Cardiff TELE*form*®

BasicScript™ Guide

Notice

Trademarks and Copyrights

Patent Information

Covered by U.S. Patents 4,893,333; 5,247,591; 5,555,101 and 5,943,137. Additional patents pending.

Cardiff Software Incorporated, Vista CA 92083

www.Cardiff.com

Document Number and Revision

Document Number 100-00012 Revision A

Effective Date: September 01, 2000

CHAPTER 1

Introducing BasicScriptTM for TELEform

CHAPTER 2

Technical Resources

CHAPTER 3

Form and Global Form Scripts

CHAPTER 4

Export Scripts

CHAPTER 5

System Script

CHAPTER 6

Custom, Periodic and Library Scripts

CHAPTER 7

Advanced Features of Scripts

Executing and Debugging Your Scripts

CHAPTER 10

Creating Custom Dialog Boxes

CHAPTER 11

Common Language Elements

CHAPTER 12

BasicScript Language Reference

CHAPTER 1

Introducing BasicScript™ for TELE*form*

About BasicScript

BasicScript™ is an integrated scripting language that lets you add extensive features such as arithmetic comparisons, financial calculations, cross-field validations, checksums, calls to external functions, and skip-and-fill logic to form processing in TELE*form*. The BasicScript language is similar in syntax to Visual Basic and requires similar programming experience to use. This version of BasicScript has been specially adapted for use with TELE*form*.

About this Chapter

This chapter provides an overview of BasicScript as it relates to TELE*form*. It explains what BasicScript is, and how BasicScript can customize and enhance your TELE*form* processing.

This chapter also gives you a quick tour of the scripting process, which includes writing your script, executing your script, and debugging your script. If you are new to BasicScript, you should read this chapter to get an idea of what steps to follow when writing your scripts.

Why use BasicScript?

You can think of BasicScript as a tool that runs behind the scenes operations during any and every part of form processing. BasicScript allows you to create customized and flexible add-ons for TELE*form*. The TELE*form* version of BasicScript has been customized to work specifically with form processing.

For example, suppose you have an order form. If you want TELE*form* to add a series of Price field values together while it is evaluating returned forms, you can do this by entering code in the appropriate entry point of your form script.

Then, you can enter code in another entry point to fix the Total field when it is incorrectly filled in. When the *Verifier* operator is in form mode correction, you can display a message box informing the operator that the value in the Total field is incorrect. The *Verifier* operator must enter the correct value in the Total field before moving to the next field.

If you want to write a script that does this type of calculation, refer to the "Sample Form Validation Script" on page 52.

BasicScript Capabilities

BasicScript provides you with increased capabilities regarding how your forms are interpreted, validated, accessed, and exported, and gives you extended control over your returned data. For example, BasicScript brings the following capabilities to TELE*form*:

During Form Evaluation

- Examine, modify and enhance form data.
- Determine if a form should be held for review.
- Change the validity status of fields and forms.

During Correction and Data Entry

- Force retries when incorrect data is entered into a field.
- Dynamically choose the next field based on entered data (skip-and-fill logic).
- Run validations before sending the form to form mode correction.
- Allow certain information to be seen only by specific *Verifier* operators.
- Route forms to specific *Verifier* operators.
- Format data before exporting it to a data file.

During Export

- Ensure that the form data is valid **before** exporting it.
- Take complete control of how your data is exported.
- React to form processing with custom operations.

General Productivity

- Create custom menus in TELE*form* for instant script access.
- Create custom dialog boxes that allow user interface with your script.
- Write scripts to automatically perform routines at regular intervals in TELE*form Reader*.

How BasicScript Works with TELE*form*

BasicScript adds a lot of functionality to TELE*form*, but does not require complex programming knowledge. If you have written programs with Visual Basic, you can probably teach yourself BasicScript. If you have never written programming code before, learning BasicScript may require extra training.

Script Types

There are seven types of scripts that you can create in BasicScript:

Form Script - There is one Form script for every TELE*form* form. The Form script is used to control form merges, validate returned data, control correction in *Verifier*, and modify data exports. The majority of BasicScript functions that you write will be accommodated by Form scripts. See “Form and Global Form Scripts” on page 21 for instructions.

Global Form Script - The Global Form Script has the same entry points that a regular Form script has; however, the entry points in a Global Form Script apply to every form template.

Export Scripts - You can write as many Export scripts as you want. Each Export script created a custom export format. See “Export Scripts” on page 57 for more information on writing and executing Export scripts.

System Script - For each TELE*form* installation, there is only one System script. The System script allows you to control batch processing, along with the starting and closing of *Designer*, Print Manager, *Reader and Verifier.* See “System Script” on page 71 for more information on writing and executing your System script.

Custom (Menu) Scripts - You can write as many Custom scripts as you want. Each Custom script creates a command in the **Script** menu of *Designer*, Print Manager, *Reader* and *Verifier*. When a user selects this command, the custom script is executed. See “Custom, Periodic and Library Scripts” on page 87 for more information.

Periodic Script - Each TELE*form* installation is allowed to have one Periodic script. The Periodic script is executed at regular intervals in *Reader*. See“Custom, Periodic and Library Scripts” on page 87 for more information.

Library Scripts - You can write as many Library scripts as you want. A Library script is not directly executed. Instead, it is used to store functions that are referenced in one of the five script types mentioned above. For example, you can consistently use a function in multiple Form scripts by putting this function into a Library script, and then calling this function in each Form script. See Chapter 5 for more information on writing and executing Library scripts.

NOTE: You are only allowed to have one System script and one Periodic Script per TELE*form* installation.

Script Entry Points (subroutines)

Each one of these scripts (except the Library script) has its own group of entry points. These entry points can be thought of as TELE*form*-specific subroutines that allow you to control each major TELE*form* operation (for example, setting up a batch in TELE*form Reader*, exporting data to a data file, and so on). You can customize any TELE*form* operation by inserting code into the appropriate entry point of a script.

As their name implies, entry points indicate those places where TELE*form* "enters" and executes that part of the script. The entry point where you type your code dictates when TELE*form* executes that code. In other words, each entry point represents a unique point in the form processing cycle.

When you understand how the TELE*form* entry points relate to the flow of data in TELE*form*, you can put your code into an entry point that TELE*form* will call at the appropriate time.

For example, when you evaluate a form image in TELE*form Reader*, the following sequence of events occurs.

1. TELE*form* begins evaluating a form.
2. TELE*form* calls the Form_Evaluate entry point.
3. If this entry point contains script, the script will be executed. Otherwise, the entry point is ignored.
4. TELE*form* continues processing the form.

You can think of the entry point as a pre-defined subroutine. Regardless of whether or not script is inserted into the entry point, TELE*form* will call the entry point every time the corresponding TELE*form* process occurs.

Overview of the BasicScript language

The BasicScript language behaves similarly to Visual Basic because both of these languages deal with objects. The TELE*form* version of BasicScript uses a special set of objects known as the TELE*form* object model. The scripts that you write communicate with TELE*form* through this object model.

When you understand the organization of TELE*form* information into unique classes and properties, you can utilize the full range of TELE*form* information.

The TELE*form* object model contains the following organization:

- The object model contains several classes of objects. Each class contains a particular set of TELE*form* information.
- Each TELE*form* class contains a unique set of properties. Each property refers to a subset of TELE*form* information. Properties allow you to reference the complete set of TELE*form* information; they are the building blocks of your script.
- Each property is classified into a data type. A data type tells you what type of value to expect from a property.

Classes

The TELE*form* object model consists of the eight classes of objects listed below. Each class allows access to a different set of TELE*form* information:

Object Class	Description
Form	Provides access to general information about the form being processed, such as the form's title, form ID, and so on.
Export**	Provides access to information about the current export session.
Fields	Provides access to all fields on the form by treating each field as an element in an array.
Field	Provides access to all data (and other field attributes) in any field on the form. This is the most frequently used type of object.
Choices*	Provides access to the choices in a choice field. Each choice is treated as an element in an array.
TopChoices*	Provides access to the 3 best-guess characters for each interpreted character (based on the field recognition). The choices are stored in a 3-element array.
Row*	Provides access fields that are part of a detail group. Each row in the detail group is treated as an element in an array.
Batch***	Provides access to batch information.

* Form scripts only

** Export scripts only

*** System Script Only

NOTE: All classes in the TELE*form* object model (except the Field class) are hidden, meaning that you cannot dimension variables of these types. Hidden classes may be used with their appropriate properties by referencing the class name directly.

Class Properties

Each object class has a unique set of properties. These properties represent the specific types of information available in the object. For example, all field objects have the property "Name", which is used to reference the field ID.

Refer to Chapters 2-4 (the Form, Export, and System Script chapters) for the following information:

- List of each classes' properties (all properties except the Batch and Export class properties are described in Chapter 2.)
- How properties are used in the object model to reference TELE*form* information
- Miscellaneous information regarding properties

One property often references another property in a particular object class. For example, the HasMask property in the Field class checks the status of the Mask property in the same field.

Data Types

Each property is classified into one of the following types:

Type	Description
String	Character string
Long	32 bit integer
Integer	16 bit integer
Float	Single precision floating point number
Double	Double precision floating point number
Variant	Universal (Generic) data type (see chapter 8)

Case Sensitivity

All object and property references are case-insensitive, so you do not need to worry about case when typing them.

City.Text = CITY.TEXT = city.text = CiTy.tExT

However, case is very important when validating alphabetic and alpha-numeric field values. Because a TELE*form* field can be configured to interpret and store both upper case and lower case characters (or both), it is important that any script which tests that field's value correspond to those settings.

For example, if the City field contains the value "San Francisco," and the script tests for the condition:

IF City.Text = "SAN FRANCISCO"

then the validation will fail, because "San Francisco" does not match "SAN FRANCISCO."

NOTE: The best way to avoid this type of problem is to select the **Convert to uppercase** check box in the **Recognition Setup Options** dialog box of all Constrained Print Fields and Image Zones used for scripting. This option stores the field value in uppercase characters. If you select this check box, make sure that you also use upper case characters in the script.

You can use Ucase$ and Lcase$ in your scripts when performing case-sensitive data comparisons. For example:

If UCase$(City.Text) = "SAN FRANCISCO"

BasicScript Tour

Now that you know a little about the way BasicScript works with TELE*form*, you are ready to take a quick tour of the BasicScript writing process. This tour will help you understand what is required to successfully incorporate BasicScript into TELE*form* processing.

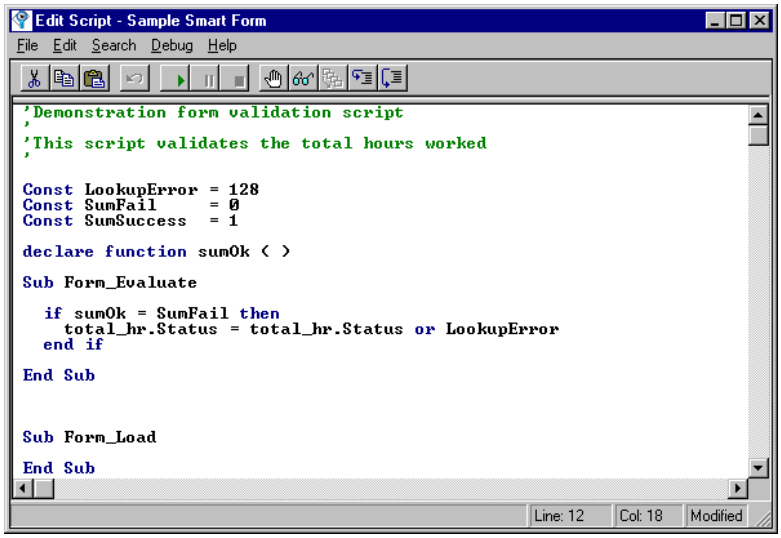
Step 1: Determine the Script Type You Want to Use

Before you can write your script, you must know which type of script you will be using. To do this, you have to pinpoint the exact part of the TELE*form* process that your script fits into (see the entry point diagrams in Chapters 2-4). For this tour, we will select the Form script, which is the script type most commonly used.

Step 2: Open Your Script in TELE*form Designer*

When you open your script, it appears in the Edit Script window. The Edit Script window is the application where you will write, edit and compile your script. For this tour, you will open the Sample Smart Form's Form script.

1. Open the Sample Smart form in TELE*form Designer*.
2. Click **Script** on the **Form** menu. The Edit Script window will appear.



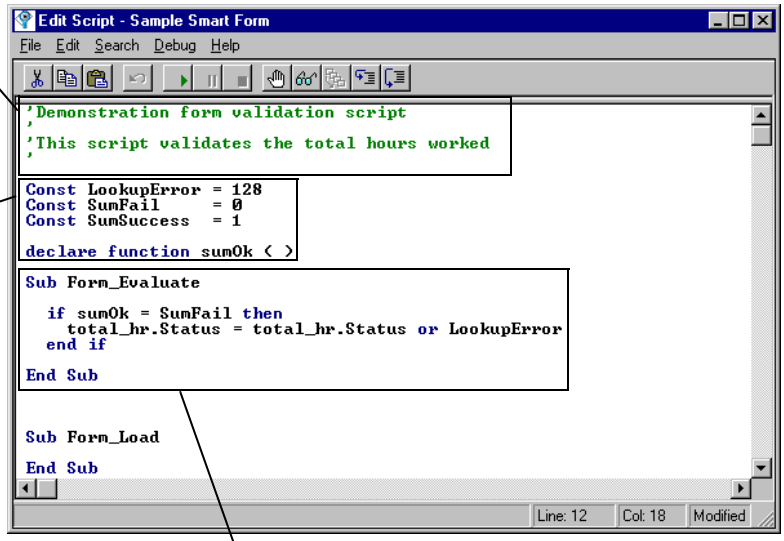
Step 3: Write Your Script

The Edit Script window includes most of the basic functions that you will find in any text editor, plus the BasicScript-specific functions that are used to create the statements in your script.

Refer to each of the callouts in the figure below to orient yourself with this script.

Notice that the script writer put comments into his/her script to inform the script viewer of the function of the script.

Notice that the script contains defined constants and a declared function. Because the constants and function are defined/declared before the entry points, they apply to all entry points in this script.



This entry point is a subroutine that will be called each time a form image is evaluated in TELE*form Reader.*

To create a script from scratch, you should adhere to the following conventions:

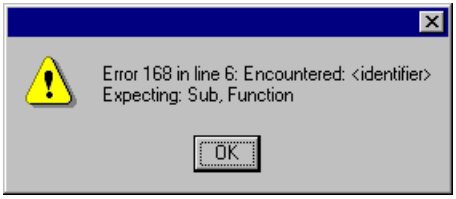
- Write comments at the top of your script to tell others what it does (and to remind yourself of what it does).
- If you want to define constants, declare public variables, or declare functions, do so at the very beginning of your script (before the entry points).
- Use blank lines in your script to group sections that logically belong together. For example, this script writer put variables in one group, and put the function in another group.
- Write at least one comment for each group in your script. This will help others see the reasoning behind your code, and it will help you make sure that this reasoning is sound.
- Use the right-click feature to reference fields on your form.

Step 4: Compile Your Script

Once you have written your script, you will need to compile it to check the syntax of your BasicScript statements. If the syntax is OK, you can proceed to step 5. If there is a problem with your syntax, you will receive an error message. This error message will point you to the line with the problem and tell you what the problem is.

IMPORTANT! Even if your script compiles OK, there is no guarantee that it will execute successfully. The compiler cannot check for every possible error in your script.

1. On the **File** menu of the Edit Script window, click **Compile**. Because this script was written for the Sample Smart form, it compiles OK.
2. If the script contained errors, you would see a message that looks something like the following:



3. Notice that this error message displays the line that contains the error, highlights this line in your script, and gives the reason for the error. When you have fixed all of the errors, recompile your script.
4. On the **File** menu, click **Save**, and then exit the Edit Script window.
5. Normally, you would also save the form itself, but you do not need to do so for this example.

Step 5: Execute Your Script

When your script has compiled OK, execute it to make sure that it performs as expected. This involves creating a condition that will force this script to do extra processing. In this example, all you need to do is evaluate the Smart sample image and correct it in TELE*form Verifier* (since this image has an incorrect value in the Total Time field.)

1. Start TELE*form Reader.*
2. On the **File** menu, click **Evaluate Image**. The **Open** dialog box appears.
3. Click **Smart.tif**, and then click **Open**.
4. TELE*form Reader* evaluates the Smart sample image. When it does this, the **Form_Evaluate** entry point is called and executed.
5. Start TELE*form Verifier.*
6. In the **Forms** list, select **Sample Smart Form**.
7. In the **Stored Images** list, select **Smart.tif** , and then click the **Correct** button.
8. Correct any suspect characters and press any key when you get to the finish flag.
9. When Form Mode Correction begins, press TAB until you get to the total_hr field. This is the field that your Form_Evaluate subroutine validated.
10. Press TAB to go to the next field. When you attempt to tab out of this field, the following message appears.

11. Click **OK** to return to the total_hr field.
12. Enter the correct time in the total_hr field (08:30), and then press TAB. TELE*form Verifier* will now accept this value and allow you to move to the next field.

If you had seen any errors or incorrect results during the execution of this script, you would want to debug it. See “Using the BasicScript Debugger” on page 141 for debugging details.

IMPORTANT! Always compile your script after any changes to the Form.

•

CHAPTER 2

Technical Resources

Need Answers?

You have many options for getting information about your TELE*form* system:

- Documentation
- Online Help
- Cardiff Web site
- Cardiff Technical Support

Documentation

Along with your program software, you receive this User Guide as an Adobe PDF file on a CD. Adobe Acrobat Reader 4.0 is available at http://www.adobe.com/acrobat/.

Printed Documents

To order printed TELE*form* documentation, go to http://www.Cardiff.com and click **Order Product Manuals**.

Downloading Revised Documents

Cardiff updates our user guides on a continuous basis. To download the latest revision of a user guide, go to http://www.Cardiff.com/Manuals.

Online Help

You can access the TELE*form* Help system from any TELE*form* module by pressing the F1 key or clicking an option from the **Help** menu. The Help system includes a Table of Contents, an Index, and a Search/Find feature.

Cardiff Web Site

The Cardiff Web site, www.Cardiff.com, provides a wealth of information about TELE*form*. The site includes:

- News updates;
- A list of Frequently Asked Questions (FAQ) that you can search for solutions to common problems;
- A Scanner Wizard to help you find scanners that are certified for use with or compatible with TELE*form*;
- A list of fax servers that are certified for use with TELE*form*;
- A library of Cardiff documents in Adobe Acrobat PDF format;
- Free downloads of Connect Agents, Service Packs, patches, and other useful software;
- A glossary of terms you may encounter when working with your system;
- "Walk-through" slide shows that demonstrate the exact steps needed to perform common tasks like exporting a form, copying your form to Microsoft Word, importing a form, setting up your primary auto export, using the Purge Log, NonForm data entry, and using the Form Merge feature;
- White Papers explaining the use of TELE*form* in real-world scenarios;
- An Operating System (OS) Wizard to help you determine if your computer OS is compatible with your version of TELE*form* and any other Cardiff products;
- Automatic e-mail contact to Cardiff's Technical Support and Sales departments.

Accessing www.Cardiff.com from TELE*form*

The **Help** menu and the **www.Cardiff.com** toolbar button provide direct links to Cardiff's Web site.

Technical Support

If you have a question about TELE*form*, you should first look in the TELE*form* user documentation, the online Help, or check the Cardiff Web site for answers. Frequently Asked Questions (FAQ) are available on the Web site's Technical Support page. If you still can't find answers to your questions, contact Cardiff's Technical Support team.

Cardiff is dedicated to providing the highest quality technical support to registered TELE*form* customers. You will receive the benefits of our Annual Support and Maintenance Plan for a full 60 days at no charge following the shipment of your TELE*form* product. After this initial 60 day period, you have the option of signing up for an Annual Support and Maintenance Plan.

The Annual Support and Maintenance Plan

The Annual Support Plan offers the following benefits:

- Unlimited technical support;
- New point releases of your TELE*form* products.

Point releases are only available to customers with an Annual Support and Maintenance Plan. Express shipping and handling is available for new point releases for a nominal fee. Contact your Cardiff sales representative for more information.

Before You Contact Technical Support…

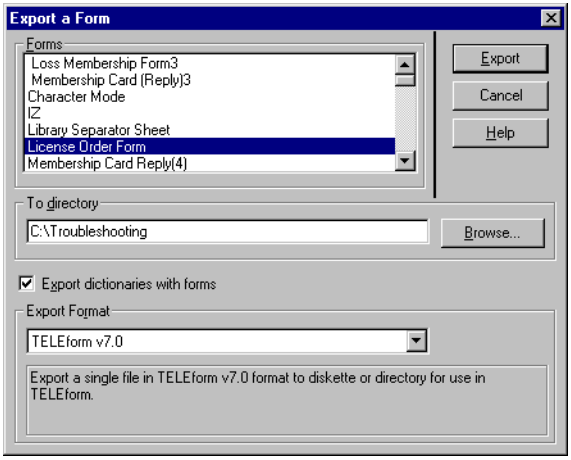
Before you call Technical Support, please have the following data available:

- The version and build number of TELE*form* that you are running. This information can be found by clicking **About** from the **Help** menu of any TELE*form* module;
- The registration code of your software. This code is printed on the CD case. You can also find it by starting TELE*form* License Manager and clicking **File** - **License Info**;
- The type of hardware you are using;
- The amount of available memory (RAM) and disk space on your system;
- A description of what you were doing when the problem occurred;
- The exact wording of any messages that appeared on your screen;
- Any other details pertinent to your problem.

If you are having recognition problems, please be ready to send the form template and several image files to Tech Support when you call.

Saving the form definition and image files

1. Create a “Troubleshooting” directory on your local drive.
2. In TELE*form Designer*, on the **File** menu, click **Export a Form**. The **Export a Form** dialog box will appear.



3. In the **Forms** window, select the form you need to troubleshoot.
4. Click the **Browse** button and navigate to your new Troubleshooting directory.
5. Make sure the **Export Format** box shows the version of TELE*form* you are currently using.
6. Click **Export**. The form template will be copied to the Troubleshooting directory.
7. Start TELE*form Verifier*.
8. Open the **Image Management Dialog**.
9. In the **Forms** window, select the form you exported.
10. In the **Stored Images** list, select 5-10 images of the form.
11. From the **File** menu, click **Save As** and save the images to the Troubleshooting directory.
12. Using a word processing application, create a “ReadMe.txt” file that contains step-by-step notes on the events leading up to the problem. Please save this file in a text (.txt) format. Save this file to the Troubleshooting directory.
13. If possible, compress the files in the Troubleshooting directory.
14. E-mail the form template and image files to your Cardiff Technical Support representative.

Contacting Technical Support from the Americas, Asia, and the Pacific Region

E-Mail: support@Cardiff.com

Web site: http://www.Cardiff.com

Fax: (760) 936-4850

Phone: (760) 936-4801. Phone calls are taken from 6 am to 5 pm Pacific Time, Monday through Friday (closed from 3 pm to 4 pm on Friday). Check the Cardiff Web site for up-to-date hours.

Mail: Cardiff
Attention: Technical Support
3220 Executive Ridge Drive
Vista, CA 92083 USA

Contacting Technical Support from Europe, Africa, and the Middle East

Cardiff, Ltd. is now providing direct technical support to customers in Europe, Africa, and the Middle East through the following methods:

E-Mail: UK_Support@Cardiff.com

Fax: +44(0) 208 326 1122

Phone: +44(0) 208 326 1111. Phone calls are taken from 9am to 5pm Greenwich Mean Time (GMT), Monday through Friday.

CHAPTER 3

Form and Global Form Scripts

About this Chapter

In this chapter, Form and Global Form scripts will be introduced and their components will be explained in detail. This chapter will also explain how to:

- Open the Edit Script window for writing Form scripts
- Execute your Form scripts

At the end of this chapter, there are examples illustrating some common uses for Form Scripts.

Overview of Form Scripts

Form scripts (also called "validation scripts") allow you to interact with TELE*form* at various points during the form processing cycle. Each TELE*form* form is assigned one Form script where validation routines can be written. When a form is evaluated, corrected, and exported, its Form script is executed in various stages throughout the processing cycle.

Form scripts are most often used for data validation (i.e. comparing returned data to some acceptable value within the script). This can include double-checking any mathematical calculations performed on the form, such as with time cards and order forms. Fields that do not pass the script's validation tests can be marked for review.

Besides validating returned data, Form scripts can be used to intelligently control the tabbing order during correction in TELE*form Verifier*. This "skip and fill logic" means that the script can decide which field to visit next based upon the value in the current field. For example, when correcting a survey form, the script could instruct *Verifier* to go to question 5 if the answer to question 3 is "No".

From the time a form is first evaluated by TELE*form Reader* to the time the form data is stored or exported, Form scripts provide many opportunities to control the action.

Overview of Global Form Script

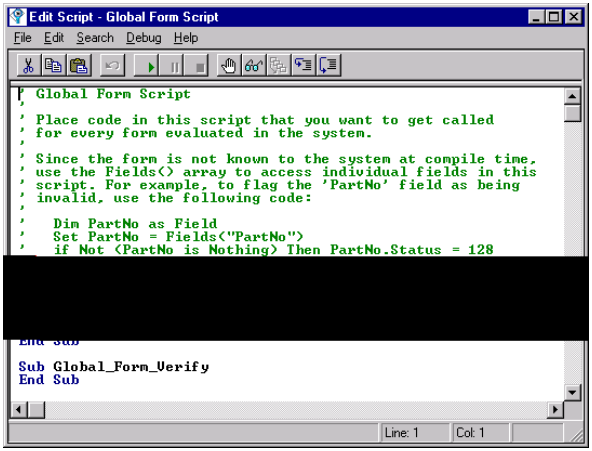
Sometimes it is preferable to have a script that runs for every form that is processed in TELE*form* (instead of for only a specific form). The Global Form Script can be used for this very situation. The Global Form Script has the same entry points that a regular Form script has; however, the entry points in a Global Form Script apply to every form template.

Recall that Form scripts (also called "validation scripts") allow you to interact with TELE*form* at various points during the form processing cycle. When a form is processed, its Form script and the Global Form Script are executed in various stages throughout the processing cycle.

Opening a Global Form Script For Script Writing

To access the Global Form Script, you must use the following procedure:

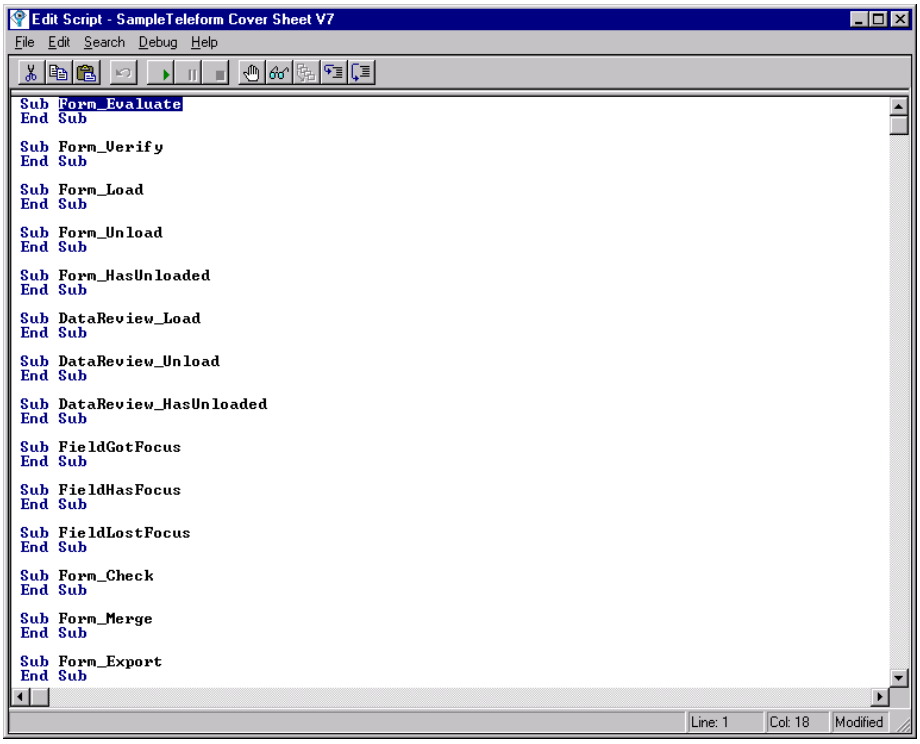
1. In TELE*form Designer*, select **Export Scripts** on the **Utilities** menu. The **Edit Script** window appears.
2. Click **Open** on the **File** menu. The **Open Script** dialog box appears.
3. Click the **Display Library** and **Custom Scripts** check box, select Global Form Script from the list, and then click **OK**. The Global Form Script is displayed in the **Edit Script** window.



Opening a Form Script for Script writing

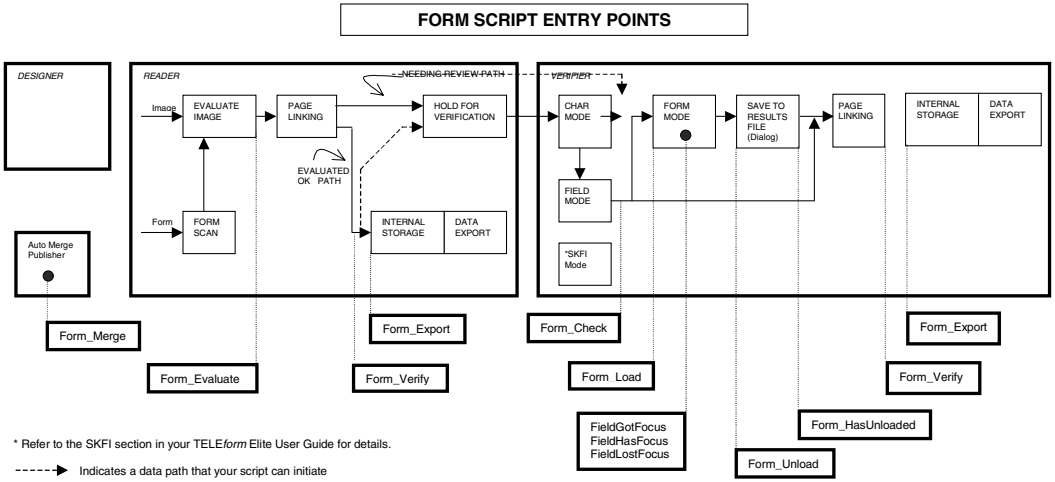
In order to write, edit and compile your script, you must use the BasicScript editor. This script editor is initiated when you open the Edit Script window in TELE*form Designer*:

1. Start TELE*form Designer.*
2. Open your form.
3. Click **Script** on the **Form** menu. The Edit Script window will display the Form script for that form.



Form Script Entry Points

The following diagram shows when each Form script entry point gets called with respect to the TELE*form* data flow.



A tabular description of each entry point is provided on the following pages. You'll probably want to refer to this table often when you first begin writing scripts. As you review these, note that the entry points form a sequence; each being executed at a slightly later point in the form processing cycle.

Typically a form script utilizes several entry points. For example, the script may use Sub Form_Evaluate to check the form data after evaluation and Sub Form_Verify to check the corrected value after verification.

Form Script Entry Point	Description
Sub Form_Merge (script) End Sub	This entry point is called at the beginning of a form merge operation in TELE*form Auto Merge Publisher*. The data to be merged onto the form corresponds to fields within the script. With this entry point, users can modify the data prior to merging it. **Note:** A merge cannot be aborted from within the form script. The Form_Merge entry point only allows editing the data that will be used in the merge.
Sub Form_Evaluate (script) End Sub	This entry point is called immediately after a form is evaluated by TELE*form Reader*. For multi-page forms: • it is run for each linked group in a multi-page form. • if the page link fields fail to associate some pages, it is run multiple times for the form. This entry point is frequently used to perform validations on form data and mark fields for review, as appropriate. This entry point is called for all HTML forms received by TELE*form* Internet Server.
Sub Form_Check (script) End Sub	When a single image is corrected in TELE*form Verifier*, this entry point is called after character and field mode correction but before entering form mode correction. You can use this to save the corrected data or run validations before going to form mode correction. When multiple images are corrected in TELE*form Verifier* at the same time, the following apply: • If all fields on a form are verified in character and field mode, Sub Form_Check is initiated as a background process while character and/or field mode continues for the remaining forms. • If form mode is required for any of the forms, Sub Form_Check is initiated prior to form mode occurring. • If your form contains SKFI zones, Form_Check is called after character mode, field mode, and all SKFI zones are complete. **Note:** If validations in Form_Check run for more than a second or two, the *Verifier* operator may get the message “Waiting for validations” when making a transition to form mode.

Form Script Entry Point	Description
Sub Form_Load (script) End Sub	This entry point is called as soon as a form enters form mode correction in TELE*form Verifier.*
Sub FieldGotFocus (script) End Sub	This entry point is called in the form mode of TELE*form Verifier* immediately prior to a field gaining the focus.
Sub FieldHasFocus (script) End Sub	This entry point is called in the form mode of TELE*form Verifier* when a field has the focus (when a field is highlighted). Sub FieldHasFocus can be used to receive information from the operator concerning the current field.
Sub FieldLostFocus (script) End Sub	This entry point is called in TELE*form Verifier* immediately after a field loses the focus (when you tab out of a field). Sub FieldLostFocus can be used to instantly check the corrected field and prevent invalid data from being entered by *Verifier* operators.
Sub Form_Unload (script) End Sub	This entry point is called before a form or partial form (i.e. missing pages) is closed in TELE*form Verifier.* This entry point will either be called right before the user is prompted to save changes, or before the user manually closes the form. This can be used to double-check edits made on the form or to change the form status to force the form to stay in TELE*form Verifier.* This entry point is also called after leaving a SKFI zone in SKFI Streaming Mode.
Sub Form_HasUnloaded (script) End Sub	This entry point is called immediately after Sub Form_Unload. If the user is prompted to save results, this prompt will appear before Sub Form_HasUnloaded is called. Sub Form_HasUnloaded allows you to close a file that you opened in Form_Load.
Sub Form_Verify (script) End Sub	This entry point is called after the *Verifier* operator corrects all images in the form image set. Complete form cross-validation can be performed in this entry point. The form can be forced to go through review again if validations do not pass. This entry point is called even if forms do not go through verification processing (those forms that are Evaluated OK in TELE*form Reader*). All page linking actions are complete before Form_Verify is entered for either TELE*form Reader* or TELE*form Verifier.*
Sub Form_Export (script) End Sub	This entry point is called immediately prior to data being exported to your data file or stored in an internal data file. Sub Form_Export can modify data immediately prior to export. Therefore, you can use Sub Form_Export when you want to avoid replacing your export format with an Export script.

Field-Specific Form Script Entry Points

During verification, it is often useful to be able to run a script in association with a particular field. This can be used to instantly check the corrected field and prevent invalid data from being entered by *Verifier* operators. To enable this type of "instant validation", BasicScript includes three entry points that can be attached to individual fields.

The following entry points are executed whenever the fields receive, have, or lose the focus during form mode correction in TELE*form Verifier*.

Field-Specific Entry Point	Description
Sub *FieldName*_GotFocus (script) End Sub	This entry point is called immediately prior to the specified field getting the focus in the form mode of TELE*form Verifier* (when you tab into the field).
Sub *FieldName*_HasFocus (script) End Sub	This entry point is called when the specified field has the focus in the form mode of TELE*form Verifier* (when the field is highlighted).
Sub *FieldName*_LostFocus (script) End Sub	This entry point is called immediately after the specified field loses the focus in the form mode of TELE*form Verifier* (when you tab out of the field).

If a script has both general entry points and field-specific entry points defined, the general entry point (i.e. FieldGotFocus or FieldLostFocus) gets called before the field-specific entry point.

Field-Specific Entry Point Examples

If the field **total** has a specific entry point, then the order of calls for GotFocus is the following:

1. **FieldGotFocus**
2. **total_GotFocus**.

