



Chapter 4: Using E-Basic

4.1 Why Use E-Basic?

Although E-Studio is a robust application, E-Basic is the underlying scripting language. Specifically, the graphical design created in E-Studio is translated to E-Basic script when the experiment is compiled or generated. While E-Studio is filled with interesting and clever functionality, chances are that it can be improved to satisfy the end user's specific or custom needs. That is where E-Basic is useful. If the design of E-Studio is too constricting, E-Basic affords the ability to accommodate the needs of individual users.

E-Studio provides the foundation for experiments in E-Prime. It is recommended that all users, regardless of their programming expertise, take full advantage of the graphical design interface in E-Studio rather than writing straight E-Basic scripts. E-Studio's graphical interface can do most, if not all, of the work. Many users will not even need to use E-Basic.

The majority of all experiments can be done using E-Studio's graphical design interface. The basic structure of almost all experiments includes blocks of trials within a session. It is more efficient to use the E-Studio design interface to set this up. It's more effective to drag and drop icons onto procedural timelines than it is to write the equivalent script.

E-Basic is the underlying scripting language of E-Prime, and the power of the E-Prime system. Where E-Studio leaves off, E-Basic may be used to extend the system. This is similar to the use of Visual Basic for Applications™ within the Microsoft Office™ suite of applications. On its own, E-Basic is not particularly useful. However when used in conjunction with E-Studio and E-Run, power and flexibility abound.

E-Basic really excels when used to extend the functionality of E-Studio. For example, the ability to exit a set of practice trials based on a percentage correct criterion is something that not all users are interested in, and for which there is no graphical option. The function can be accomplished through a few lines of E-Basic script strategically placed in an InLine object or two. The example, which follows, illustrates most of the script necessary to set up a criterion-based exit:

```
'Terminate Practice Trials if greater than 80% accuracy
If PracticeProp.Mean < .80 Then
    PracticeResults.Text = "Your accuracy for the practice"&_
        " trials was " & CStr(PracticeProp.Mean)*100 &
        "%.\n\nYou must achieve 80% accuracy in order to"&_
        " continue with the experimental trials.\n\n Press"&_
        " the spacebar to repeat the practice trials."
Else
    PracBlockList.Terminate
    PracticeResults.Text = "Your accuracy for the practice"&_
        " trials was " & CStr(PracticeProp.Mean)*100 & "%."&_
        "\n\n Press the spacebar to continue"
End If
```

Upon reading this code carefully, it is not difficult to guess what it is supposed to do. One of the nicest features of E-Basic is that the language is similar to ordinary English. If the example looks like a foreign language, rest assured, this chapter has a section devoted to the beginner user in



addition to sections for intermediate and advanced users. The goal of this chapter is to get the user accustomed to writing E-Basic script. If this is something that is believed to be beyond present abilities, the following section will recommend some additional sources.

4.1.1 Before Beginning...

Before attempting to write any script using E-Basic, a few critical pieces of information must be known. **It is recommended that even the most expert programmer read the following section.**

Learning to write script will minimally require learning the basics of programming. The complexity of the task will determine the amount and complexity of the script required, and therefore, the amount and complexity of script-writing knowledge necessary. Most experiments involving user-written script will minimally require the user to be able to write and add functions (e.g., to display an internal variable, or indicate contingent branching based on responses). For someone with programming experience, these types of functions might be accomplished in a few minutes or hours, while someone without programming experience may need a day or more to accomplish the same goal. More complicated tasks, such as creating a simulated ATM machine, will require more comprehensive programming and may require considerable programming experience. Again, the length of time required to accomplish the task will depend on programming knowledge, skill, and the task itself. Complex systems interactions, such as creating new SDKs, would require significant programming knowledge and expertise, and are best handled by professional programmers.

The present chapter discusses the E-Basic language and the basics of entering user-written script. If the reader has experience programming with languages such as Basic, C, Pascal, or experience writing MEL Professional code subroutines, the information in this chapter should be straightforward. New users to programming are advised to become familiar with script by taking a programming course, or by carefully working through script examples. E-Basic is very similar to Visual Basic for Applications. Visual Basic for Applications would be the best programming course to choose. Visual Basic would also be useful, and the knowledge would transfer well.

If there is a preference to learn about programming alone, or for more help, PST recommends the following reference sources:

For the novice user who has little to no programming experience and for those new to VBA type languages:

VBA for Dummies, Steve Cummings, IDG Books Worldwide, Inc., Foster City, CA, 1998.

For more advanced users with substantial programming experience:

VBA Developer's Handbook, Ken Getz & Mike Gilbert, Sybex Inc., San Francisco, CA, 1997

Another efficient method of learning to program using E-Basic is to examine script examples in the E-Basic Online Help and actual programs generated by E-Studio. The E-Basic Online Help may be launched from the Start button, or through the Help menu in E-Studio.

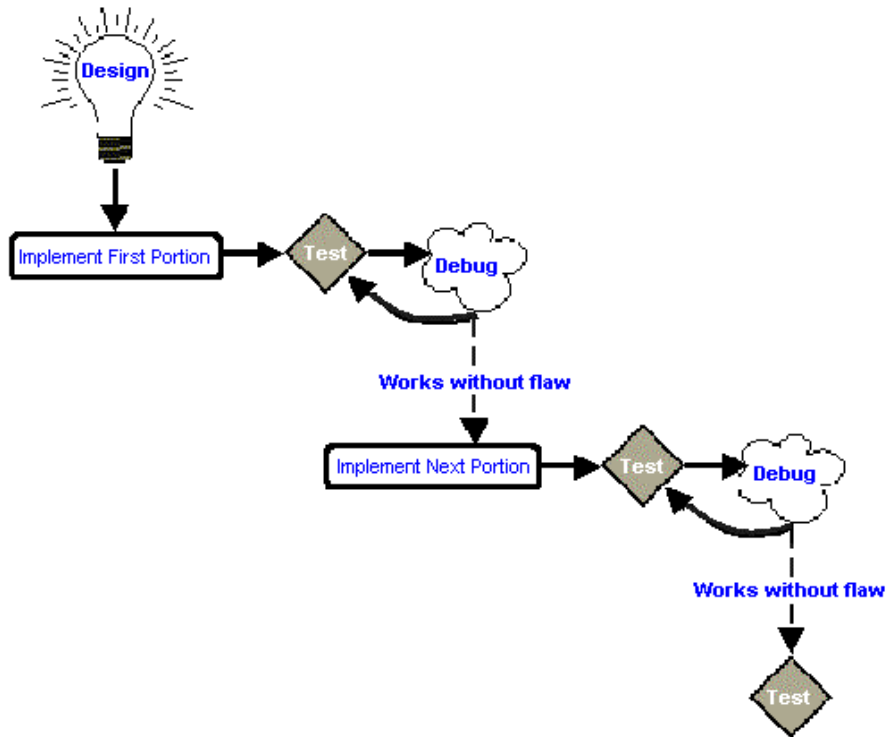
It can be very useful to view the script automatically generated by E-Studio in an EBS (E-Basic Script) file. When an experiment is compiled within E-Studio using the Generate command in the E-Run menu, or the Generate tool button, E-Studio automatically generates the script for the E-Objects defined in the ES (Experiment Specification) file into the EBS file. Examination of the EBS file can provide a great amount of information concerning the properties associated with an object, or the methods used to manipulate data related to that object. An EBS file may be



examined using the Full tab in the Script window within E-Studio, or by loading an existing EBS file into E-Run.

4.1.1.1 Introduction to Programming

Regardless of programming experience, every user should follow the same basic steps when writing script.



Design

It is critical to thoughtfully plan the desired result, rather than assuming how it will work. Often the simple act of creating a flow chart to represent the experiment can help to remain focused on the task, and not be overwhelmed with the experiment as a whole. It is extremely helpful to think of an experiment in logical pieces rather than the sum of its parts. This is true of experiments in general, but more so of script.

Implement First Segment

Once the task is broken into segments, begin to implement just the first portion or segment. Often users will try to do too much and they lose sight of the relatively simple steps required to reach the goal. Start simple. For instance, if the goal is to display a question on the screen then collect and score the response, the flowchart might resemble:

- Step 1: display a word on the screen
- Step 2: display a question on the screen
- Step 3: collect a response
- Step 4: score the response.

It is helpful to get through step one before moving on to step two and so on. Start small and build on a solid foundation.



Test

Once the first step is implemented, it **MUST** be tested thoroughly. If a problem is discovered, it is important to fix it now rather than later. Testing is particularly important in research. Precision testing is described in Chapter 2-*Using E-Studio* in this volume.

Debug

Search through the code to find the problem and then correct it. There are many tricks to make debugging less painful than taking a stab in the dark. Debugging tips are included later in this chapter, in section 4.8.

Implement the Next Segment

After the first segment of code is thoroughly tested and there is confidence that it is working as expected, begin to implement the next section of the task.

Test

Repeat the testing procedure. It is important to be confident in the code.

Debug

Repeat the debugging process if necessary.

Keep Testing

Repeat the testing procedure until there is certainty the code is working as designed. When there is confidence in the functionality of the code, be certain to run a few pilot subjects. This is critical, because it is often the case that the subject will uncover a flaw. Rather than losing an actual subject to this disaster, it is best to find the problem with a pilot subject.

4.1.2 Basic Steps

Included in this chapter is a section presenting the *Basics Steps for Writing E-Prime Script* (see Section 4.4). Even if there is familiarity with writing script, and especially if not, the basic steps offer helpful information for scripting within E-Prime. Be sure to review the *Basic Steps for Writing E-Prime Script* section as an effective tool in learning to use E-Prime.

4.2 Introducing E-Basic

E-Basic is truly a standard language, which has been customized to better fit the needs of real-time research. It is nearly identical to Visual Basic for Applications™. Essentially, the only part of VBA that will not transfer to E-Basic is the forms used in VBA.

E-Basic is a standard object-oriented programming language with over 800 commands. Some of the E-Basic commands were specially developed by PST to accommodate the unique requirements of behavioral research. Unlike BASIC, PASCAL or the MEL Language (from MEL Professional), E-Basic is object-oriented. The previous languages were command driven but still resembled ordinary English. E-Basic is object driven and still resembles ordinary English. Comparatively, E-Basic is user-friendly, unlike other more advanced languages (e.g., C++).

E-Basic is used to expand the power of E-Studio in a few ways. First, the E-Basic script generated as a result of the graphical components used to build the experiment may be used as an informative tool. The script required by an entire experiment is generated into an E-Basic script file (EBS) simply by pressing the Generate tool button. Direct editing of this script file is



not recommended, since it defeats the purpose of the graphical interface. In addition, E-Studio always overwrites the EBS file each time it generates, which would discard any edits made directly in the EBS file. However, reading through the EBS file is a good way to become familiar with E-Basic and its components.

The second and third methods of using E-Basic are contained within E-Studio. E-Basic script can be entered in objects that are placed on procedural timelines within the experiment, or E-Basic script may be entered using the User tab in the Script window. The E-Object used for inserting bits of E-Basic into the experiment is the InLine object. User-written script is generally placed in one of three places in an experiment:

1. In an InLine object to be executed at a given time during a Procedure: This is the most common placement of script. The script included in an InLine object is inserted "as-is" into the EBS file. The location is determined by the placement of the InLine object in the structure of the experiment.
2. On the User tab in the Script window to declare global variables: Refer to section 4.3.4- *Variable Declaration and Initialization*.
3. In an InLine placed at the beginning of the experiment to initialize variables: For example, global variables declared on the User tab must be initialized prior to their use, and it is common to do so at the beginning of the experiment.

For details about the InLine object as well as the User tab in the Script window, refer to sections 4.3.4- *Variable Declaration and Initialization* and 4.4- *Basic Steps for Writing E-Prime Script*, as well as Chapter 2- *Using E-Studio*.

4.2.1 Syntax

The E-Basic language is object-oriented. Each object has a list of associated properties and methods. An object appears in code using object.property or object.method where "object" equates to the object's name, and the item after the dot (.) refers to either the object's particular property or method.

4.2.1.1 Objects

Objects are essentially the core components of E-Basic. An object in E-Basic is an encapsulation of data and routines into a single unit. The use of objects in E-Basic has the effect of grouping together a set of functions and data items that apply only to a specific object type.

