arrow	196	206	210	214	132	142	146	150
ESC	27	27	155	155	27 (ESC)	27 (ESC)	155	155
LeftArrow	198	208	212	216	134	144	148	152
RightArrow	199	209	213	217	135	145	149	153
Spacebar	32 (SP)	32 (SP)	32 (SP)	32 (SP)	0 (NUL)	0 (NUL)	0 (NUL)	0 (NUL)
TAB	9	201	137	139	9	137	137	139
UpArrow	197	207	211	215	133	143	147	151

Appendix D

Suggestions On Getting Started

This appendix provides suggestions on how to get the most out of GRiDTask.

GETTING STARTED

If you have never used GRiDTask you may be asking, "What should I do first?".

The best thing you can do is to get an existing, debugged GRiDTask program and study it. Look at it with GRiDWrite until you understand everything it does.

If you have GRiDRecord and GRiDPlayback, you may want to record several series of keystrokes using GRiDRecord and then look at the "Keystrokes" files in GRiDWrite. You can modify the Keystrokes files with GRiDWrite and then play them back. You can also change the Kind of the Keystrokes files to "Task" and play them back with GRiDTask. They should work just the same.

The next step is to enter some new GRiDTask statements into the Task file. Start off with the TASKWINDOW verb and experiment with different size windows. You can look through the GRiDTask verb explanations to find other capabilities you might want to build into your GRIDTask applications.

EDITING TASK FILES

Creating a GRiDTask application is an iterative process. You edit Task program files, run them, and edit them again. You will spend such a large percentage of your time in this cycle that it pays to invest some time to facilitate it.

During development you should make all of your Task program files "Text" files. This allows you to edit them in GRIDWrite by just selecting them.

To execute a GRiDTask program file, have one file with Title and Kind " Main~Task~" and a second file with Title and Kind "Main~Text~". The first file, with Title and Kind " Main~Task~", contains a single GRIDTask statement.

```
      TASK "Main~Text~"
```

In this statement, GRiDTask is told to execute the file "Main~Text~". The space in front of " Main~Task~" causes this file to be placed ahead of other files in a file form list. Then you can easily find and select this file to run your GRiDTask application.

Note that within the second file, "Main~Text~", you may have TASK statements which cause other GRIDTask modules to be executed.

When you want to execute the program, select and confirm the file " Main~Task~".

After your program is completed and debugged you may want to change the program files from Kind "Text" to something else such as "Sub". Only the main file - the one you select to run the application - should have the Kind "Task". Thus if someone accidently selects one of the "Sub" files, they will get an Error 33 "File not found" (because no application has the Kind "Run Sub"). Thus, the GRIDTask application cannot execute beginning at the wrong location.

Appendix E

Reserved Words

The names in this list are reserved for GRiD's use. They should not be used as variable names in your GRiDTask programs. Included in this list are the functions and variables described in this manual, as well as names which are reserved for future use by GRiD. With few exceptions, if you try to use any of these names as variable names with either upper or lower-case letters, an error may occur.

ABS
ACOS
ADDKEYS
APPENDFILE
ASC
ASIN
ATN

BREAK
BREAKONKEY
BREAKRESET

CALL
CDBL
CELL$
CENTER
CHANGEKIND$
CHARWIDTH
CHR$
CINT
CLEARMSG
CLEARSCREEN
COMMANDLINE
CONCHARIN$
COPYFILE
COS
CSNG
CURSOR
CURX
CURY
CVD
CVS

DATE$
DELAY
DEVICE$
DIRECTORY$
DO
DOFORM
DOFORM$
DOMENU

ELSE
ENDIF
ENDP
EOF
EOLN
ERASEBOX
ERASEFILE
ERRORCODE
ERRORSTR$
EXP
EXIT

FALSE
FFPATCHOFF
FFPATCHON
FILEFORM
FINDTITLE$
FIRSTPAGE
FIX
FONT
FORMCHOICE
FORMCHOICE$
FRAMEBOX
FREEFONT

GETFILE$
GETPREFIX$

HEX$
HEXVAL

IF
IMP
INKEY$
INPUT$
INSTALL
INSTR
INT
INVERTBOX
INVERTLINE
ITEMCOUNT

LASTKEY$
LASTMESSAGE$
LEFT$
LEN
LINEHEIGHT
LOC
LOCATE
LOF
LOG
LOG10
LPRINT

MEMORY
MID$
MKD$
MKI$
MKS$

OCT$
OCTVAL

PAGE
PAINT
$PARSEONLY
PASSKEYS
PAUSE
PEEK
PI
PLAY
POKE
POS
PRINT
PROCEDURE

READFILE$
RETURN
RIGHT$
RND
ROUND

SCREENIMAGE
SCROLL
SCROLLBOX
SGN

SHAPE
SIN
SPACE$
SPEED
SQAR
SQR
STACKMSG
STACKSIZE
STOP
STR$
STRING$
SUBJECT$
SUBSTITUTE$
SUBSTRING$

TAB
TAN
TASK
TASKWINDOW
TESTKEYS
TIME$
TITLE
TRUE
TRUNC

UPDATESCREEN

VAL

WEND
WHILE
WINDOWHEIGHT
WINDOWMOTION
WINDOWWIDTH
WINDOWX
WINDOWY
WRITEFILE

Words Reserved When Using Data-Entry Forms Libraries

DISPOSEFORM

EDITFORM$

FORMINIT
FORMINITFROMFILE

GETALLFIELDS$
GETCURRENTFIELD
GETFIELDVALUE$

INDEXFROMNAME

NAMEFROMINDEX$

PARSEFORM$
PRINTFORM

SETALLFIELDS
SETCURRENTFIELD
SETFIELDVALUE

Appendix F

Error Messages

OVERVIEW

When GRiDTask encounters an error during execution, the following items occur:

1. Execution stops.
2. The Task window is set to the entire screen.
3. The statement where the error occurred is displayed.
4. A carat "^" is displayed to indicate where in the statement the error occurred.
5. The error type and number are displayed.
6. The error message (if any) is displayed.
7. A message saying "Confirm to exit GRiDTask" appears at the bottom of the window.
8. When you Confirm, GRiDTask exits.