Consider the case where you have five fields named Price1, Price2, Price3, Price4 and Price5. When a user leaves one of these fields, you want the focus to go to the Total field. You can write the following statement to do this:

```
Sub FieldLostFocus
    if Left$(LCase$(Form.CurField), 5) = "price" then
        Total.SetFocus
    end if
End Sub
```

Global Form Script Entry Points

Entry points are TELE*form*-specific subroutines that allow you to control each form processing operation. Each Form script entry point corresponds to a Global_Form entry point in the Global Form Script. For example, the Form_Evaluate entry point in a Form script has the following format in the Global Form Script:

```
Sub Global_Form_Evaluate
    (enter script here)
End Sub
```

The Global Form Script entry points include:

Global_Form_Evaluate	**Global_FieldHasFocus**
Global_Form_Verify	**Global_FieldLostFocus**
Global_Form_Load	**Global_Form_Check**
Global_Form_Unload	**Global_Form_Merge**
Global_Form_HasUnloaded	**Global_Form_Export**
Global_FieldGotFocus	**Global_DataReview_Load**
Global_DataReview_Unload	**Global_DataReview_HasUnloaded**

The Global Form Script entry points are always called **before** the corresponding Form script entry points. Accordingly, all Global Form Script entry points are executed for every TELE*form* form that has a Form Script associated with it. For example, if the form "Sample Andy's Time Card V6" has a Form Script written for it, the Global Form Script will be called for that form; since Global Form Script entry points apply to every form template associated with it.

If a Form script has not been written for a form, such as "Sample Comdex Show Form V6", then the Global Form Script would not be called since there is no script written for it. See "Form Script Entry Points" on page 24 for a description of the form processing cycle.

Global Form Script Entry Point Examples

Hiding a Global Form Script entry point

If you put an entry point into a regular Form script that has the Global_ prefix (such as Global_Form_Evaluate), this entry point will substitute for the corresponding entry point in the Global Form Script. In effect, doing this hides the Global Form Script's entry point from TELE*form* while you are processing this particular form.

Overriding existing Global Form Script entry points

All Global Form Script entry points are called for Forms that have Form Script associated with them. Some Forms, however, may require that the Global Form Script entry points NOT be called, or that they perform a different task. It is possible to redefine the Global Form entry points for a Form within the Form Script to override the existing Global Form Script entry point. For example, if the Global Form Script defines an entry point:

```
Sub Global_Form_Load
    MsgBox "Hello!"
End Sub
```

A message box saying **"Hello"** would appear each time a form with Form Script is loaded in *Verifier*. This behavior can be over ridden for a form by redefining this entry point in Form Script as shown in the following example:

```
Sub Global_Form_Load
End Sub
```

When this form is loaded in *Verifier* there will be no message box stating **"Helllo!"**.

Field Specific Entry Points

You can place field specific entry points in the Global Form Script, just as you can in a regular Form script. For example, if you have a field named LastName, you can create the following entry point.

```
Sub Global_LastName_GotFocus
    (enter script here)
End Sub
```

Classes compatible with the Global Form Script

You can only use the Form and Fields class properties in a Global Form Script. Although you cannot directly reference specific fields using the Field class objects, you can refer to a specific field by name in the Fields array (collection), as shown below:

```
Dim LastName as Field
Set LastName = Fields("LastName")
If Not (LastName is Nothing) Then
    LastName.Text = "Doe"
End If
```

The 'is nothing' test prevents a runtime error from occurring in the event that the script is run for a form that does not have the LastName field

Form Script Classes and their Properties

There are eight classes of objects in TELE*form*. Each class has a unique set of properties. With these properties, you can access the full range of TELE*form* information. Because the script must rely on logic and mathematics to execute properly, it is imperative that you use these properties correctly.

Form Class

Form class properties contain information about the particular form currently being processed by the Form script. The syntax for referencing Form class information is:

Form.*FormPropertyName*

where ***FormPropertyName*** is a valid property of the Form class. For example:

```
id = Form.FormId        'Assign the form ID to the variable 'id'.
name$ = Form.Title      'Assign the form's title to the variable
name$'.
```

The following properties are defined for the Form class:

Property	Type	Access	Description
Title	String	Read Only	Contains the name of the form.
FormID	Long	Read Only	Contains the form number assigned to the form.
Mode	Integer	Read Only	Contains the path of the form image being processed.
Image	String	Read Only	Contains the path of the form image being processed.
Status	Integer	Read Only	Contains and controls the disposition of the form. At form evaluation time, it indicates whether the form will be held for review (suspended) or evaluated OK. At verification, it determines how corrected data will be handled. The values of this property are described for each case in the table on page 31.
CurField	String	Read Only	Only defined in the GotFocus, HasFocus and LostFocus entry points. This property contains the name of the current field
CurGroup	String	Read Only	Only defined in the GotFocus, HasFocus and LostFocus entry points. Contains the name of the detail group that the current field belongs to (if the current field is part of a detail group). If the current field is not part of a detail group, then CurGroup is empty.

Form.Mode Property Values

Value	Value	Processing Status
Evaluation	**2**	*Reader*: evaluating a form
FormFill	**8**	Verifier: filling a new form Reader: processing an HTML or form
Suspense	**4**	Verifier: correcting a form in form mode or performing SKFI data entry in form mode.
Exporting	**16**	Reader, Verifier, Designer: exporting data
FinalValidate	**128**	Reader, Verifier: finish evaluation or correction (in the Form_Verify entry point)
FormCheck	**512**	Verifier: going from character mode, field mode or SKFI streaming mode to form mode.
FormMerge	**1024**	*Auto Merge Publisher*: merging a form (during the Form_Merge entry point). This value allows read and write access to all fields on a form before a merge. Virtual merge fields (those not printed on the form) such as Remote_Fax and Remote_Phn are excluded from this access.
SKFI	**2048**	*Verifier*: entering SKFI data in SKFI streaming mode
DataReview	**8192**	*Verifier*: performing quality control in Data Review mode.

Form.Status Property Values

The following table lists the Status property values that the Form script can be set to in TELE*form Reader* and *Verifier* for the entry points listed. For all other Form script entry points, setting Form.Status is not defined.

Value	Evaluation (Form_Evaluate) (Form_Verify)	Verification (Form_Unload)	Verification (Form_Verify)
Accept	interpreted OK	save corrections to results file	save corrections to results file
Cancel	not defined	ignore changes keep for later verification, and exit form mode	not defined
SaveAndExi	not defined	save the form in its current state and exit form mode	not defined
Suspend	needs review	keep user in form mode	store changes for later verification

Fields Collection (Array)

The Fields collection can be used by Form scripts and Export scripts. Fields is a collection that provides access to all fields on the form being processed.

As with all collections, the Fields collection includes the Count property, which indicates the number of fields on the form. This does not include fields inside detail groups, just the number of named fields at the top level of the form. Fields that are members of SKFI zones, data groups or address groups (including virtual fields) are included in this collection.

NOTE: Detail groups have a Fields collection for each one of their rows.

Referencing Fields Collection Information

The Fields collection represents the set of fields on the form, or a set of fields in the row of a detail group. These collections utilize an array structure to gain access to each item in the collection. The number 0 represents the first element in the array.

The syntax for referencing Field class properties for individual fields in the Fields collection is:

Fields(i).*PropertyName*

or:

Fields(*FieldName*).*PropertyName*

where:

i	Integer between 0 and Fields.Count - 1
FieldName	Field ID of the field on the form
PropertyName	Valid property of the Field class.

For example:

```
Dim i as integer
For i = 0 to Fields.Count - 1
    DispMsg "The " & i & "th field is " & Fields(i).Name
Next i

If Not Fields("Name") is Nothing Then
    DispMsg "The Name field contains the value " & Fields("Name").Text
End If
```

The following properties are defined for the Fields Collection:

Fields Collection Property	Type	Access	Description
Count	Integer	Read Only	Contains the number of top-level fields in the form. During export, this property is the number of exported fields.
( )	Field	Read Only	Contains the collection of top-level fields in the form. The value specified between the parentheses can be the following: • integer containing the number of the field in the count (0 to Fields.Count - 1) • string containing the field name. **NOTE:** You **cannot** use a variant in this property If the string does not identify a valid field in the fields collection, then **Nothing** is returned. If the integer is not within the range 0 to Fields.Count - 1, a runtime error occurs. (See "Nothing" in Chapter 8 for more information on uninitialized object variables.)

Field Class

The most frequently used object class is the Field class. Each object of the Field class contains the data of one field on the form. The Field class can be used in Form scripts, Export scripts and Library scripts (via parameters). In Export scripts, field objects are only accessible through the Fields collection.

Referencing Field Class Information

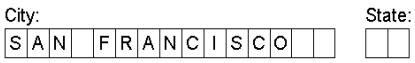
The syntax for referencing Field class information is:

FieldName.FieldPropertyName

where:

FieldName	Field ID of a field on the form
FieldPropertyName	Valid property of the Field class

For example:




```
IF City.Text = "SAN FRANCISCO" THEN State.Text = "CA"
```

Each Field object has the following properties:

Field Class Property	Type	Access	Description
Name	String	Read Only	Contains the field ID of the field.
Type	Integer	Read Only	Contains predefined values describing the format of the data field. See page page 39 for a table of values for this property.
Text	String	Read-Write	Contains the actual data associated with the field, up to 16 kilobytes (KB). Note that trailing blanks are stripped off before the script is executed. **NOTE:** As with all TELE*form* object properties, the Text property can only be modified by a direct assignment. It cannot be assigned as an argument within another function. Special handling of the Text property is performed on Choice fields. (See "ChoiceField.Text Property" on page 44 for details).
Value	Double	Read-Write	Contains the numeric value of the field. May only be used with numeric data entry fields. Note that assignments to the value property set the Text property value to be formatted according to the number of decimal places defined for the numeric field.
Status	Long	Read-Write	Contains a set of pre-defined values that describe the status of the field. When the field is evaluated and corrected, this property indicates whether the field is OK or needs review. See page 39 for a list of the Field.Status values.
Missing	Integer	Read Only	Contains the missing page status of the form page that includes the field (this property is only applicable during Form_Evaluate). **True**: the form page is missing **False**: the form page is not missing Because the individual pages of a multi-page form can be evaluated separately, you may want to test this property before running validations on a given field. For example, if SSN.Missing is true, it means that the SSN field is not in the set of pages currently being evaluated and will be handled by a different call to the script.

Field Class Property	Type	Access	Description
Mask	String	Read-Write	Contains a string of numbers (0-9) corresponding to each character in the text property. 0's indicate successful recognition and any other value indicates a failure condition as follows: **0** Successful recognition **1** Other error with character **2** Reserved **3** Not filled **4** Too many marks **5** Illegal Character **6** Indefinite Mark **7** Marks beginning of word not found in dictionary **8** Marks best guess character **9** Marks bad character, no best guess found **IMPORTANT:** This string is not always available (see the HasMask property below). Choice fields have 1 mask character per choice. **NOTE:** As with all TELE*form* object properties, the Mask property can only be modified by a direct assignment. It cannot be assigned as an argument within another function. For more information on Field.Mask property, see "Combining FieldName.Mask and FieldName.Text Properties" on page 102.
HasMask	Integer	Read Only	Contains the mask status of the field: **True**: the mask string is valid **False**: the mask string is not available and cannot be set **IMPORTANT:** References to Field.Mask when Field.HasMask is **False** will generate a run-time error, and will stop execution of your script.
Length	Integer	Read Only	Contains the maximum number of characters that may be assigned to the text property.

Field Class Property	Type	Access	Description
TabIndex	Integer	Read Only	Contains the field's order in the TAB sequence during form mode correction in TELE*form Verifier.* The tabindex value must be between 0 and Fields.Count - 1. When a new tabindex value is assigned to a field, all fields below it are moved down in the order. Attempts to access this property during the Form_Evaluate entry point will cause an error. **NOTE:** When assigning **TabIndex** to fields in BasicScript, assign fields in ascending order. Assigning fields in descending order will produce inconsistent results.
TabStop	Integer	Read-Write	Contains the TAB stop status of a field during form mode correction in TELE*form Verifier.* **True** if the value of the Field.Status property is nonzero **False** if the value of the Field.Status property is zero **NOTE:** TabStop is provided for backward compatibility. Use Field.Status instead.
TopChoices	TopChoice	Read Only	Contains a collection of TopChoice objects, each one corresponding to a character in the Text property. Note that this property is only valid if the HasChoices property is true. **IMPORTANT:** This property is not always available (see the HasChoices property below). (See page 110 for more information on the TopChoices collection.)
HasChoices	Integer	Read Only	Contains the status of the TopChoices property: **True**: the TopChoices property is defined for this field, **False**: the TopChoices property is not defined for this field . **IMPORTANT:** References to Field.TopChoices when Field.HasChoices is **False** will generate a run-time error and will stop execution of your script.

Field Class Property	Type	Access	Description
SetFocus	Method		In TELE*form Verifier*, this command results in the specified field being selected for correction. SetFocus is ignored if the mouse is used to select a specific field to go to, unless SetFocus is setting the focus back to the field the user is trying to leave. This can guarantee that valid data is entered in a field SetFocus can be used within a LosfFocus or GotFocus entry point. **NOTE:** In SKFI Streaming Mode, SetFocus calls to a field outside the current SKFI zone will be ignored.
Choices	Choice	Read Only	Contains a collection of choice objects for this field. Always **Nothing** for non-choice fields. Choices is a collection, so it has a Count property (see page 43 for more information on the Choices collection).
Count	Integer	Read Only	Contains the number of rows in a detail group. Always zero for non-detail groups.
( )	Row	Read Only	Contains a collection of rows. This collection of rows is valid only for detail groups. This value must be between 0 and *DetailField*.Count - 1. **NOTE:** A Row by itself is a collection, so it has a Count property. See page 110 for more information on the Row collection The result is a row of data inside the detail group. During verification processing in the GotFocus, HasFocus, or LostFocus entry point, a subscript of -1 indicates that the "current" row should be used.
CurRow	Integer	Read Only	Contains the current row being edited during verification. This value is only valid in the GotFocus, HasFocus and LostFocus entry points and only applies to fields that are members of a detail group.
ImagePage-Number	Integer	Read Only	Contains the page number of the form image from which the field was read. The first page of the image is page 0.

Field Class Property	Type	Access	Description
DoubleKey	Integer	Read Only	If a field is marked for Double Key, the Boolean value of *Field*.DoubleKey is 1. Otherwise *Field*.DoubleKey is 0. **NOTE:** *Field*.DoubleKey is only defined during Data Review and Form mode, and is 0 during Form mode correction.
Image-Orientation	Integer	Read Only	Contains the orientation of the form image from which the field was read. This property will be set to one or more of the following values: **ImageRotate90** - page is rotated 90 degrees **ImageRotate180** - page is rotated 180 degrees All rotations must be applied in a counter-clockwise fashion to the stored image before applying the **Left**, **Right**, **Top**, and **Bottom** field coordinates (see below). In other words, you should use this property to determine how much the image needs to be rotated before applying the coordinates.
Left, Right, Top, Bottom			**NOTE:** Each of the field coordinates (**Left, Right, Top,** and **Bottom**) are expressed in terms of the number of pixels the edge of the field is from the top or left edge of the image, after rotating this image with the Image-Orientation property.
Left	Long	Read Only	Contains the X (horizontal) coordinate of the left edge of the field (See the note in the Image-Orientation description above).
Right	Long	Read Only	Contains the X (horizontal) coordinate of the right edge of the field. (See the note in the Image-Orientation description above).
Top	Long	Read Only	Contains the Y (vertical) coordinate of the top edge of the field. (See the note in the Image-Orientation description above).
Bottom	Long	Read Only	Contains the Y (vertical) coordinate of the bottom edge of the field. (See the note in the Image-Orientation description above).

FieldName.Type Property Values

The following table lists the possible values for the Field.Type property:

FieldName.Type	Value	Description
NumberType	**1**	number
StringType	**2**	string (e.g, 'treat as text' checked in the field Attributes dialog box)
TextFileType	**3**	file name of a text file (image zone with **Store Value** and **In Separate File** selected in the Image Zone Attributes dialog box.)
ImageFileType	**4**	file name of an image (image zone with **Store Image** selected in the Image Zone Attributes dialog box)
DetailType	**5**	detail group record
ChoiceType	**6**	a choice field
KFIType	**9**	a SKFI zone

FieldName.Status Property Values

The following is a list of the possible values for the Field.Status property. These constant names are not pre-defined. Therefore, you must declare them at the top of your script (before your entry points) if you want to use them in your code. Each of these status values (except FldOK) can be combined with a bitwise OR operator.

To clear all of the *FieldName*.Status values for a field:

```
Const FldOK = 0
FieldName.Status = FldOK
```

An example of setting FldInvalid status on the **total** field is:

```
Const FldInvalid = 128
IF Not SumsAddOK Then
    total.Status = (total.status OR FldInvalid)
End If
```

By using OR, this statement preserves any existing status the **total** field may have such as FldReview.

Additional status values are available for TrueAddress fields and for setting custom status messages. Reference "Advanced Features of Scripts" on page 99 for more information on these features

The following table lists the possible values for the Field.Status property:

Constant Name	Value	Description
Const FldOK	0	Field interpreted OK
Const FldNotFilled	1	Field not filled
Const FldThreshold	2	Ambiguous choice or entry markings
Const FldRange	4	Data outside the numeric range defined in the Field Attributes dialog box
Const FldTooMany	8	Too many choices marked in a choice field
Const FldBadInterp	16	Low confidence recognition of a character in field
Const FldIOError	32	Failure to write to file (image zone)
Const FldReview	64	The Always review checkbox is selected in the Field Attributes dialog box
Const FldInvalid	128	Field validation failed
Const FldLookup	256	Database lookup error, invalid value in field
Const WordNotFound	512	Word not found in dictionary
Const FldIllegalChar	1024	Illegal character in the field
Const FldMissingPg	2048	Missing page
Const FldBlankZone	4096	Image Zone not filled in
Const FldLengthErr	8192	Length not correct (bar code)
Const FldKeyNotFound #	16384	Key for a variable location field was not found
Const FldWordChg	32768	Word changed by dictionary
Const FldInvalidDate #	268435456	Date Field contains a value that is not a valid date
Const FldBestGuessChar #	536870912	Best guess character
Const IndefiniteLocation #	1073741824	Indefinite location detected for field object

TELE*form* Virtual Fields

TELE*form* automatically includes several standard fields for use in export routines. These standard fields appear on every form and are referenced as any other TELE*form* field. These fields can also be exported along with the validated and corrected form data to your data file. Refer to "Table of TELEform Virtual Fields" on page 103 for a list of pre-defined TELE*form* virtual fields.

Route_To Field

The Route_To field (a TELE*form* virtual field) is filled whenever a form is being transferred to a specific individual or escalated for review or further verification by a Supervisor. BasicScript can read the field to determine the form's recipient. Also, BasicScript can set the field (along with Form.Status) to route or escalate the form to an individual.

NOTE: You can only use the Route_To field when the Security feature in TELE*form Verifier* is used. This field only applies to form mode correction in TELE*form Verifier*. For more information on TELE*form* Security, refer to your TELE*form* User Guide.

When the user clicks the **Send to** button in TELE*form Verifier*, the form is unloaded (causing the Form_Unload entry point to execute) and the text property of Route_To is set to the name of the form recipient.

When the user clicks the **Escalate** button in TELEform *Verifier*, the same procedure occurs, except that the text property is set to **Supervisor**. BasicScript can read the field value entered by the user and take the appropriate action.

If a user exits the form by:

- Clicking the Close button (cancelling), and then clicking **No** on the **Save corrections before closing** message (no save)
- Clicking the Close button, and then clicking **Yes** on the **Save corrections before closing** message (partial save)
- Completely correcting the form, and then clicking **Yes** on the **Save corrections to results file** prompt (full save)

Then the text property of Route_To is set to empty, and the form is neither sent nor escalated.

You can escalate or send forms in the Form_Unload entry point. In order to escalate or send the form, you must use the SaveAndExit status. Setting Form.Status to SaveAndExit tells TELE*form Verifier* to save the current state of the form, to keep the form in suspense, and to exit from Form Mode correction so that Route_To can send or escalate the form.

NOTE: Route_To and SaveAndExit have no effect in the Form_Verify entry point.

To escalate a form for review by a supervisor

```
Sub Form_Unload
    ...other code...
    Form.Status = SaveAndExit
    'Supervisor is not quoted since it is a special keyword
    Route_To = Supervisor
End Sub
```

To send a form to another user

```
Sub Form_Unload
    ...other code...
    Form.Status = SaveAndExit
    'The UserName must always be in quotes
    Route_To = "DaveL"
End Sub
```

The only way to activate the **Send to** and **Escalate** buttons is in form mode correction, as a user, when security features are enabled. For more information on Security features in TELE*form*, refer to your TELE*form* User guide.

NOTE: To send a form for review to a specific user in TELE*form Reader*, use the Form_Evaluate entry point, and set the Route_To field to the user name. To escalate a form in TELE*form Reader*, use the Form_Evaluate entry point, and enter Route_To = **Supervisor**.

Referencing Image Zone File Names

TELE*form* lets you save each image zone as a separate file.

The image file name can be accessed in BasicScript as:

PCX_*imagefieldname*.text

where:

imagefieldname is the field ID of the image zone on the form.

NOTE: In the Image Zone Attributes dialog box, select the **Store Image** check box to store each image in a separate file.

Choices Class

Only Form scripts can use the Choices class. Each field that is defined as a choice field has an additional property called Choices. The Choices property is a collection of Boolean values reflecting the settings in the Choice Field Attributes dialog box.

Referencing Choices Collection Information

The Choices collection represents the set of choices in a choice field. This collection utilizes an array structure to gain access to each choice in the choice field. The number 0 represents the first element in the array.

The syntax for referencing Choices collection information (other than Choices.Count) is:

ChoiceField*.Choices(i).*ChoicesProperty

where:

ChoiceField	Field ID of the choice field on the form
i	Integer between 0 and *ChoiceField*.Choices.Count - 1.
ChoicesProperty	Valid property of the Choices collection

The following properties are available for each choice field on your form:

Choices Class Property	Type	Access	Description
Count	Integer	Read Only	Contains the number of choice options in a choice field.
Text	String	Read Only	Contains the storage text string associated with an individual choice. For each choice, there is one storage text string.
Value	Integer	Read-Write	Contains the mark status of the choice. **True**: this choice has been marked **False**: this choice has not been marked **NOTE:** This setting this to True in a single-choice field sets all other choices to False. When the Value property is set, the *ChoiceField*.Text property is changed to reflect the new set of choices (this property is discussed below).

*ChoiceField.*Text Property

The *ChoiceField.*Text property is the text property of the choice field (a member of the Field class). The string that is assigned to the *ChoiceField.*text property is the tab-separated set of Storage strings whose choice values are marked.

- If the field has fixed storage, then there is one tab separating each possible choice, even if the choice is not marked (for example, a field with 7 choices would always contain 6 tabs in this property).
- If the field does not have fixed storage, then the number of tabs separating values depends on how many choices are marked. (For example, a 7 choice field with 3 choices marked would contain two tabs in this property).

Choices Property Example

Suppose we have a multiple choice field named X whose Storage values are Yes, No, and Maybe. The following statements are true:

○ Yes
○ No
○ Maybe

X.Choices(0).Text equals "Yes"
X.Choices(1).Text equals "No"
X.Choices(2).Text equals "Maybe"
X.Choices.Count equals 3

If we set **X.Choices(0).Value** to **True**, set **X.Choices(1)** to **False**, and set **X.Choices(2)** to **True**, then for a fixed choice field, **X.Text** is "Yes*TT*Maybe"; for a non-fixed choice field, then **X.Text** would be "Yes*T*Maybe" (where *T* is a single tab character).

Single choice vs. Multi-choice fields

If a choice field is defined to allow a single choice, then only one element in the collection may be True at a time. Setting a choice to True sets all other choices to False.

If a choice field is defined as multi-choice, then any number of choices may be True.

Referencing Choices from Scripts

Choice fields utilize the additional choices property because each field can consist of any number of choices.

For example, suppose the field ID of a choice field is Tendency. The Tendency choice field consists of five choices: Always, Usually, Occasionally, Usually Not, and Never. In this example, the storage values match the display values.

○ Always ○ Usually ○ Occasionally ○ Usually Not ○ Never

Using this choice field, the various properties of the choices collection can be demonstrated. The following script goes through each choice in the choice field. When it finds the selected choice, it tests to see if the selection was **Always**, and if so, sets the **qualify** variable equal to **1**.

NOTE: This script works for choice fields where multiple choices are allowed.

```
Sub Options_LostFocus
    Dim choicenum as integer          'Define the variable choicenum
    Dim Qualify as integer            'Define the variable Qualify

    'Loop through each of the choices
    For choicenum = 0 to Options.Choices.Count - 1
        'If a choice has been marked, examine its value
        If Options.Choices(choicenum).Value Then
            'Test to see if value is Always
            If Options.Choices(choicenum).Text = "Always" Then
                Qualify = 1
            End If
        End if
    Next choicenum
End Sub
```

Data Review Functionality

Data Review allows a *Verifier* operator to quickly review data from a batch that is ready to be committed (initiated when all of the forms in the batch have the status Evaluated OK). Depending on how your batch is set up, certain forms will be displayed in the Data Review window. Depending on how each form in the batch is designed, certain fields will be checked (or keyed) in Data Review. With this feature, a *Verifier* operator can perform quality control on the processing of each batch before the batch is committed.

This functionality only appears in TELE*form* Elite and TELE*form* Enterprise Edition. If you are running TELE*form* Standard, you will not see any DataReview-related entry points, properties or values.

For more information on the Data Review operation, refer to "Data Review" on page 136 of this Addendum.

DataReview Entry Points

The following entry points are called during Data Review in TELE*form Verifier*. These entry points are located in both the Form script of an individual form and the Global Form Script of TELE*form*.

Form Script Entry Point	Description
Sub DataReview_Load (script) End Sub	This entry point is called when a form enters Data Review Mode in TELE*form Verifier*.
Sub DataReview_Unload (script) End Sub	This entry point is called before a form or partial form (i.e. missing pages) is closed in Data Review. This entry point will either be called right before the user is prompted to save changes, or before the user manually closes the form. This can be used to double-check edits made on the form or to change the form status to force the form to stay in Data Review.
Sub DataReview_HasUnloaded (script) End Sub	This entry point is called immediately after Sub DataReview_Unload. If the user is prompted to save results, this prompt will appear before Sub DataReview_HasUnloaded is called. Sub DataReview_HasUnloaded allows you to close a file that you opened in DataReview_Load.

Refer to page 2-3 of your *BasicScript Guide* for more information on Form script entry points.

DataReview (Form.Mode Value)

The following Form.Mode value is used to distinguish Data Review mode from Form mode.

Mode	Value	Processing Status
DataReview	8192	*Verifier*: performing quality control in Data Review mode.

A generic example of using this new value in your script is provided below.

```
If Form.Mode = DataReview Then
   'Do something specific to Data Review mode.
Else
   'Do normal form mode stuff
End If
```

Refer to "Form.Mode Property Values" on page 31 for more information on Form.Mode values.

Field.DoubleKey Property

Double Key is a type of Data Review that forces the operator to re-enter the value of a field from scratch (he/she cannot see the corrected value). Then, the keyed-in value and the corrected value are compared to make sure they match. Double Key must be specified for a particular field in order for that field to undergo Double Key.

The DoubleKey field property allows a script to test whether or not a particular field is configured for Double Key.

Field Class	Type	Access	Description
DoubleKey	Integer	Read Only	If a field is marked for Double Key, the Boolean value of *Field*.DoubleKey is 1. Otherwise *Field*.DoubleKey is 0. **NOTE:** *Field*.DoubleKey is only defined during Data Review and Form mode, and is 0 during Form mode correction.

Executing Your Form Scripts

In order to execute your Form script, you must first successfully compile it and save it in the Edit Script window of TELE*form Designer*. If you receive any compile errors when compiling your Form script, you must resolve these errors **before** you attempt to execute it.

To execute a form script, use the following general procedure:

1. If your script includes the **Form_Merge** entry point, execute a form merge in TELE*form* Print Manager.
2. Evaluate a form image in TELE*form Reader*.
3. Correct this form image in TELE*form Verifier*.

   The routines in each entry point are executed automatically in response to the various events (evaluating, correcting, exporting) that occur.

IMPORTANT: Always compile your script after any changes to the Form.

If you want to isolate a particular entry point in your form script, refer to the following table:

Form Script Entry Point	Called...
Form_Merge	When TELE*form* Print Manager merges a record into your form. To start a form merge, click the **New Merge** button on the Form Merge Setup dialog box in TELE*form* Print Manager.
Form_Evaluate	When you evaluate a form image in TELE*form Reader*.
Form_Check	When you finish Character and Field Mode Correction for a form image. It is called right before you enter Form Mode Correction
Form_Load	Before you enter Form Mode Correction or SKFI Streaming Mode for a form image.
FieldGotFocus	Right before you tab into a field in Form Mode Correction
FieldHasFocus	After you tab into a field in Form Mode Correction, but before you type anything in the field.
FieldLostFocus	When you tab out of a field in Form Mode Correction

Form Script Entry Point	Called...
Form_Unload	When you close a form in TELE*form Verifier*. You can save the corrections to the results file, or cancel the edits you made to the form image. When you leave a SKFI zone in SKFI Streaming Mode
Form_HasUnloaded	After you click **Yes** on the 'Save corrections to results file' message in TELE*form Verifier*. Otherwise, immediately after Form_Unload, unless Form_Unload sets the Form.Status to Suspend
Form_Verify	After you have corrected all pages of a form in TELE*form Verifier*.
Form_Export	After you have corrected all pages of a form but before the data is exported to your data file.
DataReview_Load	When a form enters Data Review Mode in TELE*form Verifier*.
DataReview_Unload	Before a form or partial form (i.e. missing pages) is closed in Data Review. This entry point will either be called right before the user is prompted to save changes, or before the user manually closes the form. This can be used to double-check edits made on the form or to change the form status to force the form to stay in Data Review.
DataReview_HasUnloaded	Immediately after Sub DataReview_Unload. If the user is prompted to save results, this prompt will appear before Sub DataReview_HasUnloaded is called. Sub DataReview_HasUnloaded allows you to close a file that you opened in DataReview_Load.

PDF+Forms, Pdf+forms for Livelink, and HTML+Forms Evaluations

If you are using PDF+Forms, PDF+Forms for Livelink, and HTML+Forms, the entry point 'Form_Evaluate' will be called for all records received by TELE*form* Internet Server. To distinguish from paper form evaluations, the Form.Mode property will be set to "FormFill" rather than "Evaluation".

Normally, PDF+Forms, PDF+Forms for Livelink, and HTML+Forms (Filler) records are exported without going through TELE*form Verifier*. If, during the Form_Evaluate entry point, a field's status gets set to a non-OK value, then the record will be held for review. Rather than having an image as a backdrop, a filled in form template will be presented. The BasicScript entry points called during this phase will be the same as the normal Verification entry points (Form.Mode = Suspense).

Sample Form Scripts

If you want TELE*form* to perform a special function, it is very likely that you can write a Form script in BasicScript to accommodate and execute this function. This section includes examples that illustrate common BasicScript functions, and how these functions are integrated with the TELE*form* process. Although you may not be able to use these scripts word-for-word, you can modify them and then use them to do similar things in your TELE*form* system.

Using Form Scripts for TELE*form Verifier*

This section exemplifies the important role of scripts during form verification in TELE*form Verifier*.

Forcing Retries of Incorrect Data

If the SetFocus is to the current field, the user will effectively not be able to leave the field. This technique can be incorporated into a LostFocus routine to keep the focus on the current field until some acceptable value is entered.

For example, to prevent the user from entering a value for the numeric field ‘x1’ that is less than zero (0), you could write the following script:

```
Const FieldOK = 0
Sub x1_LostFocus
    If x1.value < 0 Then
        DispMsg "Amount cannot be less than zero.  Please
retry"
        x1.setfocus
     Else
        x1.status=FieldOk
    End If
End Sub
```

NOTE: Be careful not to trap the user in the current field with no way out.

Using the SetFocus Property

The SetFocus property can also be used to determine (conditionally) which field should be visited next. This property is useful if you want to employ skip-and-fill logic for the tab order in TELE*form Verifier*.

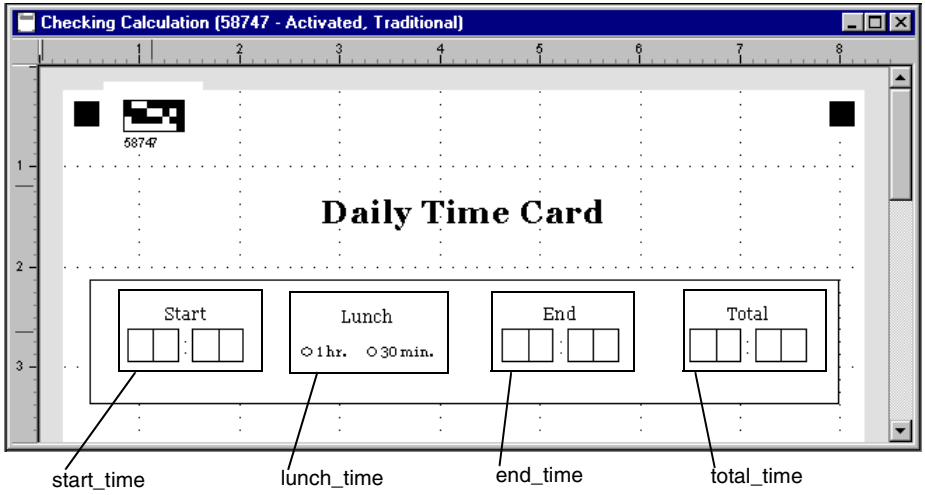
For example:

```
If q1.text = "Yes" Then
    q2.SetFocus
Else
    q3.SetFocus
End If
```

Sample Form Validation Script

Validation scripts are form scripts, and are therefore associated with a particular form. This script is for the time card form shown below. It checks to make sure that the total time equals the difference between the end time and the start time, less any lunch time.

The form contains the following fields:



Overview of the Validation Script

This script utilizes a user-defined function to convert all of the time data on the form into minutes so that calculations can be performed with the values. This function can be typed anywhere in the script file except inside another subroutine or function.

The function is called at evaluation time and will test the values on the form to see if they are correct. If the calculated "total time" value doesn't match the value written in the total_time field, the function will mark that field for review and hold the form for verification.

If the form needs review, the script will include a total_time_LostFocus event that will not allow tabbing out of the field until the correct value is entered.

‘Form Validation Script

```
Const SumFail        = 0
Const SumSuccess     = 1
Const FldInvalid     = 128          'field validation failed
Const FldOK          = 0            'field evaluated OK

‘declare the SumOK function for use throughout the script
Declare Function SumOK() as integer

Sub Form_Evaluate

    'call the SumOK function to test the values on the form.

    'if function fails, mark total_time field for review.
    If SumOK = SumFail Then
        'if total doesn't match, bitwise 'OR' turns on FldInvalid flag, forcing review.
        total_time.status = (total_time.status OR FldInvalid)
    End If
End Sub

'the following subroutine is executed when you tab out of total_time field.
Sub total_time_LostFocus

    'keep the focus on total_time field until correct value is entered.
    If SumOK = SumFail Then
        'if totals do not match, show message box.
        DispMsg "Total hours worked is not correct. Please re-enter."
        total_time.setfocus    'set focus back to the total_time field.
    Else
        total_time.status=0   'if total matches, accept value and proceed to the next field.
    End If
End Sub
```

```
'create a user-defined function that validates the total hours worked.   If this function fails,
'it returns False.

Function SumOK() as integer
'declare variables to hold "minutes" value for start, end, and total times and the calculated
'duration.
Dim beg_min  as integer                    'start time
Dim end_min  as integer                    'end time
Dim tot_min  as integer                    'total time
Dim duration as integer                    'calculated duration

'this code assumes that template characters are not stored with the data.

    'convert start time data to minutes
    beg_min  = Val(Left$( start_time.text, 2)) * 60 + Val(Mid$( start_time.text, 3,2))

    'convert end time data to minutes
    end_min  = Val(Left$( end_time.text,   2)) * 60 + Val(Mid$( end_time.text,   3,2))

    'convert total time data to minutes
    tot_min  = Val(Left$( total_time.text, 2)) * 60 + Val(Mid$( total_time.text, 3,2))

    'calculate the total hours from the individual fields
    duration = end_min - beg_min
    If duration < 0 Then                   'add 12 hours if not in 24 hour format
        duration = duration + (12 * 60)
    End If

    'subtract lunch break from calculated total hours
    duration = duration - Val( lunch_time.text )

    'compare calculated total to the entered total and set value of SumOK
    If duration = tot_min Then
        SumOK = SumSuccess
    Else
        SumOK = SumFail
    End If
End Function
```

Sample FieldGotFocus Script

A script writer can declare a global variable and set it to Form.CurField during FieldGotFocus to keep track of the last field visited at any time. This may be useful since global variables can be accessed from any entry point, including custom menu entry points. (Recall that Form.CurField is only defined in the FieldGotFocus, FieldHasFocus and FieldLostFocus entry points.) The script to accomplish this could be as follows:

```
 'Define the global variable
Public LastField as String
 ... other code here ...
Sub FieldGotFocus
    LastField = Form.CurField
End Sub
```

With the above example, 'LastField' can be used in a custom script to take action based on what field the user is currently on.