In E-Studio, there are a variety of E-Objects including TextDisplay, Slide, List and so on. The graphical representations of those objects are accessible in E-Basic using the object.property syntax. In the example below, Instructions is a TextDisplay object. The statement follows the object.property syntax to set the Text property for the Instructions object.

```
Instructions.Text = "Welcome to the experiment"
```

Object.Properties

Objects expose data items, called properties, for programmability. Usually, properties can be both retrieved (Get) and modified (Set). Just as each E-Object in E-Studio has a set of associated properties, so do objects in script. The property of the object is referenced using the object.property syntax.



Essentially, properties store information regarding the behavior or physical appearance of the object. For instance, a TextDisplay object has a Text property, which represents the text to be displayed on the screen.

Object.Methods

Objects also expose internal routines for programmability called methods. In E-Basic, an object method can take the form of a command or a function.

Object.Commands

Commands are quite literally the action or actions that the object can perform. An object command is referenced using the object.command syntax. Commands may or may not take parameters. For example, the Clear command (i.e., object.Clear), which may be used to clear the active display, requires no parameters, while the Rectangle command associated with the Canvas object requires parameters to define the position and size of the rectangle to be drawn. Note, not all commands are available to all objects. Refer to the E-Basic Online Help for a listing of commands available to each object.

Object.Functions

An object method which returns a value is called a function. An object function is invoked using the object.function syntax. Functions may or may not take parameters. For example, the Mean function (i.e., object.Mean) requires no parameters to return the mean of a collection of values contained within a Summation object. However, the GetPixel function associated with the Canvas object requires parameters in order to return the color value at a particular x, y coordinate position.

4.2.2 Getting Help

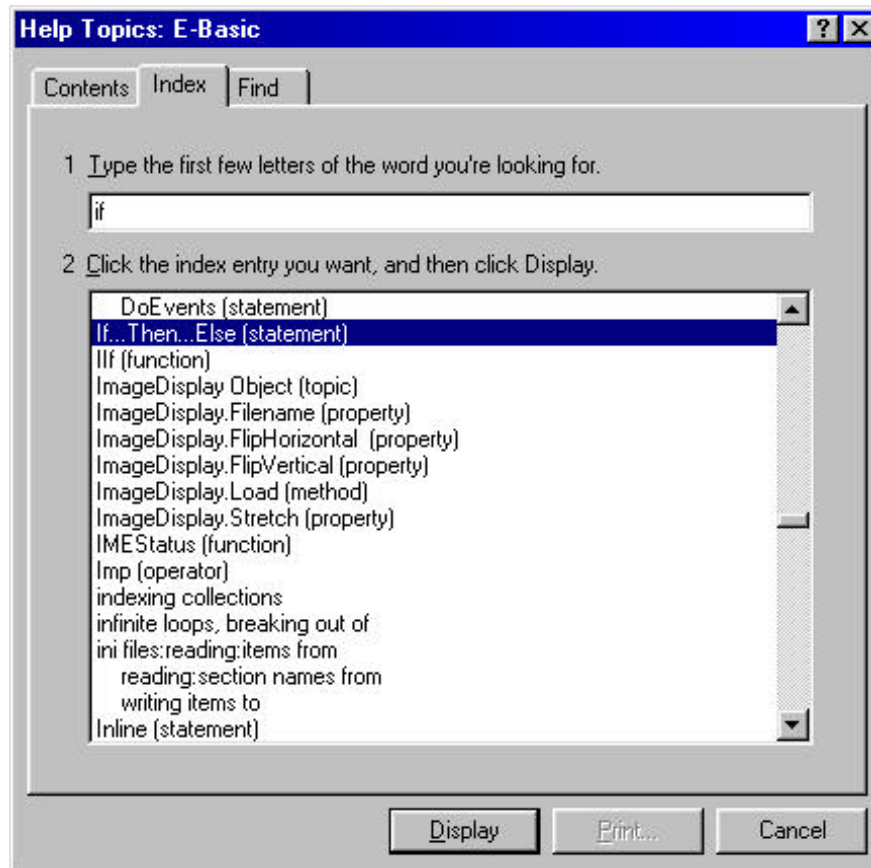
The E-Basic Online Help is launched through the Start button (i.e., select E-Basic Help from the E-Prime menu), or through the Help menu in E-Studio. The E-Basic Help command opens the Help Topics dialog, containing Contents, Index and Find tabs.

4.2.2.1 Contents

The Contents page lists a table of contents for the Help topics available within the E-Basic Online Help system. The main Help topics are displayed as books, which may be opened to display related subtopics. The topics within the Contents list may be expanded or collapsed by double clicking the book. When a topic is expanded, information concerning individual subtopics may be displayed by double-clicking the desired item.

4.2.2.2 Index

The Index page displays an alphabetical listing of topics and commands within E-Basic. This is typically the best way to find a command or function name. The Help information for a particular topic may be displayed by selecting the topic and clicking the Display button, or by double-clicking the topic. The topics may be searched by typing directly in the first field of the Help Topics dialog, or by scrolling through the topics contained in the index using the scroll bar on the right side of the Help Topics dialog.



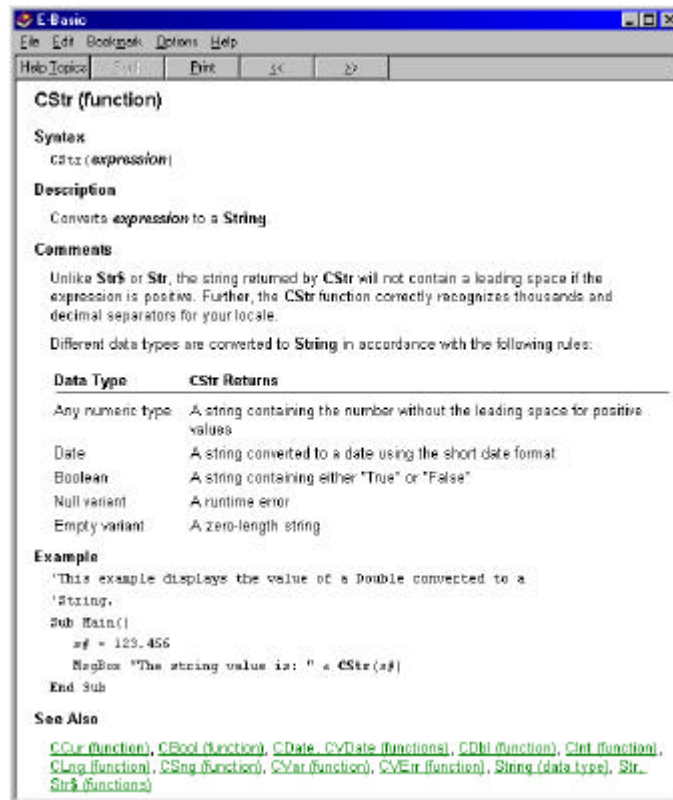
4.2.2.3 Find

The Find page allows searching for specific words or phrases within the Help topics. After typing in a word to search for in the Help topics, the topics in which the word appears are listed, as well as additional suggestions to narrow the search. The Help information for a particular topic may be displayed by selecting the topic and clicking the Display button, or by double clicking the topic.

The Find page is useful for locating all commands that reference a particular topic (e.g., all topics referencing the "Abs" string). However, since any reference to the string searched for is included, this method often gives a long list of topics that must be searched through in order to locate relevant information.

4.2.2.4 Reading Help Topics

E-Basic Help topics are presented using a standard format for all topics. Each description within the E-Basic Help topics describes the use of the statement, function, command or object, the syntax and parameters required, any considerations specific to the topic, an example, and a listing of direct links to related topics.



Section	Purpose
Syntax	Describes the parameters necessary for use with the statement, function, command or object.
Description	Describes the purpose of the statement, function, command or object.
Comments	Lists specific considerations for the statement, function, command or object.
Example	Actual script example illustrating the use of the statement, function, command or object.
See Also	Direct links to related statements, topics, functions, commands or objects.

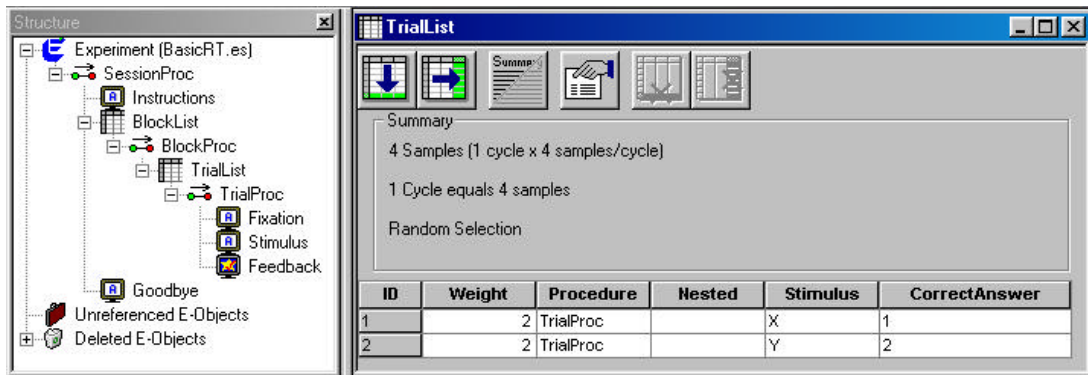
4.2.3 Handling Errors in the Script

For detailed information related to error handling, refer to the section 4.8-*Debugging in E-Prime*.

4.3 Communicating with E-Prime Objects

Before attempting to write E-Basic script, it is useful to understand how information is managed within E-Basic and E-Prime. Within E-Basic, data and routines acting on that data may be encapsulated into units called "**objects**." For example, at the trial level, the List object is an encapsulation of the trial level data (e.g., stimuli, independent variables, etc.) and the routines applied to that data (e.g., TrialProc).

In the image below, each line in the TrialList object lists a specific **exemplar**. The Stimulus and CorrectAnswer attributes contain the trial level data necessary for presenting the stimulus and scoring the input collected from the subject. The Procedure attribute indicates which routine is to be associated with the specific piece of data. In this case, the TrialProc Procedure, which calls individual objects to present a fixation, stimulus, and feedback, is applied to the specific stimulus (i.e., X or Y).

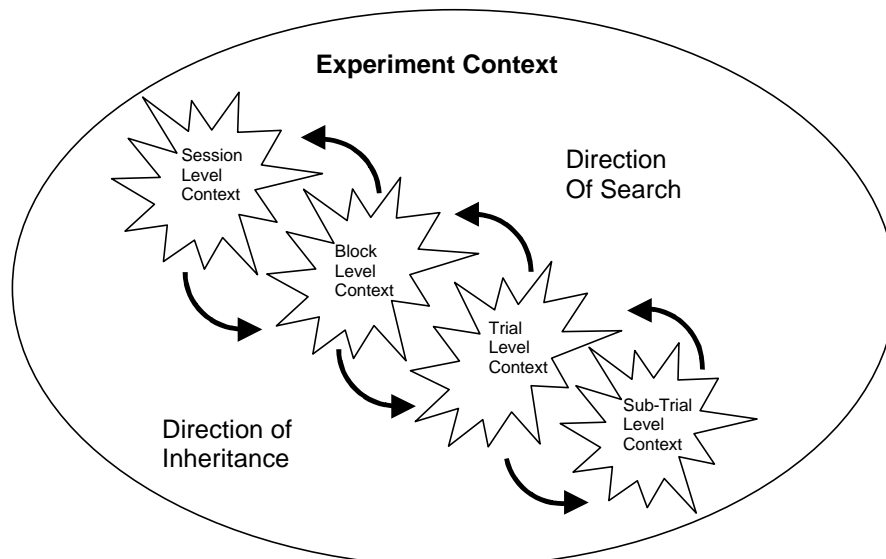


Thus, when the experiment is executed, a specific exemplar is selected from the TrialList object. An exemplar from a List is an entire row (or level). In the current example, a chosen exemplar includes the data to present the stimulus (Stimulus = either X or Y) and score the response (CorrectAnswer = 1 or 2), as well as the Procedure using that data (TrialProc).