GRiDTask has three different types of errors:

"GRiDTask errors"

"Evaluator errors"

"System errors".

GRiDTask ERRORS

GRiDTask errors are related to the operation and limits of GRiDTask. The possible errors are listed below.

0001	No program file was selected
0002	File is too long: max length is 64K
0003	Font limit exceeded: 4 fonts can be active at once
0004	This file was not found
0005	This font is not currently loaded
0006	You cannot free the current font
0008	CELL$ error: no field is currently active in application
0009	Too many parameters in function: max is 8
0010	Too many installed functions
0011	Invalid parameter: "ON" or "OFF" expected
0012	Invalid procedure name
0013	Stack underflow: reduce nesting
0014	Duplicate procedure definition
0015	Filename is too long
0016	Filename is zero length
0017	Unknown Error

EVALUATOR ERRORS

Evaluator errors occur while evaluating expressions. The most common one you see is probably "01A Unknown Command". If you spell the name of a function incorrectly, you get this error. The possible evaluator errors are listed below.

0001	Invalid expression
0002	Expression too long
0003	Invalid character
0004	Invalid value in expression
0005	Type mismatch in expression
0006	Bad function in expression
0007	Mismatched quotes in expression
0008	Out of memory
0009	Uninitialized variable or undefined function
0010	Syntax error in expression
0011	Feature not implemented
0012	Generation error
0013	Stack error while parsing expression
0014	Error while evaluating expression
0015	End of file encountered. Missing WEND for WHILE
0016	{Invalid Construct. Missing THEN for IF}
0017	End of file encountered. Missing END for IF
0018	ELSE encountered without matching IF
0019	WEND encountered without matching WHILE
0020	END encountered without matching IF
0021	Invalid assignment statement. "=" expected
0022	End of file encountered
0023	
0024	Undefined variable. String expected
0025	
0026	Invalid call to function
0027	Attempt to use a reserved word
0028	End of file encountered
0029	ENDP encountered outside of procedure
0030	Internal GRiDTask error
0031	Parameter list is too short
0032	Unknown error

SYSTEM ERRORS

System errors are GRiD-OS errors and are mostly related to the file system. For example, if you execute a TASK statement and the file specified does not exist, then you get system error 33, "File does not exist". Another example of a system error is "Out of memory". When a system error occurs, the error message displayed is retrieved from the file "@SystemErrors~Text~". If this file is not available, then only the error number is displayed.

Appendix G

Procedure Performance Issues

OVERVIEW

Performance

The following guidelines may aid in understanding GRiDTask performance issues. Certain operations such as loading fonts and initializing forms are not done by GRiDTask, so they cannot be made faster by altering the Task program.

The first time a statement is executed within a procedure, it executes at the same speed as it does from a file. Subsequent times, the same statement will execute anywhere from 5 to 10 times faster.

The reason for the difference in execution time is as follows. When a statement is first executed, a "thread" or internal form of the statement is saved. The next time(s) the statement is executed, the thread does not have to be recalculated.

A disadvantage of procedures is that they use more RAM. If a procedure is 100 bytes long in a file, it would probably take 150 to 200 bytes of RAM. The exact amount of RAM cannot be determined [without looking at specific code].

Executing From a File

Executing from a file is the slowest way to run a Task program.

The main advantage of executing from a file is that it uses the least amount of RAM. If you have enough RAM available, consider replacing Task statements with DO statements, as follows:

This	--->	TASK "filename"
Becomes This	--->	DO READFILE ("filename")

Executing statements from a string instead of directly from the file can be much faster.

Executing a String with a "DO" Statement

A string containing a sequence of statements can be executed with a DO verb.

In terms of speed improvements and RAM usage, executing a string is exactly the same as executing a procedure. The only difference is that when the DO statement finishes, all the threads are freed. Therefore the next time you execute the same string, it will have to build the threads again.

Defining a procedure is much faster than initializing a string of the same length.

Appendix H

INSTALL Verb Development

Steps Required to Create an INSTALLed Library

The files in the "library starter kit" supplied with GRiDTask are used here as an example.

These are the steps to take when developing INSTALLed libraries. Pascal will almost always be used. FORTRAN could be called from Pascal.

1) Develop the custom routine(s) in the selected programming language. In this example, Sample.Pas is a Pascal source file containing several custom routines.

2) Compile the custom routine(s). In this example, Sample.Pas.

3) Link the custom routine(s) as shown in the develop file:

```
:Link:
LINK Sample.Pas~Obj~,        ``Libs`LibraryProcs~Obj~,
``Libs`CompactSystemCalls~Lib~                    TO
``Programs`Sample~Library~  NOPRINT BIND PC(PURGE) PURGE
FASTLOAD SS(STACK(0))
```

Steps Required to Use Installed Verbs in GRiDTask

1) Place an INSTALL statement in the GRiDTask program.

   e.g.   INSTALL "Sample~Library~"

2) The new verb(s) can now be used like any other GRiDTask verb (see the example):

```
.....
PRINT CONCAT$ (str1$,str2$)
.....
```

Files Created or Used When Installing Verbs
The following files were created or used when installing the custom routine Sample.Pas in GRiDTask:

Files in the Subject: Incs

TITLE	KIND
Library.Inc	Text

Files in the Subject: Library starter kit (on the starter kit diskette)

TITLE	KIND
Sample	Develop
Sample	Task
SAMPLE.PAS	OBJ
SAMPLE.PAS	LST
Sample.Pas	Text

Files in the Subject: Libs

TITLE	KIND
LibraryProcs	Obj
CompactSystemCalls	Lib

Files in the Subject: Programs

TITLE	KIND
SAMPLE	LIBRARY

In addition, the following GRiD applications are used:

GRiDTask, GRiDWRITE, and the GRiD development environment and Pascal version 3.0

Example Files
In this example, the following functions or procedures are contained in Sample.Pas:

Name	Type
FLASH	Procedure
CONCAT	String Function
MAX	Real Number Function

The next several pages contain copies of Sample.Pas~Text~, Sample~Develop~, and 'Incs'Library.Inc~Text~.