Sample Form_Merge Script

The following example fixes the Company name 'Cardiff Software' and calculates the Total field (by summing Line1, Line2 and Line3 fields). The printed form will show the new values for the Company and Total fields.

```
Sub Form_Merge

    ' Standardize the company name
    if Left$(LCase$(Company.Text), 16) = "cardiff software" then
        Company.Text = "Cardiff Software, Inc."
    end if

    ' Set the total field to the sum of the line fields
    Total.Value = Line1.Value + Line2.Value + Line3.Value
End Sub
```

CHAPTER 4

Export Scripts

About this Chapter

In this chapter, Export scripts will be introduced and their components will be explained in detail. The Export script components include the following:

- Export script entry points
- Export script classes and properties

This chapter will also explain how to do the following:

- Open the Edit Script window for writing Export scripts
- Execute your Export scripts

At the end of this chapter, there is an example of an Export script.

Overview of Export Scripts

Export scripts allow you to write customized export routines for TELE*form*. When you create an Export script, it is added to the list of standard export formats and becomes available for both auto and manual export. Export scripts are not connected to any particular form and they assume no prior knowledge of the TELE*form* forms or the data that is being exported. Export scripts simply read the data from a form and write it out to a file or process it in other useful ways.

When an Export script is saved, you are prompted to enter a name for it. The next time you start TELE*form Designer* and open one of the Export Setup dialog boxes, this name appears in the **Format** list.

NOTE: You must exit TELE*form Designer*, and then re-start it in order to view the names of newly added export scripts in the **Format** list.

Opening an Export Script for Script Writing

In order to write, edit and compile your Export script, you must use the BasicScript editor. This script editor is initiated when you open the Edit Script window in TELE*form Designer*

To create a new export script:

1. Start TELE*form Designer*
2. Click **Export Scripts** on the **Utilities** menu.
3. The Edit Script window displays a new export script.

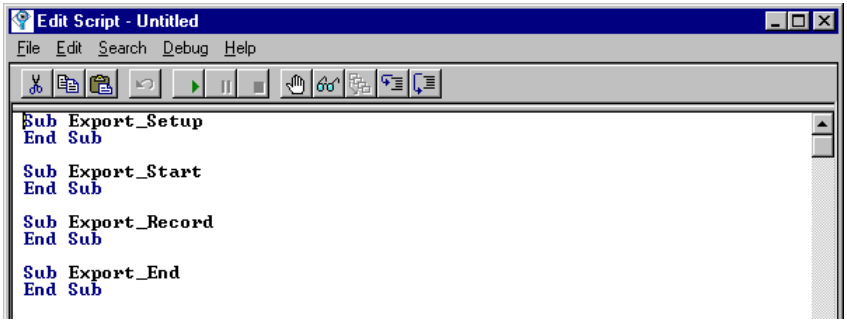
To open an existing export script:

1. Start TELE*form Designer*
2. Click **Export Scripts** on the **Utilities** menu.

   The Edit Script window appears.
3. Click **Open** on the **File** menu.
4. If a message appears, click **No** on the message to bypass saving changes to the Untitled script.

   The Open dialog box appears.
5. Click your export script in the list, and then click **OK**.

   The Edit Script window displays your export script.

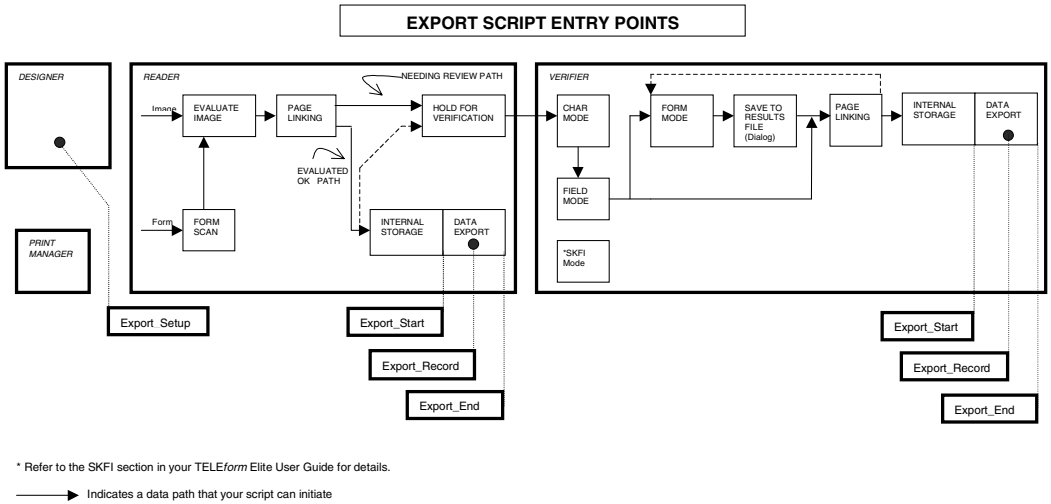


NOTE: You can only write and edit your Export script in the Edit Script window of TELE*form Designer*. If you try to edit your Export script in another TELE*form* application, you will not be able to compile and/or save it.

Export Script Entry Points

Export script entry points indicate the points where TELE*form* "enters" and executes the Export script. The entry point where you type in your code dictates at which point during the export process that code will be run.

The following diagram shows when each Export script entry point gets called with respect to the TELE*form* data flow:

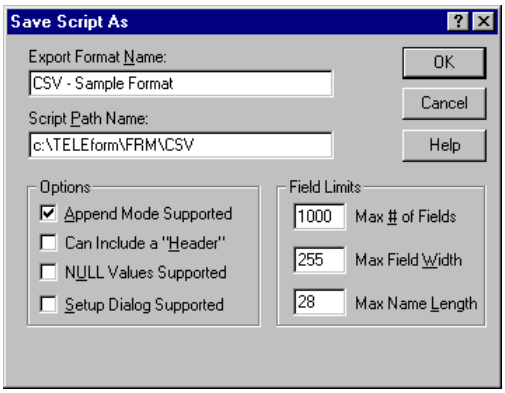


Each entry point is described in the following table.

Export Script Entry Point	Description
Sub Export_Setup (script) End Sub	This entry point is called in TELE*form Designer* when the user chooses 'Save As' on one of the Export Setup dialog boxes. This entry point will only be called when the **Setup Dialog Supported** check box is selected in the Save Script As dialog box or the Edit Capabilities dialog box (of the Edit Script window).
Sub Export_Start (script) End Sub	This entry point is called once for each export record and allows access to every exported field on the form. A typical export script goes through each field and writes the desired field information to the data file.
Sub Export_Record (script) End Sub	This entry point is called once for each export record and allows access to every exported field on the form. A typical export script goes through each field and writes the desired field information to the data file.
Sub Export_End (script) End Sub	This entry point is called at the completion of the export and is typically used to close the data file.

Saving Your Export Script

The first time you attempt to save your Export script, the Save Script As dialog box appears.



In this dialog box, you can specify the extension and name of your export format, and set up several export options. These options tell TELE*form* what your script's capabilities are. Enter these settings to accurately reflect your script's function.

NOTE: If your settings are not accurate, your data may not be exported correctly.

IMPORTANT: Always compile your script after any changes to the Form.

The following table briefly describes each option in the Save Script As dialog box. Many of these options can also be specified in the Edit Capabilities dialog box (on the **Edit** menu of the Edit Script window, click **Capabilities**):

Export Option	Description
Export Format Name	Name of the script as it will appear in the Format list of your Export Setup dialog boxes. **The first 3 characters of this name specify the file extension that will appear in the Save As dialog box when defining the data export file path and name.**
Script Path Name	Directory and file name where your Export script will be saved to
OPTIONS	
Append Mode Supported	If your Export script supports appending data to an existing file, select this check box.
Can Include a Header	If including a header in your script is optional, select this check box.
NULL Values Supported	If your script allows null (empty) values, select this check box. Otherwise, TELE*form* converts null numeric values to zero (0) for export. If your script supports null values, use the *FieldName*.Text property to test for blank fields instead of using the *FieldName*.Value property.
Setup Dialog Supported	Allows the Export_Setup entry point in your Export script to be called If you select this check box, you must fill the Export_Setup entry point with code that asks the user to fill in the name of the file to export to.
FIELD LIMITS	
Max # of Fields	Enter the maximum number of fields per form record that your Export script allows.
Max Field Width	Enter the maximum number of characters per field that your Export script allows.
Max Name Length	Enter the maximum number of characters per field ID that your script allows.

Export Classes and their Properties

As mentioned in Chapter 1, each class has a unique set of properties. With these properties, you can access the full range of TELE*form* information. Because the script must rely on logic and mathematics to execute properly, it is imperative that you use these properties correctly.

Export Class

Only Export scripts can use the Export class. This object class contains information about the export session.

Referencing Export Class Information

The syntax for referencing Export class information is:

Export.*ExportPropertyName*

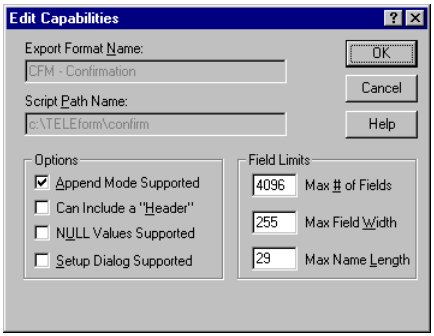
where ***ExportPropertyName*** is a valid property of the Export class

For example:

```
Open Export.Path for output as #FileNum
```

The following properties are defined for the Export class:

Export Class Property	Type	Access	Description
Path	String	Read-Write (Write only in Export_Setup)	Contains the full path of the data file. This is the file to which the script will write field data.

Capabilities	Integer	Read Only	Contains a number describing the capabilities of the export format.
			This value can be changed by clicking **Capabilities** on the **Edit** menu of the Edit Script window.
			 
			
			The value of this property equals the sum of the following constants. These are not predefined values but can be specified in the script as shown:
			Const AppendSupport = 04 'Append Mode Supported
			Const HeaderSupport = 08 'Can Include a Header
			Const AllowNull = 32 'NULL Values Supported
			Const SetupSupport = 02 'Setup Dialog Supported
			For example, if Header and Append Mode are enabled, then Export.Capabilities = 12.
			Header support can be tested by using the following syntax:
			If (Export.Capabilities And HeaderSupport) Then
			...
			End If
			The And operation is bitwise, and the result of it is only true if the Field Capabilities include header support.

Export Class Property	Type	Access	Description
Path	String	Read-Write (Write only in Export_Setup)	Contains the full path of the data file. This is the file to which the script will write field data.
MaxFields	Integer	Read Only	Contains the maximum number of fields exported from a single form at one time. This value can be changed by clicking **Capabilities** on the **Edit** menu of the Edit Script window. Range: 1-4096
MaxWidth	Integer	Read Only	Contains the maximum number of characters that can appear in any one data entry field. This value can be changed by clicking **Capabilities** on the **Edit** menu of the Edit Script window. Range: 1-16384
MaxNameLen	Integer	Read Only	Contains the maximum number of characters that can appear in the field ID of any one data entry field. This value can be changed by clicking **Capabilities** on the **Edit** menu of the Edit Script window. Range: 1-29
Format	String	Read Only	Contains the name of the export format as it appears in the **Format** list of the export dialog box.
Count	Integer	Read Only	Contains the number of forms remaining to be processed in this export session. At Export_Start, this value is the total number of forms to be processed. Each time Export_Record is called, this value is decremented by one.
Append	Integer	Read Only	Contains the append status of the export. **True**: the file already exists and you are appending **False**: the file is new

Export Class Property	Type	Access	Description
Path	String	Read-Write (Write only in Export_Setup)	Contains the full path of the data file. This is the file to which the script will write field data.
Header	Integer	Read Only	Contains the header status of the export. **True**: the user selects **Include Header** when starting the export operation **False**: the user does not select this option **Note:** Export scripts must select the **Can Include a Header** check box to allow the user to select **Include Header**. (See the Capabilities property above for more information on header support)
Result	Long	Read-Write	Contains the result of attempting to export a record. This value is returned to TELE*form* at the end of each Export_Record call. The script should set this either to a non-zero value (typically -1) to indicate failure of export or to zero (0) after successfully exporting a record.
Master	Export	Read Only	Returns the master record if the export being performed is the nested export of a detail record. Each row of a detail group is considered a separate record during export. The value of this property is: • **Nothing** for form-level exports • A value for detail group exports. (See "Nothing" in Chapter 8 for more information on uninitialized object variables.) Refer to your TELE*form* User Guide for more information on Detail groups.

Exporting detail groups

If you write an export format, special handling is usually not required for detail groups. Your export format just gets called once for each detail row as well as once for the master form record.

Because TELE*form* calls your script for detail groups before it is done with the master record, you should not use fixed file handles. Instead, use the function 'FreeFile( )' to obtain the next available file handle. See the Sample Export Script at the end of this chapter.

If you do require special handling for detail groups, the Export.Master property contains the master export object for the detail records during calls to Export_Start, Export_Record and Export_End. Export.Master is **Nothing** during export of the master form record.

Form Class

Form class properties contain information about the particular form currently being processed by the Export script. For more information on the Form class, see the "Form Class" section in Chapter 2.

Fields Collection

The Fields collection can be used by Form scripts and Export scripts. Fields is a collection that provides access to all fields on the form being processed. Therefore, you can access information from the Field class with the Fields collection. For more information on the Fields collection, see the "Fields Collection" section in Chapter 2.

Field Class

Export scripts typically use the Name, Type and Text properties of the Field class (within the syntax of the Fields collection). These and other Field class properties are discussed in the "Field Class" section of Chapter 2

Executing Your Export Scripts

In order to execute your Export script, you must first successfully compile it and save it in the Edit Script window of TELE*form Designer*. If you receive any compile errors when compiling your Export script, you must resolve these errors **before** you attempt to execute it.

To execute an Export script, use the following procedure:

1. If TELE*form Designer* is running, and you just created a new Export script, exit TELE*form Designer*.

   You must re-start TELE*form Designer* to put your newly created export format in the Format list of the Auto Export Setup dialog boxes.

2. Start TELE*form Designer*.
3. Open a form.
4. Click **Auto Export Setup** on the **Form** menu. The Auto Export Setup dialog box appears.
5. On the **Select** tab, select <none> and then click **Modify.** The Auto Export Setup dialog box for the form appears.
6. On the **Main** tab, in the Format list, select your export script name, and then click **Save As**.
7. The path that you coded in the Export_Setup Entry Point should be displayed in the dialog box.

NOTE: If nothing happens when you click **Save As**, you may have selected the **Setup Dialog Supported** check box (in the Edit Capabilities or Save Script As dialog box of the Edit Script window) without inserting any code in the Export_Setup entry point. If this is the case, go back and clear this check box.

8. Check the **Enable** checkbox to invoke this script and then click **OK**.
9. The Auto Export Setup dialog displays showing the export path and indicates it is enabled. Click **OK**.
10. Save the form.
11. Evaluate a form image in TELE*form Reader*.
12. Correct this form image in TELE*form Verifier*. The export routines are automatically executed.

NOTE: You can also execute the script by doing a manual export (click **Internal Data Export** on the **Utilities** menu).

Sample Export Script

The following Export script checks to see if a data file already exists.

- If there is an existing data file, the Export script appends exported records to this file
- If there is not an existing data file, the Export script writes a line of field names in the first line of the data file, and then adds records to this file.

For each record exported by TELE*form*, this script writes a line of data to the end of the data file.

```
Dim fileNum As Integer    'Declare global to hold file handle.  File
                           'with this handle will be opened in 'export_start'
                           'used in 'export_record' and closed in export_end'.

Dim newFile As Integer    'Declare global to remember whether file exists
                           'already.  This variable will be used to determine
                           'whether a header line should be written to the file.

'==================================================================
Sub Export_Setup

  export.path = "C:\My Directory\my file.txt"      'provide full path for export

End Sub
==================================================================



'==================================================================
Sub Export_Start

  fileNum = FreeFile()                             'Get the next available file handle

  If FileExists(export.path) Then
    Open export.path for append as #fileNum    'Open existing file for appending.
    newFile = FALSE                            'Remember that file already exists.
  Else
    Open export.path for output as #fileNum    'Create new file for writing.
    newFile = TRUE                             'Remember that file is new
  End If                                       '(so we can write a header record)

End Sub
'==================================================================
```

```
'=================================================================
Sub Export_Record

  Dim i As Integer                    'Declare integer to count through the file list
  Dim datastring As String             'Declare string to hold one line of data.


  'If the file was just created, then we need to write a header record.

  If newFile Then
    datastring =  ""                                  'Initialize to empty string.

  For i = 0 to Fields.Count - 1                       'Visit each field in the export list.
      If i > 0 Then datastring = datastring + ","      'Append a separator if not the first
                                                       'item on the line.
      datastring = datastring + fields(i).name        'Append the field name.
    Next i

    Print #fileNum, datastring                        'Write the header line to the file
    newFile = FALSE                                   'Don't go through this section again.
  End If

  'Prepare and write one record of data.

  datastring =   ""                                   'Initialize to empty string.

  For i = 0 to Fields.Count - 1                       'Visit each field in the export list.
    If i > 0 Then datastring = datastring + ","       'Append a separator if not the first item
                                                      'on the line.
    datastring = datastring + fields(i).text          'Append the field value.
  Next I

  Print #fileNum, datastring                          'Write the data line to the file.

End Sub
'=================================================================


'=================================================================
Sub Export_End

  Close #fileNum 'Close the file opened in 'export_start'

End Sub
'=================================================================
```

CHAPTER 5

System Script

About this Chapter

In this chapter, the System script will be introduced and its components will be explained in detail. The System script components include the following:

- System script entry points
- System script classes and properties

This chapter will also explain how to do the following:

- Open the Edit Script window for writing your System script
- Execute your System script

At the end of this chapter, there are examples illustrating some common uses for the System script.

Overview of the System Script

The system script brings the power of BasicScript to a global level in TELE*form*. It allows scripts to be tied to the start-up and shutdown of TELE*form Designer*, *Print Manager, Reader*, and *Verifier*, and provides script access to batch processing operations.

NOTE: There is only one system script per TELE*form* installation.

Public Variables

Besides its ability to assign script routines to the opening/exiting of TELE*form* applications and other processes, system scripts allow the declaration of public variables. Any public variables declared during the initialization of these applications will be available for every script. A good use for public variables is the storage of connection handles to a database.

Public variables are available to an application the entire time it is running. However, these variables are global only to the particular application where the variables were assigned:

- Public variables set in TELE*form Designer* cannot be read in TELEform *Reader*.
- Public variables set in TELE*form Reader* cannot be read in TELE*form Verifier*.
- Public variables set by one workstation are not available to other TELE*form* workstations.

The System script has entry points corresponding to each of the TELE*form* applications (*Designer*, Print Manager, *Reader*, and *Verifier*). Public variables that are declared at the top of the System script (before any of these entry points) are accessible for all scripts. They should be declared using the word 'Public' in the following manner:

```
Public num As Integer
```

Initializing Public variables for an application

Public variables can be used for many purposes. One of the most common uses is to initialize a variable on application start-up and then use it throughout the life of the application. For public variables used in this way, the *Application*_Init entry points are good places to initialize the variables (because these entry points will be called when the application starts up).

Opening a System Script for Script Writing

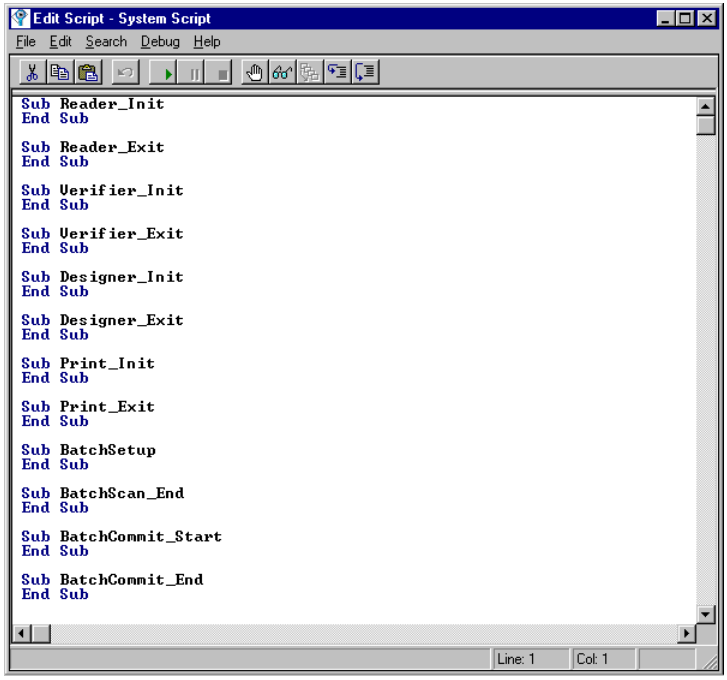
In order to write, edit and compile your script, you must use the BasicScript editor. This script editor is initiated when you open the Edit Script window in TELE*form Designer.*

1. Start TELE*form Designer.*
2. Click **Export Scripts** on the **Utilities** menu.

   The Edit Script window appears.
3. Click **Open** on the **File** menu.

   The Open Script dialog box appears.
4. Click **System Script** in the list, and then click **OK**.

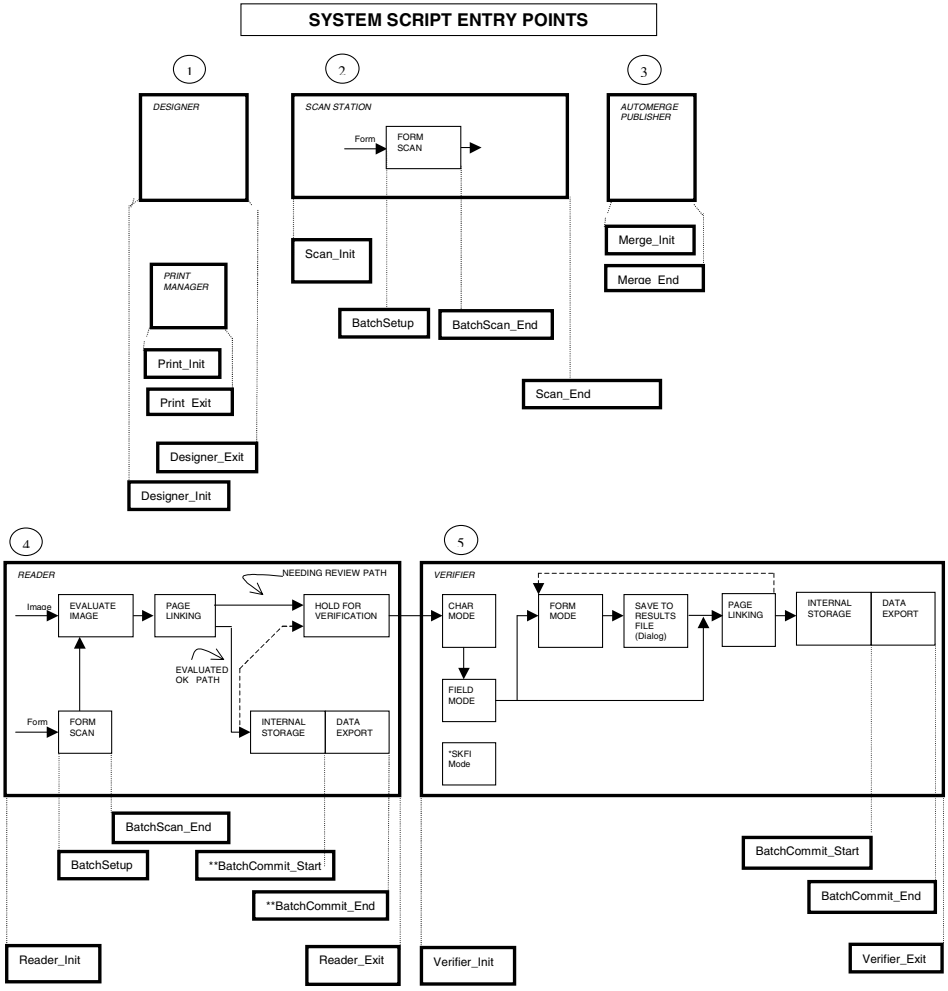
   The Edit Script window displays the System script.



NOTE: You can only write and edit your System script in the Edit Script window of TELE*form Designer.* If you try to edit your System script in another TELE*form* application, you will not be able to compile and/or save it.

System Script Entry Points

The following diagram shows when each System script entry point gets called with respect to the TELE*form* data flow:



* Refer to the SKFI section in your TELE*form* Elite User Guide for details.
** Utilities/Configuration/Local System must have "**EnabledAutoBatchCommit**" checked.
This is a toggle, IF BATCH.STATE = BATCH ERROR from *Reader* you will not go back to **BatchCommit_Start**. You will have to go to *Verifier* to correct error and re-commit.

→Indicates a data path that your script can initiate

The System script contains several entry points, each being called when the associated action or event occurs in TELE*form*.

System Script Entry Point	Description
Sub Print_Init (script) End Sub	This entry point is called each time TELE*form* Print Manager is started.
Sub Print_Exit (script) End Sub	This entry point is called each time TELE*form* Print Manager is closed.
Sub Designer_Init (script) End Sub	This entry point is called each time TELE*form Designer* is started.
Sub Designer_Exit (script) End Sub	This entry point is called each time TELE*form Designer* is closed.
Sub Reader_Init (script) End Sub	This entry point is called each time TELE*form Reader* is started.
Sub Reader_Exit (script) End Sub	This entry point is called each time TELE*form Reader* is closed.
Sub BatchSetup (script) End Sub	This entry point is called in TELE*form Reader*, TELE*form* Scan Station immediately after a user has entered batch settings in the Batch Setup dialog box and clicked the OK button. The system script is provided with properties that match each batch setting in the Batch Setup dialog box of TELE*form Reader,* TELE*form* Scan Station (see “Batch Class” on page 77). You can use Sub BatchSetup to validate batch settings and force the user back into the Batch Setup dialog box when information is invalid.

System Script Entry Point	Description
Sub Print_Init (script) End Sub	This entry point is called each time TELE*form* Print Manager is started.
Sub BatchScan_End (script) End Sub	This entry point is called after batch scanning is done but before the images are submitted into the system for evaluation. At this entry point, you have the option of: • Writing code to print or generate reports based on the batch information (Batch Object). • Modifying page or form counts so that *Verifier* operators do not need to manually calculate the number of forms based on the number of pages. • Over-riding the Accept/Reject Batch prompt so that the batch is automatically accepted. • Forcing the reject of a batch if certain criteria are not met. • Displaying a custom dialog box to collect information before handing the batch over to TELE*form.*
Sub BatchCommit_Start (script) End Sub	This entry point is called when you commit a batch of forms in TELE*form Verifier.* It is called immediately prior to storing and/or exporting the batch data records.
Sub BatchCommit_End (script) End Sub	This entry point is called after you commit a batch of forms in TELE*form Verifier.* It is called immediately after storing and/or exporting the batch data records. This is your system script's last chance to abort the batch commit process.
Sub Verifier_Init (script) End Sub	This entry point is called each time TELE*form Verifier* is started.
Sub Verifier_Exit (script) End Sub	This entry point is called each time TELE*form Verifier* is closed.

System Script Classes and their Properties

As mentioned in Chapter 1, each class has a unique set of properties. With these properties, you can access the full range of TELE*form* information.

Batch Class

Batch objects can only be used in the Batch entry points of the System script.

Referencing Batch Class Information

The syntax for referencing Batch class information is:

Batch.*BatchPropertyName*

where:

BatchPropertyName is a valid property of the Batch class

For example:

```
id = Batch.ID        'Assign the batch ID to the variable 'id'.
```

The following list describes each batch class property.

Property	Type	Access	Description
ID	Long	Read Only	Contains the unique number for each batch that is assigned by TELE*form.*
State	Integer	Read-Write	Contains the state of each batch. For more information on the Batch.State property, see page 80.
Priority	Integer	Read/Write	Verify that the Batch.Priority property in the Sub BatchSetup entry point is equal to what was set in the Batch Setup Dialog. Modify the Batch.Priority property in the Sub BatchScan_End entry point. Create a batch, then use Control Center to verify that the priority is equal to the value that was set in the entry point.
Flags	Integer	Read-Write (Write only in BatchScan_End)	These are flags used in the system to determine actions specified in the Batch Setup dialog box. **1** - Reject without prompting **2** - Accept without prompting **4** - Prompt on errors (missing pages...) **8** - NonForms expected **16** - Single form For more information, see “Batch.Flags Property” on page 81.
Pages	Integer	Read Only	Contains the number of pages expected for the batch (as entered or accepted by the user).
Forms	Integer	Read-Write (Write only in BatchScan_End)	Contains the number of forms expected for the batch (as entered or accepted by the user).
PagesEvaluated	Integer	Read Only	Contains the number of pages evaluated by TELE*form Reader.*
FormsEvaluated	Integer	Read Only	Contains the number of forms evaluated by TELE*form Reader* (a multi-page form is one form)

NonForms	Integer	Read Only	Contains the number of non-form pages in the batch.
WSName	String	Read Only	Contains the Workstation Name - the value taken from the Station Name configuration setting.
Prefix	String	Read Only	Contains the Batch file name prefix. For example, ‘0fe3’ in a file name series 0fe30000.tif, ...0fe39999.tif.
Ext	String	Read Only	Contains the Batch file extension. For example, ‘.tif’ in a file name 0fe30000.tif.
Directory	String	Read Only	Contains the batch directory. This is the same **Directory** that is displayed in the Batch Setup dialog box of TELE*form Reader*.
Comment	String	Read-Write (Write only in BatchScan_End)	Contains the batch comments. This the same **Comment** that is displayed in the Batch Options dialog box of TELE*form Verifier*. The maximum length for this property is 95 characters. If the script exceeds this length, the value will be truncated and a warning will be displayed
TrackId	String	Read-Write (Write only in BatchScan_End)	Contains the tracking ID. This is the same **Tracking ID** that is entered in the Batch Setup dialog box of TELE*form Reader*. The maximum length for this property is 19 characters. If the script exceeds this length, the value will be truncated and a warning will be displayed
UserName	String	Read-Write (Write only in BatchScan_End)	Contains the operator name. This is the same **Operator** that is entered in the Batch Setup dialog box of TELE*form Reader*. The maximum length for this property is 19 characters. If the script exceeds this length, the value will be truncated and a warning will be displayed
Date	String	Read-Write (Write only in BatchScan_End)	Contains a date. This is the same **Date** that is entered in the Batch Setup dialog box of TELE*form Reader*. The maximum length for this property is 11 characters. If the script exceeds this length, the value will be truncated and a warning will be displayed
FormId	Long	Read-Write (Write only in BatchScan_End)	Contains the form ID of the form in the batch. This property can only be set in conjunction with a Batch.Flags value of **16** (single form). See the Batch.Flags property for more information.

CommitCount	Integer	Read Only	Contains the number of times the user has attempted to commit a batch.
Time	Long	Read Only	Contains the date/time stamp of when the batch started. This value is system generated. C/C++ time_t data type. Returned number represents number of seconds from 01/01/70. **NOTE: Batch.Date** contains the date field entered by the scanner operator in the Batch Setup dialog box of TELE*form Reader.*
RecordCount	Long	Read Only	Contains the number of records that are ready to be committed by this batch. These are records that are evaluated O.K. Even if you force a batch to commit, the "Needs Review" records are still not considered as Ready and will not be reflected in the record count. The number of forms for the batch is still unknown until ALL records have the status of Evaluated OK. **NOTE:** If you want to know the number of records that will be committed even when you are forcing the batch commit with "Needs Review" records, you can use the .FormsEvaluated value. This should give you the correct number of records.

Batch.State Property

The following integer constants are predefined for use with the Batch.State value:

- BatchInProcess
- BatchReady
- BatchComplete
- BatchSetupComplete
- BatchError
- BatchSetupError
- BatchUserAbort

Upon entering BatchCommit_Start and BatchCommit_End, Batch.State is usually BatchReady. If the user decides to commit the batch before all forms have been evaluated, the Batch.State is BatchInProcess.

In BatchCommit_End, to allow the batch to commit, the script need do nothing. To go back to the state prior to the commit, the script must set Batch.State equal to BatchError.

In BatchSetup, to return the user to the Batch Setup dialog box, the script for Batch.State must contain the value BatchSetupError. The script writer must display an error message with this value to inform the user that there is a problem with the batch information. If the Batch.State property is not modified or is set to BatchSetupComplete, the batch will continue normally.

If the user manually aborts a batch from within TELE*form*, the Batch.State will contain the value BatchUserAbort. Alternatively, the script can set this value to indicate to TELE*form* that the user aborted the batch (which is useful if you allow the user to abort a batch from a BasicScript generated dialog box.)

Batch.Flags Property

The Flags property consists of a set of bit flags. This means that each flag value is added together. The following values are used for the Batch.Flags property:

Flags Value	Description
1	Reject without prompting
2	Accept without prompting
4	Prompt on errors (missing pages...)
8	NonForms expected
16	Single form

The 1, 2 and 4 values are mutually exclusive; they should never be set together. However, they can be set in conjunction with the 8 and 16 values.

NOTE: Use caution when changing the flags property.

For example, if the scan operator wants to be prompted when errors occur, and specifies a single form for the batch, the Flag property will contain the value 20 (16 + 4). However, if the script needs to change to Accept Without Prompting (for example 2 instead of 4), then the new value should be 18 (16 + 2).

Note that setting Form_ID to a non-zero value will be ignored unless the Flag property has the value 16 (single form) set.

Valid values for Flags are:

Value	Combination of...	Description
1		Rejects without prompting
9	**(1 + 8)**	Rejects without prompting and NonForms expected
17	**(1 + 16)**	Rejects without prompting and single form
25	**(1 + 8 + 16)**	Rejects without prompting, NonForms expected and single form
2		Accepts without prompting
10	**(2 + 8)**	Accepts without prompting and NonForms expected
18	**(2 + 16)**	Accepts without prompting and single form
26	**(2 + 8 + 16)**	Accepts without prompting, NonForms expected and single form
4		Prompts on errors
12	**(4 + 8)**	Prompts on errors and NonForms expected
20	**(4 + 16)**	Prompts on errors and single form
28	**(4 + 8 + 16)**	Prompts on errors, NonForms expected and single form

Invalid values are: 3, 5, 6, 7, 8, 11, 13, 14, 15, 16, 19, 21, 22, 23, 24, 27, 29 and higher.

Executing Your System Script

In order to execute your System script, you must first successfully compile it and save it in the Edit Script window of TELE*form Designer*. If you receive any compile errors when compiling your System script, you must resolve these errors **before** you attempt to execute it.

To execute the *Application*_Init and *Application*_Exit entry points:

1. Start TELE*form Designer*, *Print Manager, Reader* and/or *Verifier.*
2. Exit TELE*form Designer*, *Print Manager, Reader* and/or *Verifier.*

To execute batch entry points in your system script:

1. Start TELE*form Reader.*
2. Place filled-out forms in your scanner.
3. Click **New Batch** on the **Scan** menu.

   The New Batch dialog box appears.
4. Select **Scanner** and click **OK.**

   The Batch Setup dialog box appears.
5. Select the **General** tab.
6. Enter the batch parameters and click **OK**.

