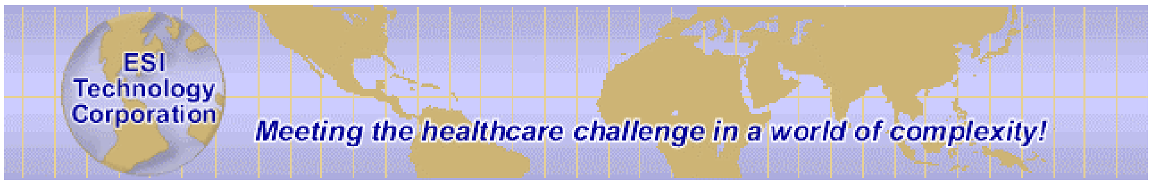


# Language Reference Guide

## EsiObjects V4.2

## *Trademarks*

# Table of Contents

# Introduction

This guide contains some language concepts for the EsiObjects programming system. It contains the following:

- Language Concepts section that covers all functionality supported within a method or property code body.
- All EsiObjects commands, special variables, functions and operators.

# Document Conventions

EsiObjects documentation uses the following typographical conventions:

| | |
|---|---|
| For more information on this subject please refer to the <u>BREAK Command section of this guide.</u> | Underlined text is used to highlight a reference to another section of this guide or another guide. |
| *Property* | In text, italicized words indicate defined terms that are usually used for the first time. Words are also italicized for emphasis. |
| **CREATE** | Words in bold and capitalized are EsiObjects commands or keywords. |
| `Set T%Test=I%Pat.Name` | This font is used for code examples. |

# Language Concepts

The EsiObjects language is based upon the ANSI standard MUMPS (M) language. The M language is a powerful string handling language that, by definition, implements a hierarchical, multi-dimensional array construct whose instances may exist persistently or non-persistently.

EsiObjects uses the M language and embedded array specification as an enabling technology. The object model implemented using M code and array structures is based on the Smalltalk object model. This model is class based. Classes contain the definitional information needed to instantiate an object. Classes contain:

- Interfaces that contain class services
- Variable definitions

## Interfaces and Services

Interfaces provide a way of partitioning a class's services. Interfaces offer the programmer a way to logically separate the services of a class. Each interface can potentially contain up to four service types. They are:

- Relationships
- Events
- Methods
- Properties

A relationship is a special type of service that is covered in detail in the Programmer's Reference Guide. Relationships are created using existing classes and code; consequently, they are not of interest here.

The EsiObjects implementation added the concept of events to the M language. It is a mechanism that permits the programmer to broadcast an event to objects that have taken out a watch on that event. The mechanism uses another EsiObjects feature added to the M language, the callback. The callback mechanism uses methods to hold the code to be executed.

Methods are code bodies that give an object its behavior. Methods are stored at the class level. Every instance of a class has access to a method that exists within an interface of the class. Method code bodies contain lines of code just like M routines. However, method code bodies contain other features not found in a normal M routine such as compiler directives and input specifications.

Properties within EsiObjects are generally used to expose the state of an object. Each EsiObjects property consists of a number of specialized methods called accessors. Accessors are:

- Assign
- Create
- Kill
- Value
- $Data
- $Get
- $Normalize
- $Order
- $Query
- $Valid

Each accessor represents a code body that is mapped to an EsiObjects language element. For example, if code exists for the Kill accessor, it will be invoked whenever the property is used within an object message of the Kill argument.

The method and property accessor support the same structure and features.

# Method and Properties

## Code Body Structure

A method is callable code body associated with a class. This code body can be invoked on any object instantiated from that class.  A message is a request that can be sent to an object that may invoke a method to be performed.

An accessor is a special purpose method that is attached to a property and controls access to the property. An accessor is an action that is associated with a property.