This is the file "Sample.Pas~Text~"

```
$NOLIST
$COMPACT (EXPORTS RegisterFunction)
MODULE SampleFunctionsPas;
$INCLUDE (``Incs`Common.Inc~Text~)
$INCLUDE (``Incs`ConPas.Inc~Text~)
$INCLUDE (``Incs`Math.Inc~Text~)
$INCLUDE (``Incs`StringTypes.Inc~Text~)
$INCLUDE (``Incs`StringProcs.Inc~Text~)
$INCLUDE (``Incs`WindowTypes.Inc~Text~)
$INCLUDE (``Incs`WindowProcs.Inc~Text~)
$INCLUDE (``Incs`MessageTypes.Inc~Text~)
$INCLUDE (``Incs`FieldTypes.Inc~Text~)
$INCLUDE (``Incs`OsPasTypes.Inc~Text~)
$INCLUDE (``Incs`OsPasProcs.Inc~Text~)

$INCLUDE (``Incs`Library.Inc~Text~)

PUBLIC SampleFunctionsPas;
  PROCEDURE RegisterLibraryFunctions;

PRIVATE SampleFunctionsPas;

{----------------------------------------------------------}
{                   SampleRoutines                         }
{----------------------------------------------------------}
PROCEDURE FlashRoutine;
VAR
  rect : rectangle;
BEGIN
  rect.topLeft.x := 0;
  rect.topLeft.y := 0;
  WinGetWindowExtent (rect.extent);
  WinInvertRectangle (rect);
  WinInvertRectangle (rect);
END;
{----------------------------------------------------------}
FUNCTION ConcatRoutine (str1, str2: StringPtr): StringPtr;
BEGIN
  ConCatRoutine := ConcatStrings (str1, str2);
END;
{----------------------------------------------------------}
FUNCTION MaxRoutine (num1, num2 :INTEGER): INTEGER;
BEGIN
  MaxRoutine := Max(num1,num2);
END;
{----------------------------------------------------------}
FUNCTION DivRoutine (num1, num2 :LongReal): LongReal;
BEGIN
  DivRoutine := num1 / num2;
END;
```

```
{-----------------------------------------------------------------}
{                   RegisterLibraryFunctions                      }
{-----------------------------------------------------------------}
PROCEDURE RegisterLibraryFunctions;
BEGIN
  RegisterFunction (NewStringLit ('FLASH '),
                    FlashRoutine,
                    statement,
                    NewString(0));

  RegisterFunction (NewStringLit ('CONCAT$ '),
                    ConcatRoutine,
                    stringFunction,
                    NewStringLit ('ss '));

  RegisterFunction (NewStringLit ('MAX '),
                    MaxRoutine,
                    IntegerFunction,
                    NewStringLit ('ii '));

  RegisterFunction (NewStringLit ('DIV '),
                    DivRoutine,
                    LongRealFunction,
                    NewStringLit ('rr '));
END;
.
```

This is the file "Sample~Develop~"

```
:Name:       Library Starter Kit
:Prefix:     'Library Starter Kit'

:Sources:
  Sample.Pas

:Link:
LINK Sample.Pas~Obj~,        ``Libs`LibraryProcs~Obj~,
``Libs`CompactSystemCalls~Lib~                  TO
``Programs`Sample~Library~  NOPRINT BIND PC(PURGE) PURGE FASTLOAD
SS(STACK(0))

:GRiDManager:
  'GRiDManager'

:Edit Sample~Task~:
GRiDWrite  Sample~Task~
```

This is the file "Incs'Library.Inc~Text~"

```
{----------------------------------------------------------}
{                       Library.Inc                        }
{----------------------------------------------------------}
PUBLIC Library;

TYPE
    SymbolType = (statement, integerFunction, stringFunction,
                  longRealFunction, booleanFunction);

PROCEDURE RegisterFunction (functionName       : StringPtr;
                            VAR functionAddress: BYTES;
                            functionType       : SymbolType;
                            parameterDescriptor: StringPtr);

{ RegisterFunction frees the two string parameters }
```

Notes on Custom Routines

The RegisterLibraryFunctions procedure (in Sample.Pas) uses the RegisterFunction procedure to:

1) supply the verb name to be used in GRiDTask.

2) supply the function or procedure address to GRiDTask.

3) supply the type of function or procedure. This is determined by the type of value returned. Possibilities are

```
* statement - no value is passed back
* string    - a string value is passed back
* Integer   - an integer value is passed back
* LongReal  - a real value is passed back
* Boolean   - a boolean value is passed back
```

4) supply the type and number of parameters passed from the installed verb statement in GRiDTask to the custom routine. The possibilities are:

Specification	Parameter Type
1) none	
2) "i"	integer
3) "s"	string
4) "r"	real

Any combination of "i", "s", "r" is allowed, up to eight maximum. Note that these parameters must match the TYPE of parameters in the custom routine.

For convenience, the example from INSTALL is printed here.

EXAMPLE

```
;-----------------------------------------------------
;  This task program illustrates how to install
;  the sample user-written library -
;  `Programs`Sample~Library~      - in GRiDTask.
;  The new functions are: CONCAT$, FLASH, MAX, and DIV
;-----------------------------------------------------
TASKWINDOW 0,0,-1,-1
INSTALL DEVICE$ + "Programs`Sample~Library~"
;--------------------------
str1$ = "One and "
str2$ = "two and ..."
PRINT CONCAT$ (str1$,str2$)
;--------------------------
FLASH: FLASH: FLASH
;--------------------------
PRINT "This is  MAX(4,5)"
PRINT STR$ (MAX(4,5))
;--------------------------
```

```
PRINT "This is  DIV (5,PI)"
PRINT STR$ (DIV (5,PI))
STACKMSG "Press any key to exit"
PAUSE ""
```

Appendix I

Data-Entry Forms

Introduction

This appendix gives:

o An overview of the steps required to use the data-entry forms package,

o Descriptions of the forms verbs,

o Descriptions of the forms specifications.