   The Forms are scanned by the scanner and the TELE*form* Reader dialog box appears.
7. Click on "**Stop Scanning".**
8. Start TELE*form Verifier*
9. Correct the batch of forms.
10. Commit the batch.

See the diagram in "System Script Entry Points" on page 74 to get a visual representation of when these entry points are called.

IMPORTANT: Always compile your script after any changes to the Form.

Common Examples of a System Script

Sample System Script

A primary feature of System scripts is the ability to declare Public variables. One practical application of Public variables is seen in the example below. This script shows how a Public variable can be passed to an Export script to track connection handles in the export database:

```
Public gblConnection as integer

Sub Reader_Init
    gblConnection = 0 'Not connected
End Sub

Sub Reader_Exit
    Disconnect( gblConnection )
End Sub
```

The above code would reside in the System script. In a separate Export script, you would set the gblConnection variable to the connection handle returned by your database Connect function. On subsequent exports, if the gblConnection variable is not 0, then you can re-use the connection and make the operation faster.

Public variables must be declared with 'Public'. To access the Public variables from another script file, the variable must be declared the same way in that script.

Sample BatchSetup Script

Sub BatchSetup is called in TELE*form Reader* immediately after a user has entered the batch settings in the Batch Setup dialog box and clicked the OK button.

Here is an example that only allows people named Roger to scan batches:

```
Sub BatchSetup
    If LCase$(Batch.UserName) <> "roger" Then
        Batch.State = BatchSetupError
        MsgBox "Sorry, unless your name is 'Roger', this batch can't go"
    End If
End Sub
```

NOTE: MsgBox is used here instead of DispMsg because the message should be forced in front of the user instead of being displayed in the *Reader* log (where the user might not see it).

Sample BatchScan_End Script

BatchScan_End is called after scanning is complete but prior to evaluating any images. The System script can change any of the writable Batch class properties in this entry point. To abort the batch, set Batch.State = BatchSetupError. Leave the State alone or set Batch.State = BatchComplete to let the batch go on normally.

Here is an example that allows a minimum of 50 pages in each batch.

```
Sub BatchScan_End
    If Batch.Pages < 50 Then
        MsgBox "You've got to scan more pages to get ahead in this company."
        Batch.State = BatchSetupError
    Else
        Batch.Forms = Batch.Pages/2  'These batches are always made of 2 page forms
    End if
End Sub
```

CHAPTER 6

Custom, Periodic and Library Scripts

About this Chapter

In this chapter, Custom, Periodic, and Library scripts will be introduced and their components will be explained in detail. These components include the following:

- Script entry points
- Script classes and properties

This chapter will also explain how to do the following:

- Open the Edit Script window for writing these scripts.
- Execute these scripts.

Overview of Custom, Periodic, and Library Scripts

The Custom, Periodic and Library scripts are rarely used by TELE*form* script writers. However, if the need arises, you can use these specialized scripts to do the following:

- **Custom Script -** Create a new menu on each TELE*form* application's menu bar that allows TELE*form* users to execute each Custom script.
- **Periodic Script -** Execute a function at fixed intervals in TELE*form Reader.*
- **Library Script -** Store commonly used functions in a library.

Custom Scripts

When you create a new Custom script, a command is added to a user-named menu in each TELE*form* application's menu bar. This type of script allows users to select a menu command when they want to execute the script. These scripts can be used for testing code and running data conversion programs.

When you create and save a Custom script, it is entered in the [Custom Menu] section of the Teleglob.ini file. By default, the name of the new menu is Script. However, this can be changed by editing the Teleglob.ini file.

To change the name of your menu:

1. Start Windows Notepad.
2. Click **Open** on the **File** menu.
3. Open your Teleglob.ini file (which is located in your **Teleform** directory).
4. Find the [Custom Menu] section in this file:

```
[Custom Menu]
Menu Title=&Script
Custom1=C:\Teleform\frm\Custom1
Custom2=C:\Teleform\frm\Custom2
```

NOTE: The **&** character precedes the shortcut key underlined on the TELE*form* menu bar.

5. In the **Menu Title** line, replace the word **Script** with the name of the menu that you want to see in your TELE*form* applications.
6. Save Teleglob.ini and close Windows Notepad.

   These changes will take effect the next time you start a TELE*form* application.

Notice that each line in the [Custom Menu] section corresponds to a command on the Custom script menu. TELE*form* allows you to add up to 20 commands (20 Custom scripts). When a user chooses the Custom script's command, TELE*form* will load the script specified and call the Sub Main entry point.

IMPORTANT: Always compile your script after any changes to the Form.

One use of a Custom script is to provide help to a *Verifier* operator that is specific to the field currently in focus. Because the Fields collection and the Field class are not available in Custom scripts, you can add code to your Form script that stores information in Public variables. See the "Sample FieldGotFocus Script" in Chapter 2. Using this sample script as an example, the Custom script knows which field is the current field by examining the LastField variable.

Assigning Accelerator Keys to Custom Scripts

Accelerator keys can be specified to execute whole scripts. Once you assign an accelerator key to your custom script, pressing the CTRL and/or SHIFT key in conjuction with an alphanumeric key will execute this script. You can also assign a function key to your custom script (for example, F6).

When an accelerator key assignment is activated, the key combination will be displayed on that script's menu item in the Script menu.

Possible key assignments

- Any function key except F1 and F10.
- A function key in combination with the control key [CTRL-F6] or in combination with the control and shift keys [CTRL-SHIFT-F6].
- Any alphanumeric key in combination with the control key [CTRL -G], or in combination with the control and shift keys [CTRL-SHIFT-G].

NOTE: The SHIFT key can only be used in combination with the CTRL key.

To assign an accelerator key to a Custom Script

1. Start Windows Notepad.
2. Open your **Teleglob.ini** file. This file is located in your TELE*form* directory.
3. Locate the **[Custom Menu]** section in this file.
4. Find the entry in this section that corresponds to the custom script for which you want to set up an accelerator key.
5. Add a description of the accelerator key to the end of this entry, separated by an asterisk (*). For example:

   Test Script=c:\teleform\test*F6

   This edit will assign the function key F6 to the custom menu script **Test Script**.
6. Repeat step **5** for each of your custom scripts.
7. Click **Save** on the **File** menu to save these edits to your **Teleglob.ini** file.
8. Exit Windows Notepad.

   These changes will take effect the next time you start a TELE*form* application.

Examples of acceptable accelerator keys

Test Script1=c:\teleform\test1*F4

Test Script2=c:\teleform\test2*CTRL-K

Test Script3=c:\teleform\test3*CTRL-SHIFT-F7

Test Script4=c:\teleform\test4*CTRL-SHIFT-K

Restricted Accelerator Keys

If the specified accelerator key conflicts with an accelerator used by Windows or TELE*form*, the custom menu accelerator will be disabled. You can tell if an accelerator was accepted by looking at the **Script** menu. If the accelerator keystroke is shown in the menu command, then the accelerator assignment was accepted.

NOTE: Make sure that you re-start TELE*form* before checking the **Script** menu for your accelerator keys.

The following table contains a list of accelerators that will **not** be allowed by TELE*form* or Windows:

NOTE: Because certain accelerator keys can be customized in TELE*form*, there may be other accelerator keys that your Custom scripts cannot use. Application-defined accelerator keys always take precedence over script-defined accelerator keys.

Application or System	Restricted Accelerator Keys
System	F1 F10 CTRL-F4 CTRL-F6
TELE*form Designer*	F1 SHIFT-F4 SHIFT-F5 CTRL-C CTRL-V CTRL-X CTRL-Z CTRL-Y CTRL-O CTRL-S CTRL-P
TELE*form* Print Manager	F1 SHIFT-F4 SHIFT-F5
TELE*form Reader*	F1 SHIFT-F4 SHIFT-F5
TELE*form Verifier*	F1 F12 (by default) CTRL-H CTRL-D CTRL-F CTRL-T CTRL-E CTRL-W CTRL-L

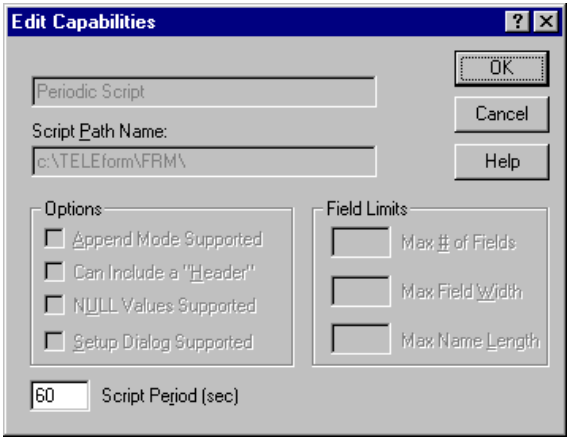
Periodic Script

The Periodic script is called by TELE*form Reader* at regular intervals. Periodic scripts are not allowed to run while TELE*form Reader* is evaluating a form and during some other processes such as scanning. As a result, the interval between script calls may be longer than the period specified.

To change the period of a Periodic script

1. Open the Edit Script window
2. Click **Capabilities** on the **Edit** menu.

   The **Edit Capabilities** dialog box appears.



3. Type the new period (in seconds) in the **Script Period (sec)** box, and then click **OK**.

Teleglob.ini and the Periodic script

When you create and save a Periodic script, it is automatically recorded in the **[Periodic Script]** section of the Teleglob.ini file. The first line in this section specifies the name of the periodic script that will be repeatedly executed. The second line specifies the interval that the script will be called. The default period is 1 minute.

For example:

```
[Periodic Script]
Periodic Script=F:\Teleform\frm\PS1
Period (secs)=60
```

In this example, ‘PS1’ is the name of the script file (without the .tfs extension) and TELE*form Reader* will call it every 60 seconds.

NOTE: When using TELE*form* Enterprise Edition, you might want the Periodic script to run on only one workstation. To accomplish this, the Periodic script section should be moved from the Teleglob.ini file (in your TELE*form* network directory) to the Teleform.ini file (in your WINNT network directory).

If you do this, do not edit the period of the script in the Edit Capabilities dialog box (of the Edit Script window). When you edit the period in this dialog box, you will place the Periodic script settings back in your Teleglob.ini file.

Library Scripts

Library scripts provide a consistent way to incorporate any code that you use into multiple forms or scripts. Once the code is in a Library script, it can be accessed from any script.

For example, suppose you have two different time card templates in TELE*form Designer*. Instead of having two copies of the validation code in your Form scripts (which validates the total hours worked by your employees), you can put the validation code into a Library script, and then call the Library script from each of your Form scripts.

The functions and subroutines defined in Library Scripts are not automatically available to other scripts. To access Library Script functions from other scripts each must be declared at the top of the other scripts. For example, if sub test MsgBox "Test" EndSub was defined in a Library Script, that same function must be declared at the top of a script that wants to access it Declare Sub Test. None of the standard TELE*form* objects (such as the Fields collection or the Form object) are available in a Library script. If you want to operate on a field, the field has to be passed in as a parameter.

For example:

```
declare Function GetName( fname as Field, lname as Field ) as integer
```

NOTE: Each library script must be compiled like a standard script.

Opening a Custom, Periodic, or Library Script for Script Writing

In order to write, edit and compile your script, you must use the BasicScript editor. This script editor is initiated when you open the Edit Script window in TELE*form Designer.*

To create a new script:

1. Click **Export Scripts** on the **Utilities** menu.

   The Edit Script window appears.

2. Point to **New** on the **File** menu.

   - If there is a submenu, go to **5**.
   - If there is no submenu, go to **3**.

3. Click **Open** on the **File** menu.

   The Open Script dialog box appears.

4. Click the **Display Library and Custom Scripts** check box, and then click **Cancel**.

5. Point to **New** on the **File** menu, and then click the type of script you want to create.

   The Edit Script window displays a new script of this type.

To open an existing script:

1. Click **Export Scripts** on the **Utilities** menu.

   The Edit Script window appears.

2. Click **Open** on the **File** menu.

   The Open Script dialog box appears.

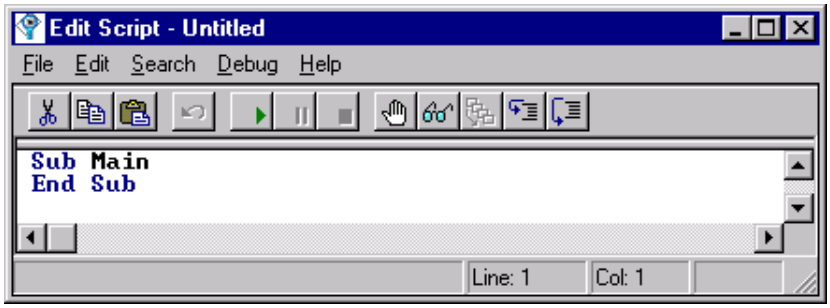
3. Select the script name in the list, and then click **OK**.

   The Edit Script window displays your script.

IMPORTANT: Always compile your script after any changes to the Form.

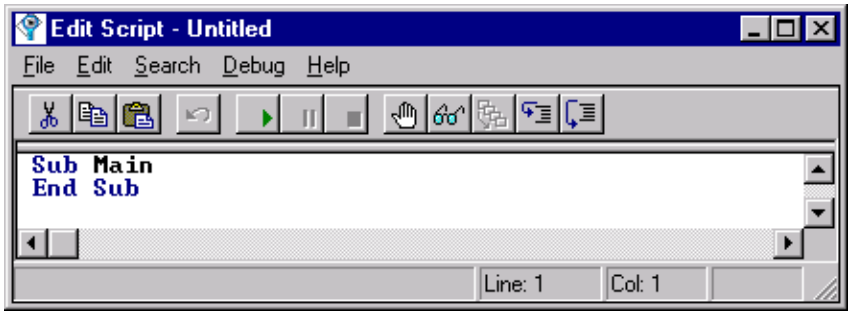
New Custom Script

The following figure shows what a new Custom script looks like:



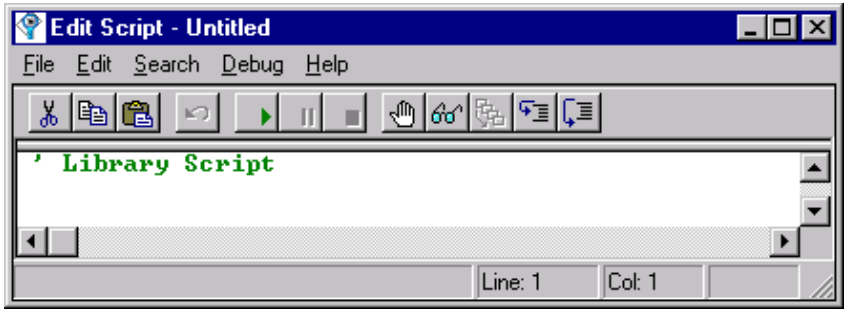
New Periodic Script

The following figure shows what a new Periodic script looks like:



New Library Script

The following figure shows what a new Library script looks like:



Notice that there are no pre-defined entry points in a Library script. Remember, a Library script is not called by TELE*form* directly. It can only be referenced by another script.

Custom, Periodic and Library Script Entry Points

Custom Script Entry Point

Custom scripts have a single entry point called Main, which contains the Custom script routine. This entry point is called when you click a command on the **Script** menu in a TELE*form* application.

Periodic Script Entry Point

The Periodic script has a single entry point called Main, which contains the Periodic script routine. By default, this script will be called every 60 seconds by TELE*form Reader.*

Library Script Entry Point

TELE*form* does not call entry points in Library scripts directly, so you can choose whatever name you like for the Library script function. Any function that is declared and written in a Library script can be called and used by any other TELE*form* script.

Executing Your Custom, Periodic and Library Scripts

In order to execute your scripts, you must first successfully compile and save them in the Edit Script window of TELE*form Designer*. If you receive any compile errors when compiling your scripts, you must resolve these errors **before** you attempt to execute your scripts.

Custom Script Execution

In a TELE*form* application, click the Custom script command on the **Script** menu (or on the menu name that you specified in the Teleglob.ini file).

Periodic Script Execution

Run TELE*form Reader*.

Because the Main entry point will be called by TELE*form Reader* at every specified interval, you must keep TELE*form Reader* open for at least this duration.

Library Script Execution

Use the execution procedure(s) that the script's function is used in.

Remember that Library scripts are used to store commonly used functions that can be referenced in other, active scripts.

NOTE: Library scripts cannot run while they are open in the Edit Script window.

To debug a Library script, open the script that will call the Library script. See Chapter 7 for more information on debugging your scripts.

CHAPTER 7

Advanced Features of Scripts

About this Chapter

The following sections explain in more detail the complex and less commonly used features of scripts.

TrueAddressFieldName.Status Values

The following values occur when a portion of a TrueAddress field cannot be validated.

TrueAddress Constant Name	TrueAddress Value	Description
Const FldAddress	**65536**	entire address
Const FldCityState	**FieldAddress * 2**	city, state or zip code
Const FldStreet	**FieldAddress * 3**	street number
Const FldStreetRange	**FieldAddress * 4**	street range
Const FldStreetName	**FieldAddress * 5**	street name
Const FldStreetDir	**FieldAddress * 6**	street direction
Const FldStreetSuffix	**FieldAddress * 7**	street suffix
Const FldStreetDirSuf	**FieldAddress * 8**	street direction (suffix)
Const FldZipRange	**FieldAddress * 9**	zip code
Const FldUndeliverable	**FieldAddress * 10**	address is undeliverable
Const FldName	**FieldAddress * 16**	name field that has low confidence character
Const FldCompany	**FieldAddress * 32**	company field has low confidence character
Const FldStreet2	**FieldAddress * 64**	street 2 field cannot be validated

Whenever a script sets the Field.Status to FldOK (0) or FldBlankZone (4096), the corresponding Field.Mask property is set to all '0' characters so that it matches the length of the corresponding Field.Text property. Any of the Status property values in the previous table can be put into either a conditional or a bitwise (boolean) Or statement with the *FieldName*.Status values, with the following exceptions:

- Each of the **Fld** values found in "FieldName.Type Property Values" on page 39 can be in an Or statement with any combination of the **FldName**, **FldCompany**, and **FldStreet2** values found in the TrueAddress table above.
- Only one of the other TrueAddress values (**FldAddress** through **FldUndeliverable**) may be included in an Or statement with the **Fld** values in "FieldName.Type Property Values" on page 39.

Custom Status Messages

Fields can be marked with any number of review conditions listed in the Status Property Values tables (see the previous section). There are also seven status codes reserved explicitly for BasicScript that can be set and/or read in a script. The corresponding status is then indicated to the user in *Verifier* when the field is corrected.

NOTE: Custom status messages appear only during form mode correction.

The following is a list of the custom status values:

Custom Constant Name	Custom Value	Description
Const FldCustom	**16777216**	custom message 1
Const FldCustomN	**FldCustom * N**	custom message N, where N is an integer between 2 and 7.
Const FldCustomPriority	**FldCustom * 8**	priority status for custom message

To specify custom status values:

1. Create the following section in your Teleglob.ini file (which is located in your TELE*form* directory):

   ```
   [Field Status Messages]
   Custom1 = My Message
   Custom2 = Name does not conform to the rules
   ```

   The message written in this file will be displayed next to the field in TELE*form Verifier* where you would normally see messages such as ‘low confidence character’ or ‘lookup failed’.

2. In your script, define the following constants:

   ```
   Const FldCustom  = 16777216
   Const FldCustom2 = FldCustom * 2
   Const FldCustomPriority  = FldCustom * 8
   ```

   Insert your custom status by choosing one of the values FldCustom through FldCustom7. Optionally, put this value into an Or statement with FldCustomPriority. Then, put your combined custom status into an Or statement with Field.Status to set the final status.

   Since a field can have multiple status messages, TELE*form* must choose which message to display. By default, TELE*form* displays built in status messages before custom status messages. If you include CustomPriority with your status value, TELE*form* will display your message first regardless of other status messages.

For example:

Suppose you have a field called **MyName** that had an unrecognized character. In BasicScript you set the status to indicate that the field does not conform to your special naming rules. (Using the definitions given earlier in this section).

If you set the status as follows:

```
MyName.Status = MyName.Status Or (FldCustom2)
```

The message 'Unrecognized Character' will appear in TELE*form Verifier* when the focus goes to the field Name. When the user corrects this character, the 'Name does not conform to the rules' message will appear.

If you set the status as follows:

```
MyName.Status = MyName.Status Or (FldCustom2) Or
FldCustomPriority
```

The message 'Name does not conform to the rules' will appear in TELE*form Verifier* when the focus goes to the field Name. However, if the field has a non-zero mask property, the 'Unrecognized Character' message will still take precedence.

Combining *FieldName*.Mask and *FieldName*.Text Properties

Most scripts never need to modify the Field.Mask property. However, if you set the Mask property of a field, it may be truncated to match the length of the Text property. Therefore, if you are setting both the Text and Mask properties of a field, the Text property needs to be set first to avoid losing part of the newly set Mask property.

An example of the wrong order is as follows:

Suppose Field.Text and Field.Mask contain 'myf' and '090' respectively. Then we write the following code.

This example will not work properly:

```
Field.Mask = "000090"
Field.Text = "myfile"
```

1. After the first line, Field.Mask is truncated to 3 characters ('000') so that it matches the length of Field.Text, which is still 'myf'.
2. After both lines have executed, Field.Text and Field.Mask contain 'myfile' and '000000', respectively

Therefore, Field.Mask gets truncated in the first step and expanded in the second.

This example will work properly

```
Field.Text = "myfile"
Field.Mask = "000090"
```

Table of TELE*form* Virtual Fields

The following table contains a description of each TELE*form* virtual field. Unless indicated otherwise, each field exports as a string with a length of 30 characters:

Virtual Field	Description
BatchCust1-5	These fields are configured in the Custom Fields tab of the Batch Setup dialog box in TELE*form Reader* or the Scan Station. They allow you to create customized data entry fields that a Reader or Scan Station operator must fill with information before a batch is processed.
BatchDir*	Field specifies the directory to hold the image files in during batch processing.
BatchNo*	Field uniquely identifies the batch of forms being processed. Exports a numeric field with a length of 10.
BatchPgCnt*	Field contains the number of pages in the batch. Exports a numeric field with a length of 4.
BatchPgDta*	Batch Page Data. Normally has scanner endorser string. Otherwise has original TIF filename under batch processing conditions.
BatchPgNo*	Each page in a batch is assigned a unique page ID corresponding to the order it was evaluated within the batch. This field contains that number. Exports a numeric field with a length of 4.
BatchRDate*	Batch Receive date
BatchScOpr*	Batch Scanner Operator
BatchTrack*	Batch customer tracking ID
CSID	The fax number (CSID) of the sending fax machine if such a number is available. If the form was scanned or received in any manner other than from a fax machine, this field is set to the file name of the image evaluated. The validity of the field (when received from a fax machine) depends on the sending fax machine, which must be manually programmed with its fax number by its owner.
Image_Seq	Image sequence - list of pages in the order they are in the file (separated by the choice field separator).

Virtual Field	Description
Form_ID	Contains the form's form_ID, range of 2-65000. Exports a numeric field with a length of 5.
Form_Notes	Notes that are entered by a *Verifier* operator during correction in TELE*form Verifier*. Exports a string with a length of 4096.
Form_Pri	Sets the priority of an individual form image. The range of values for this field are 0 (highest priority) to 255 (lowest priority). The default value is 100. FormPri is exported as a numeric field with a length of 5.
Orig_File	TIS will grab attachments from the MAPI service. They are placed in the RCV directory along with the XLQ file. The XLQ file will have an entry field for "**Orig_File**" virtual field if there is an attachment. The value of the field will be the full path to the attachment. "**Orig_File**" must be in the field order for an export (this is an optional field but required for digital signature capture).
OrigPgSeq	Contains the page number of each image in the batch, ordered consecutively. These images can either be forms or NonForm attachments.
Remote_Bid	Phone book ID for the remote user, with a range of 0-255. This field is exported as a numeric field with a length of 3.
Remote_Cmp	Contains name of the company associated with remote user, as specified in the phone book.
Orig_File	TIS will grab attachments from the MAPI service. They are placed in the RCV directory along with the XLQ file. The XLQ file will have an entry field for "**Orig_File**" virtual field if there is an attachment. The value of the field will be the full path to the attachment. "**Orig_File**" must be in the field order for an export (this is an optional field but required for digital signature capture).
OrigPgSeq	Contains the page number of each image in the batch, ordered consecutively. These images can either be forms or NonForm attachments.
Remote_Bid	Phone book ID for the remote user, with a range of 0-255. This field is exported as a numeric field with a length of 3.
Remote_Cmp	Contains name of the company associated with remote user, as specified in the phone book.

Virtual Field	Description
Remote_Fax	Contains the fax number in the phone book that matches the Remote_Uid if one exists.
Remote_Phn	Contains the phone number in the phone book that matches the Remote_Uid if one exists.
Remote_Uid	The ID for the remote user, with a range of 0 - 32767. It is exported as a numeric field with a length of 5.
Remote_User	Contains the fax sender's name, as configured in the receiving phone book.
SuspenseFile	Contains the name of the file in the /SUS directory that contains the image that produced this export record. In the export format, the column by this name will contain the name of the file in the suspended images (sus) directory. This field is only valid when forms that are Evaluated OK are saved.
Time_Stamp	The date and time the form was received or evaluated.
Route_To	Use in BasicScript to route forms to other workstations for verification. For more details on this virtual field, refer to "Route_To Field" on page 41.
Verify_Wks	For use with TELE*form* Enterprise Edition, this field holds the name of the workstation that performed the verification on the form. By default, this field is assigned the value from the "Station Name=" line in the TELEFORM.INI file. This occasionally causes a problem if more than one person uses the system. To solve this problem, you can set an environment variable called TFUSER during either a network login script or during boot-up. If TFUSER is defined, its value is automatically assigned to the Verify_Wks field. This ensures that the proper user name is associated with the person performing the TELE*form* operations. **NOTE:** If security is turned on, login name overrides "station name=" value.

* for batch scanning only

LoseFocus Field Property

The LoseFocus property is a Field class property that will initiate leaving a field while in Form Mode Correction (of TELE*form Verifier*). This property is most useful when a script dialog box has allowed the operator to correct data in a field, leaving no reason to stay in the field.

NOTE: The LoseFocus Property can only be used in the Sub FieldHasFocus entry point and the Sub *FieldName*_HasFocus entry point

Field Class Property	Type	Description
LoseFocus	Method	In TELE*form Verifier*, this command results in the highlighted field being closed and the next field in the field order being opened for correction. The LoseFocus method can only be used within a HasFocus entry point

Examples Using the LoseFocus Property

```
Sub MyName_HasFocus
    'prompt user for input with a dialog box
    ...
    MyName.Status=0
    MyName.LoseFocus
End Sub
```

Additional Batch Class Properties

There are four additional properties of the Batch class, all of which apply to the new Batch Processing capabilities. Each of these properties can be read from the Batch Setup dialog box, and/or set in the **BatchScan_End** entry point of the System Script. Refer to the following table for specific information on each property.

Batch Class Property	Access	Description
ClassificationReview	Read/Write	This property specifies the Job QC status of a batch. • ClassificationReview = 1 if Job QC is selected for the batch. Classification Review =0 if no Job QC is done on the batch.
DataReview	Read/Write	This property specifies the Data Review status of a batch. • DataReview = 1 if a review of the batch data will be done (after the batch is ready to be committed), DataReview =0 if Data Review is not specified for the batch.
DataReviewMethod	Read/Write	This property specifies what kind of review should be done of the batch data (it is only applicable if DataReview = 1): • **DataReviewAll** (0) - review all forms • **DataReviewPercent** (1) - Review N percent of the forms **DataReviewInterval** (2) - Review every Nth form.
DataReviewNumber	Read/Write	This property specifies the value of the DataReviewMethod property (it is only applicable if DataReview = 1). The significance of this value is dependent on the value of DataReviewMethod: • For DataReviewPercent, valid values are 0 to 100 (percent). • For DataReviewInterval, valid values are 1 to 255 or the number of forms in the batch, whichever is less. Otherwise, DataReviewNumber is ignored.

TopChoice Class

The TopChoice class can only be used in the Form_Evaluate entry point of your form script. The field must be set up for character recognition (OCR). During form evaluation, the character recognition engine typically produces a set of three choices for each character position in the Text string of a field. These choices are available through the TopChoices collection. Each element in this collection is a TopChoice object.

Referencing TopChoices Collection Information

The TopChoices collection represents a set of recognition values for a character. This collection utilizes an array structure to gain access to each recognition value for the character. The number 1 represents the first element in the array. Each element is a TopChoice object.

The syntax for referencing TopChoice information in the TopChoices collection is:

FieldName*.TopChoices(i).*TopChoicesProperty

where:

FieldName	Field ID of the field on the form
i	An integer between 1 and Len(*FieldName*.Text)
TopChoicesProperty	Valid property of the TopChoice class

The TopChoice class has the following properties:

Property	Type	Access	Description
Choices	Integer	Read Only	Choices is an array of three characters which are the three most likely candidates for the character.
Confidence	Integer	Read Only	Confidence is an array of three values which are the confidences of each of the three characters above.
Left*	Long	Read Only	Contains the X coordinate of the left edge of the character.
Right*	Long	Read Only	Contains the X coordinate of the right edge of the character.
Top*	Long	Read Only	Contains the Y coordinate of the top edge of the character.
Bottom*	Long	Read Only	Contains the Y coordinate of the bottom edge of the character.

* Each of the Left, Right, Top, and Bottom properties are expressed in terms of the number of pixels the edge of the character is form the left or top edge of the image after rotating the image according to *FieldName*.ImageOrientation (see page 42 for more information on the ImageOrientation property).

The 0-th element in the Choices array (and the Confidence array) is often the one stored in the Text property.

TopChoices Property Example

Given a field X at form evaluation time, the TopChoices class may be used as follows:

```
Dim i as Integer
'If the field HasChoices is true, then evaluate the field
If X.HasChoices Then
    'Loop through each character in the field text string
    For i = 1 to Len (X.Text)
        If X.TopChoices(i).Confidence(0) < 90 Then
            DispMsg "First choice < 90"
        End If
    Next i
End If
```

Unlike most Collections, the TopChoices Collection goes from 1 to the number-of-characters in the text property. This is done to correspond with the string array, which is always indexed from 1.

Row Class

Only your Form script can use the Row class. Each detail group has a collection of rows. Refer to your TELE*form* User Guide for more information on detail groups.

Referencing Row Collection Information

The Row collection represents the set of rows in a detail group. This collection utilizes an array structure to gain access to each row in the detail group. The number 0 represents the first element in the array.

The syntax for referencing Field class properties for fields within a Row object is:

DetailFieldName*(i).*FieldName.FieldPropertyName

where:

DetailFieldName	Field ID of the detail group on the form
i	Integer from 0 to *DetailFieldName*.Count - 1
FieldName	Field ID of the field in the detail group
FieldPropertyName	Valid property of the Field class.

The example below shows a detail field named Order that consists of a collection of 3 row objects. Referencing the data in a particular field now requires that you specify which row the field is in.

TABLE 1.

Order:

Item Cost		Quantity		Total Price
12.00	X	3	=	36.00
1.50	X	2	=	3.00
0.59	X	1	=	0.59

```
Sub Form_Evaluate
    Dim i as integer
    Dim sum as double
    sum=0 'initialize the variable
    For i = 0 to Order.Count-1
        sum = sum + Order(i).TotalPrice.Value
    Next i
    DispMsg "The grand total is "+ "$"  + Str$(sum)
End Sub
```

The script routine shown here goes through each row in the detail group and adds the value in the TotalPrice field to the sum variable. It then displays the net sum as the grand total.

The following property is available for each row of a detail group:

Property	Type	Access	Description
Fields	Fields	Read Only	The collection of fields within the specified row. This property has exactly the same properties as the fields declared for the entire form (in the Field class), including a Count property.

Because the Row class represents member fields in a detail group, individual fields in a detail group row can be accessed using the Fields collection.

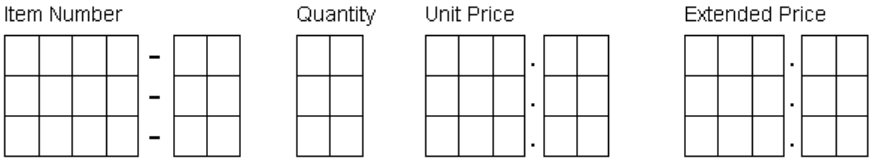
In the Row class example above, Order(i).TotalPrice is equivalent to Order(i).Fields("TotalPrice").

Or, to check all fields in all rows of the 'Order' detail group for unacceptable entries, you could use the following code:

```
Dim row as Integer
Dim f as Integer

For row = 0 to Order.Count - 1
    For f = 0 to Order(row).Fields.Count - 1
        If Order(row).Fields(f).Type = NumberType Then
            If Order(row).Fields(f).Value < 0 Then
                DispMsg "Row " & row & " field " & Order(row).Fields(f).Name & _
                        "Contains an illegal value."
            End If
        End If
    Next f
Next row
```

Row Class Example

Setting the index to -1 in a GotFocus, HasFocus or LostFocus subroutine references the current row. For example, suppose a detail group named "Order" is defined with the following fields.



The following script tests the detail group to make sure that the extended price value equals the product of the unit price and the quantity for each row.

Extended Price = Quantity * Unit Price

The script has two basic parts:

- The first part of the script is called from the Form_Evaluate entry point, which tests the data as soon as it is evaluated. It checks the extended price field in each row, and marks it as needing review if the value is not valid.
- The second part of the script is called from the Eprice_LostFocus (Extended Price) event. It tests the extended price value when you tab out of that field during verification. The script keeps the focus on the Eprice field until a valid value is entered.

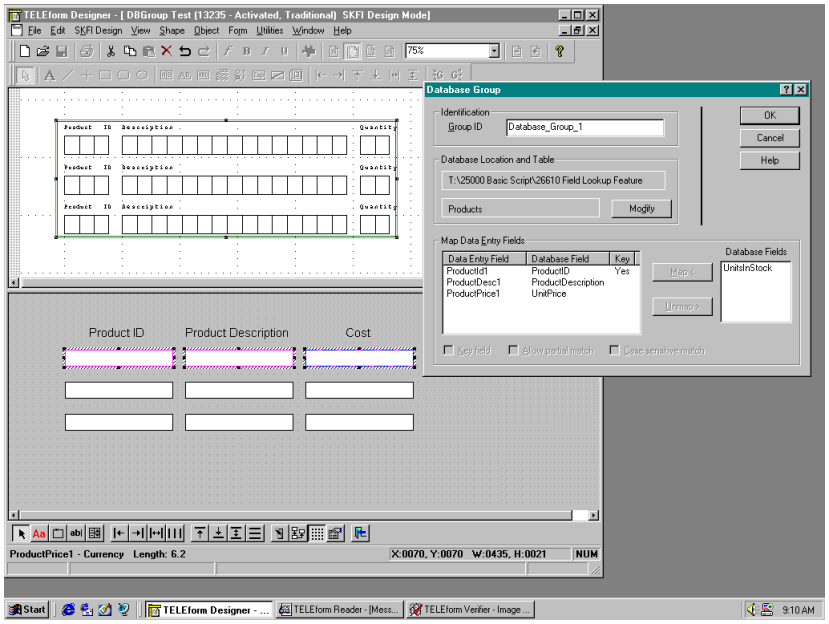
```
Const FldInvalid = 128

Sub Form_Evaluate
    Dim row as integer
    Dim CalcPrice as Double
    'Check the value of Eprice in each row.  Mark Eprice for review if incorrect
    For row = 0 to Order.Count-1
        CalcPrice = Order(row).UPrice.Value * Order(row).Qty.Value
        If Order(row).Eprice.Value <> CalcPrice Then
            'The bitwise 'or' turns on the FldInvalid flag, forcing review.
            Order(row).Eprice.Status = Order(row).Eprice.Status or FldInvalid
        End If
    Next row
End Sub

Sub Eprice_LostFocus
    Dim CalcPrice as Double
    CalcPrice = Order(-1).UPrice.Value * Order(-1).Qty.Value
    'Confirm that the extended price is correct before changing the focus.
    'If value is incorrect, set the focus back to Eprice until valid value is entered
    If Order(-1).EPrice.Value <> CalcPrice Then
        DispMsg "Extended price does not match sum of unit prices"
        'set the focus back to the Eprice field
        Order(-1).Eprice.SetFocus
    Else
        'accept value and proceed
        Order(-1).Eprice.Status = 0
    End If
End Sub
```

NOTE: Because of the (-1) row index, this subroutine works properly regardless of which row has the focus.