The TrialList object calls the TrialProc object (containing the Fixation, Stimulus and Feedback objects) using the Stimulus and CorrectAnswer information from the chosen exemplar. In this way, the trial level information encapsulated in the TrialList object is used to run a series of trials. Likewise, the block level data and routines are encapsulated in the BlockList and BlockProc objects.

4.3.1 Context

As with the block and trial levels, the data and routines involved in the overall experiment are encapsulated in object form at a higher level. The experiment data and associated routines are combined to define the **Context object**. Within the Context object, information is hierarchical. This hierarchical nature allows values from upper levels to be either inherited or reset at lower levels, while search procedures to determine the value of attributes occur at the current level and continue in an upward direction until the information is located.





For example, if an attribute is defined at the block level, this attribute will be available at the block level and all levels subordinate to the block level (e.g., trial, sub-trial, etc.). When values are requested, the current level is searched first, and the search continues upward through the levels until a value is found or until all levels have been exhausted. This defines a hierarchy of context levels within the overall Context. If a value for an attribute is requested during a trial, E-Run will search the trial level context first, then the next level of context (e.g., block level), and so on, to resolve the value for the attribute.

For example, perhaps a series of blocks of trials is to be run, with the instructions for a series of trials changing only when the next block is run. An attribute named "Instructions" likely be created at the block level, since the value of the Instructions attribute would only need to vary from block to block. However, perhaps the instructions are to be displayed at the beginning of each trial as a reminder of the task. A display during the trial Procedure would necessarily reference the Instructions attribute (i.e., [Instructions]), which is defined at the block level. E-Run would first check the trial level context, then the block level context, and then the session level context in order to resolve the value of Instructions. In this example, the search would terminate at the block level, where the Instructions attribute is defined.

4.3.1.1 Attributes

Data within the Context object is manipulated through the creation and setting of attributes. Attributes differ from variables in that they generally define the experimental conditions and are logged by default. The logging may be disabled for individual attributes.

Attributes must be entered into the Context of the experiment before they may be referenced or modified. Although the Context object outlines the overall experiment, the Context is hierarchically structured. Attributes defined at a specific level of the Context may be seen by any level lower than the one at which the attribute is defined, and upper level information may be inherited by lower level attributes. However, attributes defined at lower levels of the Context cannot be "seen" beyond the level at which they are defined. Thus, in order to reference or modify an attribute, it must be placed in the Context and set at the appropriate level.

4.3.2 Object.Properties

Properties are data items associated with objects that may be both retrieved and modified. For example, the Stimulus object in the example above has properties that may be accessed and modified through E-Basic script. Properties of an object are accessed using the dot operator, which separates the property from the object with which it is associated. The example below illustrates the assignment of the Text property associated with the Stimulus object. In this example, the Text property is assigned a value of "X."

```
Stimulus.Text = "X"
```

Not all properties may be modified. Read-only properties may not be modified through E-Basic script. A listing of all properties associated with particular objects is available in the E-Basic Online Help.

4.3.3 Object.Methods

Objects also have associated methods, which cause objects to perform certain actions. Like properties, methods are accessed using the dot operator in conjunction with the object. Methods may be broken down into Commands and Functions.



4.3.3.1 Commands

Commands are used to instruct the program to perform an operation. For example, the Run command is used to launch the object named "Stimulus" within the experiment script.

```
Stimulus.Run
```

Commands may or may not take parameters. A listing of all commands associated with particular objects is available in the E-Basic Online Help.

4.3.3.2 Functions

Like commands, functions are used to instruct the program to perform an operation. In contrast to commands, however, functions are used to perform an operation that returns a value. Functions may also be used within a script statement. For example, a function could be written to calculate the mean value for a set of values. Once defined as a function, a single line of script would be necessary to run the function to calculate a mean.

4.3.4 Variable Declaration and Initialization

4.3.4.1 Declaring Variables

Variables are distinguished from attributes in that they are not specifically related to the Context object. Variables are defined and accessible within a particular scope. That is, variables are temporary, and are discarded after the scope (e.g., Procedure) in which they are defined is exited. Note, variables declared in the User tab of the Script window are defined globally, so their scope spans the entire program.

A **Dim statement** within a Procedure, subroutine or function declares variables locally to that Procedure, subroutine or function. Variables declared within a particular scope are not automatically "seen" outside of that scope. Variables are declared within a particular scope using the Dim statement. Once that scope is exited, the variable is discarded. For example, a variable declared at the trial level with the Dim command is discarded after the trial Procedure is completed, and is not automatically logged in the data file. A variable must be set as an attribute of the Context object in order to be logged in the data file.

Variables declared at the top level of the experiment (i.e., **global variables**) must be declared on the User tab in the Script window using the Dim statement. Global variables may then be initialized within the structure of the experiment using an InLine object. Most commonly, the initialization of global variables would be entered in an InLine object at the beginning of the experiment structure. Variables declared globally may be accessed at any point in the structure of the experiment (i.e., their scope includes the entire experiment).

4.3.4.2 Naming Variables

Variable names must start with a letter, and may contain letters, digits and the underscore character¹⁸. Punctuation is not allowed, although the exclamation point (!) may appear in a position other than the first or last character. If the exclamation point is entered as the last character, it is interpreted as a type-declaration character by E-Basic. Variable names may not exceed 80 characters in length, and cannot be from among the list of **reserved words** (see the Keywords topic in E-Basic Online Help for a listing of reserved words in the E-Basic language).

¹⁸ E-Objects named in E-Studio do not permit the use of the underscore character.



Rules

- Must begin with a letter.
- Numbers are acceptable but may not appear in the first character position.
- Punctuation is not permitted, with the exception of the underscore character and the exclamation point (see above paragraph for details).
- Illegal characters include: @#\$%^&*(){}-+[]=><~`~;`;
- Spaces are not permitted.
- Maximum number of characters is 80.
- Cannot duplicate a reserved word.
- Cannot use the same name more than once within the same scope.
- The backslash character ("\") may not be used. This is an escape character within E-Basic, which signals E-Basic to interpret the character following the backslash as a command (e.g., "\n" would be interpreted by E-Basic as "new line").

As long as the rules above are followed, variables may be named almost anything. However, it is highly recommended that the user follow a logical naming scheme, because this makes programming much easier in the long run. Specifically, give variables logical names to quickly remember their purpose. If the variable is nothing more than a counter, give it a single letter name such as "i," "j" and so on. If the variable's purpose is to keep track of a birth date, name the variable something like "subject_birthdate" rather than "xvariable."

4.3.5 User Script Window vs. InLine Object

The **User Script** window is accessible via the Script command in the View menu. Notice the two tabs at the bottom of the Script window, User and Full. The Full tab is useful for viewing the code generated by E-Studio. It is not possible to edit script on the Full tab. The User tab allows entering the user's own high-level script.

The User Script tab in the Script window may be used to declare global variables. When declaring variables, the variables are only accessible throughout the scope of the Procedure in which they are declared. For example, if a variable is declared at the trial level, the variable can only be seen, or referenced, within the scope of the trial. In order for variables to be accessible at any level of the experiment (e.g., at both the block and trial level), the variables must be declared within the scope of the entire experiment rather than within a specific Procedure. Script on the User Script tab is entered into the EBS file globally.

The User Script tab in the Script window may also be used to declare functions and subroutines. In addition to the accessibility issue described above (i.e., functions and subroutines declared on the User Script tab may be used at any level of the experiment), there is also a syntax constraint related to the declaration of functions and subroutines. The syntax used to declare functions and subroutines (e.g., Sub..End Sub) will interfere with the script automatically generated by E-Prime for the Procedure object if entered outside of the User tab. Therefore, user-defined subroutines may not be entered using an InLine object, and must be entered on the User tab.

InLine objects are used to insert segments of user-written script within an experiment. Script contained within InLine objects is inserted as a unit into the EBS file. The point of insertion is in relation to the location of the InLine object within the structure of the experiment. For example, if an InLine object is called by the trial Procedure, the script contained within the InLine will be inserted in the EBS file at the point at which the trial Procedure is run. As alluded to above, global variables should not be defined in an InLine, rather they should be declared in the Script window on the User tab.



4.4 Basic Steps for Writing E-Prime Script

The following steps, to be covered in this section, introduce helpful information for writing E-Basic script or accessing values determined through script.

1. Determine the purpose and placement of the script
2. Create an InLine object and enter the script
3. Determine the scope of variables and attributes
4. Set or reference values in script
5. Reference script results from other objects
6. Debug
7. Test

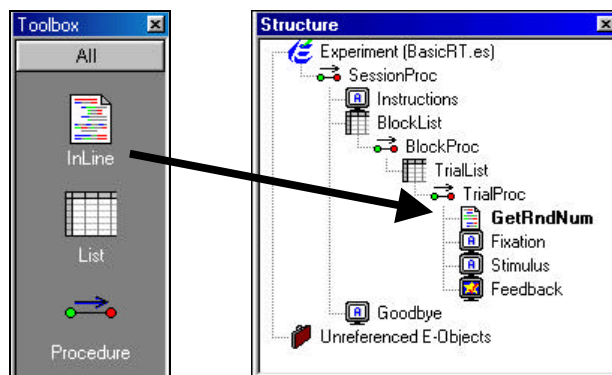
The basic steps refer specifically to writing script in an E-Prime experiment. For additional information, it is recommended that the user continue in the current chapter, working through the *Programming* sections.

4.4.1 Determine the purpose and placement of the script

This step requires the user to consider what task the script is supposed to accomplish and when the action must occur. For example, an experimenter might want to choose a random number from 1-499 to be used as the stimulus display for each trial. The purpose of the script, therefore, is to select a random number and to place the selected value into a form useable by E-Prime. The placement of the script must be within the Procedure driving the events of the trial. Specifically, since the random number is to be used during the display of the stimulus, the script determining the value must be entered in the trial Procedure prior to the object displaying the stimulus.

4.4.2 Create an InLine object and enter the script

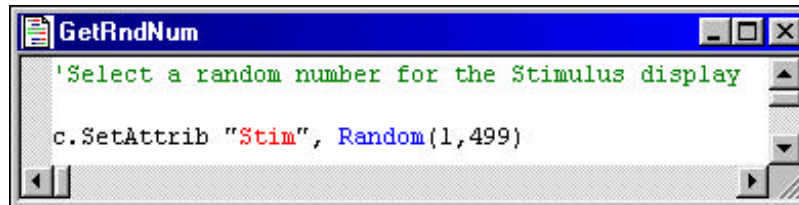
Once the appropriate placement of the script has been determined, create an InLine object at that location. Continuing the example from Step 1, if the script is to determine a random number during the trial Procedure (TrialProc) prior to the stimulus display, the most appropriate location for the InLine object containing this script is as the first event in the TrialProc.



Actually, any time prior to the event displaying the stimulus would be appropriate, but it is good practice to separate the setup events from the critical events of the trial (e.g., Fixation-Stimulus-Feedback).



After placing an InLine object in the appropriate location, double click the InLine object to open it, and enter the script required to accomplish the task. To select a random number (1-499), and to place the selected value into a form useable by E-Prime, the following script would be entered:

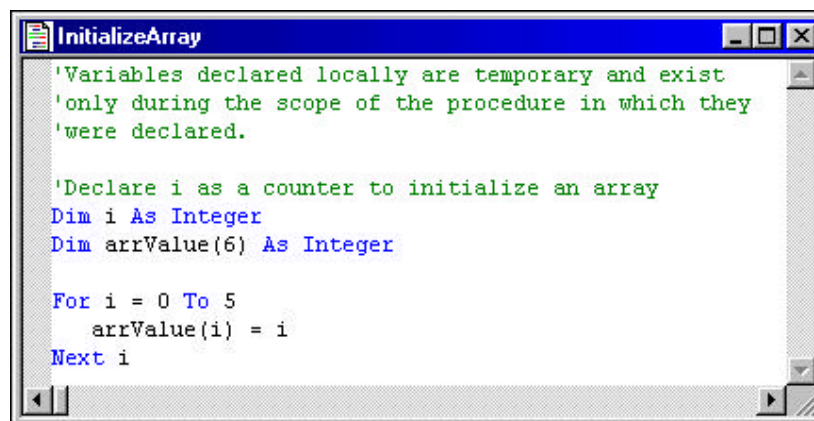


```
GetRndNum
'Select a random number for the Stimulus display
c.SetAttrib "Stim", Random(1,499)
```

The Random function is used to select a random value from 1-499, and the SetAttrib command is used to place that value into an attribute for later access. Notice the single quote character used to enter a comment in the script above. The single quote character is used to skip all characters between the apostrophe and the end of the current line. Similarly, the Rem statement may be used to enter comments. Refer to the Contents topic in the E-Basic Online Help for more information.