Overview

Data-entry forms are custom forms such as a consumer loan application. Data-entry forms are displayed on the computer screen, and the user fills them out by entering information on the keyboard. Each item of information is entered in a given field and can be checked as it is entered. In addition, calculations using this information can be performed.

The specifications for these forms are written in a GRiDWrite text file, and a GRiDTask program can be written to display and process the forms.

You might take the following steps to use the data-entry forms package.

1. Design the form(s) and enter the specifications in a GRiDWrite text file.

2. Write the GRiDTask program that uses the form(s). Install the "DataEntryForms~Library" using the INSTALL verb as follows:

   INSTALL "DataEntryForms~Library"

   "DataEntryForms~Library~" contains the library routines that support the forms verbs. See "INSTALL" in Chapter 4 of the GRiDTask manual for more information.

   Some forms verbs that might typically be used in your GRiDTask program are as follows:

- FORMINIT, which reads a forms specification from a string or FORMINITFROMFILE, which reads a forms specification from a text file.
- EDITFORM$, which displays the form on the screen.
- GETALLFIELDS$, which retrieves information entered by the user.
- PRINTFORM, which prints the information in the form.
- DISPOSEFORM, which removes the form from RAM memory.

You can also use any other GRiDTask verbs (described in Chapter 4 of the GRiDTask manual) as required in your program.

DISPOSEFORM

```
DISPOSEFORM formNum
```

NOTES

DISPOSEFORM frees the memory associated with the form defined by formNum.

To get formNum, use the FORMINIT verb.

EXAMPLES

```
MainForm = FORMINIT(READFILE$("Form4549"))
         o
         o
         o
DISPOSEFORM MainForm
```

EDITFORM$

```
lastKey$ = EDITFORM$ (formNum, topLeftX, topLeftY, widthX, heightY,
mode)
```

NOTES

EDITFORM$ displays the form identified by formNum (from FORMINIT) and waits for input from the user. After the user fills in the form and presses ESC, CODE-RETURN (for confirm), or any other CODE-key sequence, control is returned to the GRiDTask program. The string variable lastKey$ is set to the key pressed by the user. The form contains the latest data, regardless of the key pressed.

The font set when EDITFORM$ executes must be the same font set when FORMINIT initialized the form.

The parameters for EDITFORM$ are as follows:

formNum is a number returned by FORMINIT that identifies the form.

topLeftX and _topLeftY_ specify the pixel location of the top left corner of the form. _widthX_ and _heightY_ identify the size of the space in which the form appears.

Be sure to reserve one line at the bottom of the screen where messages can be displayed. The height of the line must be equal to the height of the current font. To determine this height, use the LINEHEIGHT verb.

mode specifies how the form is to be displayed. The choices are as follows:

1 The form area is erased and the form is drawn normally. This option is the conventional method for displaying forms.

2 The form area is not erased and only the contents of the fields are drawn. This option is useful for displaying field data if the form is already displayed on the screen. It prevents an annoying screen refresh.

3 The form is not displayed, but all of the field values are recalculated. This option is useful for updating field values in forms whose fields are linked to data in other forms. This mode does not modify the display in any way.

4 The form area is not erased before the form is drawn. This option can be used to display a form on top of another image, such as a logo.

EXAMPLE

```
currentForm     = FORMINIT (READFILE$ (filename$))
lastkey$        = EDITFORM$ (currentForm, 0, 0,_
                      WINDOWWIDTH-2,_
                     (WINDOWHEIGHT-(2*LINEHEIGHT)), 1)
```

The width and height items are set so that the form is displayed within the highlighted frame that surrounds the window.

FORMINIT

```
formNum = FORMINIT (formStr$)
```

NOTES

FORMINIT accepts a string representing a form definition, initializes the data structures associated with the form, and returns to formNum a number used to identify the form.

formStr$ represents the form to be initialized. The contents of this string are found on page I-17.

The font set when FORMINIT executes must also be the same when you specify the form in the EDITFORM$ and PRINTFORM verbs.

EXAMPLES

```
MainForm = FORMINIT(READFILE$("Form4549"))
```

FORMINITFROMFILE

```
formNum = FORMINITFROMFILE (pathName$)
```

NOTES

FORMINITFROMFILE is an integer function which returns a number used to identify the form when you edit or print the form. "pathName$" is the pathname of a file containing a form specification.

In GRiDTask, FORMINITFROMFILE gives the same result as:

```
formNum = FORMINIT (READFILE$ (pathName$))
```

GETALLFIELDS$

```
values$ = GETALLFIELDS$ (formNum, delimiter$)
```

NOTES

GETALLFIELDS$ is a function which returns a string containing all the field values in the form identified by formNum (obtained using the FORMINIT verb). Individual field values in the returned string are separated by delimiter$. GETALLFIELDS$ does not change the contents of the original form.

GETCURRENTFIELD

```
currentField = GETCURRENTFIELD (formNum)
```

NOTES

GETCURRENTFIELD returns an integer that identifies the field where the cursor is currently positioned - the "current field" - in the form identified by formNum.

The first field in the form is number 1, the second is 2, the third is 3, etc.

GETFIELDVALUE$

```
value$ = GETFIELDVALUE$ (formNum, fieldNum)
```

NOTES

GETFIELDVALUE$ is a string function which returns the contents of a specified field in the form identified by formNum (obtained using the FORMINIT verb).

INDEXFROMNAME

```
fieldIndex = INDEXFROMNAME (formNum, name$)
```

NOTES

INDEXFROMNAME returns a number representing a specific field in a form.

The items used with INDEXFROMNAME are as follows:

formNum is the identifier of the form. It is obtained using FORMINIT.

name$ is the name of the field. This is the name specified in the field definition section of the over-all form definition.