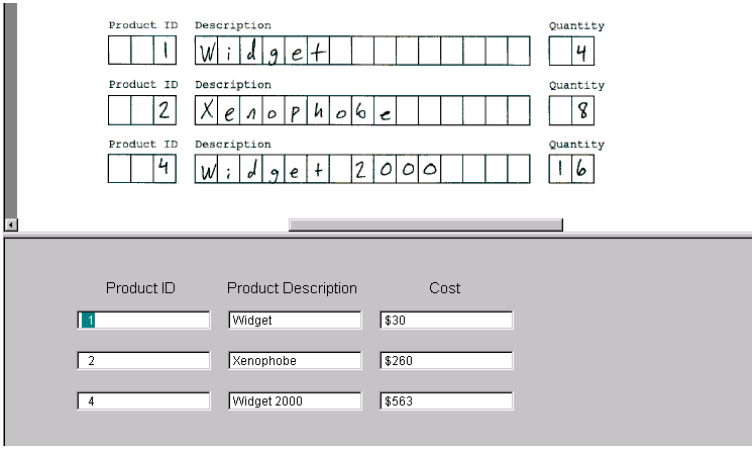
Automatic Field Lookups in SKFI Database Groups

Sometimes it is beneficial to combine a BasicScript call with a SKFI database group. For example, you may have an order form that has the standard order form fields (Item Number, Item Description, Quantity, Unit Price and Total Price.). Using a Form script, you can combine these data entry fields with a SKFI zone that looks up the Price and Description based on the Item Number.

Keep in mind that you would have to create two sets of fields in order to do this: one set that is set up for character recognition and one set that is located inside the SKFI zone. The SKFI zone would then have to be linked (via script) to the former set of fields so that the key SKFI field (Item Number) would automatically be filled in based on the recognized Item Number value.



Instead of having to visit each Item Number field in order to update the corresponding information, TELE*form* automatically updates the information whenever the Item Number is changed by your script. In effect, every time a script event is called, fields with database lookups will be updated to reflect the current values of key SKFI fields.



Automatic field lookups are enabled for the FieldGotFocus and FieldLostFocus entry points.

For more information on SKFI database groups, refer to your *User Guide*.

For information on field-specific Form script entry points, refer to your *BasicScript Guide*.

CHAPTER 8

Writing and Editing Your Scripts

About this Chapter

This chapter shows you how to write and execute scripts, including how to reference TELE*form* objects.

Writing Scripts

Opening your Script in the Edit Script window

Scripts are created and modified in the Edit Script window.

If you want to edit your script, open the Edit Script window in TELE*form Designer.* If you want to debug your script, open the Edit Script window in the TELE*form* applications that execute your script. (See "Executing and Debugging Your Scripts" on page 9-135 for more information on debugging your scripts).

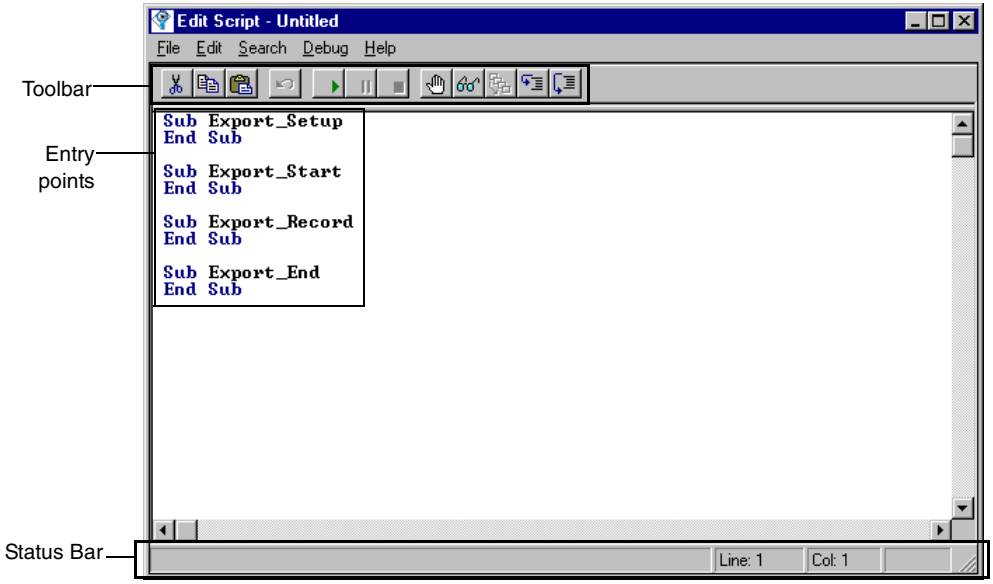
NOTE: Scripts are read-only when they are opened in TELE*form* Print Manager, *Reader* and *Verifier.* If you make edits in any of these applications, they cannot be saved to your script.

To open a specific type of script, refer to that script's chapter:

- For more information on opening a Form script, refer to "Overview of Form Scripts" on page 21.
- For more information on opening an Export script, refer to "Opening an Export Script for Script Writing" on page 58.
- For more information on opening your System script, refer to "Opening a System Script for Script Writing" on page 73.
- For more information on opening another type of script, refer to "Opening a Custom, Periodic, or Library Script for Script Writing" on page 94.

Overview of the Edit Script Window

When you open the Edit Script window in TELE*form Designer*, Print Manager, *Reader* or *Verifier*, the following is displayed:



Edit Script Window Toolbar

The following list briefly explains the purpose of each of the tools on the Edit Script toolbar. These tools will be explained in more detail in the following sections. For the buttons in the toolbar that relate to debugging, see"Debugging Your Scripts" on page 136.

Button	Tool	Function
	Cut	Cuts the selected text from the Edit Script window and places it in the Clipboard.
	Copy	Copies the selected text to the Clipboard.
	Paste	Pastes the contents of the Clipboard to the script editor.
	Undo	Reverses the most recent edit.

Edit Script Window Status Bar

The status bar of the edit script window displays the following

- The compile status when you click **Compile** on the **File** menu
- The line number and column number of your insertion point
- The edit status of your script - Modified appears in the right corner

Editing Your Script

This section explains how to edit BasicScript code in the Edit Script window. You'll learn how to move around within your script, select and edit text, add comments to your script, break long BasicScript statements across multiple lines, search for and replace selected text, and perform a syntax check of your script.

Navigating within a Script

The navigating keyboard shortcuts listed below allow you to move the insertion point to any location in your script

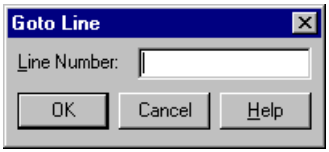
Key(s)	Function
UP ARROW	Moves the insertion point up one line.
DOWN ARROW	Moves the insertion point down one line.
LEFT ARROW	Moves the insertion point left by one character position.
RIGHT ARROW	Moves the insertion point right by one character position.
PAGE UP	Moves the insertion point up one page.
PAGE DOWN	Moves the insertion point down one page.
CTRL + PAGE UP	Scrolls the insertion point x columns to the left.
CTRL + PAGE DOWN	Scrolls the insertion point x columns to the right.
CTRL + LEFT ARROW	Moves the insertion point to the start of the next word to the left.
CTRL + RIGHT ARROW	Moves the insertion point to the start of the next word to the right.
HOME	Places the insertion point before the first character in the current line.
END	Places the insertion point after the last character in the current line.
CTRL + HOME	Places the insertion point before the first character in the script.
CTRL + END	Places the insertion point after the last character in the script.

You can also reposition the insertion point with the mouse or the **Goto Line** command.

To move the insertion point to a specific line

1. In the Edit Script window, press F4.

   The Goto Line dialog box appears.



2. Enter the line number in your script that you want to move the insertion point to, and then click **OK**.

   The insertion point is positioned at the start of this line.

Edit Procedures

The editing keyboard shortcuts are listed below:

Key(s)	Function
DELETE	Deletes the selected text or removes the character following the insertion point.
BACKSPACE	Deletes the selected text or removes the character preceding the insertion point.
CTRL+Y	Deletes the entire line containing the insertion point.
TAB	Inserts a tab character.
ENTER	Inserts a new line, ending the current line.
CTRL + C	Copies the selected text and places it on the Clipboard.
CTRL + X	Removes the selected text from the script and places it on the Clipboard.
CTRL + V	Inserts the contents of the Clipboard at the insertion point.
SHIFT + any navigating shortcut	Selects the text between the initial location of the insertion point and the point to which the keyboard shortcut would normally move the insertion point. (For example, pressing SHIFT + DOWN ARROW selects the current line and the line below it; pressing SHIFT + CTRL + LEFT ARROW selects the word to the left of the insertion point; pressing SHIFT + CTRL + HOME selects all the text from the location of the insertion point to the start of your script.)
CTRL + Z	Reverses the most recent edit change.

The following sections provide more detailed instructions on the editing operations you can perform in the Edit Script window.

Inserting Text

In the script editor, inserting text and other characters such as tabs and line breaks works about the same way as it does in a word-processing program: you position the insertion point at the desired location in the script and start typing.

Pressing ENTER

In the script editor, text does not wrap. If you keep entering text on a given line, eventually you will reach a point at which you can enter no more text.

Press ENTER when you want to insert a new line in your script. The effect of pressing ENTER depends on where the insertion point is located:

- If you press ENTER with the insertion point at or beyond the end of a line, a new line is inserted after the current line.
- If you press ENTER with the insertion point at the start of a line, a new line is inserted before the current line.
- If you press ENTER with the insertion point within a line, the current line is broken into two lines at that location.

Pressing TAB

If you press TAB , a tab character is inserted at the insertion point. Any text after the tab moves to the next tab stop.

Adding TELE*form* References

When you are editing Form scripts, Export scripts, and the System script, the Edit Script window has a right-click feature that simplifies the task of adding TELE*form* field, object class, and property references to your script.

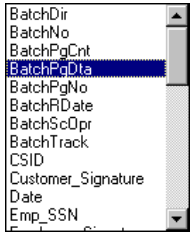
NOTE: In Export scripts, the only available classes are export level classes.

In the System script, the only available class is the Batch class.

To add a TELE*form* field reference to a Form script

1. Move the pointer to the desired location in your script.
2. Click the right mouse button.

   A pop up list of all the top-level fields on the form appears.



3. Double-click the field.

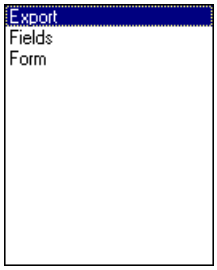
   The field name is inserted in your script.

NOTE: The top-level field list also contains form level classes.

To add a TELE*form* object class reference to an Export or System script

1. Move the pointer to the desired location in your script.
2. Click the right mouse button.

   A pop-up list of the available classes for this script appears.



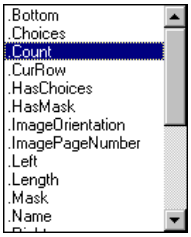
3. Double-click the desired object class.

   The object class name is inserted in your script.

To add a property reference

1. Move the insertion point to a TELE*form* field (or object class) on your script
2. Click the right mouse button.

   A pop-up list of available properties for this field type (or object class) appears.



3. Double-click on a property.

   The field (or class) property is inserted in your script using the proper syntax.

Selecting Text

You can select either a portion of one script line or a series of whole script lines. When you select multiple lines, the Edit Script window automatically extends the selection to include each line in its entirety.

Once you have selected text within your script, you can perform a variety of other editing operations on it, including deleting the text, placing it on the Clipboard and pasting it.

To select a portion of one line with the mouse

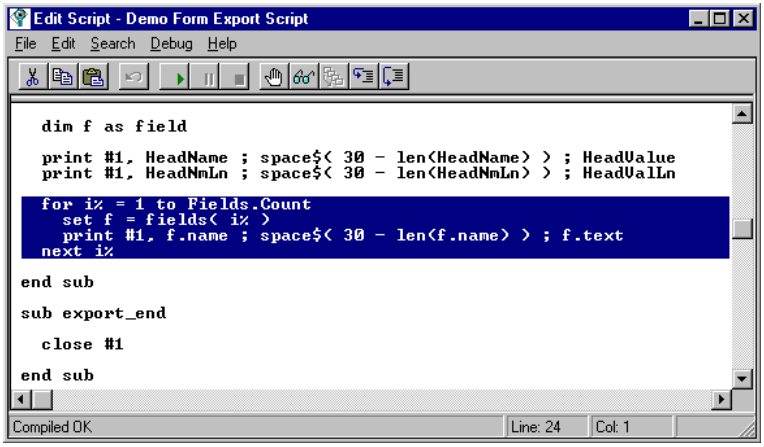
1. Point to where you want your selection to begin.
2. Drag to the end of your selection.

   The selected text is highlighted.

To select multiple lines with the mouse

1. Point to the left margin of the first line you want to select.
2. Drag up or down to select multiple lines.

   The selected lines are highlighted.



To select text with the keyboard:

1. Place the insertion point where you want your selection to begin.
2. Press SHIFT + one of the navigating keyboard shortcuts (see the preceding table) to extend the selection to the desired ending point.

   The selected text is highlighted.

To select an entire line

NOTE: When you intend to select an entire single line of text in your script, it is important to remember to extend your selection far enough to include the hidden end-of-line character, which is the character that inserts a new line in your script.

1. Place the insertion point at the beginning of the line.
2. Press SHIFT + END to select both the text and any hidden spaces that may be present at the end of the line.
3. Press SHIFT + RIGHT ARROW to select the hidden end-of-line character.

Deleting Text

When you delete text, it is removed from your script. If you accidentally delete text, click **Undo** on the **Edit** menu to restore it.

- To delete a single character to the left of the insertion point, press BACKSPACE once;
- To delete a single character to the right of the insertion point, press DELETE once.
- To delete selected text , press BACKSPACE or DELETE.
- To delete an entire line, place the insertion point in this line and press CTRL + Y.

Undoing Edits

You can undo editing operations that produce a change in your script, including:

- Typing text.
- Pasting text.
- Cutting or deleting text

You cannot undo operations that produce no changes in your script, such as moving the insertion point, selecting text, or copying material to the Clipboard.

To undo an edit

- Press CTRL + Z.

  The effect of the preceding editing operation is reversed. You may click this again to undo more editing operations.

Using the Clipboard

You can place text from your script on the Clipboard by either cutting it or copying it. You can then paste this text to another part of your script, or another application.

To cut text

1. Select the text you want to cut.
2. Press CTRL + X.

   The selected text is removed from your script and placed on the Clipboard.

To copy text

1. Select the text you want to copy.
2. Press CTRL + C.

   The selected text remains in your script, and a copy of it is placed on the Clipboard.

To paste text

1. Place the insertion point where you want to paste the text.
2. Press CTRL + V.

   The text is inserted.

To replace text on the script with text on the Clipboard

1. Select the text you want to replace.
2. Press CTRL + V

   The selected text is replaced with the Clipboard text.

Searching for and Replacing Text

The Edit Script window makes it easy to search for text in your script and automatically replace this text with other text.

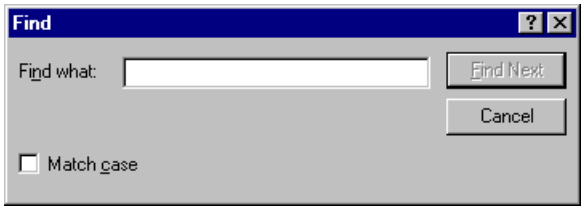
Finding Text in Scripts

1. Place the insertion point where you want to start your search.

   (To start at the beginning of your script, press CTRL + HOME.)

2. Click **Find** on the **Search** menu.

   The Find dialog box appears.



3. Type the text you want to search for in the **Find what** box.
4. Click the **Match case** check box if you want the search to be case-sensitive.
5. Click **Find Next**.

   The Find dialog box remains displayed, and the Edit Script window searches for this text.

   - If it finds the text, it highlights the text in the script.
   - If does not find the text, it displays a message telling you so.

6. To search for other occurrences of the text, click **Find Next** again.

NOTE: If the Find dialog box obstructs your view of the specified text, you can drag it out of your way and continue with your search, or you can close it and press F3 to find the next occurrence of this text.

Replacing Text in Scripts

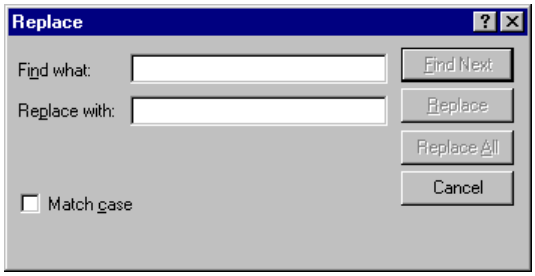
The script editor lets you automatically replace either all instances or selected instances of text.

1. Place the insertion point where you want to start your search and replace.

   (To start at the beginning of your script, press CTRL + HOME.)

2. Click **Replace** on the **Search** menu.

   The Replace dialog box appears.



3. Type the text you want to search for in the **Find what** box.
4. Type the text you want to replace the **Find what** text with in the **Replace with** box.
5. Click the **Match case** check box if you want the search to be case sensitive.
6. Click one of the following buttons:
   - To replace all instances of the search text, click **Replace All**.

     If no instances of the search text are found, a message appears.
   - To replace certain instances of the specified text, click **Find Next**, and follow the rest of the procedure.
7. If the specified text has been found, either click **Replace** to replace that instance of it or click **Find Next** to leave that instance in your script and highlight the next instance.

   Each time you click **Replace**, the highlighted text is replaced and you proceed to the next instance of the search text.

Adding Comments to Your Script

You can add comments to your script to remind yourself or others of the reasoning behind your code. Comments are ignored when your script is executed. In BasicScript, the apostrophe symbol (’) is used to indicate that the text from the apostrophe to the end of the line is a comment.

To add a full-line comment

1. Type an apostrophe (’) at the start of the line.
2. Type your comment.

   When your script is run, BasicScript will ignore this line.

To add comments to the end of a line of code

1. Place the insertion point in the empty space beyond the end of the line of code.
2. Type an apostrophe (’).
3. Type your comment.

   When your script is run, BasicScript will ignore all text to the right of the apostrophe.

Notes on using comments:

- If a comment uses a carriage return to force it onto another line, there must also be an apostrophe at the beginning of that line.

  For example:
  'This is a valid comment line
  'and so is this, but
  this line needs an apostrophe to be a comment

- Although you can place a comment at the end of a line containing executable code, you cannot place executable code at the end of a line containing a comment because the presence of the apostrophe at the start of the comment will cause the whole the line (including the code) to be ignored.

Right	**If Age.Value < 18 'check the age field**
Wrong	**'check the age field If Age.Value < 18**

Extending a BasicScript Statement into Multiple Lines

By default, a single BasicScript statement can extend only as far as the right margin; each line break represents a new statement. However, you can override this default if you want to extend a long statement into two or more lines.

To extend a BasicScript statement into multiple lines

1. Type the BasicScript statement on multiple lines, exactly the way you want it to appear.
2. Place the insertion point at the end of the first line in the statement.
3. Press the SPACEBAR once to insert a single space.
4. Type an underscore (_).

   The underscore is the line-continuation character, which indicates that the BasicScript statement continues on the following line.
5. Repeat steps 2-4 to place the underscore at the end of each line in the statement except the last line.

   When you run your script, the code on this series of lines will be executed as a single BasicScript statement.

Creating Dialog Boxes

Inserting a new dialog box into your script

1. Place the insertion point in the entry point where you want the dialog box to appear.
2. In the Edit Script window, click **New Dialog** on the **Edit** menu.

   The Dialog Editor window appears, displaying the default dialog box (which contains an **OK** and **Cancel** button).
3. Create your dialog box using commands on the Dialog Editor window and then save it.
4. Click **Exit and Return** on the **File** menu.

   You return to the Edit Script window. The dialog box code is inserted into your script.

Editing existing dialog boxes in a script

1. Select all the lines in your script from **Begin Dialog** to **End Dialog** (make sure to include **Begin Dialog** and **End Dialog** in your selection).
2. Click **Edit Dialog** on the **Edit** menu.

   The Dialog Editor window appears.
3. Edit your dialog box and then save it.
4. Click **Exit and Return** on the **File** menu.

- You return to the Edit Script window. Your dialog box code will reflect the changes you made to the dialog box.

NOTE: Refer to "Creating Custom Dialog Boxes" on page 149 for more information on creating and editing a custom dialog box to place in your script.

Compiling Your Script (Checking the Syntax)

Before executing a script, you must compile and save it. Compiling checks the syntax of the script, making sure that BasicScript commands are properly used.

To compile a script

1. In TELE*form Designer*'s Edit Script window, click **Compile** on the **File** menu.
   - If the script compiles successfully, the status bar displays **Compiled OK**.
   - If the script does not compile correctly, an error message appears, displaying the first line in your script where an error has been found and briefly describing the nature of that error.
2. If an error message is displayed, write down the error. Click **OK**.

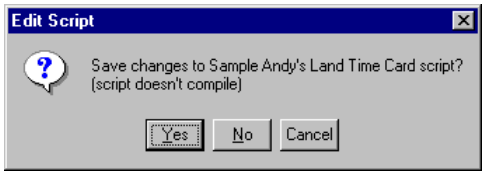
   If there is a syntax error, the line containing the error is highlighted on your display.
3. Correct the event that is causing the syntax error.
4. Repeat steps 1-3 until you find and correct all syntax errors.
5. Save the corrected script.

IMPORTANT: Always compile your script after any changes to the Form.

Exiting the Edit Script window

To exit the Edit Script window

1. Click **Close** on the **File** menu.
2. If your script compiles OK, and you made any unsaved changes to your script, a message appears asking whether you want to save the script.
   - Click **No** to close the Edit Script window without saving your changes.
   - Click **Yes** to save your changes. The Edit Script window closes after compiling and saving your script.
3. If your script does not compile OK, you will receive the following message

   Do one of the following:

   - Click **Yes** to save the script, including the errors, and close the Edit Script window.

   NOTE: Your script cannot be executed until you fix the errors.

   - Click **No** to close the Edit Script window without saving your changes.
   - Click **Cancel** to cancel the **Close** command.
4. If you clicked **Cancel** in step **3**, click **Compile** on the **File** menu to see the first line that is causing the compile-time error.

   Refer to the preceding section for more information on the Compile procedure.

CHAPTER 9

Executing and Debugging Your Scripts

About this Chapter

This chapter explains the fundamentals of executing and debugging your scripts. The debugging process includes identifying procedure calls, setting breakpoints, controlling which lines in your script are traced, and monitoring selected variables in your script.

Executing Your Scripts

If your script has a problem when you execute it, an error message will appear on the screen.

To fix your script

1. Open the Edit Script window in TELE*form Designer*.
2. Fix the problem.
3. Save and compile the script.
4. Test the script again.

The procedure above is known as debugging a script (which is explained in more detail in the following sections).

For more information on executing your script, refer to the chapter that explains your script type:

- For more information on executing your Form scripts, refer to “Writing Scripts” on page 117.
- For more information on executing your Export scripts, refer to “Executing Your Export Scripts” on page 67.
- For more information on executing your System script, refer to “Executing Your System Script” on page 83.
- For more information on executing your other scripts, refer to “Executing Your Custom, Periodic and Library Scripts” on page 98.

Debugging Your Scripts

The Edit Script window contains some powerful debugging tools to help you troubleshoot your scripts. These tools are available when you are operating in debug mode. They will help you track variables and locate errors in your script.

When the debugger is in use, the Edit Script window appears on top of all other applications so the various debugging tools can be accessed.

This section presents some general information that will help you debug your script. It also explains how to trace the execution of your script, how to set and remove breakpoints, and how to add watch variables and modify their values.

Debugging Toolbar

The following table lists the buttons on the toolbar that relate to executing and debugging your script:

Button	Function	Description
	Start	Continues execution of a script after the debugger stops on a breakpoint. Remember that scripts cannot be started in the Edit Script window. To start a script, you must run the TELE*form* application that calls that script's subroutine (see "Executing Your Scripts" on page 135 for more information on starting your script.)
	Pause	Pauses execution of a script.
	End	Stops execution of a script.
	Toggle Breakpoint	Adds or removes a breakpoint on a line of BasicScript code.
	Add Watch	Displays the Add Watch dialog box, where you can add the name of a BasicScript variable. BasicScript will display the value of the specified variable in the watch pane of the Edit Script window (above the code).
	Calls	Displays the list of procedures called by the currently executing BasicScript script. Available only during break mode.
	Single Step	Executes the next line of a script and then suspends execution. If the script calls another BasicScript procedure, execution will continue into each line of that procedure.
	Procedure Step	Executes the next line of a script and then suspends execution. If the script calls another BasicScript procedure, it will run the procedure in its entirety, but will not step through each line.

Debugging Keyboard Shortcuts

Key(s)	Function
SHIFT + F9	Chooses the Add Watch tool.
DELETE	Removes the selected watched variable from the Watch pane.
F6	If the watch pane is open, switches the insertion point between the watch pane and the code pane.
F8	Chooses the Single Step tool. (See the Toolbar section above).
SHIFT+F8	Chooses the Procedure Step tool. (See the Toolbar section above).
CTRL + BREAK	Suspends execution of an executing script and places the instruction pointer on the next line to be executed.
F9	Sets or removes a breakpoint on the line with the insertion point.
F5	Chooses the Start tool (see “Debugging Toolbar” on page 137).

Starting Debug Mode

To start debug mode for a particular script (or a portion of the script), that script must be opened in the TELE*form* application that calls the script. For example, if your Form script is called at the **Sub Form_Load** entry point, open your Form script in the Edit Script window in TELE*form Verifier,* and then correct one of these forms.

You can debug a Form script and another script at the same time. For example, you can debug your Form script and your Export script simultaneously. After Form_Export is called in your Form script, Export_Start is called in your Export script.

Starting Debug Mode for Form Scripts

In TELE*form Reader* or *Verifier*:

1. On the **Utilities** menu, point to **Debug Script** , and then click **Form**.

   The Select Form dialog box appears.

2. Click the form that you want to debug, and then click **OK**.

   The Edit Script window appears, displaying the form script.

3. If you want, minimize the Edit Script window.

4. Execute your Form script using the procedure in "Executing Your Form Scripts" on page 48.

   TELE*form* will put the Edit Script window into the foreground when the first line of your script is executed.

5. When you are done with the debugging process, point to **Debug Script** (on the **Utilities** menu), and then click **Form** to clear the check mark.

   Clearing this check mark will stop the debug process for your form script.

Starting Debug Mode for Other Scripts

In TELE*form Print Manager, Reader or Verifier*

1. On the **Utilities** menu, point to **Debug Script** , and then click **Other**.

2. Click **Export Scripts** on the **Utilities** menu.

   The Edit Script window appears.

3. Click **Open** on the **File** menu.

   The Open Script dialog box appears.

4. Click your script, and then click **OK**.

   Your script is displayed in the Edit Script window.

5. If you want, minimize the Edit Script window. Execute your script using the appropriate execution procedure (see "Executing Your Scripts" on page 135 for a reference to your script type's execution procedure).

   TELE*form* will put the Edit Script window into the foreground when the first line of your script is executed.

In TELE*form Reader* or *Verifier*

1. Point to **Debug Script** on the **Utilities** menu, and then click **Other**.

   The Edit Script window appears.

2. Click **Open** on the **File** menu.

   The Open Script dialog box appears.

3. Click your script, and then click **OK**.

   Your script is displayed in the Edit Script window.

4. If you want, minimize the Edit Script window.

5. Execute your script using the appropriate execution procedure (see "Executing Your Scripts" on page 135 for a reference to your script type's execution procedure).

   TELE*form* will put the Edit Script window into the foreground when the first line of your script is executed.

6. When you are done with the debugging process, point to **Debug Script** (on the **Utilities** menu), and then click **Other** to clear the check mark.

   Clearing this check mark will stop the debug process for your script.

IMPORTANT: The Edit Script window must be opened in the application that the script is executing in to debug it. Opening the script in TELE*form Designer* will not debug a script called from *Reader* or *Verifier*.

Using the BasicScript Debugger

While debugging, you are actually executing the code in your script line by line. Therefore, to prevent any changes to your script while it is being run, the Edit Script window is read-only during the debugging process. You can move the insertion point throughout the script, select text and copy it to the Clipboard as necessary, set breakpoints, and add and remove watch variables, but you cannot make any changes to the script until you stop running it.

NOTE: You can only make changes to your script in the Edit Script window of TELE*form Designer.* Do not attempt to make changes in any other application, because you will not be able to save these changes to your script.

NOTE: The following procedures assume that you have already successfully compiled and saved your script in TELE*form Designer*, that you have opened the appropriate TELE*form* application and your script's debugger, and that you are executing your script in this TELE*form* application.

IMPORTANT: Always compile your script after any changes to the Form.

Instruction pointer

To let you follow and control the debugging process, the Edit Script window displays an instruction pointer on the next line of code to be executed. When the instruction pointer is on a line of code, the text appears in black against a gray background.

Tracing Script Execution

The Edit Script window gives you two ways to trace script execution: single step and procedure step. Both involve moving through your code line by line.

- The single step process traces into every line in your script.
- The procedure step process does not trace into the individual lines of a procedure.

To step through a script

NOTE: Make sure that the Edit Script window is started in the TELE*form* application that will execute your script, and make sure that your script is opened in this window (for Debug mode).

1. Initiate the action in the TELE*form* application that will execute your script.

   BasicScript will transfer the Edit Script window to the foreground and place the instruction pointer on the first line in your script's code (most likely a subroutine).

2. To trace the execution of your script line by line:

   - Press F8 to move to the next line in the single step process.
   - Press SHIFT + F8 to move to the next line in the procedure step process.

   The Edit Script window executes the line containing the instruction pointer and moves the instruction pointer to the next line.

3. Repeat step **2** for each line that you want to debug.
4. When you finish tracing the execution of your script, do one of the following:

   - Click  on the toolbar to run the rest of the script at full speed.
   - Click  on the toolbar to halt execution of the script.

NOTE: If your script contains any compile errors, it cannot be executed.

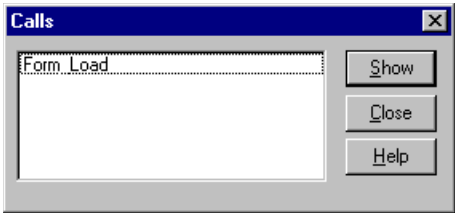
To identify the procedure calls in a subroutine

When stepping through a subroutine, you can display the Calls dialog box to help you quickly identify the subroutine calls that brought you to this point in the script.

1. Click  on the toolbar.

   The **Calls** dialog box appears when a subroutine call occurs.

   For example, when an image is opened for correction in TELE*form Verifier*, the following **Calls** dialog box appears.



2. Click the subroutine you want to view, and then click **Show**.

   The Edit Script window highlights the line in the subroutine you selected which brought you to the current point in the script.

NOTE: During this process, the instruction pointer remains in its original location in the subroutine. Therefore, the call point is highlighted and the current line contains the instruction pointer.

To move the instruction pointer to another line

When you are stepping through a subroutine, use the **Set Next Statement** command to move the instruction pointer to another line within that subroutine. This command is useful if you want to repeat or skip a part of your code.

1. Place the insertion point in the line where you want to resume stepping through the script.
2. Click **Set Next Statement** on the **Debug** menu.

   The instruction pointer moves to the line you selected

3. Resume stepping through your script.

NOTE: You can only use the **Set Next Statement** command to move the instruction pointer within the same subroutine. If you place the insertion point on a line outside this subroutine, the **Set Next Statement** command will be unavailable.

Debugging one or more parts of a long script

If you want to debug certain parts of a long script, set one or more breakpoints at selected lines in your script. The Edit Script window suspends execution of your script when it reaches a line containing a breakpoint. Suspending execution allows you to begin or resume stepping through the script from that line.

Valid breakpoints can only be set on lines in your script that contain code, including lines in functions and subroutines. When you compile and run the script, invalid breakpoints (breakpoints on lines that don't contain code) are automatically removed. While you are debugging your script, the Edit Script window will beep if you try to set a breakpoint on a line that does not contain code.

You can set breakpoints to begin the debugging process partway through your script, to continue debugging at a line outside the current subroutine, and to debug only selected portions of your script.

Debugging partway through a script

1. Place the insertion point in the line where you want to start debugging.
2. Click on the toolbar to set a breakpoint on that line

   The line on which you set the breakpoint now appears in a contrasting color.
3. Initiate the action in the appropriate TELE*form* application that will execute your script.

   The Edit Script window runs your script at full speed from the beginning and then places the instruction pointer on the breakpoint line to designate it as the line that will be executed next.
4. Either start debugging or resume running the script.
5. If you want to continue debugging at another line within this subroutine, skipping all lines in between that line and the current line, use the **Set Next Statement** command (discussed in the preceding section).
6. If you want to continue debugging at a line in another subroutine, set a

   breakpoint in the line where you want to continue debugging, and click on the toolbar .

Debugging selected portions of a script

1. Set a breakpoint at the start of each script section that you want to debug. (see above)

NOTE: Up to 255 lines in your script can contain breakpoints.

2. Initiate the action in the appropriate TELE*form* application that will execute your script.

   The script executes at full speed until it reaches the line containing the first breakpoint and then pauses with the instruction pointer on that line.

3. Debug the script section.

4. Click  on the toolbar to move to the next breakpoint.

   Every time you click this button, you will move to the next breakpoint that you set in step **1**.

Removing Breakpoints

Breakpoints can be removed either manually or automatically.

1. Place the insertion point on the line containing the breakpoint that you want to remove.

2. Click  on the toolbar.

   The breakpoint is removed, and the line no longer appears in a contrasting color

3. If you want remove all breakpoints, click **Clear All Breakpoints** on the **Debug** menu.

NOTE: When you exit the Edit Script window, all breakpoints are cleared.

Monitoring Selected Variables

As you debug your script, you can use the Watch pane to monitor selected variables. For each of the variables in the watch variable list, the Edit Script window displays the name of the variable, where it is defined, its value (if the variable is not in scope, its value is shown as **variable not defined in context**), and other key information such as its type and length (if it is a string). The values of the variables on the watch list are updated each time you enter debug mode, and each time you execute a line of code.

To add a watch variable to your script

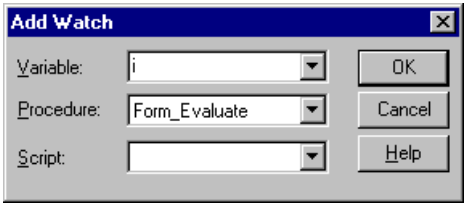
NOTE: The BasicScript debugger cannot recognize variables that are declared outside the scope of a subroutine, unless these variables are declared as public variables (see Chapter 5 for more information on public variables).

1. Initiate the action in the appropriate TELE*form* application that will execute your script.

   When your script is executed, the Edit Script window will appear in the foreground.

2. Click  on the toolbar.

   The **Add Watch** dialog box appears.



3. In the **Procedure** list, select the subroutine that contains the variable you want to add.
4. In the **Variable** list, select the variable that you want to add to the watch variable list.
5. Click **OK**.

   The Watch pane expands far enough to display the variable you just added

6. Single step through your procedure.

   The Watch pane displays the current value of the variable. This value will change whenever the variable's value is re-assigned in the script.

NOTE: Although you can add as many watch variables to the list as you want, the watch pane expands to fill at most half of the Edit Script window.

The list of watch variables is maintained between script executions.

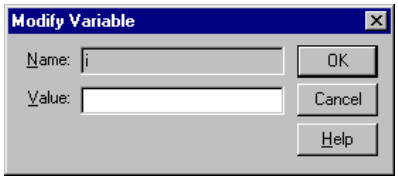
To delete a watch variable

1. Select the variable on the watch pane and press DELETE.

To modify the value of a watch variable

1. Initiate the action in the appropriate TELE*form* application that will execute your script. When your script is executed, the Edit Script window will appear in the foreground.
2. Press F8 until the instruction pointer highlights the variable you want to modify.
3. Click **Modify** on the **Debug** menu.

   The Modify Variable dialog box appears.