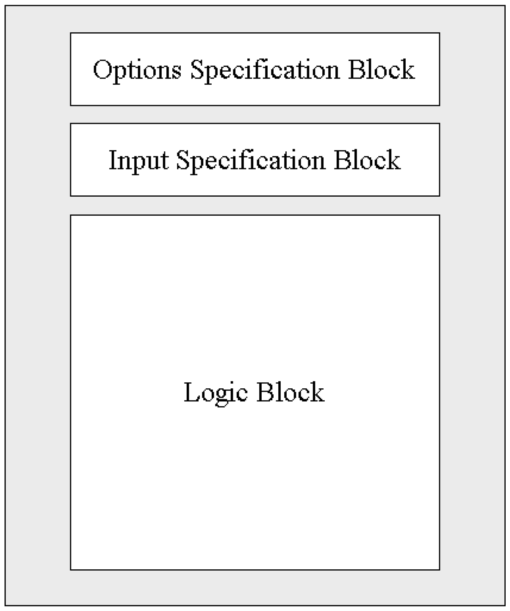
Illustrated below is the general structure of a method code body.

A method may begin with an **Options Specification Block**, which defines the options that define the method. That is, what type it is, whether it is inheritable, what values is passes back, etc. If the Options block is defined, it must precede the Input Specification.

The **Input Specification Block**, if present, declares parameters and their internal mappings.

The combination of the Options Specification Block and the Input Specification Block form a full specification of the method or property. Combined, these two blocks can be used to generate a textual specification of the method or property.

The remainder of the method body consists of the **Logic Block** consisting of one or more lines of EsiObjects code and compiler directives.



**Code Body Components**

All EsiObjects properties have accessor methods. Note that these are standard methods, and they share the method structure described in this section.

## Options Specification Block

An EsiObjects Options Specification lets you define a specification of the method or property accessor code body such that:

1. A total textual representation can be produced outside of the EsiObjects environment
2. And, strong typing can be enforced.

It must be placed at the beginning of the code body and before the Input Specification if it exists. Specifying options actually has an affect on the compiler. Consequently care must be taken to insure accuracy. The options are also used by various code generation add-ons (e.g. The Java Proxies builder).

The following is the syntax for an options specification:

```
 optionspec ::={ws} OPTIONS:{ws}({ws} L optionitem) {comment text} eol
```

where:

| | |
|---|---|
| **ws** | **is optional white space**<br>**can be one of the following:**<br>**SP**<br>**TAB**<br>**Eol**<br>**; {comment text} eol** |
| **OPTIONS** | **is the Options: which is not case sensitive** |
| **optionitem** | **is a list of option items enclosed in parentheses (or curly brackets) and separated by commas** |
| **comment text** | **are optional comments (comments can be preceded by a semicolon for clarity, but it is not required)** |

The following is an example of an Option Specification that contains two option items and a comment:

```
Options:(Type=Base$Set,Method) ; Return type is Base$Set and it’s a Method
```

White space in an Options Specification can contain tab characters, spaces, end-of-line characters, and comment text (preceded by a semicolon and terminated by an end-of-line character). White space allows option specifications to extend beyond a single line. For example:

```
Options:  (
          Type=Base$Set,      ; Return type is Base$Set

          Method              ; This code body is a Method

          )
```

In the previous example, each option item is specified on a separate line and its purpose is documented by a comment. This is the preferred method of coding option specifications because it is easier to read and understand.

The following table outlines the various option items currently available for EsiObjects.

| Option Items | Description |
|---|---|
| **Abstract** | **Abstract** indicates that the service is a placeholder and must be implemented by a subclass. |
| **AnyKeyword** | **AnyKeyword** indicates the method or property will accept any keyword parameter as being valid. |
| **Class** | **Class** flags the code body as being a part of the Factory interface. It is basically a class level method. |
| **Constant** | **Constant** indicates that the service returns a constant value. The parser may resolve this constant during the compile. |