4.4.3 Determine the scope of variables and attributes

Variables or attributes declared within an E-Prime experiment are limited to the scope of the Procedure in which they are defined. Variables to be used only during a single trial may be declared and initialized at the trial level. For example, a variable used as a counter may be declared locally, as in the following example:



```
InitializeArray
'Variables declared locally are temporary and exist
'only during the scope of the procedure in which they
'were declared.

'Declare i as a counter to initialize an array
Dim i As Integer
Dim arrValue(6) As Integer

For i = 0 To 5
  arrValue(i) = i
Next i
```

When InitializeArray is placed in the trial Procedure, both "i" and "arrValue" will be accessible during the trial. At the end of the trial Procedure, the "i" and "arrValue" variables will be discarded (e.g., the value of arrValue will not be accessible by an event at the block level).

If a variable is to be maintained or accessed across multiple executions of a Procedure (e.g., performance over a series of blocks), the variable must be declared globally. Global variables are declared using the User tab in the Script window. For example, a variable might be declared to keep track of the total number of trials. Use the View menu to display the Script window in E-Studio, and select the User tab to declare a global variable.



```
'Global variables must be declared in the User Script
'Declare global variable trial count
Dim g_nTotalTrial as Integer
```

Initialization of global variables cannot occur on the User tab; instead, an InLine object would be used for this purpose. To initialize a global variable prior to its use (e.g., to initialize the number of trials to 0), use an InLine object placed appropriately. It is a good practice to initialize variables as the first event in the Procedure in which they will be used as part of the Procedure setup events. In this case, global variables exist throughout the scope of the experiment, so initialization should take place as the first event in the Session Procedure.

The Structure window shows a tree view of the experiment hierarchy: Experiment (BasicRT.es) > SessionProc > InitializeVariables > Instructions > BlockList > BlockProc > TrialList > TrialProc > Fixation > Stimulus > Feedback > Goodbye > Unreferenced E-Objects. An arrow points from the InitializeVariables object to a script window titled 'InitializeVariables' containing the following code:

```
'Initialize global variables at the
'beginning of the Session Procedure

g_nTotalTrial = 0
```

The global variable may then be updated during the trial level Procedure to maintain a count across multiple executions of the Procedure. To use the global variable to update the trial count, insert an InLine object as the first event in the trial Procedure, and enter script to increase the count.

The Structure window shows the same hierarchy as above, but with an additional object, 'TotalTrialCount', added under the TrialProc object. An arrow points from this object to a script window titled 'TotalTrialCount' containing the following code:

```
'Update the total trial count
'Set value as an attribute for display

g_nTotalTrial = g_nTotalTrial+1
c.SetAttrib "TrialCount", g_nTotalTrial
```

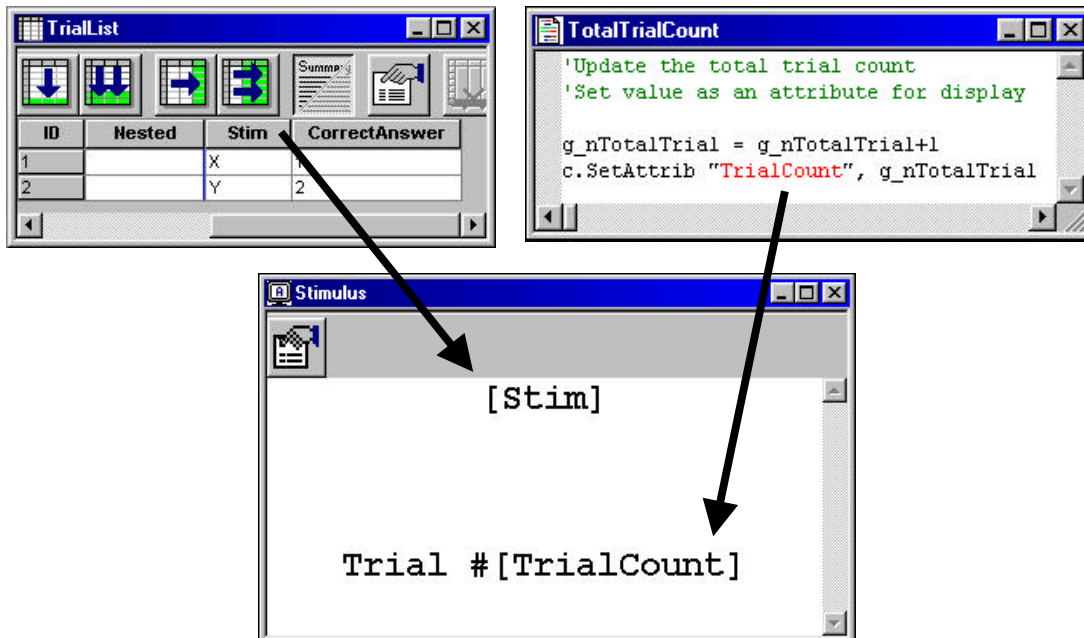


The total trial count variable is updated simply by increasing the previous value by one (i.e., `g_nTotalTrial+1`). The second line of script referring to the `c.SetAttrib` command is used to place the value of `g_nTotalTrial` into an attribute for access by another object and to log the value in the data file. Refer to the next step for information pertaining to setting values as attributes.

4.4.4 Set or reference values in script

If variables are to be retrieved and/or logged, they must be placed into the experimental context as attributes. Otherwise, they exist only temporarily, and are discarded at the conclusion of the Procedure in which they are defined. Attributes defined in a List object are automatically placed into the experimental context. For example, if a List object defines an attribute named "Stim," and the values of Stim are used to define the stimulus displayed during each trial, the value of Stim for each trial will automatically be logged in the data file.

In order to enter a temporary variable (e.g., `g_nTotalTrial` from the previous step) into the experimental context, either for the purpose of logging that value in the data file or for later access by another object, use the `c.SetAttrib` command. In the previous step, the value of the `g_nTotalTrial` variable was set as an attribute (i.e., `c.SetAttrib "TrialCount", g_nTotalTrial`). The TrialCount attribute may then be accessed by another object using bracket notation, and the value of TrialCount will be logged in the data file.



The values of attributes and properties from the experimental context may be accessed via script using the `c.GetAttrib` command. For example, the stimulus might display a random number selected from a range of values depending on the condition. "Condition" could be entered as a List attribute defining the possible conditions, and this value could be retrieved at runtime to set the possible range of values for the random number selection.

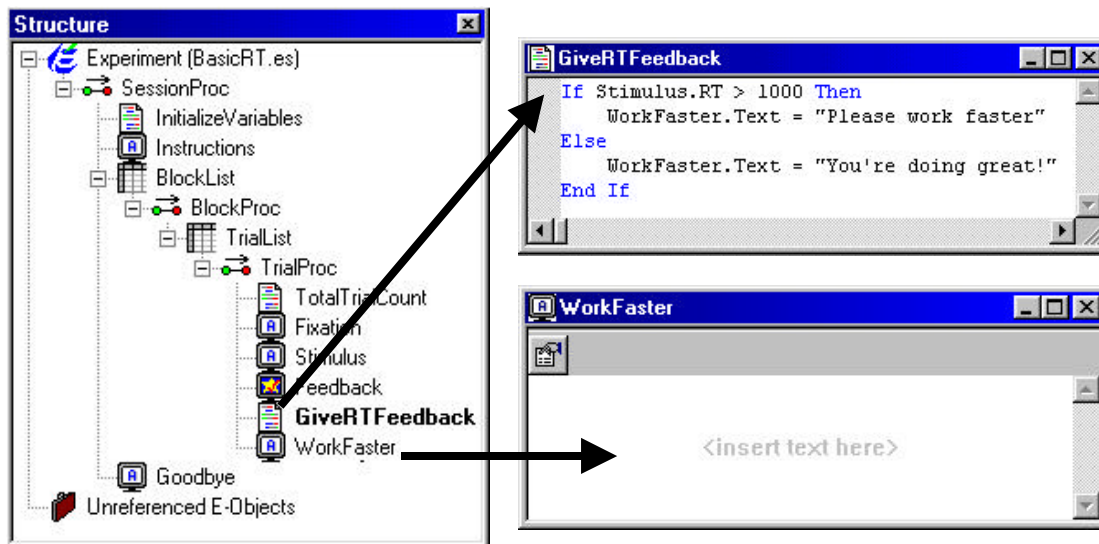


```
SetStimPerCondition
'Set the value of Stim in relation
'to the Condition attribute

Select Case c.GetAttrib("Condition")
  Case "below500"
    c.SetAttrib "Stim", Random (101,499)
  Case "above500"
    c.SetAttrib "Stim", Random (501,899)
  Case Else
    MsgBox "Bad Condition: " & c.GetAttrib("Condition")
End Select
```

Note the use of the "Else" condition in the script above. It is a good practice, and ultimately the programmer's responsibility, to cover all possibilities when writing script. If the values of the Condition attribute are carefully entered, the "Else" condition should not be necessary. However, it is better to consider the possibility of error than to have the program fail. Here, we put up a message box to tell the experimenter a bad stimulus condition has occurred.

The properties of objects may also be set or retrieved through script as long as the property is not read-only or design-time only (i.e., not able to be modified at runtime). For example, it is possible to vary messages presented at runtime based on the speed of the response collected. Such a procedure would require accessing one value (i.e., the reaction time from the input object Stimulus.RT), and setting another (i.e., the text to be displayed by the object presenting the message).



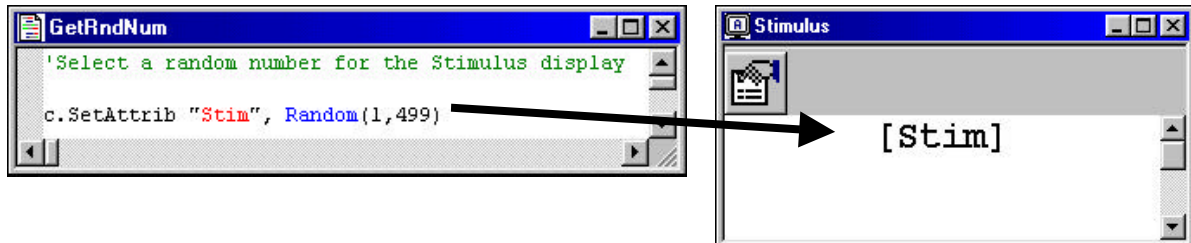
The GiveRTFeedback Inline object containing the script above sets the Text field for the WorkFaster TextDisplay object at runtime. Thus, no value need be entered for the Text field in the WorkFaster object in E-Studio.

4.4.5 Reference script results from other objects

After a variable has been entered into the experimental context as an attribute, that attribute may then be accessed by other E-Prime objects occurring within the same scope. For example, once



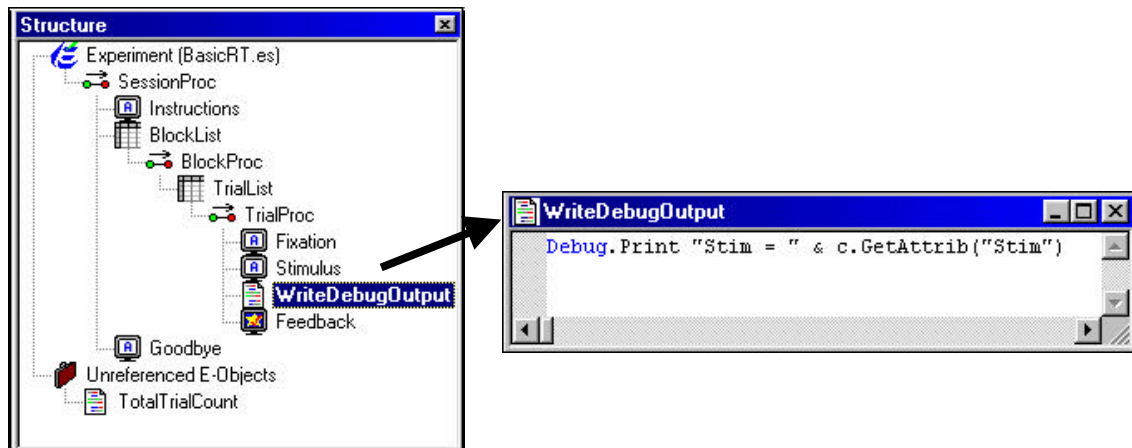
the random value has been set as an attribute at the beginning of the trial Procedure, that attribute may be referenced by a TextDisplay object in order to display the value as the stimulus during the trial. To refer to an attribute, use square brackets surrounding the attribute name (e.g., [Stim]) in the Text field of the TextDisplay object.