   If the instruction pointer is highlighting a variable, this dialog box will be pre-filled with the variable name.
4. If it is not already entered, enter the variable name in the **Name** box.
5. Enter the new value for this variable in the **Value** box, and then click **OK**.

   When you continue execution of your script, the new value of your variable is displayed in the Watch pane. Your variable will start out with this value.

Debugging Script in the Form_Check and Export Entry Points

Form_Check and Export entry points (including Form_Export in your Form scripts and all entry points in your Export scripts) are always run as background processes in TELE*form Verifier* and cannot be debugged using the conventional debug procedures described in this Chapter.

To debug one of these entry points in TELE*form Verifier*, or to debug Print-Init and Print-Exit use one or more of the following methods:

- Write to a text file and view the file with Windows Notepad.
- Display variables using a message box.

With Export scripts, you can open the Edit Script window in TELE*form Reader*, and then use the **Manual Data Export** command on the **Utilities** menu of TELE*form Reader*.

CHAPTER 10

Creating Custom Dialog Boxes

About this Chapter

This chapter shows you how to use the Dialog Editor. You will also learn how to troubleshoot (debug) your dialog box, how to insert your dialog box into your script, and what additional script you will need to enter in order to make your dialog box functional.

What You Can Use Custom Dialog Boxes for

Sometimes your script will need to obtain information from the user. In many cases, you can obtain this information by using one of BasicScript's predefined dialog boxes in your script (see "Predefined Dialogs" on page 260 for more information on the predefined dialog box language elements). When you must go beyond the information-gathering capabilities provided by predefined dialog boxes, you can use the Dialog Editor to create a custom dialog box for your script.

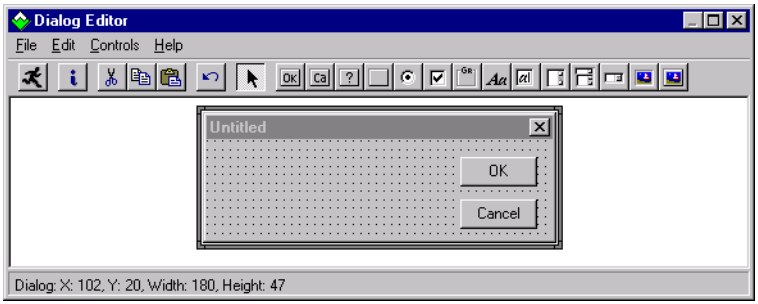
Overview of the Dialog Editor

With the Dialog Editor, you can create and modify custom dialog boxes for use in your BasicScript scripts. The Dialog Editor makes it easy to generate BasicScript statements needed for your custom dialog boxes. These BasicScript statements include code for the dialog box display and the dialog box functions (for example, selecting a check box).

The Dialog Editor is a tool that allows you to generate a dialog box structure in BasicScript simply by editing an on-screen dialog box display. When you are done editing this display, BasicScript will insert the code for this dialog box into your script

Dialog Editor Window

In the Edit Script window, when you click **New Dialog** or **Edit Dialog** on the **Edit** menu, the following window appears.



Toolbar

The toolbar is a collection of shortcut buttons. The most common Dialog Editor commands can be accessed by clicking these buttons. Each toolbar button corresponds to a menu option on the menu bar.

Button	Function
	Runs the dialog box, which makes it functional for testing purposes
	Displays the Information dialog box for the selected dialog box or control
	Cuts the selected control or dialog box and places it on the Clipboard
	Copies the selected control or dialog box to the Clipboard
	Inserts the Clipboard contents into dialog editor (for more information on this function, see the Keyboard Shortcuts section below).
	Reverses the effect of the preceding editing change.
	Lets you select items and position the insertion point.

The following buttons add a control to your dialog box

Button	Function
	Adds an OK button to your dialog box.
	Adds a Cancel button to your dialog box.
	Adds a Help button to your dialog box.
	Adds a push button to your dialog box.
	Adds an option (radio) button to your dialog box.
	Adds a check box to your dialog box.
	Adds a group box to your dialog box.
	Adds text to your dialog box.
	Adds a text box to your dialog box.
	Adds a list box to your dialog box.
	Adds a combo box to your dialog box.
	Adds a drop list box to your dialog box.
	Adds a picture to your dialog box.
	Adds a picture button to your dialog box.

Dialog Box Display

The dialog box display is the visual layout of the dialog box that you are currently creating or editing. You can think of this dialog box as a preview of the custom dialog box that will appear in TELE*form* when you execute your script.

By default, every new dialog box contains an **OK** button and a **Cancel** button.

Status bar

The status bar shows the following

- Name of the currently selected control or dialog box.
- Position of the pointer, or position and dimensions of the selected control or dialog box
- Name of the control you are adding to the dialog box.

NOTE: Dialog boxes created with Dialog Editor appear in Helvetica 8-point font, both in Dialog Editor and when the corresponding BasicScript code is run. If you want to change this font, refer to page 164.

Keyboard Shortcuts

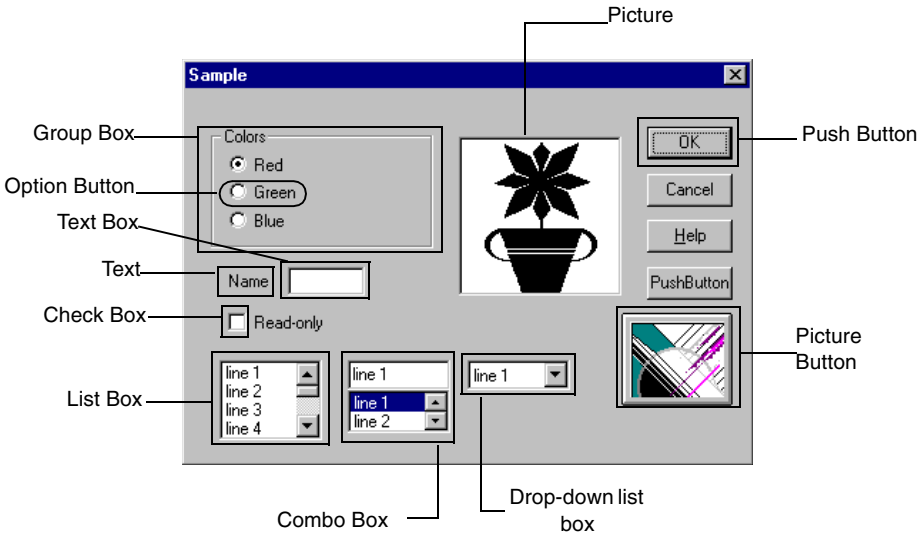
There are many keyboard shortcuts built into the dialog box editor to speed up common editing processes. The following table contains a complete list of keyboard shortcuts:

Key(s)	Function
ALT+F4	Closes Dialog Editor.
CTRL+C	Copies the selected dialog box or control and places it on the Clipboard.
CTRL+D	Creates a duplicate copy of the selected control.
CTRL+G	Displays the Grid dialog box.
CTRL+I	Displays the Information dialog box for the selected dialog box or control.
CTRL+V	Inserts the contents of the Clipboard into Dialog Editor. If the Clipboard contains BasicScript statements describing one or more controls, then those controls are added to the current dialog box. If the Clipboard contains BasicScript statements for an entire dialog box, then Dialog Editor creates a new dialog box from these statements.
CTRL+X	Removes the selected dialog box or control from Dialog Editor and places it on the Clipboard.
CTRL+Z	Undoes the preceding operation.
DELETE	Removes the selected dialog box or control from Dialog Editor.
F1	Displays the Help system contents.
F2	Runs the dialog box, which makes it functional for testing purposes.
F3	Resizes the controls to fit their label text.
F4	Selects the entire dialog box.
F10	Toggles menu bar activation.

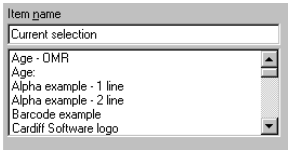
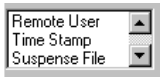
Creating a Custom Dialog Box

This section describes the types of controls that Dialog Editor supports. It also explains how to create controls and position them within your dialog box.

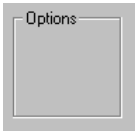
Control and Design Elements



Control elements allow the user to communicate and interact with the dialog box (and therefore TELE*form)*. The following table provides a description of each dialog box control:

Control Element	Example	Description
Check box		Square box that users select to turn on an option and clear to turn off an option.
Combo box		Combination of a text box and a list box. Users can either select an item from the list or type text in the text box. If the user selects an item from the list, it is highlighted in the list and placed in the text box.
Push button		Rectangular button that initiates an action.
Drop-down list box		Type of list box where the list is displayed only when the user clicks on the drop-down arrow. Once they select an item from the list, the list disappears and the newly selected item is displayed in the box.
List box		Rectangular box containing a list of items, from which the user selects one item. The selected item is highlighted.
Option (radio) button		One of a group of mutually exclusive options. Users can only select one option per group box.
Picture button	**See Figure Above**	Type of command button where a Windows bitmap or metafile is the label.
Text box		Rectangular box that the user types text into. If there is default text, the user can delete this text and type new text. You can choose between the default setting, in which this field holds a single line of nonwrapping text, and the Multi-line setting, in which the field holds multiple lines of wrapping text.

Design elements organize the dialog box, inform the user about controls, and enhance dialog box design. The following table provides a description of each dialog box design element.

Design Element	Example	Description
Group box		Rectangular frame that encloses a set of related controls. You can use the group box label as a title for controls in the box.
Picture	**See Figure Above**	Windows bitmap (.bmp format) or metafile (.wmf format), which you can obtain from a file or a library.
Text		Text displayed to inform the user. The text in this field wraps, and the field can contain a maximum of 255 characters. There are two types of text: • Stand-alone text • Label text (defined as part of another element such as a group box)

Planning Your Dialog Box

Creating dialog box elements in random order might seem like the fastest approach. However, the order in which you create elements has important implications; a little planning can save you a lot of work.

Here are three features of dialog box creation that you should understand.

NOTE: You can fix problems in your dialog box when testing it. However, adding elements in the right order will save you time and trouble.

Tab order

Users can select dialog box controls by pressing TAB. As users press TAB, the focus is changed from one control to the next. The order in which you create controls (not their position on the dialog box) determines the tab order. See page 175 for more information on the tab order of your dialog box controls.

You should create controls in the tab order you want. The fewer tab-order adjustments you have to make, the less time you will spend.

Option buttons

If you want a series of option buttons to work together as a mutually exclusive group, you must create all of them at the same time. If you create a different type of control before you have finished creating all of the option buttons, you will split the option buttons into separate groups.

Accelerator keys

In addition to clicking on a control to focus on it, users can also have keyboard access to controls with accelerator keys.

To assign accelerator keys to controls without labels, create the text or group box, then create the control. If you do not create the control immediately after you create the text or group box, your accelerator key will not work.

Saving Your Dialog Box

To save your dialog box for use in this script

1. Click **Update** on the **File** menu.

   Your dialog box is converted into a series of BasicScript statements and placed at the insertion point in your script.

2. Click **Exit and Return** on the **File** menu.

   You return to the Edit Script window.

3. Click **Save** on the **File** menu of the Edit Script window to save your dialog box with your script.

To put your dialog box code in another location:

1. If it is not already selected, select the dialog box code (it starts with **Begin Dialog** and ends with **End Dialog**)
2. Click the **Cut** button on the toolbar.
3. Move your insertion point to the new location.
4. Click the **Paste** button on the toolbar.

To save your dialog box for use in another script

1. Click **Save As** on the **File** menu.

   The Save Dialog File dialog box appears.

2. On the **Save in** list, select the folder that you want to save your dialog box in.
3. In the **File name** box, type the name of your dialog box file.
4. Click **Save**.

   Your dialog box code is now saved to a file, and can be opened to include in other scripts.

Adding a Title to Your Dialog Box

The title of your dialog box is located on the title bar at the top of the dialog box. By default, the title of your dialog box is "Untitled".

To change your dialog box title

1. Double-click the title bar of your dialog box.
2. Type the new title in the **Text$** box.
3. If the value in the **Text$** box should be used as a variable name instead of a literal string, click the **Variable Name** check box.
4. Click **OK**.

   The new title is displayed on the title bar or on the control.

Using the Dialog Box Grid

The borders of your dialog box contain a dot grid. Displaying the grid and changing its X and Y spacing can help you position elements more precisely in your dialog box.

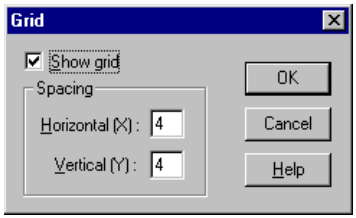
This grid includes the following features:

- The X (horizontal) axis and the Y (vertical) axis intersect in the upper left corner of the dialog box. This intersection point is (0,0).
- As you move the pointer down from the intersection point, the Y value increases.
- As you move the pointer to the right of the intersection point, the X value increases.
- Think of each grid dot as a specific location in your dialog box.

To change the grid spacing

1. Press CTRL + G.

   The Grid dialog box appears.



2. Type a number in the **Horizontal (X):** box to set the horizontal spacing of your grid dots.
   - A lower number allows more precise horizontal positioning.
   - A higher number allows less precise horizontal positioning.
3. Type a number in the **Vertical (Y):** box to set the vertical spacing of your grid dots.
   - A lower number allows more precise vertical positioning
   - A higher number allows less precise vertical positioning
4. Click **OK**.

   Dialog Editor displays the grid with the spacing you specified.

NOTE: Grid units represent increments of 8 point Helvetica font.

- Each X unit represents an increment equal to 1/4 of that font.
- Each Y unit represents an increment equal to 1/8 of that font.

Adding Elements to a Dialog Box

In this section, you'll learn how to add elements to your dialog box. The following points should be noted:

- A single dialog box can contain no more than 255 controls
- The dialog box must contain at least one push button.

To add an element to your dialog box

1. On the **Controls** menu, click the element that you want to add.

NOTE: You can only insert an element within the borders of the dialog box you are creating. You cannot insert an element on the title bar or outside dialog box borders.

2. If you are within the dialog box borders, the pointer becomes an image of the element.

   If you are outside the dialog box borders, the pointer becomes a circle with a line through it.

3. Place the element pointer at the desired location and click there.

   The upper left corner of the element is inserted at the coordinate you chose.

   The element you just added is surrounded by the selection frame.

4. To add this element to another part of your dialog box, press CTRL + D.

   The duplicate element is now selected.

Selecting Your Elements

In order to edit an element, you must first select it.

You can select an element in one of two ways:

- Click the **Select** button on the toolbar, and then click the element
- Click the **Select** button on the toolbar, and then press TAB repeatedly until the focus moves to the desired element.

When you select an item, a thick frame surrounds it. This frame is called the selection frame.

Selecting Your Dialog Box

Select your dialog box in one of two ways:

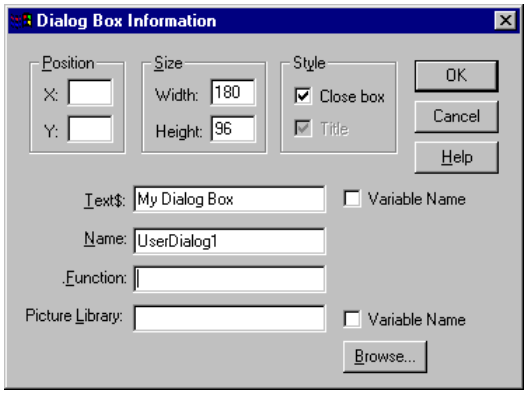
- Click the **Select** button on the toolbar, and then click on the title bar of your dialog box.
- Click the **Select** button on the toolbar, and then press TAB repeatedly until the focus moves to the dialog box.

Configuring Element and Dialog Box Attributes

The Information dialog box allows you to configure various attributes of elements and dialog boxes.

To open the Information dialog box

1. Click the element or dialog box to select it.
2. Double-click the item you selected.



The Information dialog box appears, displaying the name of the element in the title bar.

NOTE: Each element type contains a unique Information dialog box with specialized attributes. Some of these attributes must be specified, while others are strictly optional.

3. Enter the attributes you want for this element and then click **OK**.

NOTE: If the **OK** button in the Information dialog box is unavailable, then one or more required attributes is missing. Enter the missing attributes, or click **Cancel** to revert to the previously entered attributes.

Dialog Box Attributes

The following table lists each dialog box attribute and whether or not the attribute is required by Dialog Editor.

Attribute	Required (Yes/No)	Description
Position	No	X and Y coordinates on the display, in grid units (see page 159).
Size	Yes	Width and height of the dialog box, in grid units (see page 159).
Style	No	Options that determine whether the close button and title bar are displayed.
Text$	No	Text displayed on the title bar of the dialog box.
Name	Yes	Name of the dialog box. This will be referenced in your BasicScript code.
.Function	No	Name of a BasicScript function in your dialog box.
Picture Library	No	Picture library where you get pictures for your dialog box.

To change the font of your dialog boxes

If you want to change the font of the text in your dialog box (including the list box labels), use the following procedure:

1. Start Windows Notepad
2. Open your **Teleglob.ini** file. This file is located in your TELE*form* directory.
3. Locate the **[Script Editor]** section of the **Teleglob.ini** file, and enter the following line into this section:

   Dialog Font=FontName,PointSize,isBold,isItalics

where **isBold** and **isItalics** should contain the following values:

- Enter **1** if you want the font to be bold/italic.
- Enter **0** if you do not want the font to be bold/italic.

For example, enter the following line if you want Times New Roman, 9 pt and bold for your dialog box font:

Dialog Font=TimesNewRoman,9,1,0

4. Save your **Teleglob.ini** file.
5. Exit Windows Notepad.

Element Attributes

The following table lists each element attribute, whether or not the attribute is required by Dialog Editor, and which elements the attribute applies to.

Attribute	Required (Yes/No)	Applies to	Description
Position	Yes	All elements	X and Y coordinates within the dialog box, in grid units (see page 159).
Size	Yes	All elements	Width and height of the element, in grid units (see page 159).
Text$	No	Push button, option button check box, group box and text	Text displayed on or beside an element as a label.
FileName$	No	Help button	Name of the help file opened when the user clicks the help button.
Font	No	Text	Font in which text is displayed.
Multiline	No	Text box	Option that determines whether users can enter a single line of text or multiple lines of text.
.Identifier	No	Push button, option button, group box, text	Name of the element. This will be referenced in your BasicScript code.
	Yes	Check box, text box, list box, drop- down list box, combo box	Name of the element, and container of the control value after the dialog box has been processed.
	No	Picture, picture button	Name of the file containing a picture that you want to display or the name of a picture from a specific picture library.
Frame	No	Picture	Creates a 3 dimensional frame for your picture
Array$	Yes	List box, drop-down list box, and combo box	Name of an array variable in your BasicScript code.
.Option Group	Yes	Option button	Name given to a group of option buttons. This will be referred in your BasicScript code.

Adding/Changing Titles and Labels

By default, when you begin creating a dialog box, its title is **Untitled**. When you first create group boxes, option buttons, push buttons, text controls, and check boxes, they have generic-sounding default labels, such as **Group Box** and **Option Button**.

To change a dialog box title or a control label

1. Display the Information dialog box for the dialog box whose title you want to change or for the control whose label you want to change.
2. Enter the new title or label in the **Text$** box.

   Dialog box titles and control labels are optional. Therefore, you can leave the **Text$** box blank.
3. If the information in the **Text$** box should be interpreted as a variable name rather than a literal string, select the **Variable Name** check box.
4. Click **OK**.

   The new title or label is now displayed on the title bar or on the control.

Although OK and Cancel buttons also have labels, you cannot change them. The remaining controls (text boxes, list boxes, combo boxes, drop list boxes, and picture buttons) do not have their own labels, but you can position a text element above or beside these controls to serve as a label.

Moving and Sizing Elements

To move an element

1. Click the element in your dialog box with the Selection pointer.
2. Drag the element to its new location
3. If you need to move the element with precision, press an arrow key on your keyboard.

NOTE: You can only drag an element in increments of a grid dot. If you need to position the element with more precision, use the arrow keys, or open the Grid dialog box and change the grid spacing (see page 159).

To size an element (or dialog box)

1. Click the element or dialog box that you want to resize with the Selection pointer.
2. Point to the border or a corner of the selected item.
3. Drag the border or corner until the item expands or contracts to the desired size.

NOTE: Pictures in.wmf format always expand or contract proportionally to fit within the picture or picture button.

Pictures in.bmp format are of a fixed size. If you place a bitmap in an element that is smaller than the bitmap, part of the picture will be cut off. If you place a bitmap in an element that is larger than the bitmap, the picture is centered within the borders of the element.

Assigning Accelerator Keys to Your Controls

An accelerator key allows users to access a dialog box control simply by pressing ALT + a specified keyboard letter. This letter must appear in the control label, and is underlined in the label when it is assigned. For example, users can employ accelerator keys to select an option button, toggle a check box on or off, and move the insertion point into a text box.

An accelerator key can be linked to any control except OK and Cancel buttons, (because their labels cannot be edited). If the control does not have a label (for example, a combo box), you can create an associated text element and assign an accelerator key to the text.

To assign an accelerator key

1. Double-click the control in your dialog box.

   The Information dialog box appears.

2. In the **Text$** box, type an ampersand (&) before the desired letter.
3. Click **OK**.

   The accelerator letter is now underlined on the control label.

NOTE: Accelerator key assignments in a dialog box must be unique. If you attempt to assign the same accelerator key to more than one control, Dialog Editor displays a message that the letter has already been assigned.

Adding Pictures to Your Picture Elements

The picture element is an empty outline until you specify the picture that you want it to display. A picture element can display Windows bitmaps or Windows metafiles, which you can obtain from a file.

NOTE: If you use a picture library, all the pictures in your dialog box must come from the same library.

To add a picture from a file

1. Double-click the picture element.

   The Picture Information dialog box appears.

2. Under **Picture source**, click **File**.
3. Enter the full path of the picture file in the **Name$** box.
4. If you do not know the full path, click **Browse** to display the Select a Picture File dialog box, and then search for the file in your directory.

   When you select the file and click **OK**, the full path is pasted into the **$Name** box.

5. Click **OK**.

   The picture element now displays the picture you specified.

Creating and Modifying Picture Libraries

NOTE: Creating a picture library requires fundamental C programming skills. To add pictures to your picture elements, you can also use a picture file (see the preceding section)

A picture library is a DLL (dynamic link library) that contains a collection of pictures. Currently, both Windows bitmaps and Windows metafiles are supported.

Each picture is placed into the DLL as a resource with a unique identifier. This identifier is the name used in the Picture statement of BasicScript to refer to the picture.

The following resource types are supported in picture libraries:

Resource Type	Description
2	Windows Bitmap. This is defined in windows as RT_BITMAP.
256	Windows Metafile. Since there is no resource type for metafiles, 256 is used.

To create a picture library

1. Create a C file containing the minimal code required to establish a DLL. The following code can be used:

```
#include <windows.h>
int CALLBACK LibMain(
    HINSTANCE hInstance,
    WORD wDataSeg,
    WORD wHeapSz,
{
    UnlockData(0);
    return 1;
}
```

2. Use the following code to create a DEF file for your picture library:

```
LIBRARY
DESCRIPTION "My Picture Library"
EXETYPE WINDOWS
CODE LOADONCALL MOVABLE DISCARDABLE
DATA PRELOAD MOVABLE SINGLE
HEAPSIZE 1024
```

3. Create a resource file containing your pictures. The following example shows a resource file using a bitmap called sample.bmp and a metafile called usa.wmf.

```
#define METAFILE 256
USA METAFILE "usa.wmf"
MySample BITMAP "sample.bmp"
```

4. Create a make file that compiles your C module, creates the resource file, and links everything together

To modify an existing picture library

1. Make a copy of the picture library you want to modify.
2. Modify the copy by adding pictures with a resource editor such as Borland's Resource Workshop or Microsoft's App Studio.

NOTE: When you use a resource editor, you need to create a new resource type for metafiles (using the value 256).

Duplicating Your Elements

If you need one or more copies of a particular element, you can create the first element and then use Dialog Editor's duplication feature rather than creating each of the additional elements separately.

To duplicate an element

1. Select the element you want to duplicate and then press CTRL + D.

Deleting Your Elements

To delete a single element

1. Select the element you want to delete and then press DELETE.

To delete all elements in a dialog box

1. Select the dialog box and then press DELETE.

   If the dialog box contains more than one control, Dialog Editor displays a message confirming your decision.

2. Click **Yes**. All the elements disappear, but the title bar and close button remain on the dialog box.

Undoing Editing Operations

You can undo editing operations that produce a change in your dialog box, including:

- Addition of an element
- Insertion of one or more elements from the Clipboard
- Deletion of an element
- Changes made to an element or dialog box,

To undo an editing operation

- Press CTRL + Z.

  The editing operation is reversed.

Using an Existing Dialog Box

There are three ways to use an existing dialog box in Dialog Editor:

1. Copy the dialog box code (or part of it) to the Clipboard and paste it into Dialog Editor.
2. Click **Capture Dialog** on the **File** menu to capture a dialog box from another application and place it in Dialog Editor.
3. Open a dialog box that has been saved to a file.

Pasting Existing Dialog Box Code into Dialog Editor

To paste an existing dialog box into Dialog Editor

1. Copy the dialog box code from your script to the Clipboard (include the Begin Dialog and End Dialog lines).
2. Open Dialog Editor.
3. Press CTRL + V.
4. Click **Yes** on the message box.

   Dialog Editor creates a new dialog box corresponding to the code on the Clipboard.

To paste an existing element into Dialog Editor

1. Copy the element(s) code from your script to the Clipboard.
2. Open Dialog Editor.
3. Press CTRL + V.

   Dialog Editor adds one or more elements to your current (or new) dialog box.

NOTE: When you paste a dialog box into Dialog Editor, the tabbing order of the controls is determined by the order in which the controls appear in the script.

When you paste one or more elements into Dialog Editor, they will come last in the tabbing order, following the elements that are already present in the current dialog box.

Capturing a Dialog Box from Another Application

To capture an existing dialog box

1. Open the dialog box that you want to capture.
2. Open Dialog Editor.
3. Click **Capture Dialog** on the **File** menu.

   The Select the Dialog Box to Capture dialog box appears.
4. Select this dialog box from the **Available Dialogs** list, and then click **OK**.

   A prompt appears asking whether you want to replace the current dialog box with this dialog box.
5. Click **Yes** to place the captured dialog box in the Dialog Editor.

NOTE: Dialog Editor only supports standard Windows controls and standard Windows dialog boxes.

Opening a Dialog Box File

1. Click **Open** on the **File** menu.

   The **Open Dialog File** dialog box appears.
2. Select the dialog box file and click **Open**.

   Dialog Editor creates a dialog box from the statements in the file.

NOTE: If there are any errors in the BasicScript statements that describe the dialog box, the Dialog Translation Errors dialog box will appear. This dialog box shows the lines of code containing the errors and provides a brief description of the nature of each error.

Editing Existing Dialog Boxes in a Script

1. In the Edit Script window, select all the lines in your script starting with **Begin Dialog** and ending with **End Dialog**.
2. Click **Edit Dialog** on the **Edit** menu.

   The Dialog Editor window appears.
3. Edit your dialog box, and then click **Update** on the **File** menu.

   Your updated dialog box code replaces the dialog box code you selected in step **1**.
4. Click **Exit and Return** on the **File** menu.

   You return to the Edit Script window. Your dialog box code will reflect the changes you made to the dialog box in the Dialog Editor.

Testing Your Dialog Box

Dialog Editor lets you run your edited dialog box for testing purposes. When you click the Test button on the toolbar, your dialog box is active. This gives you an opportunity to make sure it functions properly and fix any problems before you incorporate your dialog box into your script.

Before you test your dialog box, check the following:

- Your dialog box contains all the necessary push buttons.
- Your dialog box contains a Help button if one is needed.
- Your elements are aligned and sized properly.
- Your element labels, text elements, and dialog box title are spelled and capitalized correctly.
- Your elements fit within the borders of the dialog box.
- Group boxes are added so that the element order is obvious to the user.
- You used text elements to describe unlabeled elements.
- You assigned accelerator keys consistently (if you assigned them).

When your dialog box complies with the above list, it is ready to be tested.

Testing your dialog box is an iterative process that involves running your dialog box to see how well it works, identifying and fixing problems, and then running the dialog box again to make sure these problems are fixed and to identify any additional problems.

To test your dialog box

1. Save your dialog box (see page 158).
2. Press F5.

   The dialog box becomes active
3. Check your dialog box functions (see the next section).
4. To stop the dialog box, press F5.
5. Make any necessary corrections to your dialog box.
6. Repeat steps 2-4 until your dialog box works properly.

Checking Your Dialog Box Functions

Tab Order

When you press TAB, the focus should move through the controls in a logical order. Because users cannot interact with design elements, the focus skips over these elements.

To correct your tab order

1. Press F5 to stop the test.
2. Cut and paste your elements in the tab order you want.

   For example, cut element 1, and paste it. Then cut element 2, and paste it.

NOTE: If you click the **Cut** button, and then the click the **Paste** button, your element will not change its location in the dialog box.

3. Press F5 to confirm that the tab order is correct.

Option Button Grouping

Your option buttons should be grouped correctly. Selecting one option button in your group should automatically clear all other option buttons in your group.

To merge separate option button groups into a single group

1. Press F5 to stop the test.
2. Change the**.Option Group** box in every Option Button Information dialog box of the group so that each option button contains the same value.
3. Press F5 to confirm that the option buttons work properly.

Accelerator Keys

If you have assigned an accelerator key to a text element or group box in order to provide user access to a text box, list box, combo box, or drop-down list box, the accelerator keys should put the focus on the control.

To assign your accelerator keys correctly

1. Press F5 to stop the test.
2. Cut and paste your design element, then cut and paste the associated control.

NOTE: If you click the **Cut** button, and then the click the **Paste** button, your element will not change its location in the dialog box.

3. Press F5 to confirm that the accelerator keys work properly.

NOTE: Cutting and pasting a design element may affect the tab order of your elements. Therefore, you may need to re-cut and re-paste your elements to establish the correct tab order.

Adding an Element to Your Script

Dialog box elements can be transferred from the Dialog Editor to your script. When you place your element on the Clipboard, it is converted to a BasicScript statement.

To add an element to your script

1. Select the element that you want to add to your script.
2. Press CTRL + C.
3. Click **Exit and Return** on the **File** menu.
4. In the Edit Script window, click the location where you want the statement to go, and then click Paste on the toolbar.
5. Save your script.

Adding Your Dialog Box to Your Script

To insert a dialog box into your script

1. Click **Update** on the **File** menu.

   Your dialog box is converted into a series of BasicScript statements and placed at the insertion point in your script.
2. Click **Exit and Return** on the **File** menu.

   You return to the Edit Script window.
3. To put your dialog box code in another location:
   - Click the **Cut** button on the toolbar.
   - Move your insertion point to the new location.
   - Click the **Paste** button on the toolbar.

Incorporating Your Dialog Box into Your Script

After using Dialog Editor to insert a custom dialog box into your script, make the following modifications to your script:

1. Create a dialog record using the Dim statement.
2. Assign values to dialog box controls.
3. Display the dialog box using either the Dialog() function or the Dialog statement.
4. Retrieve values from the dialog box after the user closes it.

Each of these steps is explained in more detail in the subsections below.

Sample Script

Steps 1-4 will use this sample script as an example, and will add statements to this script to make it functional.

```
Sub Main()
    'Initialize list box array.
    Dim ListBox1$()
    'Define the dialog box template.
    Begin Dialog UserDialog,,163,94,"Grocery Order"
        Text 13,6,32,8,"&Quantity:",.Text1
        TextBox 48,4,28,12,.TextBox1
        ListBox 12,28,68,32,ListBox1$,.ListBox1
        OKButton 112,8,40,14
        CancelButton 112,28,40,14
    End Dialog
End Sub
```

Step 1: Creating a Dialog Record

To store the values retrieved from the custom dialog box, create a dialog record with a Dim statement, using the following syntax:

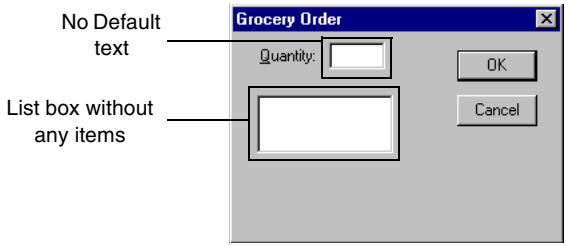
```
Dim DialogRecord As DialogVariable
```

In the sample script above, the Dim statement is as follows:

```
...
End Dialog
Dim b As UserDialog 'Create the dialog record.
Dialog b 'Display the dialog box.
...
```

Step 2: Assigning Values to Dialog Box Controls

If you open and run the sample script shown in the preceding subsection, you'll see a dialog box that resembles the following:



This custom dialog box isn't very useful; the user cannot see any items in the list box.

To assign values to dialog box controls, modify the control statements in your script. The following table lists the dialog box controls that you can assign values to:

Control	Type of Value
List box	Items
Drop-down list box	Item
Combo box	Item
Text box	Default text
Check box	Boolean Value

Adding an Item to Your Script

You can add items to the list box in the sample script above by creating an array and then assigning values to the elements of that array.

For example, you could add the following statements to initialize an array with three elements, where each element is assigned the name of a fruit:

```
Sub Main
    Dim Fruit as String      ‘Define the variable fruit.
    Dim ListBox1$(2)         'Initialize list box array.
    ListBox1$(0) = "Apples"
    ListBox1$(1) = "Oranges"
    ListBox1$(2) = "Pears"
...
```

You can create an array for your drop-down list box and combo box using the same method.

Adding Default Text to a Text Box

You can set the default value of the text box in the sample script above to 12 with the following statement:

```
...
Dim b As UserDialog     'Create the dialog record.
b.TextBox1 = "12"       'Make the default value of the text box 12
Dialog b                'Display the dialog box.
...
```

NOTE: The default text statement above must be entered after the dialog record statement but before the dialog display statement (as shown in the example code above).

Step 3: Displaying the Custom Dialog Box

To display a custom dialog box, you can use either a Dialog() function or a Dialog statement.

Using the Dialog() Function

You can use a Dialog() function to determine how the user closed your custom dialog box.

For example, the following statement will return a value when the user performs an action:

```
...
Dim b As UserDialog          ’Create the dialog record.
response% = Dialog(b)        ‘Dialog() function - display dialog box.
End Sub
```

The Dialog() function returns any of the following values:

Action	Value Returned
User clicks the OK button.	–1
User clicks the Cancel button	0
User clicks another push button. The returned number corresponds to the tab order of the push button.	(greater than 0) 1 = first push button 2 = second push button and so on.

Using the Dialog Statement

Use the Dialog statement when there is only one push button on your dialog box. The following is an example of the correct use of the Dialog statement:

```
...
Dim b As UserDialog          'Create the dialog record.
Dialog b                     'Dialog statement - display dialog box.
End Sub
```

Step 4: Retrieving Values from the Custom Dialog Box

After displaying a custom dialog box for your user, your script must retrieve the values from the dialog controls. You retrieve these values by referencing the appropriate identifiers in the dialog record.