| | |
|---|---|
| **Experimental** | Flags the method or properties as being **Experimental.** |
| **Id = Identifier** | Associates an **identifier** to this service. This identifier may by presented to the user during errors or debugging. |
| **Inheritable** | **Inheritable** means the method can be seen by subclasses. |
| **Method** or<br>**Property_Set** or<br>**Property_Kill** or<br>**Property_Get** or<br>**Property_Create** or<br>**Property_DataFn** or<br>**Property_GetFn** or<br>**Property_OrderFn** or<br>**Property_QueryFn** or<br>**Property_NormalizeFn** or<br>**Property_ValidateFn** | Identifies the type of the code body. Only those listed are valid and must be spelled properly. |
| **Name = Full Name** | Name of the code body (Method or Property). |
| **Platform = Platform Expected** | Where Platform Expected can be DSM, MSM, Cache, GT.M or All. It is strictly a documentation flag at this point. |
| **Private or Public** | **Private** identifies the method as private to the class. It is not a part of the classes' protocol.<br>**Public** means the method is exposed as a part of the classes protocol. |
| **Privileged** | **Privileged** means the method has privileges to execute those language elements that required privileges such as $OIDPTR and $PTROID. |
| **Static** | This keyword indicates that the service is called in a static manner (i.e. without an instance). When this option is present references to instance variables is illegal. A Parser code PARSE_STATIC_NOINVAR will result. |
| **Throws = Exception** | Identifies the service as throwing the noted exception. Each exception that is thrown by the service should appear |

| | |
|---|---|
| | in the options block. The exception must be a subclass of the class **ESI$Exception**. This option is available from V4.1 and higher. |
| **Type = Return Type or Void** | Defines the return type of the method or property if it is applicable. If you use the **Type=** format, the compiler will insure that the value returned from the method is of the type specified. For example:<br>`Options:(Type=HIS$Patient)`<br>will insure that the return value is an object of the HIS$Patient class.<br><br>Alternatively, you may specify **Void**, there will be no return type returned. |
| **UsesIO** | **UsesIO** flags the method or property body as using direct IO devices that are specific to the M implementation. |
| **Virtual** | **Virtual** flags the method or property as being a part of a virtual class. It cannot access instance variables since virtual objects do not have state. |

## Input Specification Block

An EsiObjects Input Specification must be placed between the Options Specification Block, if it exists, and the Logic Block. It defines the method's input parameters and their internal mappings.

The following is the syntax for an Input Specification:

inputspec ::={ws} INPUT:{ws}({ws} L inputitem) {comment text} eol

where:

| | |
|---|---|
| **ws** | **is optional white space** |
| | **can be one of the following:** |
| | **SP** |
| | **TAB** |
| | **Eol** |
| | **; {comment text} eol** |
| **INPUT** | **is the Input: keyword, which is not case sensitive** |
| **inputitem** | **is a list of input items enclosed in parentheses** |
| **comment text** | **are optional comments (comments can be preceded by a semicolon for clarity, but it is not required)** |

The following is the syntax of an input item:

| | | | |
|---|---|---|---|
| **inputitem ::=** | **{{paramopt} keyword:}** | **{pers}** | **{ws}** |
| | | **{paramopt}glvn pers** | |
| | | **{paramopt}glvn=expr** | |

An input item can be null, but can contain up to three parts. The first argument specifies a keyword if the parameter can be passed by keyword. The second argument specifies a variable mapping and possibly an expression to calculate the default value. Either of these arguments can specify a parameter option. The third argument of an input item consists of ignored white space and comment lines.

The following is an example of an Input Specification that contains two input items and a comment:

```
Input:(P%Dir,P%Index) ; direction and index name
```

White space in an Input Specification can contain tab characters, spaces, end-of-line characters, and comment text (preceded by a semicolon and terminated by an end-of-line character). White space allows the Input Specification to extend beyond a single line, for example:

```
Input: (
       P%FieldId,   ; Prevalidated field number (ID)
       P%Value      ; Value to be validated.
       )
```

In the previous example, each variable is specified on a separate line and its purpose is documented by a comment. This is the preferred method of coding input specifications because it is easier to read and understand.