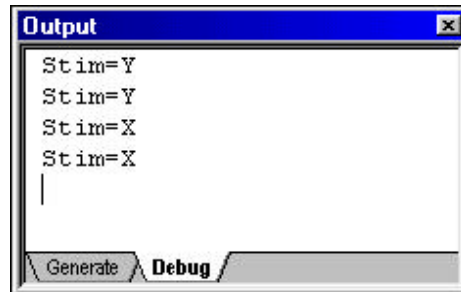
Note most properties may be changed through script as well. For example, to change the location of the display area defined by a TextDisplay object, the X property for the TextDisplay could be set as `TextDisplay1.X = 100`. This command sets the horizontal location of the display area to begin at pixel location 100.

4.4.6 Debug

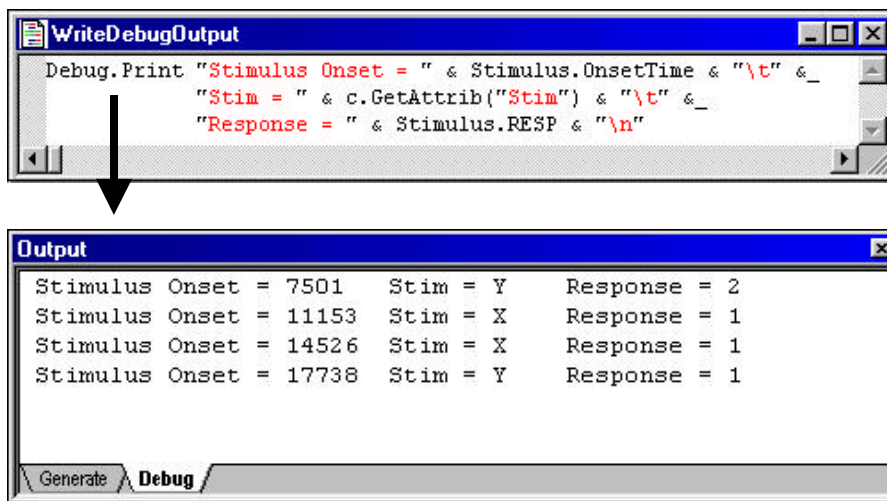
Debug commands are useful in tracking down problems, verifying values, or simply reviewing the execution of the program. The Debug object, when used with the Print method, is used to send information at runtime to the Debug tab of the Output window for examination following the run. For example, the following script could be used to write the value of the Stim attribute to the Debug tab in the Output window during each trial.



The script above will send the value of the Stim attribute per trial to the Debug tab in the Output window. After the run, the Debug output may be viewed by displaying the Output window within E-Studio. Select Output from the View menu, and in the Output window, select the Debug tab.



Debug.Print may also be used to monitor timing and response events. The following script uses Debug.Print to write the stimulus onset time and the subject's response to the Output window.



The contents of the Debug tab may be copied to the clipboard by right clicking in the Output window. Then, the Debug output may be pasted into Excel® to print it or use spreadsheet calculations to check the timing of events.

The Debug object is most appropriate during the development and testing of new programs. After an experiment has been fully tested, the Debug commands may be disabled by setting the Debug object's Enabled property to "false" (i.e., Debug.Enabled = false). This allows the user to leave Debug commands in the script for future testing purposes, but to improve the efficiency of the program during data collection.

A final method of debugging involves the use of comments to indicate the purpose of certain sections of script. This method may be employed more often as a preventative measure than as a diagnostic tool, but can be of tremendous importance during debugging (especially if someone other than the author of the script is given the task of debugging). There is no standard style for adding comments, but in general they should be brief descriptions of the purpose of the segment of script. Many of the script examples used in this chapter make use of brief comments to summarize the purpose of the script for the reader. In E-Basic, comments are separated from script that is compiled through the use of the apostrophe (') keyword or the Rem statement. Refer to the Comments topic in the E-Basic Online Help within E-Prime for further discussion of comments.



4.4.7 Test

After the experiment is functioning without errors, it is important to completely test it prior to running subjects. The Debug object is especially useful for testing an experiment. Use the Debug.Print command to send variable values to the Output window, and keep track of the progress of the experiment with a pencil and paper as well. After running the experiment, examine the output on the Debug tab to view the values. Or, the contents of the Debug tab may be copied to the clipboard and pasted into another application for more thorough viewing or analysis. A written-record and the Debug output should be compared to the data file to verify that all of the variables are being logged, the values are in the correct range, and the sampling is occurring as expected. It is important to test the script completely, including tests of unlikely responses and invalid ranges. For more discussion concerning debugging and testing, refer to *Stage 7: Testing the Experiment* in Chapter 2-Using E-Studio.

4.5 Programming: Basic

4.5.1 Logical Operators

Logical operators allow comparison and conditional execution (e.g., If trial > 5 Then...). Often, in script, expressions are used, which compare items using simple mathematical expressions.

>	Greater than
<	Less than
=	Equal to
>=	Greater than or equal to
<=	Less than or equal to
<>	Not equal to

Logical operators are also commonly used in compound expressions. Essentially, logical operators evaluate two expressions and use a set of rules to determine if the total expression is true or false.

Operator	Returns True	Example	Result
And	If both expressions are true.	5 > 2 And 6 + 3 = 9	True
		3 * 3 = 9 And 7 < 6	False
Or	If either expression is true.	5 > 7 Or 8 * 2 = 16	True
		8 < 4 and 3 > 2	False
Xor	If only one expression is true. Note that it is False if expressions are either both true or both false.	3 + 2 = 5 Xor 5 + 5 > 10	True
		3 + 2 = 5 Xor 5 + 5 = 10	False

4.5.2 Flow Control

Controlling the flow of the script is a critical component of programming. There are two major types of controlling the flow: conditional statements and loops.

4.5.2.1 Conditional Statements

Conditional statements determine or specify which part of the script should be executed based on a condition. Contingent branching is often utilized in behavioral research and is based on whether a specific item is true or false. Specifically, If...Then statements are very commonly used to control the flow of the code.



If...Then statements simply are equated to making a choice (e.g., if I have money, then I can go to the movies). If...Then statements are the foundation of all logic. Although they seem to be very simple statements, they are quite powerful in a programming language.

There are actually 3 conditional expressions available in E-Basic. They are: If...Then, Select Case, and Do...Loop. These expressions are conditional statements simply because they perform a task based on a single true or false test. The If...Then and Select Case constructions will be discussed in this section. The discussion of Do...Loop is reserved for section 4.5.2.2 *Loops* in this chapter.

If...Then statements

The single most commonly used flow control statement is If...Then. Simply put, an If...Then statement will execute a block of code if the condition is true. If the condition is false, it will do nothing unless "Else" is used in the flow control statement.

```
If condition Then
    <Block of code statements to execute if the condition is true>
End If
```

Rules:

- Notice the "Then" portion of the statement is on the same line as the "If" portion. If there is a need to drop it to the next line, indicate that to the E-Basic compiler. This is done by placing an underscore (i.e., E-Basic line continuation character) at the end of the line to be continued.
- The End If statement is critical. It identifies the last statement in the block of code to be executed based on the condition.
- When the If...Then statement is only a one-line statement, the End If is not used. In fact, it will produce an error if used. In the following two examples, the code works exactly the same, but it is syntactically different.

Example 1: One-line If...Then statement

```
Dim a As Integer
a = 12

If a > 10 Then MsgBox "A is greater than 10."
```

Example 2: If...Then...End If statement

```
Dim a As Integer
a = 12

If a > 10 Then
    MsgBox "A is greater than 10."
End If
```



If...Then...Else statements

If the program must choose between two alternative blocks of code to execute based on the conditional, then the Else statement is included.

```
Dim a As Integer
a = 12

If a > 10 Then
  MsgBox "A is greater than 10."
Else
  MsgBox "A is not greater than 10."
End If
```

Select Case statements

Nested If...Then statements are ideal for testing different values before executing the next block of code. Select Case statements are more appropriate for testing the same value against many different conditions.

```
Select Case variable
  Case test1
    <block of code statements to be executed if the
      value of variable meets test1 criteria>
  Case test2
    <block of code statements to be executed if the
      value of variable meets test2 criteria>
  Case Else
    <block of code statements to be executed if the
      value of variable doesn't meet any of the
      listed Case criteria above>
End Select
```

The Select Case structure indirectly uses condition expressions. An expression may be $A + B > C$. In the example above, think of the variable being what is to the left of the operator ($A + B$) and test as everything to the right, including the operator ($>C$).

Rules:

- An unlimited number of cases may be used as test criteria.
- The Else case is optional. It is not required to be used, but can be useful in behavioral research.

4.5.2.2 Loops

The **loop** flow control structures are also quite commonly used in programming. They are useful when a block of code needs to be executed more than once. There are three major types of loops available in E-Basic: Do...Loop, For...Next, and For Each...Next.

Type of Loop	Function
Do...Loop	Repeats the block of code until a condition is true.
For...Next	Repeats the block of code a specified number of times.
For Each...Next	Repeats the block of code for each object within a collection.



Do...Loops

There are a variety of Do ...Loops available in E-Basic.

Statement	Description
Do...Loop	Repeats the block of code until a condition is true and then executes an End Do statement.
Do...While...Loop	Repeats the block of code only while a condition is true.
Do Loop...While	Executes the block of code once and then repeats it until the condition is false.
Do Until...Loop	Executes and repeats the block of code only while the condition is false.
Do...Loop Until	Executes the block of code once and then repeats it until the condition is true.

Do While... Loop

The most typically used Do...Loop is the Do While...Loop. The basic syntax is:

```
Do While condition
  <block of statements that are executed while condition is true>
Loop
```

E-Basic evaluates the condition when it encounters a Do While statement. If the condition is true, then the block of code within the Loop structure will be executed. When it reaches the Loop statement, the entire process is repeated including reevaluation of the condition. When the condition is false, the entire loop structure is skipped and E-Basic executes the next statement immediately following the Loop statement of the structure. In theory, no limit exists on the amount of times the block of code within the structure may be executed.

Do...Loop While

The only difference between a Do While...Loop and a Do... Loop While is the location of the condition. The Do While ...Loop evaluates the condition *before* executing the block of code within the structure. The Do Loop...While evaluates the condition *after* executing the block of code within the structure. The While in this case determines if the block of code within the structure should be repeated based on the value of the condition. The major resulting difference is that a Do... Loop While will always execute the block of code at least once.

```
Do
  <block of statements that are executed while condition is true>
Loop While condition
```

The Do...Loop While structure is useful when the block of code within the structure sets a value for the condition before it is evaluated. This structure is also useful when performing an action on an item which has more than one element (e.g., a string or an array). Since the item has at least one element, execute the block of code at least once and then repeat it based on the total number of elements within the item.

Do Until...Loop

The Do Until Loop is essentially equivalent to the Do While...Loop structure. They both execute a block of code after evaluating a condition. The Do While structure will repeat a block of code until the condition is false. A Do Until structure repeats a block of code until the condition is true.

```
Do Until condition
  <block of statements that are executed while condition is true>
Loop
```



Do...Loop Until

Like the Do While Loop varieties, the Do Until Loop also offers the option of setting when the condition is evaluated. In this case, the condition is evaluated at the end of the block of code. Therefore, the block of code is executed at least once.

```
Do
  <block of statements that are executed while condition is true>
Loop Until condition
```

Do Loops with If..Then or Select Case statements

Exit Do

Occasionally, the Do Loop structure doesn't quite meet the need for the intended flow. For instance, a loop may need to be broken immediately within the block of code contained within the structure. This is accomplished by nesting an If...Then...End If or Select Case structure within the block of code executed by the Do Loop.

```
Do While condition1
  If condition2 Then
    Exit Do
  End If
  <block of statements that are executed while condition is true>
Loop
```

Exit Do is particularly helpful in debugging code. Specifically, Exit Do will allow the loop to be bypassed without having to manually comment out the entire Do Loop structure.