For example, the following statements retrieve the value of the text box and the list box to display a message.

```
...
response%=Dialog(b) 'Dialog() function - display dialog box.
'Create a message box with a display that is contigent upon the user
'action.
Select Case response%
    Case -1
      Fruit$=ListBox1$(b.ListBox1)
      MsgBox "Thank you for ordering" + b.TextBox1+" "+Fruit$+"."
    Case Else
      MsgBox "Your order has been canceled."
End Select
...
```

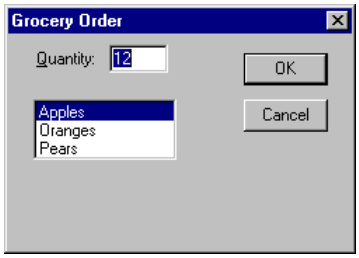
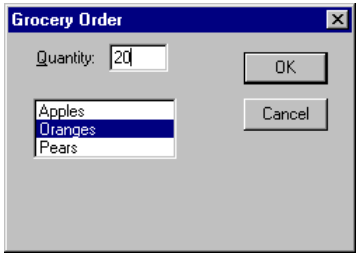
Example of Your Finished Script

If you inserted the sample code of these four steps into the sample script, your script might look something like this:

```
Sub Main
    Dim Fruit As String
    Dim ListBox1$(2)
    Dim response%

    ListBox1$(0) = "Apples"
    ListBox1$(1) = "Oranges"
    ListBox1$(2) = "Pears"

    Begin Dialog UserDialog,,163,94,"Grocery Order" _
        Text 13,6,32,8,"&Quantity:",.Text1
        'First control in the dialog box gets focus
        TextBox 48,4,28,12,.TextBox1
        ListBox 12,28,68,32,ListBox1$,.ListBox1
        OKButton 112,8,40,14
        CancelButton 112,28,40,14
    End Dialog

    Dim b As UserDialog          'Create the dialog record.
    b.TextBox1 = "12"            'Set the default value of the text box to 1 dozen.
    response = Dialog(b)         'Display the dialog box.

    'Create a message box with a display that is contigent upon the user action.
    Select Case response%
        Case -1
            Fruit$=ListBox1$(b.ListBox1)
            MsgBox "Thank you for ordering" + b.TextBox1 + " " +Fruit$+"."
        Case Else
            MsgBox "Your order has been canceled."
    End Select
End Sub
```

Dialog box and message boxes

The following table of figures show you what the dialog box and message box looks like. If you entered this script into your Edit Script window, you can run the appropriate TELE*form* operation (for example, evaluating a form image) to test it. (see Chapter 7 for more information on executing and debugging your script in the Edit Script window).

Dialog/Message Box	Description
	This is the dialog box when the user initiates it • The default text in the text box is 12. • The default item in the list box is Apples.
	This is the dialog box when the user changes these values. • The user enters 20 in the text box. • The user clicks **Oranges** in the list box.
	The user clicks **OK** in the Grocery Order dialog box, and receives this message box.
	The user clicks **Cancel** in the Grocery Order dialog box, and receives this message box.

Making Your Dialog Box Dynamic

As shown in the previous section, you can retrieve the values from dialog box controls after the user dismisses the dialog box by referencing the identifiers in the dialog record.

You can also retrieve values from a dialog box while the dialog box is displayed. To do this, you must make your dialog box dynamic.

Using a Dialog Function

With a dialog function, your script can carry out certain actions, such as hiding, changing, and disabling dialog box controls. This can be done *while* the dialog box is active.

Before BasicScript displays a custom dialog box (by executing a Dialog statement or Dialog() function), it must initialize the dialog box. During this initialization process, BasicScript checks to see whether you have defined a dialog function as part of your dialog box and calls it.

After completing its initialization process, BasicScript displays your custom dialog box. When the user clicks a control, BasicScript will again call your dialog function.

In the dynamic dialog box example below, the dialog function is the Function/End Function part of the script

Responding to User Actions

A BasicScript dialog function can respond to six types of user actions:

Function *FunctionName*(ControlName$, Action%, SuppValue%) As Integer

Action	Description
1	This action is sent immediately before the dialog box is displayed for the user.
2	This action is sent when: • A push button is clicked. • A check box is selected or cleared. • An option button is clicked. ControlName$ contains the name of the option button that was clicked SuppValue contains the index of the option button as it relates to the option group (1,2, and so on) • The current selection is changed in a list box, drop-down list box, or combo box. ControlName$ contains the name of the list box, combo box, or drop list box, SuppValue contains the index of the new item as it relates to the list (1,2, and so on).
3	This action is sent when the content of a text box or combo box has been changed *and* that control loses focus
4	This action is sent when a control gains the focus
5	This action is sent continuously when the dialog box is idle
6	This action is sent when the dialog box is moved.

The following script contains the important concepts you need to make your dialog box dynamic.

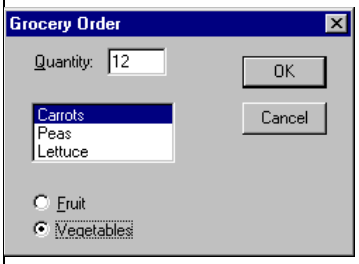
NOTE: This section does not explain in full detail how to make your dialog box dynamic. However, the comments in this code do provide you with the reasoning behind each statement.

```
'Dim "Fruits" and "Vegetables" arrays here to make them accessible to all procedures.
Dim Fruits(2) As String
Dim Vegetables(2) As String

'Dialog procedure - must precede the procedure that defines the dialog box.
Function DialogControl(ctrl$, action%, suppvalue%) As Integer
    Select Case action%
        Case 1
            'Fill list box with items before dialog box is visible.
            DlgListBoxArray "ListBox1", fruits
            'Set default value to first item in list box.
            DlgValue "ListBox1",0
        Case Else
            'Fill the list box with names of fruits or vegetables when the user selects an
            'option (radio) button.
            If ctrl$ = "OptionButton1" Then
                DlgListBoxArray "ListBox1", fruits
                DlgValue "ListBox1", 0
            ElseIf ctrl$ = "OptionButton2" Then
                DlgListBoxArray "ListBox1", vegetables
                DlgValue "ListBox1", 0
            End If
    End Select
End Function

Sub Main()
    Dim ListBox1$()     'Initialize array for use by ListBox statement in dialog box.
    Dim Produce$'       'Assign values to elements in the "Fruits" and "Vegetables"
                        'arrays.

    Fruits(0) = "Apples"
    Fruits(1) = "Oranges"
    Fruits(2) = "Pears"
    Vegetables(0) = "Carrots"
    Vegetables(1) = "Peas"
    Vegetables(2) = "Lettuce"
```

```
    'Define the dialog box.
    Begin Dialog UserDialog,,163,94,"Grocery Order",.DialogControl
        Text 13,6,32,8,"&Quantity:",.Text1
        'First control in template gets the focus.
        TextBox 48,4,28,12,.TextBox1
        ListBox 12,28,68,32,ListBox1$,.ListBox1
        OptionGroup.OptionGroup1
        OptionButton 12,68,48,8,"&Fruit",.OptionButton1
        OptionButton 12,80,48,8,"&Vegetables",.OptionButton2
        OKButton 112,8,40,14
        CancelButton 112,28,40,14
    End Dialog

    Dim b As UserDialog         'Create the dialog record.
    b.TextBox1 = "12"           'Set the default value of the text box to 1 dozen.
    response% = Dialog(b)       'Display the dialog box.

    Select Case response%
        Case -1
            If b.OptionGroup1 = 0 Then
                produce$ = fruits(b.ListBox1)
            Else
                produce$ = vegetables(b.ListBox1)
            End If
            MsgBox "Thank you for ordering " & b.TextBox1 & " "& produce$ & "."
        Case Else
            MsgBox "Your order has been canceled."
    End Select
End Sub
```

Dialog box and message boxes

The following table of figures show you what the dynamic dialog box and message boxes look like. If you entered this script into your Edit Script window, you can run the appropriate TELE*form* operation (for example, evaluating a form image) to test it. (See Chapter 7 for more information on executing and debugging your script in the Edit Script window.).

Dialog/Message Box	Description
	This is the dynamic dialog box when the user initiates it: • The default text in the text box is 12. • The default option is **Fruit**. • The default item in the list box is **Apples**.
	This is the dynamic dialog box when the user clicks on the **Vegetables** option • The items in the list box change to vegetables (this is dynamic). • The default item in the list box changes to **Carrots**.
	The user clicks **OK** in the Grocery Order dialog box, and receives this message box.

Dialog/Message Box	Description
	The user clicks **Cancel** in the Grocery Order dialog box, and receives this message box.

CHAPTER 11

Common Language Elements

About this Chapter

In this chapter, you can reference the commonly used BasicScript language elements. This list of elements is not exhaustive, but it does provide you with the vocabulary and syntax that is used regularly by script writers.

Common Language Elements

If you look at Appendix A, you will notice that the BasicScript language includes a large number of functions, statements, methods and operators. You might wonder which language elements are necessary to know, and which ones are used sparingly if at all. In this section, we will describe in detail each of the language elements most commonly used when writing your scripts.

IMPORTANT: Many of the language elements in this chapter are not explained in full detail. For the complete description of any of these language elements, refer to the BasicScript online help.

The following is a list of the BasicScript language elements that are explained in this chapter, sorted by the category to which they belong.

Variant

A variant is a universal (generic) data type.

Comments

Comments are not part of the compiled and executed script. Comments only describe the reasoning behind each line (or multiple lines) of the script.

- **Comments** - page 196

Declarations

These are the statements that declare constants, variables and functions in your script:

- **Const** - page 195
- **Dim** - page 197
- **Public** - page 199

Flow Control

These are the conditional and looping statements that control the sequence of events in your script:

- **If...Then...Else** - page 199
- **For...Next** - page 201

Logical Operators

These operators combine two elements in a line of your script:

- **And** - page 203
- **Or** - page 205

String Operators

The string operators convert a string into a number, and convert a number into a string:

- **Str$** - page 207
- **Val** - page 208

User Interface

These statements allow the user to interface with your script actively and passively:

- **InputBox$** - page 209
- **MsgBox** - page 210
- **DispMsg** - page 212

File Operators

These elements perform operations on external files:

Calling Functions

These elements call defined subroutines and functions:

Reserved Words

These words are reserved by BasicScript. Therefore, you cannot create functions, statements, etc. using reserved words as names.

Miscellaneous

Common language elements that do not fit into the above categories

Variant

The Variant variable can store any type of data that you want to put in it. BasicScript interprets the data based on the context with which it is used. You can think of this variable as a universal variable. Because of this, it is commonly used in BasicScript. However, be aware of the following BasicScript conventions.

Function Variant

The default data type for a function is Variant. Functions that are declared as Integers may generate a compile time message indicating that the function data type is different than a prior declaration.

To correct this problem, function declarations such as this:

```
declare function myFunc() as integer
function myFunc()

end function
```

must be changed to:

```
declare function myFunc() as integer
function myFunc() as integer

end function
```

Variable Variant

The default data type for a variable is 'Variant'. You must explicitly declare your variables in cases where you want a specific data type to be assigned to a variable

For example, the following script:

```
For i = 0 to Fields.Count - 1
    DispMsg Fields(i).Text
Next i
```

must be changed to:

```
Dim i as Integer
For i = 0 to Fields.Count - 1
    DispMsg Fields(i).Text
Next i
```

in order to avoid a runtime error message

IMPORTANT: As an alternative to declaring all your variables, you can put the statement **Option Default Integer** at the beginning of the script. Doing so will automatically treat all undeclared variables and functions as type **Integer**.

Declarations

Const (statement)

The **Const** statement declares a constant for use within the current script.

Syntax

Const name [As type] = expression1, name [As type] = expression2...

The **name** is only valid within the current BasicScript script. Constant names must follow these rules:

- Must begin with a letter.
- May contain only letters, digits, and the underscore character.
- Must not exceed 80 characters in length.
- Cannot be a reserved word.

NOTE: Constant names are not case-sensitive.

The expression must be assembled from literals or other constants. Calls to functions are not allowed except calls to the **Chr$** function, as shown below:

```
Const s$ = "Hello, there" + Chr(44)
```

Constants can be given an explicit type by declaring the name with a type-declaration character, as shown below:

```
Const a% = 5          'Constant Integer whose value is 5
Const b# = 5          'Constant Double whose value is 5.0
Const c$ = "5"        'Constant String whose value is "5"
Const d! = 5          'Constant Single whose value is 5.0
Const e& = 5          'Constant Long whose value is 5
```

The type can also be given by specifying the As type clause:

```
Const a As Integer = 5     'Constant Integer whose value is 5
Const b As Double = 5      'Constant Double whose value is 5.0
Const c As String = "5"    'Constant String whose value is "5"
```

```
Const d As Single = 5        'Constant Single whose value is 5.0
Const e As Long = 5          'Constant Long whose value is 5
```

You cannot specify both a type-declaration character and the type:

```
Const a% As Integer = 5          'THIS IS ILLEGAL.
```

Constants defined within a **Sub** or **Function** are local to that subroutine or function. Constants defined outside of all subroutines and functions can be used anywhere within that script.

Example

This example displays the declared constants in a dialog box (**crlf** produces a new line in the dialog box).

```
Const crlf = Chr$(13) + Chr$(10)
Const s As String = "This is a constant."
Sub Main()
    MsgBox s$ & crlf & "The constants are shown above."
End Sub
```

Comments

Comments can be added to BasicScript code using one of the following three methods:

1. All text between a single quotation mark and the end of the line is ignored:

   ```
   MsgBox "Hello"  'Displays a message box.
   ```

2. The REM statement causes the compiler to ignore the entire line:

   ```
   REM This is a comment.
   ```

3. BasicScript supports C-style multi-line comment blocks /*...*/, as shown in the following example:

   ```
   MsgBox "Before comment"
   /* This stuff is all commented out.
   This line, too, will be ignored.
   This is the last line of the comment. */
   MsgBox "After comment"
   ```

NOTE: C-style comments can be nested.

Dim (statement)

The **Dim** statement declares a list of local variables and their corresponding types and sizes.

Syntax

Dim name [(<subscripts>)] [As [New] type] [,name [(<subscripts>)] [As [New] type]]...

If a type-declaration character is used when specifying name (such as %, @, &, $, or !), the optional **[As type]** expression is not allowed. For example, the following are allowed:

```
Dim Temperature As Integer
Dim Temperature%
```

The **type** parameter specifies the type of the data item being declared. It can be any of the following data types: String, Integer, Long, Single, Double, Currency, Object, data object, built-in data type, or any user-defined data type. When specifying explicit object types, you can use the following syntax for type:

```
module.class
```

Where module is the name of the module in which the object is defined and class is the type of object. For example, to specify the OLE automation variable for Excel's Application object, you could use the following code:

```
Dim a As Excel.Application
```

Note: Explicit object types can only be specified for data objects and early bound OLE automation objects—i.e., objects whose type libraries have been registered with BasicScript.

- A **Dim** statement within a subroutine or function declares variables local to that subroutine or function.
- If the **Dim** statement appears outside of any subroutine or function declaration, then that variable has the same scope as variables declared with the **Private** statement.

NOTE: Private variables are not visible when you use the Watch Variable function in the Edit Script window during debug mode.

Naming Conventions

Variable names must follow these naming rules:

- Must start with a letter.
- May contain letters, digits, and the underscore character (_); punctuation is not allowed. The exclamation point (!) can appear within the name as long as it is not the last character, in which case it is interpreted as a type-declaration character.
- The last character of the name can be any of the following type-declaration characters: #, @, %, !, &, and $.
- Must not exceed 80 characters in length.
- Cannot be a reserved word

Examples

The following examples use the Dim statement to declare various variable types.

```
Sub Main()
        Dim i As Integer
        Dim l&                          'Long
        Dim s As Single
        Dim d#                          'Double
        Dim c$                          'String
        Dim MyArray(10) As Integer      '10 element integer array
        Dim MyStrings$(2,10)            '2-10 element string arrays
        Dim Filenames$(5 to 10)         '6 element string array
        Dim Values(1 to 10, 100 to 200) '111 element variant array
End Sub
```

NOTE: For more information on the Dim statement, refer to the BasicScript online help system.

Public (statement)

The **Public** statement declares a list of public variables and their corresponding types and sizes.

Syntax

Public name [(subscripts)] [As type, name] [(subscripts)] [As type]

Public variables are global to all Subs and Functions in all scripts.

If a type-declaration character is used when specifying name (such as %, @, &, $, or !), the optional **[As type]** expression is not allowed.

For example, the following are allowed:

```
Public foo As integer
Public foo%
```

Refer to the **Dim** (statement) section for more information on variables.

Flow Control

If...Then...Else (statement)

The **If...Then...Else** statement conditionally executes a statement or group of statements.

Syntax 1

```
If condition Then statements [Else else_statements]
```

Syntax 2

```
If condition Then
    [statements]
[ElseIf else_condition Then
    [elseif_statements]]
[Else
    [else_statements]]
End If
```

NOTE: There can be as many ElseIf conditions as are required.

Syntax 1 Parameters

The single-line conditional statement (syntax 1) has the following parameters:

Syntax 1 Parameter	Description
condition	Any expression evaluating to a Boolean value
statements	One or more statements separated with colons. This group of statements is executed when **condition** is True
else_statements	One or more statements separated with colons. This group of statements is executed when **condition** is False

Syntax 2 Parameters

The multi-line conditional statement (syntax 2) has the following parameters:

Syntax 2 Parameter	Description
condition	Any expression evaluating to a Boolean value.
statement	One or more statements to be executed when **condition** is True.
else_condition	Any expression evaluating to a Boolean value. The **else_condition** is evaluated if **condition** is False.
elseif_statements	One or more statements to be executed when **condition** is False and **else_condition** is True.
else_statements	One or more statements to be executed when both condition and else_condition are False

Example

This example inputs a name from the user and checks to see whether it is MICHAEL or MIKE using three forms of the If...Then...Else statement. It then branches to a statement that displays a welcome message depending on the user's name.

```
Sub Main()
    uname$ = UCase$(InputBox$("Enter your name:","Enter
Name"))
    if uname$ = "MICHAEL"  GoSub MikeName
    if uname$ = "MIKE" Then
        GoSub MikeName
        Exit Sub
    Else If uname$ = "" Then
        MsgBox "Since you have no name, I'll call you MIKE!"
        uname$ = "MIKE"
        GoSub MikeName
    Else
        GoSub OtherName
    End If
    Exit Sub

    MikeName:
        MsgBox "Hello, MICHAEL!"
        Return
    OtherName:
        MsgBox "Hello, " & uname$ & "!"
        Return
End Sub
```

For...Next (statement)

Repeats a block of statements a specified number of times, incrementing a loop counter by a given increment each time through the loop.

Syntax

```
For counter = start To end [Step increment]
    [statements]
    [Exit For]
    [statements]
Next [counter [,nextcounter]... ]
```

For statement

The For statement takes the following parameters:

For Parameter	Description
counter	Name of a numeric variable. Variables of the following types can be used: Integer, Long, Single, Double, Variant
start	Initial value for counter. The first time through the loop, counter is assigned this value
end	Final value for counter. The statements will continue executing until counter is equal to end.
increment	Amount added to counter each time through the loop. If end is greater than start, then increment must be positive. If end is less than start, then increment must be negative. If increment is not specified, then 1 is assumed. The expression given as increment is evaluated only once. Changing the step during execution of the loop will have no effect.
statements	Any number of BasicScript statements.

The **For...Next** statement continues executing until an **Exit For** statement is encountered or counter is greater than end.

For...Next statements can be nested. In such a case, the **Next [counter]** statement applies to the innermost **For...Next**.

Example

This example adds the numbers 1 through 10 using a For loop:

```
Sub Main()
    Dim i as integer
    Dim s as integer
    s = 0
    For i = 1 to 10
        s = s + 1
    Next i
    DispMsg "The sum is " & str$(s)
End Sub
```

Logical Operators

And (operator)

The **And** operator performs a logical or binary conjunction on two expressions.

Syntax

```
result = (expression1) And (expression2)
```

If both expressions are either Boolean, Boolean variants, or Null variants, then a logical conjunction is performed as follows:

If expression 1 is	and expression 2 is	then the result is
True	True	True
True	False	False
True	Null	Null
False	True	False
False	False	False
False	Null	False
Null	True	Null
Null	False	False
Null	Null	Null

Binary Conjunction

If the two expressions are integer, then a binary conjunction is performed, returning an integer result. All other numeric types (including Empty variants) are converted to Long, and a binary conjunction is then performed, returning a Long result.

Binary conjunction forms a new value based on a bit-by-bit comparison of the binary representations of the two expressions according to the following table:

If bit in expression 1 is	and bit in expression 2 is	then the result is
1	1	1
0	1	0
1	0	0
0	0	0

Example

```
Sub Main()
    n1 = 9          ‘1001 binary
    n2 = 12         ‘1100 binary
    b1 = True
    b2 = False

    'This expression performs a numeric bitwise And operation and
    ‘stores the result in N3.
    n3 = n1 And n2
    'This example performs a logical And comparing B1 and B2
    'and displays the result.
    If b1 And b2 Then
        MsgBox "b1 and b2 are both True; n3 is: " & n3
    Else
        MsgBox "b1 and b2 are not both True; n3 is: " & n3
    End If
End Sub
```

Or (operator)

The **Or** statement performs a logical or binary disjunction on two expressions.

Syntax

result = (expression1) Or (expression2)

If both expressions are either Boolean, Boolean variants, or Null variants, then a logical disjunction is performed as follows:

If expression 1 is	and expression 2 is	then the result is
True	True	True
True	False	True
True	Null	True
False	True	True
False	False	False
False	Null	Null
Null	True	True
Null	False	Null
Null	Null	Null

Binary Disjunction

If the two expressions are integer, then a binary disjunction is performed, returning an integer result. All other numeric types (including Empty variants) are converted to Long and a binary disjunction is then performed, returning a Long result.

Binary disjunction forms a new value based on a bit-by-bit comparison of the binary representations of the two expressions according to the following table:

If bit in expression 1 is	and bit in expression 2 is	then the result is
1	1	1
0	1	1
1	0	1
0	0	0

Examples

This example shows the use of logical **Or**.

```
Dim s$ As String
s$ = InputBox$("Enter a string.")
If s$ = "" Or Mid$(s$,1,1) = "A" Then
    s$ = LCase$(s$)
End If
```

This example shows the use of binary **Or**.

```
n1 = 9              '1001 binary
n2 = 12             '1100 binary
b1 = True
b2 = False
'This expression performs a numeric bitwise Or operation and stores
'the result as n3
n3 = n1 Or n2

'This example performs a logical Or that compares b1 and b2 and
'displays the result
If b1 Or b2 Then
    MsgBox "b1 or b2 are True; n3 is: " & n3
Else
    MsgBox "b1 and b2 are both false; n3 is: " & n3
End If
```

String Operators

Str$ (function)

The **Str$** function converts a given number into a string. This function is the opposite of the **Val** function listed in the next section.

Syntax

```
Str[$](number)
```

The number parameter is any numeric expression or expression convertible to a number. If number is negative, then the returned string will contain a leading minus sign. If number is positive, then the returned string will contain a leading space.

These functions only output the period as the decimal separator and do not output thousands separators. Use the CStr, Format, or Format$ function for more control over these options.

Example

In this example, the Str$ function is used to display the value of a numeric variable.

```
Sub Main()
    x# = 100.22
    MsgBox "The string value is: " + Str(x#)
End Sub
```

Val (function)

The **Val** function converts a given string expression into a number. This function is the opposite of the **Str$** function listed in the previous section.

Syntax

```
Val(string)
```

The string parameter can contain any of the following:

- Leading minus sign (for nonhex or octal numbers only)
- Hexadecimal number in the format &Hhexdigits
- Octal number in the format &Ooctaldigits
- Floating-point number, which can contain a decimal point and an optional exponent

NOTE: Spaces, tabs, and line feeds are ignored.

If string does not contain a number, then 0 is returned.

The **Val** function continues to read characters from the string up to the first non-numeric character.

The **Val** function always returns a double-precision floating-point value. This value is forced to the data type of the assigned variable.

Example

This example gets a number string from an InputBox and converts it to a number variable.

```
Sub Main()
    a$ = InputBox$("Enter anything containing a number", "Enter Number")
    b# = Val(a$)
    MsgBox "The value is: " & b#
End Sub
```

User Interface

InputBox$ (function)

The **inputbox$** function displays a dialog box with a text box into which the user can type.

Syntax

```
InputBox$[prompt],[title],[default],[xpos,ypos],[helpfile,context]
```

The content of the text box is returned as a String. A zero-length string is returned if the user selects Cancel.

Some of the named parameters of the InputBox$ function are:

InputBox$ Parameter	Description
prompt	Text to be displayed above the text box. The **prompt** parameter can contain multiple lines, each separated with an end-of-line (a carriage return, line feed, or carriage-return/line-feed pair). A runtime error is generated if **prompt** is Null
title	Caption of the dialog box. If this parameter is omitted, then no title appears as the dialog box's caption. A runtime error is generated if **title** is Null.
default	Default response. This string is initially displayed in the text box. A runtime error is generated if **default** is Null.

Example

```
Sub Main()
    s$ = InputBox$("File to copy:","Copy","sample.txt")
End Sub
```

NOTE: For more information on the InputBox$ function, see the BasicScript online help.

MsgBox (function)

The **MsgBox** function displays a message in a dialog box with a set of predefined buttons, returning an Integer representing which button was selected.

Syntax

MsgBox[prompt],[buttons],[title],[helpfile,context]

Some of the named parameters of the MsgBox function are:

MsgBox Parameter	Description
prompt	Message to be displayed: any expression convertible to a String. End-of-lines can be used to separate lines (either a carriage return, line feed, or both). If a given line is too long, it will be word-wrapped.
buttons	Integer specifying the type of dialog box (see below).
title	Caption of the dialog box. This parameter is any expression convertible to a String. If it is omitted, then "BasicScript" is used. A runtime error is generated if **title** is Null.

The **MsgBox** function returns one of the following values:

Constant	Value	Description
ebOK	**1**	OK was pressed
ebCancel	**2**	Cancel was pressed
ebAbort	**3**	Abort was pressed
ebRetry	**4**	Retry was pressed
ebIgnore	**5**	Ignore was pressed
ebYes	**6**	Yes was pressed
ebNo	**7**	No was pressed

The buttons parameter is the sum of any of the following values:

Constant	Value	Description
ebOKOnly	**0**	Displays OK buttons only
ebOKCancel	**1**	Displays OK and Cancel buttons
ebAbortRetryIgnore	**2**	Displays Abort, Retry and Ignore buttons
ebYesNoCancel	**3**	Displays Yes, No, and Cancel buttons
ebYesNo	**4**	Displays Yes and No buttons
ebRetryCancel	**5**	Displays Retry and Cancel buttons
ebCritical	**16**	Displays the “stop” icon
ebQuestion	**32**	Displays the “question mark” icon
ebExlamation	**48**	Displays the “exclamation point” icon
ebInformation	**64**	Displays the “information” icon
ebDefaultButton1	**0**	First button is the default button
ebDefaultButton2	**256**	Second buton is the default button
ebDefaultButton3	**512**	Third button is the default button
ebApplicationModal	**0**	Application Modal - the current application is suspended until the dialog box is closed
ebSystemModal	**4096**	System Modal - all applications are suspended until the dialog box is closed.

The default value for buttons is 0 (display only the OK button, making it the default).

Breaking Text across Lines

The prompt parameter can contain end-of-line characters, forcing the text that follows to start on a new line. The following example shows how to display a string on two lines:

```
MsgBox "This is on" + Chr(13) + Chr(10) + "two lines."
```

The carriage-return or line-feed characters can be used by themselves to designate an end-of-line.

Example

```
Sub Main
        MsgBox "This is a simple message box."
        MsgBox "This is a message box with a title and an
icon.",ebExclamation,"Simple"
        MsgBox "This message box has OK and Cancel
buttons.",ebOkCancel,"MsgBox"
        MsgBox "This message box is system
modal!",ebSystemModal
End Sub
```

NOTE: For more information on the MsgBox function, see the BasicScript online help.

DispMsg (statement)

Messages can be displayed in TELE*form* using either the **MsgBox** statement or the **DispMsg** statement. For more information on the **MsgBox** statement, see the previous language element.

- **MsgBox** can give the user more than one push button to respond with.
- **DispMsg** only has an **OK** button.

IMPORTANT: Using the **MsgBox** statement during form evaluation will cause TELE*form Reader* to halt until the message box is cleared.

Whenever possible, use **DispMsg** instead of **MsgBox** so that TELE*form Reader* can evaluate forms without being monitored. During evaluation, **DispMsg** will write the message to the TELE*form Reader* Message Log. In all other cases, it will generate a message box.

Some script entry points, such as Export_Record, can run after evaluation in TELE*form Reader* or after correction in TELE*form Verifier*. In either case, **DispMsg** will execute correctly.

Syntax

DispMsg [message], [mode value]

- **message** is a string to be displayed
- **mode value** is an integer describing the severity of the message (**mode** is optional).

Mode Values

The values for **mode** are:

Mode	Icon	Value
Unused		1
Note - this is the default setting		2
Warning- displays yellow triangle with exclamation point.		3
Error - displays "Error:" in front of message.		4
Fatal - displays red circle with white X.		5

Example

```
DispMsg "This is a note explaining an event"
DispMsg "This is a warning that should warrant special attention",3
```

File Operators

Open (statement)

The **Open** statement opens a file for a given mode, assigning the open file to the supplied filenumber.

Syntax

Open filename$ [For mode] [Access accessmode][lock] As[#] filenumber[Len = reclen]

- The **filename$** parameter is a string expression that contains a valid filename.
- The **filenumber** parameter is a number between 1 and 255.
- The **FreeFile** function (described in the next section) can be used to determine an available file number. Use of **FreeFile** is recommended for TELE*form* scripts.

File Mode Parameter

The **mode** parameter determines the type of operations that can be performed on that file. The following table lists each **mode** value and its description:

Mode Value	Description
Input	Opens an existing file for sequential input (**filename$** must exist). The value of **accessmode**, if specified, must be Read.
Output	Opens an existing file for sequential output, truncating its length to zero, or creates a new file. The value of **accessmode**, if specified, must be Write.
Append	Opens an existing file for sequential output, positioning the file pointer at the end of the file, or creates a new file. The end-of-file character, if present, is not removed by BasicScript. The value of **accessmode**, if specified, must be Read Write
Binary	Opens an existing file for binary I/O or creates a new file. Existing binary files are never truncated in length. The value of **accessmode**, if specified, determines how the file can subsequently be accessed.
Random*	Opens an existing file for record I/O or creates a new file. Existing random files are truncated only if **accessmode** is Write. The **reclen** parameter determines the record length for I/O operations.

*If the **mode** parameter is missing, then Random is used.

File Access Parameter

The **Access** parameter determines what type of I/O operations can be performed on the file. The following table lists each **accessmode** value and its description:

Accessmode Value	Description
Read	Opens the file for reading only. This value is valid only for files opened in Binary, Random, or Input mode
Write	Opens the file for writing only. This value is valid only for files opened in Binary, Random, or Output mode
Read Write	Opens the file for both reading and writing. This value is valid only for files opened in Binary, Random, or Append mode.

Example

This example opens several files in various configurations.

```
Sub Main()
    Open "test.dat" For Output As #2
    Close #2
    Open "test.dat" For Input As #1
    Close #1
    Kill "test.dat"
End Sub
```

NOTE: For more information on the Open statement, see the BasicScript online help.

Close (statement)

The **Close** statement closes the specified files.

Syntax

```
Close [# numberoffile1], [# numberoffile2], ...
```

If no arguments are specified, then all files are closed.

Example

This example opens three files and closes them in various combinations. (See the **Open** (statement) section above for more information on using **Open** in your scripts.)

```
Sub Main()
    Open "test1" For Output As #1
    Open "test2" For Output As #2
    Open "test3" For Random As #3
    MsgBox "The next available file number is :" & FreeFile()
    Close #1 'Closes file 1 only.
    Close #2, #3 'Closes files 2 and 3.
End Sub
```

FreeFile (function)

The **FreeFile** function refers to the integer containing the next available file number.

Syntax

FreeFile [rangenumber]

This function returns the next available file number within the specified range. If rangenumber is 0, then a number between 1 and 255 is returned. If rangenumber is 1, then a number between 256 and 511 is returned. If rangenumber is not specified, then a number between 1 and 255 is returned.

The function returns 0 if there is no available file number in the specified range.

The number returned is suitable for use in the **Open** statement.

Example

This example assigns **f** to the next free file number and displays it in a dialog box.

```
Sub Main()
    f = FreeFile()
    Open "test1" For Output As #f
    MsgBox "We used file number:" & f
    Close #f
End Sub
```

FileExists (function)

The **FileExists** function returns True if filename$ exists, and returns False otherwise.

Syntax

```
FileExists(filename$)
```

- This function determines whether a given filename$ is valid.
- This function will return False if filename$ specifies a subdirectory.

Example

This example checks to see whether there is an autoexec.bat, and then displays a message box that tells the user whether or not the file exists. This file is in the root directory of the C drive.

```
Sub Main()
    If FileExists("c:\autoexec.bat") Then
        Msgbox "This file exists!"
    Else
        MsgBox "File does not exist."
    End If
End Sub
```

Calling Functions

Sub...End Sub (statement)

The **Sub...End Sub** statement declares a subroutine.

Syntax

```
[Private | Public] [Static] Sub name[(arglist)]
    [statements]
End Sub
```

where **arglist** is a comma-separated list of the following (up to 30 arguments are allowed):

```
[Optional] [ByVal | ByRef] parameter[()] [As type]
```

Some of the parts of the Sub statement are:

Sub Part	Description
Name	Name of the subroutine, which must follow BasicScript naming conventions: 1.Must start with a letter. 2.May contain letters, digits, and the underscore character (_). Punctuation and type-declaration characters are not allowed. The exclamation point (!) can appear within the name as long as it is not the last character. 3.Must not exceed 80 characters in length.
Parameter	Name of the parameter, which must follow the same naming conventions as those used by variables. This name can include a type-declaration character, appearing in place of As type.
Type	Type of the parameter (i.e., Integer, String, and so on). Arrays are indicated with parentheses. For example, an array of integers would be declared as follows: Sub Test(a() As Integer) End Sub

- A subroutine terminates when one of the following statements is encountered:

  ```
  End Sub
  Exit Sub
  ```

- Subroutines can be recursive.

Passing Parameters to Subroutines

Parameters are passed to a subroutine either by value or by reference, depending on the declaration of that parameter in **arglist**.

- If the parameter is declared using the **ByRef** keyword, then any modifications to that passed parameter within the subroutine change the value of that variable in the caller.
- If the parameter is declared using the **ByVal** keyword, then the value of that variable cannot be changed in the called subroutine.
- If neither the **ByRef** nor the **ByVal** keyword is specified, then the parameter is passed by reference.

For more information on the **Sub...End Sub** statement, refer to the BasicScript online help.

Function...End Function (statement)

The **Function...End Function** statement creates a user-defined function.

Syntax

```
[Private | Public] [Static] Function name[(arglist)] [As ReturnType]
    [statements]
End Sub
```

where **arglist** is a comma-separated list of the following (up to 30 arguments are allowed):

```
[Optional] [ByVal | ByRef] parameter [()] [As type]
```

Some of the parts of the Function statement are:

Function Part	Description
Name	Name of the function, which must follow BasicScript naming conventions: 1.Must start with a letter. 2.May contain letters, digits, and the underscore character (_). Punctuation and type-declaration characters are not allowed. The exclamation point (!) can appear within the name as long as it is not the last character, in which case it is interpreted as a type-declaration character. 3.Must not exceed 80 characters in length.Additionally, the name parameter can end with an optional type-declaration character specifying the type of data returned by the function (i.e., any of the following characters: %, &, !, #, @).
Parameter	Name of the parameter, which must follow the same naming conventions as those used by variables. This name can include a type-declaration character, appearing in place of As type
Type	Type of the parameter (Integer, String, and so on). Arrays are indicated with parentheses. For example, an array of integers would be declared as follows: **Function Test(a() As Integer)** **End Function**
Returntype	Type of data returned by the function. If the return type is not given, then Variant is assumed. The **ReturnType** can only be specified if the function name (i.e., the name parameter) does not contain an explicit type-declaration character.

- A function returns to the caller when either of the following statements is encountered:

  End Function
  Exit Function

- Functions can be recursive.

Returning Values from Functions

To assign a return value, an expression must be assigned to the name of the function, as shown below:

```
Function TimesTwo(a As Integer) As Integer
    TimesTwo = a * 2
End Function
```

If no assignment is encountered before the function exits, then one of the following values is returned:

Value	Data Type Returned
0	Integer, Long, Single, Double, Currency
Zero-length string	String
Nothing	Object (or any data object)
Error	Variant
December 30, 1899	Date
False	Boolean

The type of the return value is determined by the As **ReturnType** clause on the **Function** statement itself.

Passing Parameters to Functions

Parameters are passed to a function either by value or by reference, depending on the declaration of that parameter in **arglist**.

- If the parameter is declared using the **ByRef** keyword, then any modifications to that passed parameter within the function change the value of that variable in the caller.
- If the parameter is declared using the **ByVal** keyword, then the value of that variable cannot be changed in the called function.
- If neither the **ByRef** or **ByVal** keywords are specified, then the parameter is passed by reference.