The following sections describe the valid parameter options (**paramopt**) associated with the keyword (**keyword:**).

## Parameter Options

### *Required or Optional*

The **Required** parameter option is used to force the system to generate an error if a value is not passed in. If it is not specified or if **Optional** is specified, then the value need not be passed in and an error will not be generated. The default is **Optional**.

The following example contains an Input Specification with two mappings.

```
   Input:((Required)Field:P%Field,Value:P%Value="")
```

In the previous example, the first parameter is **Required**, is identified by the keyword Field, and maps into the symbol P%Field. The second parameter is **Optional**, is identified by the keyword Value, and it maps into the symbol P%Value. If the second parameter is not specified, it receives the default value of NULL ("").

### *In, Out or InOut*

The **In**, **Out** and **InOut** parameter options give you control over the destination of parameter values.

If neither option is specified (or the In option only), the compiler will default to the In option. This means a value may only be passed into the specified variable. It will not be passed back to the caller.

If the Out parameter option is specified by itself, this means that a value cannot be passed into the specified variable. The variable and value associated with the Out option will be passed back through the calling contexts and be made available to those contexts. This mechanism is classically used to return special conditions that occurred such as an error that terminated processing. It is a replacement for the traditional M call-by-reference syntax that violates encapsulation.

If the In and Out parameters are specified simultaneously (In, Out or InOut), this means that a value can be passed in and bound to the specified variable. Additionally, it will be passed back to the calling contexts.

The method that uses the In, Out and InOut parameters must declare them.

For example:

```
Input:    (
     (In)P%Name,
     (InOut)P%ErrStat,
     (Out)P%ErrMsg
     )
```

The first parameter declares the parameter variable P%Name as an In only parameter. This means that a value may only be passed into the current context. It cannot be passed out.

The second parameter is declared as both In or Out (could have been specified as In,Out). This means a value may be passed in and it will be passed back by the system once the current execution is popped from the stack.

The third parameter is declared as strictly an **Out** parameter. This means it cannot accept a value passed in and can only pass a value back to its caller.

On the caller’s side, the parameters must specify how the values are being passed. This needs to match the way the Input Specification is declared. An example of the calling syntax for the Input Specification above follows:

```
     Do T%Patient.Validate(“Doe, John D”,[InOut]T%ErrStat,[Out]T%ErrMsg).
```

EsiObjects has replaced the pass-by-value and pass-by-reference mechanism of standard MUMPS with the In/Out mechanism because it does not break encapsulation. It uses the messaging mechanism to pass back values to a calling context in a safe way. The call-by-reference mechanism of standard MUMPS breaks encapsulation by letting one object directly access the state of another object.

### *Alias*

The **Alias** parameter option lets you use different keywords when mapping parameter values. This option is useful when two or more callers use different keywords.

The following example illustrates how a typical Input Specification would look using the Alias option.

```
Input:     (
           PatientName:P%PatNam,
           (Alias)Name:P%PatNam,
           DateOfBirth:P%DOB
           )
```

Parameters can be passed in by keyword or by the traditional M approach - by position. Passing by keyword lets the caller to specify the parameters in any order as long as the keyword is specified. The following illustrates how passing values in by keyword would work when two different objects call the Lookup method.

Object 1:

```
Do T%PatObj.Lookup(DateOfBirth:”10-Jan-42”,PatientName:”Doe, John D”)
```

Object 2:

```
Do T%PatObj.Lookup(Name:”Doe, John D”, DateOfBirth:”10-Jan-42”,)
```

Notice that Object 1 uses the PatientName keyword and the order is different from the Input Specification. Because keywords are used, the order does not matter.

Object 2 passes the patient’s name in using the Name keyword specified by the Alias parameter option.