Do

While the previously described varieties of Do Loops evaluate a condition at either the beginning or the end of a block of code to be executed (and possibly repeated), it is also possible to evaluate the condition within the actual block of code itself. This requires the use of a nested If..Then or Select Case structure.

```
Do
  (block of statements to be executed while in the Loop structure)
  If condition Then
    Exit Do
  End If
  <block of statements that are executed while condition is true>
Loop
```

This process is useful when part of the block of code within the loop should be executed, but not the entire block of code.

For...Next Loops

If the number of times a block of code should be repeated is definite, a For...Next loop structure is most appropriate. With this structure, the loop is repeated based on the start and end values supplied. These values can be integers, variable or expressions. A counter is used to keep track of the number of times the loop is repeated.

```
For counter = start To end
  <block of statements to be executed>
Next counter
```



When the loop begins, the counter is set to start. When the Next statement is executed, the counter is incremented by one. When the counter is equal to the end value supplied, the loop terminates and the next line of code outside the loop structure is executed.

Tips:

- Keep it simple. Unless there is a specific reason to start the counter at another value, use 1 to n.
- Although the counter variable is not required to follow the Next statement, it is good practice to include it. It may seem verbose, but makes parsing through the code a bit easier.
- Avoid changing the value of the counter within the loop structure manually. Let the Next statement increment the counter unless there is a specific reason to change the counter (e.g., set it to the end value to terminate early).
- For...Next loops are particularly useful when working with arrays. An array is similar to a storage bin with numbered slots. Arrays are covered in more detail in section 4.7-*Programming: Advanced*.

Exit For

Similar in concept to Exit Do, the Exit For statement allows the loop to terminate early. This is typically used in conjunction with If..Then and Select Case statements within the For...Next loop.

For Each...Next Loop

This version of the For...Next Loop is similar to the previous version. However, the primary distinction is that it is used to perform a block of code for each element within a set, rather than for a specified number of times. For Each...Next requires an element or variable that corresponds to the object types within the collection.

```
For Each variable In collection  
  <block of statements to be executed>  
Next variable
```

Notice a counter is not specifically used in this version of a For Loop. Instead, E-Basic figures out how many times to repeat the block of code based on the items in the collection specified. This is particularly useful when debugging. If a problem is suspected with perhaps one of the TextDisplay objects within a block, try 'stepping' through the processing of each object displaying a marker to the screen to help track down the problem.

4.5.2.3 Interrupting the Flow

GoTo Label

When a design calls for jumping to another place within the script based on a specific flag, the Goto statement is useful. In behavioral research, this is often required for contingent branching experiments. For example, perhaps the execution flow should continue uninterrupted until a subject responds by pressing "a" key. If the subject presses any other letter, the execution flow should jump, or Goto a specific location within the code. In the example below, a Label is placed prior to an input statement (i.e., AskBox). The input is examined, and if it is not the required input, the execution of the program jumps back to the Label at the beginning of the script in order



to perform the response collection again. If the required input is entered (i.e., "a"), the program jumps to a point later in the script.

```
Dim answer As String
LabelB:
answer = AskBox ("Type in a letter:")

If answer = "a" Then
  Goto LabelA
Else
  MsgBox "That is the wrong letter, try again!"
  Goto LabelB 'Ask for another letter
End If

LabelA:
MsgBox "Way to go!"
```

4.5.3 Examples and Exercises

To begin adding user code, it is important that the basics of programming are conceptually understood completely. The following examples illustrate the simplicity of E-Basic code at its best. Be sure to follow through these examples before progressing to more advanced topics. It is recommended that each user actually implement each example, run it and verify that it works as expected before moving on to the next section.

4.5.3.1 Example 1: Display "Hello World" on the screen

For this example to work, create a new E-Studio experiment which has only an InLine object on the SessionProc. The InLine should contain the script provided below.

```
'The following statement will display a dialog box on the
'screen with the text "Hello World." By Default, an OK
'button is also displayed.
MsgBox "Hello World"
```

The result of the above example is:



4.5.3.2 Example 2: Setting Attributes in the Context Object

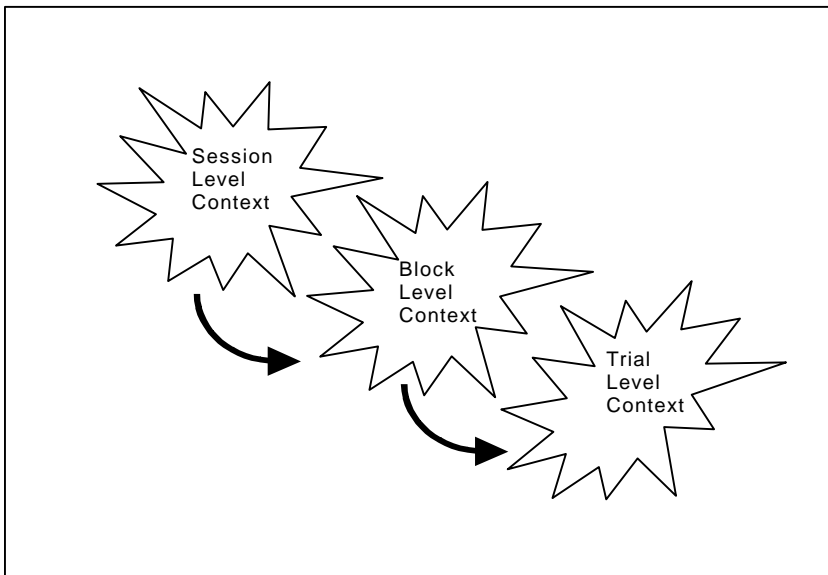
The `SetAttrib` method is used to create an attribute and assign it a value. Thus, the `SetAttrib` method is used to place the attribute in the context so that it may be assigned, modified, and referenced. In the example below, "c" has been defined as the Context object. This is done internally by E-Prime, and need not be explicitly entered into the script. Thus, the `SetAttrib` method is used in conjunction with the dot operator to declare `TotalTrial` as an attribute of the context object ("c") and assign it a value of 10.



```
'Set TotalTrial=10  
c.SetAttrib "TotalTrial," 10
```

Once an attribute is put into the context, the information is logged in the data file and the attribute may be used to display information by way of a display object (e.g., a TextDisplay).

The `TotalTrial` attribute will be available in the context during the scope of the level at which it has been defined. For example, if `TotalTrial` is defined (using `SetAttrib`) during the block Procedure, it will be available during the context of the block level, and any context levels subordinate to the block (e.g., trial level, sub-trial, etc.). However, outside of that scope, the `TotalTrial` attribute value will not be available. There is no backward inheritance possible, which would allow higher levels to inherit attribute information from lower levels.



An attribute must be defined before it is referenced, or error messages will result indicating that the attribute does not exist. For example, if an attribute is defined at the trial level, and referenced at the block level (prior to the trial), an error will occur related to the declaration of the attribute.





The inheritance of the value to assign to an attribute follows the hierarchical structure, and values may only be inherited by levels lower than the level at which the attribute is defined. Thus, if an attribute is defined at the block level, it may be referenced at the trial or sub-trial level.

Inheritance occurs in a downward direction, while the search for a value occurs in an upward direction. For example, if an attribute is defined at the block level and referenced at the trial level, the value at the trial level will be inherited from the block level (i.e., downward). The resolution of the value occurs by first searching the trial (i.e., current) level, then continuing the search at the next highest level (e.g., block), and upward until the value is resolved.

4.5.3.3 Example 3: Getting Attributes from the Context Object

The `GetAttrib` method is used to retrieve a value for an existing attribute in the context. Like `SetAttrib`, the `GetAttrib` method is used in conjunction with the dot operator.

In the example below, the `TextDisplay` object displays a stimulus "X" or "Y." In the script, the `GetAttrib` clause is used to evaluate the current stimulus and set its display color. The `GetAttrib` method will retrieve the value of the current stimulus. When `GetAttrib` returns a value of "X" the `ForeColor` property of the `TextDisplay` will be set to "Green," so that all X's will be displayed in the color green. When `GetAttrib` returns a value of "Y," the `ForeColor` property is set to "Blue" so that all Y's will be displayed in the color blue.

```
'Retrieve value of "Stimulus" and set display color  
  
If c.GetAttrib ("Stimulus") = "X" Then  
    TextDisplay.ForeColor = CColor("Green")  
ElseIf c.GetAttrib ("Stimulus") = "Y" Then  
    TextDisplay.ForeColor = CColor("Blue")  
End If
```

4.5.3.4 Example 4: Global Variables

Often, it is desirable or useful to determine the subject's performance over the course of the experiment, or perhaps after a specified number of blocks or trials. One method of assessing performance is to use a `FeedbackDisplay` object, which can automatically calculate summary statistics (e.g., mean accuracy, mean RT, etc.). Another method of assessing performance involves the use of the `Summation` object. This method involves more involvement from the user, but does not require the user to provide feedback. The example below uses a `Summation` object to determine average accuracy after a specified number of trials.

To use a `Summation` object to determine the average accuracy per condition or block, the `Summation` object must be declared in the Script window on the User tab. In most cases, accuracy would only be examined after a minimum number of trials. Thus, in order to start evaluating mean accuracy after a certain number of trials had been run, it would also be necessary to declare a counter to manually count the number of trials that have occurred. In the script below, the `PracticeProp` summation variable is declared for use in evaluation of the practice trial performance. The `TrialCount` integer variable is declared in the User Script window as well, to keep track of the running trial count.

```
'Declare Variables  
Dim PracticeProp As Summation  
Dim TrialCount As Integer
```



Once declared in the User Script window, the variables are available globally, or for the scope of the entire experiment. The variables must be initialized prior to the point in the experiment at which they are referenced. This would be accomplished using an InLine object, inserted on the Session Procedure timeline. The InLine may occur at any point prior to the referencing of the variables; however, it is a good practice to insert the initialization InLine as the first event in the Session Procedure. This InLine will serve to set up or initialize any variables that will be used.

In the example below, the Set command is used to initialize the PracticeProp summation variable. This command defines the variable as a new instance of an existing object type. The TrialCount variable is initialized to zero since no trials have yet been run.

```
'Initialize Summation Variable
Set PracticeProp = New Summation

'Initialize TrialCount Variable
TrialCount = 0
```

Once initialized, the variables may be assigned values and referenced within the context in which they were defined. In this case, the defined context was the top-level context (i.e., User tab at the experiment level). Thus, the variables are defined globally and may be referenced at any point in the program.

The script below illustrates how the Summation and counter variables are assigned values on each trial. The counter is manually incremented by one on each trial. The Summation variable collects accuracy statistics across trials during the entire block. In the script below, the responses are collected by the "Stimulus" object. Thus, the InLine that contains this script would follow the Stimulus object on the trial Procedure.

```
'Increase counter by 1
TrialCount = TrialCount + 1

'Add accuracy stats to summation variable
PracticeProp.AddObservation Cdbl (c.GetAttrib("Stimulus.ACC"))

'When trial count = 5, evaluate accuracy stats
If TrialCount >= 5 Then
  'If accuracy is 80% or better exit block
  If PracticeProp.Mean >= .80 Then
    TrialList.Terminate
  End If
End If
```

At five trials or more, the mean of the values collected by the Summation is evaluated. If mean accuracy is greater than 80%, the currently running List (i.e., TrialList) is terminated. Thus, the script examines the overall accuracy of the block and terminates the block when a minimum accuracy of 80% is reached.