EXAMPLE

```
;  get identification number of form
formCaseInfo = FORMINIT (READFILE$ ("ddForm"))
;  get index of personID field
caseIndex = INDEXFROMNAME (formCaseInfo, "personID")
;  get personid info based on caseindex
personID$ = GETFIELDVALUE$ (formCaseInfo, caseIndex)
```

NAMEFROMINDEX$

```
name$ = NAMEFROMINDEX$ (formNum, currentField)
```

NOTES

NAMEFROMINDEX$ is a string function which returns the name of the field identified by _currentField_ from the form identified by _formNum_.

"formNum" is obtained using the FORMINIT verb.

PARSEFORM$

```
parsedSpec$ = PARSEFORM$ (fileSpec$)
```

NOTES

PARSEFORM$ reduces form initialization time for complex forms. PARSEFORM$ accepts a string containing a form specification and returns another string containing the same form specification, except that all equations and ranges are parsed. You can save this parsed form specification in a file. When initializing the form with FORMINIT or FORMINITFROMFILE, use the parsed form specification for much faster initialization.

For most forms you will not need to use PARSEFORM$. Only forms which have many equations or ranges and take a long time to initialize will benefit from pre-parsing.

PARSEFORM$ should only be used as part of the development cycle. The final GRiDTask application should not call PARSEFORM$.

EXAMPLE

```
;-----------------------------------------------------------
; This is a very simple Task program illustrating
;   how to use the new PARSEFORM$ function in the
;   dataEntryForms library.
;-----------------------------------------------------------
WindowMotion "off"
TaskWindow 0,0,-1,-1
codeEsc$ = CHR$(155)
BreakOnKey codeEsc$

INSTALL "DataEntryForms"

formFileName$ = GETFILE$ ("Select input file")
parsedFormFileName$ = GETFILE$ ("Select output file")

IF (parsedFormFileName$ <> formFileName$) THEN
  fileSpec$ = READFILE$ (formFileName$)
  PRINT "Parsing form"
  parsedSpec$ = PARSEFORM$ (fileSpec$)
  WRITEFILE parsedSpec$, parsedFormFileName$
ENDIF
```

This program reads a form specification file into a string that is parsed by PARSEFORM$. The parsed string is then written back to a different file.

PRINTFORM

error = PRINTFORM (formNum, printMode, destination$, topMargin, bottomMargin, leftMargin, printSize, formFeed)

NOTES

PRINTFORM sends a printer copy of a form to the printer or to a file you specify. If an error occurs during printing, the approriate GRiD-OS error code is returned. If a user halts printing by pressing ESC, no message is displayed and an error code of 0 is returned.

The font set when PRINTFORM executes must be the same font set when FORMINIT initialized the form.

The PRINTFORM parameters are as follows:

formNum (obtained using the FORMINIT verb) identifies the form to be printed.

printMode determines the contents and format of the printed form. Specify printMode as follows:

1 Prints the entire form
2 Prints the entire form with the field contents in bold typeface
3 Prints the field contents only

destination$ identifies where the printed copy is to be placed. Specify destination$ as follows:

'printer sends a copy of the form to the printer.

a pathname$ specifies a file to recieve the copy of the form.

topMargin is the number of lines from the top of the page where the first printed line is to appear.

bottomMargin is the line number of the last line to be printed on each page.

leftMargin is the left margin where the printed lines are to start.

printSize is the size of the typeface used by the printer. Specify printSize as follows:

1 for Normal typeface
2 for Condensed typeface
3 for Enlarged typeface

formFeed specifies when the pages or forms are ejected from the printer. Choices are as follows:

1 This specifies a form feed before printing.

2 This specifies a form feed after printing.

3 This specifies a form feed before printing and a form feed after printing.

4 This specifies no form feed.

SETALLFIELDS

```
SETALLFIELDS formNum, values$, delimiter$
```

NOTES

SETALLFIELDS sets a value in every field of the form identified by formNum (obtained using the FORMINIT verb). The value(s) to be set are placed in values$ and are separated by delimiter$. Two successive delimiters set the corresponding field to blank.

The value specified before the first delimiter in values$ is set into the first field, the value before the second delimiter into the second field, and so forth. If the number of fields in the form exceeds the delimited items in values$, the extra fields are left blank.

SETCURRENTFIELD

```
SETCURRENTFIELD formNum, currentField
```

NOTES

SETCURRENTFIELD moves the cursor to the field specified by currentField in the form identified by formNum.

SETFIELDVALUE

```
SETFIELDVALUE formNum, fieldIndex, newValue$
```

NOTES

SETFIELDVALUE changes the specified field to a new value.

formNum (obtained using the FORMINIT verb) identifies the form containing the field.

fieldIndex identifies the field

newValue$ contains the value to be placed in the field.

Form Specifications

You can define forms for data entry using GRiD forms specifications. Using the verbs described in this appendix, these forms can be displayed, filled in with data, processed by GRiDTask, and stored in files. The user moves through fields using the function keys described in Table 1-2 at the end of this appendix.

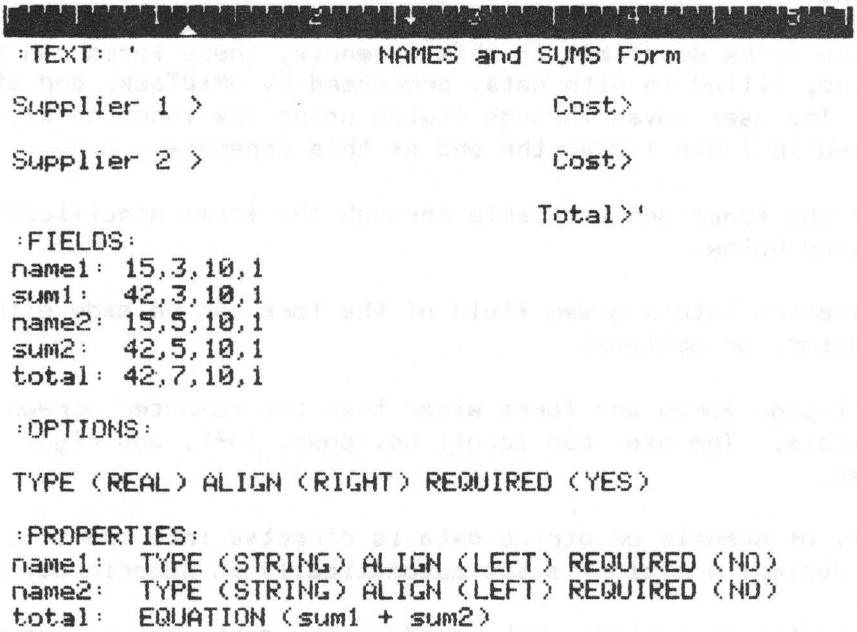
Some of the functions available through the forms specification are summarized below.