Use of the Alias parameter option in the Input Specification does not take up a position. That is, an Object 3 could call the Lookup method using positional parameters and the values would map properly. For example:

```
Do T%PatObj.Lookup(”Doe, John D”,”10-Jan-42”,)
```

### *Type=*

The **Type** parameter option lets you restrict a parameter value to an OID of an object of a particular Library$Class. For example:

```
Input:     (
           (Type=HIS$Patient)PatientName:P%PatNam,
           (Alias)Name:P%PatNam,
           (Type=Base$TimeStamp)DateOfBirth:P%DOB
           )
```

By specifying the Type=HIS$Patient on the first parameter, the compiler will generate the proper runtime code to insure that the value being passed in is an instance OID of the Patient class in the HIS library. Additionally, the Type=Base$TimeStamp will force the runtime module to make sure the values is an OID of the TimeStamp Call in the Base library.

The Type parameter option helps eliminate errors due to bad parameter passing.

### *System*

The **System** keyword flags the parameter a being system generated. It is required and should never be deleted. Additionally, the System keyword should be attached to those system-generated parameters. Specifically, the first parameter on the following property accessors should be flagged as System.

- Assign
- $Get
- Create
- $Order
- $Normalize
- $Valid

## Parameter Variable Assignment

The mapping of input parameters to their associated symbols can involve complex interactions.

The list of input parameters can consist of positional parameters, keyword parameters, and void parameters. All the parameters in an actual method parameter list are assumed to be positional until the first keyword parameter is encountered. After the first keyword is encountered, all the parameters are assumed to be keyword.

The following is a list of the input item keyword syntaxes:

| | |
|---|---|
| **keyword:** | A keyword is declared, but no mapping is associated with it. This is known as a void mapping. |
| **keyword:variable** | A keyword is mapped into a variable. |
| **keyword:var=value** | A keyword maps into a variable. A certain value is used as the default if no value is passed as a parameter. |
| **variable** | A positional parameter maps into the specified variable. |
| **variable...** | The variable name is used as the root of an array. All remaining parameters appear as array nodes, and the base of the array contains the address of the highest-numbered parameter to receive a value. |

| | |
|---|---|
| **variable=value** | A positional parameter maps into the specified variable. A default value is specified in case the parameter is not passed. |

### Example 1: Simple Parameter Passing

This example shows the simplest form of positional parameter passing. Three variables P%Name, P%Tag, and P%ParamList are declared in the Input Specification. The three values (1, 2, and 3) are passed positionally in the actual method parameter list. Therefore, inside the method body P%Name is equal to 1, P%Tag is equal to 2, and P%ParamList is equal to 3.

#### *Input Specification*

```
Input:(P%Name,P%Tag,P%ParamList)
```

#### *Method Call*

```
DO Object.Method(1,2,3)
```

#### *Internal Mappings*

```
P%Name=1
P%Tag=2
P%ParamList=3
```

### Example 2: Array Parameter Passing

This example illustrates the use of an array parameter in the Input Specification. Three periods are used to specify that all remaining parameters should be included as array nodes of P%ParamList.

The first two parameters are passed positionally into the variables P%Name and P%Tag. The third input item in the input specification is the array P%ParamList. Because there are a total of 6 parameters, parameters 3 through 6 appear as array nodes P%ParamList(1) through P%ParamList(4), and the root node C contains the total number of array nodes.

#### *Input Specification*

```
Input:(P%Name,P%Tag,P%ParamList...)
```

#### *Method Call*

```
DO Object.Method(1,2,3,4,5,6)
```

#### *Internal Mappings*

```
P%Name=1
P%Tag=2
P%ParamList=4     P%ParamList(1)=3
     P%ParamList(2)=4
     P%ParamList(3)=5
     P%ParamList(4)=6
```

### Example 3: Using Keywords on Parameters

In this example, the keyword WindowSize is declared as the sixth parameter, which is to be mapped into the variable P%Height. P%ParamList is declared as an array of all the remaining parameters. Therefore, when the keyword WindowSize is specified, the associated value appears in two places:

- In the array node P%ParamList(4) because it is the sixth parameter
- In the variable P%Height because it uses the keyword WindowSize

#### *Input Specification*

```
Input:(P%Name,P%Tag,P%ParamList...,,,WindowSize:P%Height)
```

#### *Method Call*

```
DO Object.Method("Text",WindowSize:12)
```

#### *Internal Mappings*

```
P%Name="Text"
P%ParamList=4
P%ParamList(4)=12
P%Height=12
```

### Example 4: Using Keyword and Positional Parameters

In this example, the Input Specification declares void mappings for the keywords Key and Type, and maps WindowSize into P%Height. These three keywords are declared as the fourth through sixth parameters, respectively. Also, the keyword W maps to the array P%ParamList, which is an array containing all parameters after the third. The method actual list passes values into the keywords Key, Type, and WindowSize. They map positionally into the array P%ParamList and WindowSize maps explicitly into the variable P%Height.

#### *Input Specification*

```
Input:(P%Name,P%Tag,W:P%ParamList...,Key:,Type:,WindowSize:P%Height)
```

#### *Method Call*

```
DO Object.Method(Key:1,Type:10,WindowSize:15)
```

#### *Internal Mappings*

```
P%ParamList=4     P%ParamList(2)=1
    P%ParamList(3)=10
    P%ParamList(4)=15
P%Height=15
```

### Example 5: Defaulting Parameter Values

In this example, the Input Specification maps keyword Key into variable P%SVal with a default value of Text and maps keyword Type into the variable P%Height with a default value of 10. Key is passed the string EsiObjects, which is passed to P%SVal. P%Height gets the default value 10 because a value was not passed in by the method call.

#### *Input Specification*

```
Input:(Key:P%SVal="Text",Type:P%Height=10)
```

#### *Method Call*

```
DO Object.Method(Key:"EsiObjects")
```

#### *Internal Mappings*

```
P%Height=10
P%SVal="EsiObjects"
```

### Example 6: Positional and Keyword Parameters

In this example, two positional parameters P%Name and P%Tag are declared, and P%Tag defaults to P%Name. A single value (10) is passed into the first parameter and it maps into P%Name. P%Tag is also 10 by default because it refers to P%Name.

#### *Input Specification*

```
Input:(P%Name,P%Tag=P%Name)
```

#### *Method Call*

```
DO Object.Method(10)
```

#### *Internal Mappings*

```
P%Name=10
P%Tag=10
```

### Example 7: Unspecified Parameters

In this example, two positional parameters P%Name and P%Tag are declared and P%Name defaults to P%Tag. A single value (10) is passed into the first parameter and it maps into P%Name. P%Tag is not defined.

#### *Input Specification*

```
Input:(P%Name=P%Tag,P%Tag)
```

#### *Method Call*

```
DO Object.Method(10)
```

### *Internal Mappings*

```
P%Name=10
```

### Example 8: More Unspecified Parameters

In this example, two positional parameters P%Name and P%Tag are declared and P%Name defaults to P%Tag. A single value (10) is passed into the first parameter and it maps into P%Name. Because a value is passed into P%Name, its default is not used. P%Tag is not defined.

### *Input Specification*

```
Input:(P%Name=P%Tag,P%Tag)
```

### *Method Call*

```
DO Object.Method(10)
```

### *Internal Mappings*

```
P%Name=10
```

### Example 9: Positional and Keyword Errors

In this example, three positional parameters P%SVal, P%Name, and P%Tag are declared. P%Name defaults to P%Tag. A single value (10) is passed into the first parameter, so it maps into P%SVal. An error occurs in this example when an attempt is made to assign a default value to P%Name. Its default value is P%Tag, but P%Tag is undefined. It is less misleading if the Input Specification had declared P%Name as a Required parameter.