4.5.3.5 Example 5: Trial Level Variables

Declaring global variables permits the reference of these variables at any point in the experiment. Variables to be used only within a specific context (e.g., block or trial Procedure) may be declared using the Dim command in an InLine object in the appropriate Procedure. In the script below, the Dim command is used to declare a variable to be used to collect a response on each trial. This script is entered in an InLine object, which is called from the trial Procedure object.



```
'Collect response from AskBox
Dim strAnswer As String

strAnswer = AskBox ("Type in the recalled word:")
RecallStim.RESP = strAnswer
RecallStim.CRESP = c.GetAttrib("CorrectAnswer")
```

4.5.3.6 Example 6: Using Attribute References to Pass Information

Attributes allow the passing of variable information during a Procedure. For example, in order to vary the stimulus presented per trial, the stimulus values (e.g., text, name of the picture file, name of the audio file) would be entered as separate exemplars for an attribute (e.g., Stimulus) on a List object. In order to obtain the value of a specific exemplar within the Stimulus attribute listing, the `c.GetAttrib` command is used. In the example below, a basic Stroop task is presented in which the display color of the word varies per trial. The valid display colors (Red, Green, Blue) are entered as RGB values in the "Ink" attribute on a List object.

Summary
15 Samples (1 cycle x 15 samples/cycle)
1 Cycle equals 15 samples
Random Selection

ID	Weight	Proced	Nested	Word	Color	Congruency	Ink	Answer	Ans1
1	3	TrialProc		red	red	congruent	255,0,0	red	1
2	1	TrialProc		red	green	incongruent	0,255,0	green	2
3	1	TrialProc		red	blue	incongruent	0,0,255	blue	3
4	1	TrialProc		green	red	incongruent	255,0,0	red	1
5	3	TrialProc		green	green	congruent	0,255,0	green	2
6	1	TrialProc		green	blue	incongruent	0,0,255	blue	3
7	1	TrialProc		blue	red	incongruent	255,0,0	red	1
8	1	TrialProc		blue	green	incongruent	0,255,0	green	2
9	3	TrialProc		blue	blue	congruent	0,0,255	blue	3

Then, an InLine object is created to retrieve this information, and is inserted on the Trial Procedure. The InLine object uses the `c.GetAttrib` command to retrieve the value of "Ink," and assign that value to the ForeColor property for the Stimulus TextDisplay object.

```
Stimulus.ForeColor = CColor(c.GetAttrib("Ink"))
```

The result is the variation of the display color of the stimulus per trial based on the values in the Ink attribute.



4.5.4 Additional Information

For more details concerning E-Basic, refer to Chapter 2-*E-Basic* in the E-Prime Reference Guide. The complete E-Basic scripting language is fully documented in the E-Basic Online Help. This is accessible via the E-Studio Help menu, or in the E-Prime menu.

4.6 Programming: Intermediate

4.6.1 More on Variables

There is much more information pertaining to variables that should be learned. Rather than overwhelm the beginning programmer, the information will be presented in stages of complexity. Intermediate programmers will need to use variables on a frequent basis. The following section details more information on declaration of, and working with variables.

4.6.1.1 DataTypes

When using the Dim statement to declare a variable, more information is required than just the name. The type of data the variable can hold must be specified. The script below not only reserves the word "subject_dob" as a variable, it also indicates that the variable is to hold a date (i.e., the subject's date of birth).

```
Dim subject_dob As Date
```

There are a variety of data types available within E-Basic. The following table illustrates the type and an explanation of the type:

Data Type	Description
Boolean	True (-1) or False (0) value.
Integer	Whole number ranging from -32767 to 32767
Long	Whole number ranging from -2,147,483,648 to 2,147,483,647
Single	Used to declare variables capable of holding real numbers with up to seven digits of precision: Negative: -3.402823E38 to -1.401298E-45, Positive: 1.401298E-45 to 3.402823E38
Double	Used to declare variables capable of holding real numbers with 15-16 digits of precision: Negative: -1.797693134862315E308 to -4.94066E-324, Positive: 4.94066E-324 to 1.797693134862315E308
Currency	Used to declare variables capable of holding fixed-point numbers with 15 digits to the left of the decimal point and 4 digits to the right (-922,337,203,685,477.5808 to 922,337,203,685,477.5807)
Date	Used to hold date and time values.
Object	Used to declare variables that reference objects within an application using OLE Automation.
String	Used to hold sequences of characters, each character having a value between 0 and 255. Strings can be any length up to a maximum length of 32767 characters.
Variant	Used to declare variables that can hold one of many different types of data. Refer to the Variant data type topic in the E-Basic Online Help.
User-Defined	Requires the Type statement.

Conversion between data types

Once a variable has been declared as a certain data type, it may hold only information of that type. E-Basic does not automatically convert data to the appropriate type. At times, it may be



necessary to convert information from one type to another in order to store it in a particular variable. E-Basic contains several functions for the purpose of conversion between data types.

Function	Description
CCur	Converts any expression to a Currency.
CBool	Converts expression to True or False, returning a Boolean value.
CDate, CVDate	Converts expression to a date, returning a Date value.
CDbl	Converts any expression to a Double.
CInt	Converts expression to an Integer.
Chr, Chr\$, ChrB, ChrB\$, ChrW, ChrW\$	Returns the character whose ASCII value is charcode.
CLng	Converts expression to a Long.
CSng	Converts expression to a Single.
CStr	Converts expression to a String.
Cvar	Converts expression to a Variant.
Hex, Hex\$	Returns a String containing the hexadecimal equivalent of number.
Str, Str\$	Returns a string representation of the given number.
Val	Converts a given string expression to a number.

Note, some functions offer the optional use of the "\$" character. When the "\$" character is used, the value returned is a string. When "\$" is not used, a string variant is returned.

4.6.1.2 Declaring Multiple Variables

Users are not limited to one variable declaration per line. Multiple variables may be declared on a single line using only a single Dim statement. However, the user should take care to specify the data type for each variable declared, even if they are on the same line. Any variable that is not specifically declared with a data type will be declared as a Variant. In the example below, subject_dob is declared as a date. Stim is declared as a string and k is declared as an integer. The j variable is declared as a variant because it is not specifically stated to be any other data type.

```
Dim subject_dob As Date, j, k As Integer, stim As String
```

Declaring multiple variables on the same line may save space, but it also increases the likelihood of human error. A compromise might be to not mix data types within a single line to help organize variables and reduce the chance of error.

```
Dim subject_birthdate As Date  
Dim j As Integer, k As Integer  
Dim stimulus As String, Dim recall As String
```

4.6.1.3 Initialize and Assign Values

Once a variable is declared, it must be **initialized** before it can be used. Initializing a variable is nothing more than assigning it an initial or starting value. Most often the purpose of a variable is to hold information that is assigned. An assignment statement simply consists of the variable name followed by an equal sign and then the **expression** (variable name = expression).

```
j = 1
```

In the example above, the counter variable "j" (used in a previous example) is assigned the expression or value of 1. String values are also assigned using an assignment statement, but the expression value must be enclosed in quotes.



```
Dim stringVal As String  
stringVal = "This is the value of the string."
```

If the expression extends past a single line, the expression statement must be divided into separate strings, which are concatenated. The ampersand (&) is used for this purpose. No linefeeds or carriage returns are included in the concatenation of strings, unless the new line (\n) character is included.

```
stringVal = "This is the value of a very long string "&  
            "extending over more than one line. The "&  
            "individual parts will be joined to form "&  
            "a continuous display." &  
            "\n\nThis will be displayed two lines lower"
```

Variables themselves may be used in expression statements. For example, in an assignment statement, the current value of a variable could be used to set the value of another variable.

```
Dim i As integer, j As Integer, k As Integer  
j = k * i
```

Rather than a literal assignment, in this case, the current values of the "k" and "i" variables are determined, and those values are used to calculate the value of "j." Or a variable may be used to pass information to a command or function. For example, the value of stringVal defined above could be used with the MsgBox command to display the intended string.

```
MsgBox stringVal
```

4.6.1.4 Constants

A **constant** is used when requiring the use of a value that does not change. Users could technically use a variable to hold the value, but it makes more sense to use a constant because it cannot be modified. To declare a constant, use a Const statement in the same manner a Dim is used to declare a variable. The only difference is that a value is specified immediately after the data type.

```
Const speed_of_light As String = "Really, really fast!"  
Const exercise As Boolean = True
```

4.6.2 Writing Subroutines

Subroutines are composed of a series of commands combined into a unit. This unit may then be run by a call to the subroutine from within an InLine object. A subroutine is defined using the Sub...End Sub statement. For example, a simple subroutine may be created to "clear" the background of a **Canvas object** to a particular color.

4.6.2.1 Example: Clear the screen to the current color

In the script below, the ClearToRed subroutine sets the fill color to red, and then uses the Clear command to clear the screen using the current FillColor setting. The script should be entered on the User script tab in the Script window. Once defined, the subroutine need simply be referred to by name in order to launch it.



```
'Subroutine containing the script necessary to set the
'background to red.

Dim cnvs As Canvas

Sub ClearToRed
  cnvs.FillColor = CColor("Red")
  cnvs.Clear
End Sub
```

In the example below, the `ClearToRed` subroutine is called to quickly set the background to red before 10 circles are drawn on the Canvas. The script below would be placed in an `InLine` object called during the experiment.

```
'Clear the screen to red and draw 10 circles of random
'size.

Dim i As Integer
Dim x As Integer
Dim y As Integer
Dim rad As Integer

Set cnvs = Display.Canvas
ClearToRed

x = 50
y = 100

For i = 1 To 10
  cnvs.Pencolor = CColor("white")
  cnvs.Fillcolor = CColor("white")
  rad = Random (3, 20)
  x = x + 50
  cnvs.Circle x, y, rad
Next I

Sleep 1000
```

Subroutines are most useful when a section of script is used repetitively. The use of subroutines aids in the prevention of errors, and minimizes script maintenance.

4.6.3 Writing Functions

Like subroutines, **functions** are units composed of a series of script commands. Functions differ from subroutines in that they may be used in a command, and may return a value (e.g., if the function requested a response from the user, or performed a data transformation).

4.6.3.1 Example: Calculate a mean value

In the example below, the `DoMean` function is passed two parameters (total and count). Total is divided by count (using the `/` operator) to determine the value for `DoMean`. This script is entered on the User tab of the Script window to define the `DoMean` function.

```
Function DoMean(total As Double, count As Integer)As Double
  DoMean = total/count
End Function
```



Thus entered in the User Script window, the DoMean function may be used at any time during the experiment. The CalcMean InLine object below calls the DoMean Function to calculate the mean of 5 randomly chosen numbers.

```
Dim total As Double
Dim count As Integer
Dim i As Integer
Dim next_val As Integer

total = 0
count = 5

For i = 1 To count
    next_val = Random (1, 20)
    MsgBox "Value #" & i & ": " & next_val
    total = total + next_val
Next i

MsgBox "The total is " & CStr(total) & "\n" & _
"The count is " & CStr(count) & "\n" & _
"The mean is " & DoMean (total, count)
```

4.6.4 Additional Information

For more details concerning E-Basic, refer to Chapter 2-*E-Basic* in the E-Prime Reference Guide. The complete E-Basic scripting language is fully documented in E-Basic Online Help. This is accessible via the Help menu within E-Studio.

4.7 Programming: Advanced

Before moving to this section, be comfortable with working with variables and data types. This section introduces the use of arrays and user defined data types. These are very powerful features, but understanding the basics is essential.

4.7.1 Arrays

It is often the case that multiple pieces of information need to be used as a single variable. The best way to handle this is through the use of an **array**. An array is a storage unit for many items of the same type. The storage unit is comprised of multiple items of one type, each being stored in their own **index** within the array. Think of it as a filing drawer that may contain many files in a specific order which all pertain to a single topic. The array would be the drawer while the individual files within would be the indices.

4.7.1.1 Declaring Arrays

To work with an array, refer to the name of the array and the index. An array can be of any data type, but can only hold a single data type. Specifically, declare an array that holds strings and an array that holds integers, but an array cannot hold both strings AND integers. However, this is easily remedied because users may declare arrays of variant data type, which holds any kind of data. Be careful though; since variant data typically is more memory intensive than other data types, an array of variants can easily become overwhelming overhead. Use this option wisely.