You can override passing a parameter by reference by enclosing that parameter within parentheses. For instance, the following example passes the variable j by reference, regardless of how the third parameter is declared in the **arglist** of **UserFunction**:

```
i = UserFunction(10,12,(j))
```

For more information on the **Function...End Function** statement, see the BasicScript online help.

Declare (statement)

The **declare** statement creates a prototype for either an external routine or a BasicScript routine that occurs later in the source module or in another source module.

Syntax

```
Declare Sub | Function name[TypeChar] [CDecl | Pascal | System |StdCall] [Lib]_
"LibName$" [Alias "AliasName$"] [ParameterList] As [type]
```

- Declare statements must appear outside of any Sub or Function declaration.
- Declare statements are only valid during the life of the script in which they appear.

For more information on the **Declare** statement, refer to the BasicScript online help.

Where **ParameterList** is a comma-separated list of the following (up to 30 parameters are allowed):

```
[Optional] [ByVal | ByRef] [ParameterName]() As
[ParameterType]
```

Some of the parameters of the Declare statement are:

Declare Parameter	Description
name	Any valid BasicScript name. When you declare functions, you can include a type-declaration character to indicate the return type. This name is specified as a normal BasicScript keyword— i.e., it does not appear within quotes.
"LibName$"	Must be specified if the routine is external. This parameter specifies the name of the library or code resource containing the external routine and must appear within quotes. The **"LibName$"** parameter can include an optional path specifying the exact location of the library or code resource.
"AliasName$"	Alias name that must be given to provide the name of the routine if the name parameter is not the routine's real name. Use an alias when the name of an external routine conflicts with the name of a BasicScript internal routine or when the external routine name contains invalid characters. The **AliasName$** parameter must appear within quotes.
As type	Indicates the return type for functions. For external functions, the valid return types are: Integer, Long, String, Single, Double, Date, Boolean, and data objects **NOTE:** Currency, Variant, fixed-length strings, arrays, user-defined types, and OLE Automation objects cannot be returned by external functions.
ParameterName	Name of the parameter, which must follow BasicScript naming conventions: 1. Must start with a letter. 2. May contain letters, digits, and the underscore character (_). 3. Punctuation and type-declaration characters are not allowed. The exclamation point (!) can appear within the name as long as it is not the last character, in which case it is interpreted as a type-declaration character. 4. Must not exceed 80 characters in length. Additionally, **ParameterName** can end with an optional type-declaration character specifying the type of that parameter (i.e., any of the following characters: %, &, !, #, @).

Declare Parameter	Description
ParameterType	Specifies the type of the parameter (e.g., Integer, String, Variant, and so on). The As **ParameterType** clause should only be included if **ParameterName** does not contain a type-declaration character. In addition to the default BasicScript data types, **ParameterType** can specify any user-defined structure, data object, or OLE Automation object. If the data type of the parameter is not known in advance, then the **Any** keyword can be used. This forces the BasicScript compiler to relax type checking, allowing any data type to be passed in place of the given argument.The **Any** data type can only be used when passing parameters to external routines.

Example

```
Declare Function IsLoaded% Lib "Kernel" Alias "GetModuleHandle" (ByVal name$)
Declare Function GetProfileString Lib "Kernel" (ByVal SName$,ByVal KName$,_
    ByVal Def$,ByVal Ret$,ByVal Size%) As Integer
Sub Main()
    SName$ = "Int"                          ‘Win.ini section name.
    KName$ = "sCountry”                     ‘Win.ini country setting.
    ret$ = String$(255, 0)                  'Initialize return string.
    If GetProfileString(SName$,KName$,"",ret$,Len(ret$)) Then
        MsgBox "Your country setting is: " & ret$
    Else
        MsgBox "There is no country setting in your win.ini file."
    End If
    If IsLoaded("Explorer") Then
        MsgBox "Explorer is loaded."
    Else
        MsgBox "Explorer is not loaded."
    End If
End Sub
```

NOTE: For much more information on the Declare statement, refer to the BasicScript online help.

Call (statement)

The **Call** statement transfers control to the given subroutine, optionally passing the specified arguments.

Syntax

```
Call subroutine_name [arguments]
```

Using this statement is equivalent to:

```
subroutine_name [arguments]
```

Use of the **Call** statement is optional. The **Call** statement can only be used to execute subroutines; functions cannot be executed with this statement.

The subroutine to which control is transferred by the **Call** statement must be declared outside of the **Main** procedure, as shown in the following example.

Example

This example demonstrates the use of the **Call** statement to pass control to another function.

This subroutine is declared externally to **Main** and displays the text passed in the parameter **s$**.

```
Sub Example_Call(s$)
    MsgBox "Call: " & s$
End Sub
```

This subroutine assigns a string variable to display, then calls subroutine **Example_Call**, passing parameter **S$** to be displayed in a message box within the subroutine.

```
Sub Main()
    s$ = "DAVE"
    Example_Call s$
    Call Example_Call("SUSAN")
End Sub
```

Reserved Words

Keyword

A keyword is any word or symbol recognized by BasicScript as part of the language.

All keywords are reserved by BasicScript. Therefore, you cannot create a variable, function, constant, or subroutine with the same name as a keyword. However, you are free to use all keywords as the names of structure members.

All of the following are keywords that you must restrict your use for:

Access	Alias	And	Any	Append	As
Base	Begin	Binary	Boolean	ByRef	ByVal
Call	CancelButton	Case	CDecl	CheckBox	Chr
ChrB	ChrW	Close	ComboBox	Compare	Const
CStrings	Currency	Date	Declare	Default	DefBool
DefCur	DefDate	DefDbl	DefInt	DefLng	DefObj
DefSng	DefStr	DefVar	Dialog	Dim	Do
Double	DropListBox	Else	ElseIf	End	Eqv
Error	Exit	Explicit	For	Function	Get
Global	GoSub	Goto	GroupBox	HelpButton	If
Imp	Inline	Input	InputB	Integer	IsLen
Let	Lib	Like	Line	Listbox	Lock
Long	Loop	LSet	Mid	MidB	Mod
Name	New	Next	Not	Nothing	Object
Off	OKButton	On	Open	Option	Optional
OptionButton	OptionGroup	Or	Output	ParamArray	Pascal
Picture	PictureButton	Preserve	Print	Private	Public
PushButton	Put	Random	Read	ReDim	REM
Resume	Return	RSet	Seek	Select	Set
Shared	Single	Spc	Static	StdCall	Step
Stop	String	Sub	System	Tab	Text
TextBox	Then	Time	To	Type	Unlock
Unit	Variant	WEnd	While	Width	Write
Xor					

For all other keywords in BasicScript (such as MsgBox, Str, and so on), the following restrictions apply:

- You can create a subroutine or function with the same name as a keyword.

- You can create a variable with the same name as a keyword as long as the variable is first explicitly declared with a **Dim**, **Private**, or **Public** statement.

Getting Around Reserved Words in BasicScript

The Fields collection provides a solution for calling reserved field names from a script. The ability of the Fields collection to accept a string allows you to rename variables so that they do not conflict with reserved BasicScript words.

For example:

Statement	Validity
Name.Text = **"Fred Jones"**	Illegal, "Name" is reserved.
Fields("Name").Text = **"Fred Jones"**	Legal
Dim Full_Name as Field **Set Full_Name = Fields("Name")**	Define an alias to "Name"
Full_Name.Text = **"Fred Jones"**	Legal

Miscellaneous

Nothing

Any property or Collection element that identifies another object may not always refer to a valid object. In these cases, the resulting object is **Nothing**. The reserved word **Nothing** can be used to determine whether an object variable is uninitialized.

Uninitialized object variables reference no object. In cases where it is possible for an object variable to be uninitialized, the user is expected to write scripts to take this into account by comparing an object with **Nothing**.

Examples

```
If Fields("foo") is Nothing Then
    DispMsg "No field named 'foo'!"
End If
```

You cannot use **Nothing** to check for an empty field value:

Incorrect

```
If Fields("fieldname").Name is Nothing Then
```

Correct

```
If Fields("fieldname") is Nothing Then    'no such field exists.
```

Incorrect

```
If fieldname.text is Nothing Then
```

Correct

```
If fieldname.text = "" Then    ' the field has an empty value.
```

Let (statement)

The **Let** statement assigns the result of an expression to a variable.

Syntax

```
[Let] variable = expression
```

The use of the word **Let** is supported for compatibility with other implementations of BasicScript. Normally, this word is dropped.

When assigning expressions to variables, internal type conversions are performed automatically between any two numeric quantities. Thus, you can freely assign numeric quantities without regard to type conversions. However, it is possible for an overflow error to occur when converting from larger to smaller types. This happens when the larger type contains a numeric quantity that cannot be represented by the smaller type.

For example, the following code will produce a runtime error:

```
Dim amount As Long
Dim quantity As Integer
amount = 400123          'Assign a value out of range for int.
quantity = amount        'Attempt to assign to Integer.
```

Example

```
Sub Main()
    Let a$ = "This is a string."
    Let b% = 100
    Let c# = 1213.3443
End Sub
```

CHAPTER 12

BasicScript Language Reference

About this Chapter

This chapter contains a summary of the BasicScript Language.

NOTE: The functions listed here may or may not be available in Windows 95, 98 and Windows NT (and therefore TELE*form*). See the next section for more information

Notes on this Reference

The following table summarizes the functions, statements, methods and other items that belong to the BasicScript language. Items are grouped by the tasks you want to perform. Full descriptions of each item can be viewed and printed from the online help system.

As previously noted, function availability will vary from one operating system to another. This availability is indicated at the bottom of each function's description in the Online Help. Only functions marked 'All Platforms' or 'Win32' (referring to Windows 95, 98 and Windows NT) are available in TELE*form.* Functions marked 'Windows' (referring to Windows 3.x) are not available in TELE*form.*

NOTE: Full descriptions of each language element in this list can be viewed and printed from the BasicScript Online Help.

Language Element Categories

The following table contains a description of each language element category in "Summary of the BasicScript Language" on page 235:

Language Element Category	Description	Refer to
Arrays	Contains elements that describe an array. An array is a collection (set) of objects.	page 235
BasicScript Information	Contains elements that relate to the CPU, the platform, the operating system and BasicScript	page 236
Clipboard	Contains elements that control and describe the contents of the Clipboard.	page 237
Comments	Contains elements that allow for comments (which are neither compiled nor executed)	page 237
Controlling Other Applications	Contains elements that describe and control external applications	page 238
Controlling Menus in Other Applications	Contains elements that describe and control the menu commands in external applications	page 239
Controlling Windows in Other Applications	Contains elements that describe and control the windows in external applications	page 240
Conversion	Contains elements that convert one type of object into another type of object, and describe whether or not an object can be converted	page 241
Date/Time	Contains elements that control and describe the system date and time, and manipulate objects that are related to the date and time	page 241
Desktop	Contains elements that manipulate the icons, windows and background of the desktop	page 243
Dialog Manipulation	Contains elements that describe and manipulate the controls and values of a dialog box	page 244
Dynamic Data Exchange (DDE)	Contains elements that control the DDE conversation with other applications	page 245

Language Element Category	Description	Refer to
Event Queue	Contains elements that describe and manipulate the system events	page 246
Error Handling	Contains elements that describe and manipulate the properties and attributes of the Error object	page 247
File I/O	Contains elements that describe and manipulate the operations of external files	page 248
File System	Contains elements that describe and manipulate the organization and attributes of the files in the File system	page 249
Financial	Contains elements that perform financial calculations	page 251
Flow Control	Contains elements that control the flow (sequence of events) of statements and functions	page 252
INI Files and Registry	Contains elements that describe and control the INI files and registry of your system	page 253
Logical/Binary Operators	Contains elements that perform logical or binary operations on two expressions	page 253
Math	Contains elements that perform mathematical operations on a number or an angle	page 254
Miscellaneous	Contains elements that do not belong to any of the other categories in this table	page 255
Network	Contains elements that describe and control the environment and configuration of your network	page 256
Numeric Operators	Contains elements that perform numeric operations on two expressions	page 256
Objects	Contains elements that describe and create OLE automation objects and compare object variables	page 257

Language Element Category	Description	Refer to
Open Database Connectivity (ODBC)	Contains elements that describe and manipulate the ODBC functionality in general and a single database in particular	page 257
Operating Environment	Contains elements that describe the operating environment and allows the starting and exiting of this operating environment.	page 258
Parsing	Contains elements that describe and return the attributes of a string	page 258
Predefined Dialogs	Contains elements that describe and manipulate the functionality of predefined dialog and message boxes	page 260
Printer	Contains elements that describe and control the current printer orientation	page 261
Printing	Contains elements that allow for the printing of certain objects and data	page 261
Procedures	Contains elements that call external routines, predefined subroutines and predefined functions	page 261
Screen Resolution	Contains elements that describe the screen and dialog resolution	page 262
Strings	Contains elements that describe and manipulate the attributes of one or more strings	page 262
User Dialog Boxes	Contains elements that set up and describe user-defined dialog boxes (see Chapter 9 for more information on creating these dialog boxes in BasicScript)	page 264
Variables and Constants	Contains elements that define constants, declare variables and set data types	page 266
Variants	Contains elements that describe the attributes of a variant	page 267
Viewport	Contains elements that control the Viewport	page 267

Summary of the BasicScript Language

Arrays

Array Language Element	Tasks
ArrayDims (function)	Return the number of dimensions of an array
ArraySort (statement)	Sort an array
Erase (statement)	Erase the elements in one or more arrays
LBound (function)	Return the lower bound of a given array dimension
Option Base (statement)	Change the default lower bound for array declarations
ReDim (statement)	Re-establish the dimensions of an array
UBound (function)	Return the upper bound of a dimension of an array

BasicScript Information

BasicScript Information Language Element	Task
Basic.Architecture$ (property)	Return the CPU architecture of the current system
Basic.Capability (method)	Returns the capabilities of the platform
Basic.CodePage (property)	Returns the code page of the current locale
Basic.EoIn$ (property)	Returns the end-of-line character for the platform
Basic.FreeMemory (property)	Returns the available memory
Basic.HomeDir$ (property)	Return the directory where BasicScript is located.
Basic.Locale (property)	Return the locale of the current system
Basic.OperatingSystem$ (property)	Return the name of the current operating system
Basic.OperatingSystem Vendor$ (property)	Return the name of the vendor of the current operating system
Basic.OperatingSystem Version$ (property)	Return the version of the current operating system
Basic.OS (property)	Return the platform ID
Basic.PathSeparator (property)	Return the path separator character for the platform
Basic.Processor$ (property)	Return the name of the CPU of the current system
Basic.ProcessorCount (property)	Return the number of CPU's installed on the current system
Basic.Version$ (property)	Return the version of BasicScript

Clipboard

Clipboard Language Element	Task
Clipboard$ (function)	Return the content of the clipboard as a string
Clipboard$ (statement)	Set the content of the clipboard
Clipboard.Clear (method)	Clear the clipboard
Clipboard.GetFormat (method)	Get the type of data stored in the clipboard
Clipboard.GetText (method)	Get the text from the clipboard
Clipboard.SetText (method)	Convert the content of the clipboard to text

Comments

Comments Language Element	Task
REM (statement)	Comment to end-of-line
’ (keyword)	Add a comment

Controlling other Applications

Controlling other Applications Language Element	Task
AppActivate (statement)	Activate the application
AppClose (statement)	Close an application
AppFileName$ (function)	Return the file name corresponding to an application
AppFind, AppFind$ (functions)	Return the full name of an application
AppGetActive$ (function)	Return the name of the active application
AppGetPosition$ (function)	Get the position and size of an application
AppGetState (function)	Get the window state of an application
AppHide (statement)	Hide an application
AppList (statement)	Fill an array with a list of running applications
AppMaximize (statement)	Maximize an application
AppMinimize (statement)	Minimize an application
AppMove (statement)	Move an application
AppRestore (statement)	Restore an application
AppSetState (statement)	Set the state of an application's window
AppShow (statement)	Show an application
AppSize (statement)	Change the size of an application
AppType (function)	Return the type of an application
DoKeys (statement)	Simulate keystrokes in another application
SendKeys (statement)	Send keystrokes to another application

Controlling other Applications Language Element	Task
Shell (function)	Execute another application

Controlling Menus in other Applications

Controlling Menus... Language Element	Task
Menu (statement)	Execute a menu command in another application
MenuItemChecked (function)	Determine if a menu item is checked in another application
MenuItemEnabled (function)	Determine if a menu item is enabled in another application
MenuItemExists (function)	Determine if a menu item exists in another application

Controlling Windows in other Applications

Controlling Windows... Language Element	Task
WinActivate (statement)	Activate a window
WinClose (statement)	Close a window
WinFind (function)	Given a window's name, find it
WinList (function)	Fill an array with window objects, one for each top-level window
WinMaximize (statement) **WinMinimize (statement)** **WinRestore (statement)** **WinSize (statement)**	Change the size of a window
WinMove (statement)	Move a window
HLine (statement)	Scroll the active window left/right by a specified number of lines
HPage (statement)	Scroll the active window left/right by a specified number of pages
HScroll (statement)	Scroll the active window left/right to a specified absolute position
VLine (statement)	Scroll the active window up/down by a specified number of lines
VPage (statement)	Scroll the active window up/down by a specified number of pages
VScroll (statement)	Scroll the active window up/down to a specified absolute position

Conversion

Conversion Language Element	Task
Asc, AscB, AscW (functions)	Return the value of a character
CBool, CCur, CDate, CDbl, CInt, CLng, CSng, CStr, CVar, CVDate, Fix, Int (functions)	Convert one numeric value to another
Chr, Chr$, ChrB, ChrB$, ChrW, ChrW$ (functions)	Convert a character value to a string
CVErr (function)	Convert a character to an error
Hex, Hex$ (functions)	Convert a number to a hexadecimal string
IsDate (function)	Determine if an expression can be converted to a date
IsError (function)	Determine if a variant contains a user-defined error value
IsNumeric (function)	Determine if an expression can be converted to a number
Oct. Oct$ (functions)	Convert a number to an octal string
Str, Str$ (functions)	Convert a number to a string
Val (function)	Convert a string to a number

Date/Time

Date/Time Language Element	Task
Date, Date$ (functions)	Return the current date
Date, Date$ (statements)	Change the system date
DateAdd (function)	Add a number of date intervals to a date
DateDiff (function)	Subtract a number of date intervals from a date
DatePart (function)	Return a portion of the date
DateSerial (function)	Assemble a date from date parts
DateValue (function)	Convert a string to a date
Day, Hour, Minute, Month, Second, Weekday, Year (functions)	Return a component of the date value
Now (function)	Return the current date and time
Time, Time$ (functions)	Return the current system time
TIme, Time$ (statements)	Set the system time
Timer (function)	Return the number of elapsed seconds since midnight
TimeSerial (function)	Assemble a date/time value from time components
TimeValue (function)	Convert a string to a date/time value

Desktop

Desktop Language Element	Task
Desktop.ArrangeIcons (method)	Arrange the icons on the desktop
Desktop.Cascade (method)	Cascades all non-minimized applications
Desktop.SetColors (method)	Set the desktop colors
Desktop.SetWallpaper (method)	Set the desktop wallpaper
Desktop.Snapshot (method)	Capture an image, placing it in the clipboard
Desktop.Tile (method)	Tiles all non-minimized applications

Dialog Manipulation

Dialog Manipulation Language Element	Task
ActivateControl (statement)	Activate a control
ButtonEnabled, CheckBoxEnabled, ComboBoxEnabled, EditEnabled, ListBoxEnabled, OptionEnabled (functions)	Determine if a control in another application's dialog box is enabled
ButtonExists, CheckBoxExists, ComboBoxExists, EditExists, ListBoxExists, OptionExists (functions	Determine if a control in another application's dialog box exists
GetCheckBox, GetComboBoxItem$, GetComboBoxItemCount, GetEditText$, GetListBoxItem$, GetListBoxItemCount, GetOption (functions)	Retrieve a value from a control in another application's dialog box
SelectButton, SelectComboBoxItem, SelectListBoxItem (statements)	Select a control in another application's dialog box
SetCheckBox, SetEditText, SetOption (statement)	Set the state of a control in another application's dialog box

Dynamic Data Exchange (DDE)

DDE Language Element	Task
DDEExecute (statement)	Execute a command in another application
DDEInitate (function)	Initiate a DDE conversation with another application
DDEPoke (statement)	Set a value in another application
DDERequest, **DDERequest$** **(functions)**	Return a value from another application
DDESend (statement)	Establish a DDE conversation, then set a value in another application
DDETerminate **DDETerminateAll** **(statements)**	Terminate one or more conversations
DDETimeOut (statement)	Set the timeout used for non-responding applications

Event Queue (all statements)

Event Queue Language Element	Task
QueEmpty	Empty a queue
QueFlush	Play back all events stored in a queue
QueKeyDn	Add key down event to the queue
QueKeys	Add key down/up events to the queue
QueKeyUp	Add key up event to the queue
QueMouseClick	Add mouse click to the queue
QueMouseDblClk	Add mouse double-click to the queue
QueMouseDblDwn	Add mouse down--up--down event to the queue
QueMouseDn	Add mouse down event to the queue
QueMouseMove	Add mouse move event to the queue
QueMouseMoveBatch	Add many mouse move events to the queue
QueMouseUp	Add mouse up event to the queue
QueSetRelativeWindow	Make all mouse positions in a queue relative to a window

Error Handling

Error Handling Language Element	Task
Err.Clear (method)	Clear the properties of the Error object
Err.Description (property)	Set or retrieve the description of the Error object
Err.HelpContext (property)	Set or retrieve the help context ID of the Error object
Err.HelpFile (property)	Set or retrieve the help file associated with the Error object
Err.LastDLLError (property)	Return the last error generated by a call to a DLL
Err.Number (property)	Return or set the number of the Error object
Err.Raise (method)	Generate a runtime error
Err.Source (property)	Set or retrieve the source of a runtime error
Erl (function)	Set the value of the error
Error (statement)	Simulate a trappable runtime error
Error, Error$ (functions)	Return the text of a given error
On Error (statement)	Trap an error
Resume (statement)	Continue execution after an error trap

File I/O

File I/O Language Element	Task
Close (statement)	Close one or more files
Eof (function)	Determine if the end-of-file has been reached
FreeFile (function)	Return to the next available file number
Get (statement)	Read data from a random or binary file
Input# (statement)	Read data from a sequential file into variables
Input, Input$, InputB, InputB$ (functions)	Read a specified number of bytes from a file
Line Input # (statement)	Read a line of text from a sequential file
Loc (function)	Return the record position of the file pointer within a file
Lock, Unlock (statements)	Lock or unlock a section of a file
Lof (function)	Return the number of bytes in an open file
Open (statement)	Open a file for reading or writing
Print # (statement)	Print data to a file
Put (statement)	Write data to a binary or random file
Reset (statement)	Close all open files
Seek (statement/ function)	Set/Return the byte position of the file pointer within a file
Width# (statement)	Specify the line width for sequential files
Write# (statement)	Write data to a sequential file

File System (see also predefined dialogs)

File System Language Element	Task
ChDir (statement)	Change the current directory
ChDrive (statement)	Change the current drive
CurDir, CurDir$ (functions)	Return the current directory
Dir, Dir$ (functions)	Return files in a directory
DiskDrives (statement)	Filll an array with valid disk drive letters
DiskFree (function)	Return the free space on a given hard drive
FileAttr (function)	Return the mode in which a file is open
FileCopy (statement)	Copy a file
FileDateTime (function)	Return the date and time when a file was last modified
FileDirs (statement)	Fill an array with a subdirectory list
FileExists (function)	Determine if a file exists
FileLen (function)	Return the length of a file in bytes
FileList (statement)	Fill an array with a list of files
FileParse$ (function)	Return a portion of a file name
FileType (function)	Return the file type
GetAttr (function)	Return the attributes of a file
Kill (statement)	Delete files from hard disk drive

File System Language Element	Task
MacID (function)	Return a value representing a collection of same-type files on the Macintosh
MkDir (statement)	Create a subdirectory
Name (statement)	Rename a file
RmDir (statement)	Remove a subdirectory
SetAttr (statement)	Change the attributes of a file

Financial (all functions)

Financial Language Element	Task
DDB	Return depreciation of an asset using double-declining balance method
FV	Return the future value of an annuity
IPmt	Return the interest payment for a given period of an annuity
IRR	Return the internal rate of return for a series of payments and receipts
MIRR	Return the modified internal rate of return
NPer	Return the number of periods of an annuity
NPV	Return the net present value of an annuity
Pmt	Return the payment of an annuity
PPmt	Return the principal payment for a given period of an annuity
PV	Return the present value of an annuity
Rate	Return the interest rate for each period of an annuity
SLN	Return the straight-line depreciation of an asset
SYD	Return the Sum of Years' Digits depreciation of an asset

Flow Control

Flow Control Language Element	Task
Call (statement)	Call a subroutine
Choose (function)	Return a value at a given index
Do...Loop (statement)	Execute a group of statements repeatedly
DoEvents (statement/func.)	Yield control to other applications
End (statement)	Stop execution of a script
Exit Do/For (statement)	Exit a Do/For loop
For...Next (statement)	Execute a block of statements repeatedly
GoSub (statement)	Execute at a specific label, allowing control to return later
Goto (statement)	Execute at a specific label
If...Then...Else (statement)	Conditionally execute one or more statements
IIf (function)	Return one of two values depending on a condition
Main (statement)	Define a subroutine where execution begins
Return (statement)	Continue execution after the most recent GoSub
Select...Case (statement)	Execute one of a series of statements
Sleep (statement)	Pause for a specified number of milliseconds
Stop (statement)	Suspend execution, returning to a debugger (if present)
Switch (statement)	Return one of a series of expressions depending on a condition

INI Files and Registry

INI Files and Registry Language Element	Task
DeleteSetting (statement)	Delete a setting from the system registry or an INI file
GetAllSettings (statement)	Return the values of all keys or settings within the system registry
GetSetting (function)	Return the value of a key or setting within the system registry
ReadIni$ (function)	Read a string from an INI file
ReadIniSection (statement)	Read all of the item names from a given section of an INI file
SaveSetting (statement)	Update the value of a key or setting within the system registry
WriteIni (statement)	Write a new value to an INI file

Logical/binary operators (all operators)

Operator Language Element	Task
And, Eqv , Imp, Not, Or, Xor	Perform logical or binary operations on two expressions

Math (all functions)

Math Language Element	Task
Abs	Return the absolute value of a number
Atn	Return the arc tangent of a number
Cos	Return the cosine of an angle
Exp	Return e raised to a given power
Fix	Return the integer part of a number
Int	Return the integer portion of a number
Log	Return the natural logarithm of a number
Random	Return a random number between two values
Randomize	Initialize the random number generator
Rnd	Generate a random number between 0 and 1
Sgn	Return the sign of a number
Sin	Return the sine of an angle
Sqr	Return the square root of a number
Tan	Return the tangent of an angle

Miscellaneous

Misc. Language Element	Task
#Const (directive)	Define a preprocessor constant for the BasicScript compiler
#If... Then... #Else (directive)	Direct the BasicScript compiler to include or exclude sections of code based on conditions
( ) (keyword)	Force parts of an expression to be evaluated before others
_ (keyword)	Add a line continuation character
Beep (statement)	Make a sound
IMEStatus (function)	Return the status of the Input Method Editor
Inline (statement)	Allows execution or interpretation of a block of text
MacScript (statement)	Execute an AppleScript script
Mci (function)	Execute an MCI command
Option Default (statement)	Set the default data type of variables and return values
Option Explicit (statement)	Prevent implicit declarations of variables and return values
PrintFile (function)	Print a file using the application to which the file belongs

Network (all methods)

Network Language Element	Task
Net.AddCon$	Redirect a local device to a shared device on a network
Net.Browse$	Display a dialog box requesting a network directory or printer resource
Net.CancelCon	Cancel a network connection
Net.Dialog	Display a dialog box allowing configuration of the network
Net.GetCaps	Return information about the capabilities of the network
Net.GetCon$	Return the name of the network resource associated with a local device
Net.User$	Return the name of the user on the network

Numeric Operators (all operators)

Numeric Operators Language Element	Task
*****	Multiply
+	Add
-	Subtract
/	Divide
****	Integer divide
^	Raise to a power
Mod	Determine the remainder

Objects

Objects Language Element	Task
CreateObject (function)	Create an OLE automation object
GetObject (function)	Return an OLE automation object from a file, or return a previously created OLE automation object
Is (operator)	Compare two object variables
Dim	Declare a local variable
Nothing	Value indicating no valid object

Open Database Connectivity (ODBC) (all functions)

ODBC Language Element	Task
SQLBind	Specify where to place results with SQLRetrieve
SQLClose	Close a connection to a database
SQLError	Return error information when an SQL function fails
SQLExecQuery	Execute a query against a database and return the number of rows or columns affected by the query
SQLGetSchema	Return information about the structure of a database
SQLOpen	Establish a connection with a database
SQLRequest	Run a query against a database, returning the results as an array
SQLRetrieve	Retrieve all or part of a query
SQLRetrieveToFile	Place the results of a query in a file

Operating Environment

Operating Environment Language Element	Task
Command, Command$ (functions)	Return the command line
HWND.Value (property)	Return the operating system value of a window
Environ, Environ$ (functions)	Return the value of an environment variable
System.FreeMemory (property)	Return the free memory in the operating environment
System.FreeResources (property)	Return the free resources in the operating environment
System.TotalMemory (property)	Return the total available memory in the operating environment
System.WindowsDirectory$ (property)	Return the directory containing Windows
System.WindowsVersion$ (property)	Return the Windows version
System.Exit (method)	Exit the operating environment
System.MouseTrails (method)	Toggle mouse trails on and off
System.Restart (method)	Restart the operating environment

Parsing

Parsing Language Element	Task
Item$ (function)	Return a range of items from a string
ItemCount (function)	Return the number of items in a string
Line$ (function)	Retrieve a line from a string
LineCount (function)	Return the number of lines in a string

Parsing Language Element	Task
.Word$ (function)	Return a sequence of words from a string
WordCount (function)	Return the number of words in a string

Predefined Dialogs

Predefined Dialogs Language Element	Task
AnswerBox (function)	Display a dialog box asking a question
AskBox, AskBox$$ (functions)	Display a dialog box allowing the user to type a response
AskPassword, AskPassword$ (functions)	Display a dialog box where the user enters a password
InputBox, InputBox$ (functions)	Display a dialog box allowing the user to type a response
MsgBox (function)	Display a dialog box containing a message and some buttons
Msgbox (statement)	Display a dialog box containing a message and some buttons
Msg.Close (method)	Close a modeless message box
Msg.Open (method)	Open a modeless message box
Msg.SetText (property)	Set the message contained within a modeless message box
Msg.SetThermometer (property)	Set the percentage of the thermometer in a modeless message box
OpenFilename$ (function)	Display a dialog box requesting a file to open
PopupMenu (function)	Display a popup menu containing items from an array
SaveFilename$ (function)	Display a dialog box requesting the name of a new file
SelectBox (function)	Display a dialog box allowing the selection of an item from an array

Printer

Printer Language Element	Task
PrinterGetOrientation (function)	Retrieve the current printer orientation
PrinterSetOrientation (statement)	Set the printer orientation

Printing

Printing Language Element	Task
Print (statement)	Print data to the screen
Spc (function)	Print a number of spaces within a Print statement
Tab (function)	Used with Print to print spaces up to a column position

Procedures

Procedures Language Element	Task
Declare (statement)	Define an external routine or a forward reference
Exit Function (statement)	Exit a function
Exit Sub (statement)	Exit a subroutine
Function...End (statement)	Create a user-defined function
Sub...End (statement)	Create a user-defined subroutine

Screen Resolution

Screen Resolution Language Element	Task
Screen.DlgBaseUnitsX (property)	Return the x dialog base units
Screen.DlgBaseUnitsY (property)	Return the y dialog base units
Screen.Height (property)	Return the height of the display, in pixels
Screen.TwipsPerPixelX (property)	Return the number of twips per pixel in the x direction
Screen.TwiipsPerPixelY (property)	Return the number of twips per pixel in the y direction
Screen.Width (property)	Return the width of the display, in pixels

Strings (see also Parsing, Conversion)

Strings Language Element	Task
& (operator)	Join two strings together
Format, Format$ (functions)	Return a string formatted to a given specification
InStr, InStrB (functions)	Return the position of one string with another
LCase, LCase$ (functions)	Convert a string to lower case
Left, Left$, LeftB, LeftB$ (functions)	Return the left portion of a string
Len, LenB (functions)	Return the length of a string or the size of a data item
Like (function)	Compare a string against a pattern

Strings Language Element	Task
LSet (function)	Left align a string or user-defined type within another
LTrim, LTrim$ (functions)	Remove leading spaces from a string
Mid, Mid$, MidB, MidB$ (functions)	Return a substring from a string
Mid, Mid$, MidB, MidB$ (statements)	Replace one part of a string with another
Option Compare (statement)	Change the default comparison between text and binary
Option CStrings (statement)	Allow interpretation of C-style escape sequences in strings
Right, Right$, RightB, RightB$ (functions)	Return the right portion of a string
RSet (statement)	Right align a string within another
RTrim, RTrim$ (functions)	Remove trailing spaces from a string
Space, Space$ (functions)	Return a string of spaces
StrComp (function)	Compare two strings
StrConv (function)	Convert a string based on a conversion parameter
String, String$ (functions)	Return a string consisting of a repeated character
Trim, Trim$ (functions)	Trim leading and trailing spaces from a string
UCase, UCase$ (functions)	Return the upper case of a string

User Dialog Boxes

User Dialog Boxes Language Element	Task
Begin Dialog (statement)	Begin definition of a dialog box
CancelButton, CheckBox, ComboBox, DropListBox, GroupBox, ListBox, OKButton, OptionButton, OptionGroup, Picture, PictureButton, PushButton, Text, TextBox (statements)	Add a control to a dialog box
Dialog (function)	Initiate a dialog box, and return the button that was selected by the user
Dialog (statement)	Initiate a dialog box
DlgCaption (function)	Return the caption of a dynamic dialog box
DlgCaption (statement)	Change the caption of a dynamic dialog box
DlgControlID (function)	Return the ID of a control in a dynamic dialog box
DlgEnable (function)	Determine if a control is enabled in a dynamic dialog box
DlgEnable (statement)	Enables or disables a control in a dynamic dialog box
DlgFocus (function)	Return the control with the focus in a dynamic dialog box
DlgFocus (statement)	Set the focus to a control in a dynamic dialog box
DlgListBoxArray (statement)	Set the content of a list box or combo box in a dynamic dialog box
DlgSetPicture (statement)	Set the picture of a control in a dynamic dialog box

User Dialog Boxes Language Element	Task
DlgText (statement)	Set the content of a list box or combo box in a dynamic dialog box
DlgText$ (function)	Return the content of a control in a dynamic dialog box
DlgValue (function)	Return the value of a control in a dynamic dialog box
DlgValue (statement)	Set the value of a control in a dynamic dialog box
DlgVisible (function)	Determine if a control is visible in a dynamic dialog box
DlgVisible (statement)	Set the visibility of a control in a dynamic dialog box

Variables and Constants (all statements)

Variables/Constants Language Element	Task
=	Assignment
Const	Define a constant
DefBool, DefCur, DefDate, DefDbl, DefInt, DefLng, DefObj, DefSng, DefStr, DefVar	Set the default data type
Dim	Declare a local variable
Global	Declare variables for sharing between scripts
Let	Assign a value to a variable
Private	Declare variables accessible to all routines in a script
Public	Declare variables accessible to all routines in all scripts
Set	Assign an object variable
Type	Declare a user-defined data type

Variants (all functions)

Variant Language Element	Task
IsEmpty	Determine if a variant has been initialized
IsError	Determine if a variant contains a user-defined error
IsMissing	Determine if an optional parameter was specified
IsNull	Determine if a variant contains valid data
IsObject	Determine if an expression contains an object
VarType	Return the type of data stored in a variant

Viewport (all methods)

Viewport Language Element	Task
Viewport.Clear	Clear the contents of the viewport
Viewport.Close	Close the viewport
Viewport.Open	Open a viewport

A

B

C

D

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W