- Data-entry into a given field of the form can be made either mandatory or optional.
- Multi-page forms and forms wider than the computer screen are possible. The user can scroll up, down, left, and right in the forms.
- Entry of numeric or string data is directed into specific fields. The decimal position is set automatically in numeric data.
- Arithmetic operations such as addition, subtraction, and others can be performed automatically on data as it is entered.
- Data-entry fields can be underlined, displayed in boxes, or with the characters displayed in inverse video.
- Help text can be displayed for individual fields in the form.
- Data can be aligned automatically as it is entered.

The GRiDTask program can prompt the user while the form is filled out. After the user confirms, control returns to GRiDTask.

Specification Overview Figure 7-1 shows a simple form prepared with a form specification..

Figure 7-1. Sample Form



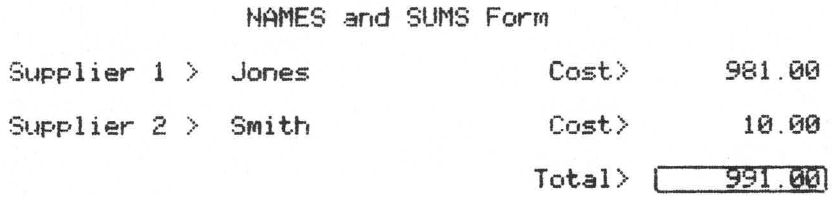
```
:TEXT: '                  NAMES and SUMS Form
Supplier 1 >                       Cost>

Supplier 2 >                       Cost>

                                  Total>'
:FIELDS:
name1: 15,3,10,1
sum1:  42,3,10,1
name2: 15,5,10,1
sum2:  42,5,10,1
total: 42,7,10,1

:OPTIONS:

TYPE (REAL) ALIGN (RIGHT) REQUIRED (YES)

:PROPERTIES:
name1:  TYPE (STRING) ALIGN (LEFT) REQUIRED (NO)
name2:  TYPE (STRING) ALIGN (LEFT) REQUIRED (NO)
total:  EQUATION (sum1 + sum2)
```

To prepare a form, a text file is created using GRiDWrite. The four basic elements that comprise the form specification are as follows.

1) The screen layout of the form
2) Definition of the fields
3) Options for each field
4) Properties for each field

Figure 7-2 shows the form specification that created the sample form shown in Figure 7-1.

Figure 7-2. Sample Form Specification



```
                NAMES and SUMS Form

Supplier 1 >  Jones              Cost>       981.00

Supplier 2 >  Smith              Cost>        10.00

                                Total> [    991.00]
```

The four elements in the form specification must appear in the order as described below.

1 The screen layout is the text that is to appear on the screen. The screen layout section is identified by the token _:TEXT:_.

2 Field definitions are numeric coordinates that define the symbolic name, the size, and the location of each field in the form. The field definition section is identified by the token _:FIELDS:_.

3 Field options are instructions specifying how to treat the data entered into each field in the form. For example, field options specify whether a field is to contain string or numeric data, where the decimal point is placed, and if the entry of data is required or optional. The field options section is identified by the token _:OPTIONS:_.

4 Field properties determine how to handle the data entered into a specific field. A field property overrides a conflicting field option. The field properties section is identified by the token _:PROPERTIES:_.

The following sections describe these elements in more detail.

Screen Layout This defines the text as it appears to the user. Enter the token :TEXT: followed by a blank and a single quotation mark. Then enter the text for the entire screen, followed by a single quote mark.

Field Definitions Field definitions begin with the token :FIELDS: followed by definitions in the following format:

fieldname: leftDistance, topDistance, length, numberLines

fieldname is a symbolic name made up of one or more valid ASCII characters. Use the field name when assigning a property to a field in a field definition. Field names can also be used to assign values to the parameters used with the DEFAULT, EQUATION, and RANGE keywords. Each field name you specify must be unique. If a fieldname contains a non-alphabetic character (for example a blank), it must be enclosed within single quotes ('). e.g. 'total sum': 42,7,10,1

Together, _leftDistance_ and _topDistance_ define the starting position of the field.

leftDistance is the number of character positions, starting from the left-hand side of the form, where the field is to begin.

topDistance is the number of character positions, starting from the top of the form, where the field is to begin.

length specifies the length of the field in characters.

numberLines defines the number of lines in the field. When the user fills all character positions of one line of the field, the cursor automatically goes to the next line, if defined.

Note that since the user can scroll a form up, down, left or right, the size of the form isn't limited by the size of the screen. However, it is not recommended to have an individual field wider or higher than the screen on which it is displayed.

When entering a field specification with GRiDWrite, it may be helpful to press CODE-O and set the "ruler" to Yes. The ruler provides an aid in determining the starting and ending character positions of each field.

Field Options and Properties Options and properties - summarized in Table 1-1 on the next page - determine the format, appearance, and other characteristics of the various fields in the form. A field option consists of a keyword followed by one or more parameters. A field property consists of a field name followed by a colon and a space, a keyword, and one or more parameters.

The following rules apply to field options and field properties:

- The same keywords can be used both as a field option and a field property. A field option applies to every field definition in the form. A field property applies only to the definition specified by the field name. A field property overrides a conflicting field option.

- The field options must be preceded by the text :OPTION: .

- The first field property must be preceded by the text :PROPERTY: .