As with any variable, before using it, first declare and initialize it. The declaration of an array is similar to any other variable. The Dim statement is used, and the only significant difference is the dimensions value.



```
Dim position_array ( ) As Integer
```

Fixed arrays

The dimensions of fixed arrays cannot be adjusted at execution time. Once declared, a fixed array will always require the same amount of storage. Fixed arrays can be declared with the Dim statement by supplying explicit dimensions. The following example declares a fixed array of eleven strings (assuming the option base is 0, see *Using an Array Index* below):

```
Dim a(10) As String
```

Fixed arrays can be used as members of user-defined data types. The following example shows a structure containing fixed-length arrays:

```
Type Foo  
    Rect(4) As Integer  
    Colors(10) As Integer  
End Type
```

Only fixed arrays can appear within structures. Refer to section 4.7.3 for a discussion of user-defined types.

Dynamic arrays

Dynamic arrays are declared without explicit dimensions, as shown below:

```
Public Ages() As Integer
```

Dynamic arrays can be resized at execution time using the ReDim statement:

```
ReDim Ages (100)
```

ReDim modifies the dimensions of an array, specifying a new upper and lower bound for each dimension. After they are declared, dynamic arrays can be redimensioned any number of times. When redimensioning an array, the old array is first erased unless the Preserve keyword is used, as shown below:

```
ReDim Preserve Ages (100)
```

Dynamic arrays cannot be members of user-defined data types.

4.7.1.2 Using an Array Index

Unless otherwise specified, items within an array are indexed beginning with zero (i.e., arrays are zero-based). In other words, the first element within an array is located within index number 0. For instance, if an array is designed to hold 10 elements, the array should be dimensioned as in the following:

```
Dim arrResponses (9) As String
```

In the previous statement, the arrResponses array is dimensioned to hold 10 elements. Because array indices begin at 0, the "9" in the dimension statement indicates the largest legal index within the array, and the total number of elements that the array may contain is one greater than this number.



Addressing an element within an array

The individual elements within an array may be accessed or set simply by listing the array name, and using subscript notation (i.e., the index number enclosed in parentheses) to refer to the appropriate index. The index of the array includes an integer value for each dimension of the array. For example, `my_array(3)` refers to, or identifies the value in the fourth slot in the array called `my_array` (given a zero-based numbering system). Given this scheme, data contained within an array may be used like any other variables:

Assign a value to an array element.

```
Dim a(10) As Integer  
a(1) = 12
```

Assign a value stored in an array to another variable.

```
x = a(1)
```

Use the value of an array element in an expression:

```
x = 10 * a(1)
```

4.7.1.3 Assigning Data

When an array is declared using the `Dim` statement, the elements composing the array are not initialized. That is, the elements contain no valid information. Before accessing the array elements, they must be assigned meaningful values. Array elements are assigned values using an assignment expression (i.e., `ArrayName(Index) = Expression`).

```
Dim a(10) As Integer  
a(1) = 12
```

The most efficient method of assigning values to an entire array at one time (e.g., to initialize an array to consecutive values) is to use a `For...Next` loop.

```
Const Size As Integer = 10  
Dim a(10) As Integer  
Dim x As Integer, i As Integer  
  
For i = 0 To Size-1  
    a(i) = x  
    x = x + 1  
Next i
```

Arrays may also be multi-dimensional. Arrays containing more than one dimension are similar to a spreadsheet-like organization, with different dimensions handling different tables or lists of information. Multi-dimensional arrays are declared just like one-dimensional arrays, with commas separating the values specifying the size of each dimension in the array.

```
Dim multi_array (10, 7, 9) As Integer
```

The total number of elements held by a multi-dimensional array is equal to the product of the sizes of the individual dimensions. The example above would result in an array holding 630 elements.



4.7.2 Timing

Refer to Chapter 3-*Critical Timing* in the User's Guide for a detailed discussion of critical timing issues and techniques.

4.7.3 User-Defined Data Types

E-Basic allows the user to define data types. User-defined data types are very useful for organizing related data items of various types. A user-defined data type is declared using the Type statement, and the data items organized by the data type are listed in the Type statement. For example, to draw and modify a grid in which some of the cells are drawn in color, it would be useful to keep track of an ID number for each cell, the color in which each cell is drawn, the location of the cell, and other possible information about each cell. The script below uses the Type statement to declare the CellInfo data type, organizing the data relevant to each cell in the grid.

```
Type CellInfo 'keep track of cell info
  nID As Integer
  nColorState As Integer 'Is the cell a color or non-color
  nColor As Long 'Specific color used to draw the cell
  nRow As Integer 'Row in the grid where cell appears
  nColumn As Integer 'Column in the grid
End Type
```

Within the Type declaration of the CellInfo data type, variables are declared to organize the information pertaining to each cell (i.e., the cell ID, color of the cell, row, column, etc.). Once the data type has been declared, the Dim command is used to declare a new instance of the type. For example, the script below declares the CurrentCell variable as an instance of the CellInfo type (i.e., a single cell in the grid).

```
Dim CurrentCell As CellInfo
```

Like assigning values to object properties, the dot operator is used to assign values to the component variables of a user-defined type. Below, the CurrentCell variable is assigned values for the ID number, row, column, color state, and color components.

```
'Define cell info
CurrentCell.nID = 12
CurrentCell.nRow = 2
CurrentCell.nColumn = 3
CurrentCell.nColorState = 1 '1 = color, 0 = white
CurrentCell.nColor = CColor("Blue")
```

4.7.4 Examples

4.7.4.1 Contingent Branching

Launching a specific Procedure based on the subject's response.

This example assumes a structure in which an input object named "subject" collects a response, and two separate List objects (List1 and List2) call separate Procedures.



```
'If the subject enters "1" run Procedure 1, otherwise run  
'Procedure 2.  
  
If subject.RESP = "1" Then  
    List1.Run  
Else  
    List2.Run  
End If
```

4.7.4.2 Arrays

Creating a single dimension array, assigning values, and accessing values for display.

```
'Creating and assigning values to a single dimension array  
  
Dim WordList(4) As String  
Dim i As Integer  
  
WordList(0) = "Every"  
WordList(1) = "good"  
WordList(2) = "boy"  
WordList(3) = "does"  
WordList(4) = "fine"  
  
For i = 0 To 4  
    MsgBox WordList(i)  
Next i
```

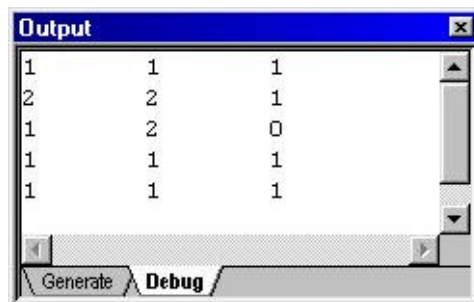
4.7.4.3 Debugging

Using Debug.Print to verify logging accuracy.

This example assumes a structure in which an input object named "StimDisplay" collects a response.

```
'Evaluate the response collected by the StimDisplay object.  
'Send the response entered by the subject (RESP), the  
'correct response (CRESP), and the accuracy (ACC) to the  
'OUTPUT window separated by tabs. View using the DEBUG tab  
'in the OUTPUT window.  
  
Debug.Print StimDisplay.RESP & "\t" & StimDisplay.CRESP &_  
            "\t" & StimDisplay.ACC
```

The script above will send information to the Debug tab in the Output window as follows:





4.7.5 Additional Information

For further details concerning E-Basic, refer to Chapter 2-*E-Basic* in the E-Prime Reference Guide and the E-Basic Online Help. The complete E-Basic scripting language is fully documented in Online Help. This is accessible via the E-Studio Help menu, or through the E-Prime menu via the Start button.

4.8 Debugging in E-Prime

Errors may occur during both the generation and the running of an experiment. When an experiment is generated within E-Studio, the Output window provides feedback concerning the status of the generation procedure. The Output window may be displayed using the View menu in E-Studio. Within the Output window, the Generate tab displays information concerning the generation process of the program. For example, if an experiment is generated without errors, the Generate tab in the Output window will display messages indicating that the script was generated successfully.

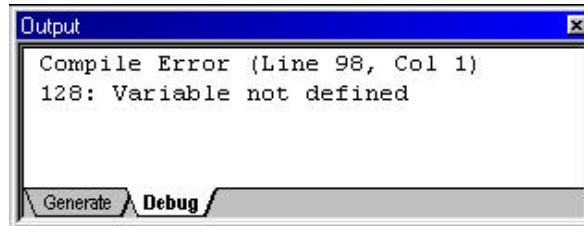


When any errors are produced as a result of generating script within E-Studio, the errors will be reported by a dialog box.

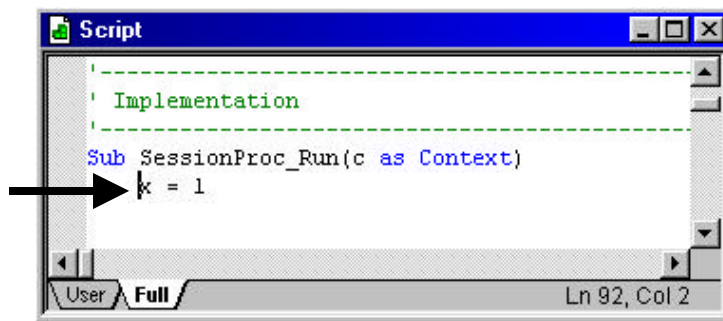




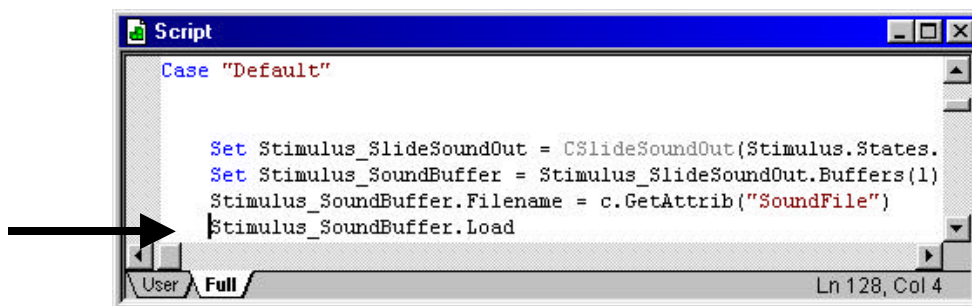
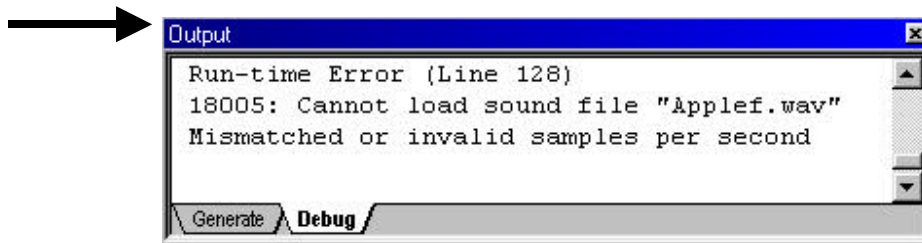
The errors will also be sent to the Debug tab in the Output window. After dismissing the error dialog, the errors may be redisplayed by opening the Output window (View menu) and clicking the Debug tab.



Each error will display the line number in the script at which the error occurred, and the Script window will open automatically in E-Studio to display the full experiment script. Within the Script window, the cursor will blink at the line at which the error was encountered.



When errors occur during the running of an experiment (i.e., runtime error), a dialog will be displayed indicating the runtime error, and the line in the script at which the error occurred. As with compile errors, runtime error messages are sent to the Output window, and the Script window is opened in E-Studio with the cursor placed at the error location.





4.8.1 *Tips to help make debugging easier*

- Run in fixed order to verify proper stimulus selection before adding randomization.
- Think small -- test small portions of the program for functionality before expanding the program.
- Use Debug.Print to send information to the Output window during runtime. This information may be evaluated after the run terminates in order to verify values.
- In an Inline object, use Display.Canvas.Text in conjunction with a Sleep command to display debugging information to the screen at runtime without requiring input from the user to continue.
- The MsgBox command may be used to display values during the run of an experiment. The MsgBox will not allow the program to continue until the user presses enter to dismiss the dialog box.
- Assign values as attributes so that they may be displayed on a TextDisplay or Slide object during the run of an experiment, or logged to the data file for later examination.
- Include comments in the script to delimit and identify the purpose of various sections of script.