- You can abbreviate keywords and parameters, as indicated in each description. For example, you can specify either AL or ALIGN as a keyword, and either RI or RIGHT as its parameter. The abbreviated keyword must contain at least the first two letters of the keyword.

If a field option or property has more than one keyword, insert one of the following between each keyword and the preceding keyword parameter:

- One or more blank characters

- A comma

- One or more carriage-returns (press RETURN)

For example, the following three definitions for ssnum are all valid:

```
ssnum:  AL(RI) CO (UPPER)    EX(NO)
ssnum:  AL(RI), CO (UPPER), EX(NO)

ssnum:  AL (RI)
        CO (UPPER)
        EX (NO)
```

Summary of Option and Property Keywords

Table 1-1 briefly describes the option and property keywords, and their respective parameters. The sections following the table, given in alphabetical order, give a detailed description of each of these items

Table 1-1. Summary of Options and Properties

Option/Property	**Default**	**Function**
AL (LE ¦ RI ¦ CE)	Left.	Align. Places the characters in the form field to the right or left, or centers them.
CO (UP ¦ NO)	None.	Convert. Changes alphabetic characters to uppercase when they are retrieved from the form.
ED (YES ¦ NO)	Yes.	Editable. When set to NO, the user cannot enter or change data in the field to which the keyword applies.
EQ (expression)	None.	Equation. Sets the value of the field to expression, which can consist of any of the following: strings, numbers, field names, arithmetic operators, and logical operators.
EX (NO ¦ YES)	Yes.	Expandable. When set to Yes, the number of characters typed in by the user can exceed the length of the field: otherwise, the number of characters cannot be greater than the length in the field definition.
FO (n)	2	Format. The number of digits to the right of the decimal point to display.
HE (string)	None.	Help. Specifies help text retrievable by the user after pressing CODE-?.
HI (UN ¦ OU ¦ IN ¦ NO)	No.	Highlight. Specifies if the field is to be underlined (UN), enclosed in a box (OU for outlined), inverted (IN) or not highlighted (NO).

MA (mask) (message)	None.	Mask. Forces the user to enter characters in a specified format. Incorrectly entered characters cause a message to appear.
PR (message)	None.	Prompt. Causes a prompt to appear in the message line at the bottom of the screen.
RA (expression) (message)	None.	Range. Defines a maximum and minimum value that the user can enter in a field. An incorrectly entered range causes a help message to appear.
RE (YES ¦ NO ¦ LE)	No.	Required. When set to Yes, the user must enter at least one character in the field; when set to LE (for length), the user must enter a character in every position of the field.
TY (ST ¦ NU)	String.	When set to NU (for numeric), the user must enter numeric characters in the field. If you omit TY or specify ST, the user can enter any printable ASCII character (A-Z, a-z, 0-9, and punctuation and other special characters)

AL -- ALIGN

Default: AL (LEFT)

The ALIGNMENT keyword has the following format:

AL (LEFT ¦ RIGHT ¦ CENTER)

Enter LE for left, RI for right, or CE for centered to specify data alignment in the field as it is entered.

CO -- CONVERT

Default: CO (NONE)

The CONVERT keyword has the following format:

CO (UPPER ¦ NONE)

When you specify UPPER, all lower-case alphabetic characters are changed to uppercase when converted to strings using GETFIELDVALUE$

and GETALLFIELDS$.

ED -- EDITABLE

Default: ED (YES)

The EDITABLE keyword has the following format:

ED (YES | NO)

If you specify NO, the user cannot enter or change data in the field to which the keyword applies. Specify NO if you want to display only data that should not be modified.

Note that the user cannot modify the data in fields for which you specify an equation (described in the next section).

EQ -- EQUATION

The EQUATION keyword has the following format:

EQ (expression)

EQ causes the field to be assigned the value of the expression. expression can consist of the following:

- Any numeric or string value
- Any fieldname or formula that produces a numeric value
- Arithmetic and logical operators
- IF/THEN/ELSE conditional expressions
- Built-in functions

All of the following are valid numeric expressions:

```
25.27
10 + 15.2
totalfld + taxfld
ABS (totalfld + taxfld)
```

A field name can be used in a conditional expression. For example, the following expression is valid:

```
IF fieldA > 4 THEN 0 ELSE fieldB
```

Note that the user cannot modify the data in a field for which you specify an equation.

Arithmetic Operators You can specify arithmetic operators within an expression. Refer to Section 3.3, "Real Variable Operators" (Chapter 3) for a list.

Conditional Expression--IF/THEN/ELSE You can use the IF/THEN/ELSE expression in a field definition. This conditional expression places one of two different values in a field, depending on whether the specified condition is met.

The conditional expression takes the format shown here.

If (condition) THEN (expression) ELSE (expression)

For condition, the conditional operators shown in Section 3.3, "String Operators" (Chapter 3) can be used.

Logical Operators The logical operators shown in Section 3.3, "Real Variable Operators" (Chapter 3) can be used.

Built-In Functions You can specify built-in functions within an expression. These include DATE$, TIME$, and the Real Number Functions described in Chapter 4, Section Two.

EX -- EXPANDABLE

Default: EX (NO)

The EXPANDABLE keyword has the following format:

EX (NO ! YES)

If you specify NO, the number of characters typed in by the user cannot be greater than the number of characters specified for the length of the field (length) in the field definition.

If you specify YES, the number of characters typed in by the user can exceed the number of characters specified for the length (length) in the field definition. These characters are available for processing although they won't be visible on the screen.

FO -- FORMAT

Default: FO (2)

The FORMAT keyword has the following format:

FO (numberDigits)

Format lets you specify where the decimal point appears in a number. numberDigits is the number of digits to be displayed to the right of the decimal point. The format does not alter the value of the number. It changes only the number of decimal places that are displayed.

HELP

The HELP keyword has the following format:

HE (string)

The parameter string is text that you can insert to provide information or instructions to the user about a particular field. When the current cursor position is in fieldname and the user presses CODE-?, the current screen is erased and the help text is displayed.

The form reappears when the user confirms or presses ESC.

You can format the text in string using carriage-return/line-feed characters. For example:

```
name:  HELP (Enter one of the following:

>  My name
>  Your name
>  Her name)
```

HI -- HIGHLIGHT

Default: HI (NO)

The HIGHLIGHT keyword lets you change the appearance of fields. It has the following format:

HI (UNDERLINE | OUTLINE | INVERT | NONE)

The Highlight parameters are used as follows:

UN Underlined. The fields are underlined. If the field has more than one line, only the last line is underlined.

OU Outlined. A highlighted outline surrounds the entire field.

IN Inverted. The characters in the field are displayed in inverse video - they appear black against an amber background.

NO None. The characters in the field are displayed in their normal appearance - amber against a black background.

MA -- MASK

The MASK keyword lets you force the entry of specific characters position-by-position in the specified field. Mask has the following format:

```
MA (mask) (message)
```

mask is a character string that defines the characters to be entered and can consist of the following:

Mask Character	**Required Character in Corresponding Field Position**
A	Alphabetic character.
D	Numeric integer.
!	Character immediately following the explanation mark (!)
B	Blank
?	Any character

message (which is optional) is a character string to provide help information to the user in filling out the specified field correctly. message appears at the bottom of the screen when the characters entered by the user don't match those specified in the mask. The message appears when the user either confirms or tries to move to another field.

Note that the message string must not exceed the width of the computer screen. Characters that exceed the width are cut off.

For example, the following mask forces the entry of three pairs of numerics separated by hyphens:

```
MA (DD!-DD!-DD) (Enter date in format mm-dd-yy)
```

PR -- PROMPT

The PROMPT keyword has the following format:

PR (message)

Prompt causes the character string specified for message to be shown at the bottom of the display when the cursor enters the field.

Note that the character string must not exceed the width of the computer screen. Characters that exceed the width are cut off.

RA -- RANGE

Default: None

The RANGE keyword defines minimum and maximum values that the user can enter in a field. Ranges can be set for both string and numeric fields and have the following format:

RA (expression) (message)

You can enter the same operators and values for expression as you can in the Equation keyword. See the description of Equation in this section for details. expression is evaluated as a BOOLEAN. If you have an expression such as (fldA + fldB), an even number is interpreted as false and any odd value is interpreted as true.

message (which is optional) is a character string that is displayed at the bottom of the screen if the user attempts to enter a value outside the specified range.

Note that the character string must not exceed the width of the computer screen. Characters that exceed the width are cut off.

```
fldC: RA (fldC > fldA+fldB)
```

the value entered in the field specified by fldC must be greater than the sum of fldA and fldB.

Logical operators such as OR make the following types of expressions possible:

```
fldC: RA ((fldC>fldA+fldB) OR (fldC=2000))
```

RE -- REQUIRED

Default: RE (NO)

The REQUIRED keyword specifies whether a user must enter data in a field. It is used in the following format:

RE (YES ! NO ! LENGTH)

If you omit the keyword altogether, or specify NO, the user can either enter data or skip to the next field, leaving the field blank.

If you specify YES, the user is required to enter at least one character of data before confirming.

If you specify LE (for length), the user must enter a character in every position of the field. Thus, the number of characters entered must be the same as the length specified in the length parameter of the field definition.

All required fields must be filled in before the user is allowed to exit the form. If you specify LE or YES, and the user confirms the form without filling in the required field, the cursor moves to the field and the user is prompted to fill it in.

TY -- TYPE

Default: TY (ST)

The TYPE keyword specifies if the data entered by the user is treated as a numeric or as a character string. It has the following format:

TY (STRING ! NUMERIC)

If you specify NU, the user must enter numeric characters in the field. If you omit TY or specify ST, the user can enter any acceptable ASCII character.

A message is displayed if the user tries to enter data other than as specified.

Key Operation

To summarize, the form specification defines a series of fields seen by the user on the display. The user fills in the fields according to the rules you define in the specifications. Table 1-2 on the next page shows the keys and key combinations available to the user to move about the form and perform other functions.

Table 1-2 Key Operation

Key	**Result When Pressed**
CODE-RETURN	Confirms that the contents of the form are correct and returns control to GRiDTask. All fields must be filled in according to the field option and field property items in the form specifications. Otherwise, after pressing CODE-RETURN, a message is issued and the cursor moves to an incorrect field. Note that pressing ESC or any other CODE-key returns from EDITFORM$( ) regardless of whether the field values agree with the corresponding field option and field property items.
TAB	Moves the cursor forward from field to field in the order that each field definition is specified following FIELDS in the form specifications.
SHIFT-TAB	Moves the cursor backward from field to field in the order the fields are defined in the form specifications.
SHIFT-RightArrow	Moves the cursor to the right from field to field in the order the fields appear on the screen. Does not move the cursor past the right-most field. Pressing the RightArrow key alone performs the same function when the field is empty, or when the cursor is positioned after the last character entered.
SHIFT-LeftArrow	Moves the cursor to the left from field to field in the order the fields appear on the screen. Does not move the cursor past the left-most field. Pressing the LeftArrow key alone performs the same function when the field is empty, or when the cursor is positioned before the first character entered.
SHIFT-UpArrow	Moves the cursor to the field above the current field. Pressing the UpArrow key alone

performs the same function in a single-line field, and in multi-line fields when the cursor is on the top line.

SHIFT-DownArrow
Moves the cursor to the field below the current field. Pressing the DownArrow key alone performs the same function in a single-line field, and in multi-line fields when the cursor is on the last line.

RETURN
In a multi-line field, moves the cursor to the next line. If the cursor is in the last line of a field, moves the cursor to the next field.

CODE-SHIFT-UpArrow
Moves the cursor to the first field in the form.

CODE-SHIFT-DownArrow
Moves the cursor to the last field in the form.

CODE-DownArrow
In a multi-page form, moves the cursor to the first field on the next page.

CODE-UpArrow
In a multi-page form, moves the cursor to the first field in the previous page.