### *Input Specification*

```
Input:(P%SVal,P%Name=P%Tag,P%Tag)
```

### *Method Call*

```
DO Object.Method(10)
```

### *Internal Mappings*

```
P%SVal=10
ERROR: P%Tag is undefined!
```

### Example 10: Positional and Keyword Mapping

In this example, two keyword parameters Key and Type are specified. Key maps into P%SVal, which defaults to P%Height and Type maps into P%Height. The value 10 is passed to Type, so P%Height becomes 10. P%SVal also gets this value because its default is P%Height.

### *Input Specification*

```
Input:
(
(Required)Key:P%SVal=P%Height,
Type:P%SVal
)
```

### *Method Call*

```
DO Object.Method(Type:10)
```

### *Internal Mappings*

```
P%Height=10
P%SVal=10
```

## Logic Block

### Four Types of Lines

The syntax described in this section defines the structure of a line of EsiObjects code found in the Logic Block. The following is the syntax of a line:

| | | |
|---|---|---|
| **line ::=** | **Formalline** | **Eol** |
| | **Codeline** | |
| | **Directive** | |
| | **Blankline** | |

The lines in an EsiObjects method fall into the following categories:

- **Formal lines** A formal line (formalline) begins with a label, followed by a formal parameter list (a list of local variable names in parentheses), a line-start indicator (space or tab), and the body of the line.

  formalline ::= label formallist ls linebody

- **Code lines** A code line (codeline) optionally can begin with a label that is followed by a line-start indicator (a space or a tab), and the body of a code line.

  codeline ::= { label } ls { li ... } linebody

- The body of a code line can begin optionally with level indicators, which indicate the dot-indent level in an argumentless **DO** block. Level indicators consist of periods and any number of optional spaces. The total number of periods indicates the line's actual level.

- **Preprocessor directives** A preprocessor directive (directive) alters the compilation of the method. For more information about preprocessor directives, see the Preprocessor Directives section of this guide.
- **Blank lines** These lines contain no text and are ignored.

The line body (**linebody**) consists of one or more commands, separated by one or more spaces. The last command on the line never requires spaces after it. The following is the syntax of a line body:

```
linebody ::=    commands {cs comment}    Eol
                Comment
```

The following is the syntax for a comment and command space:

```
comment ::=    ; {;} { commenttext }
```

cs ::= SP ...

A semicolon can occur in the command position, which indicates that the remainder of the line is a comment text. The compiler ignores comments. Comments preceded by two contiguous semicolons are never stripped by the compiler and can be accessed with the **$TEXT** function (subject to certain restrictions). For more information, see the description of the $TEXT function.

Command space **cs** consist of one or more spaces.

The commands (**commands**) on the line consist of one or more commands and their arguments, separated by at least one space. The following is the syntax of a command:

```
commands ::=    command {SP command ...}
```

As defined by the previous syntax, a **command** consists of a command and its argument, or consists of an argumentless command followed by a single space. Therefore, there must be at least two spaces between an argumentless command such as **ELSE** and the next command on the line. However, if the last command on the line is argumentless (as in the case of **QUIT**), it does not require any spaces before the end of the line.

## Line Syntax Examples

The following example contains a formal line and four code lines.

```
GETITEM(N,OBJECT) ; Return element N of OBJECT
    IF $GET(OBJECT)="" DO
    . SET OBJECT=$SELF.GetBaseItem   ; Create object
    . DO OBJECT.LoadElements
    QUIT $GET(OBJECT.Elements(N))
```

Note the following about the previous example:

- The first line is a formal line, which contains a starting label, a formal parameter list with two parameters, and a comment. The line-start indicator is a space.
- The last four lines begin with a tab, followed immediately by the line body. Tabs are stored internally as spaces.

- The third and fourth lines have a level indicator (a period), indicating that both lines are at level 1.

The following example contains four code lines:

```
; Calculate return value and exit...
IF '$DATA(T%Result) SET $RETURN=11   ; Descendant+value
ELSE  SET $RETURN=T%Result           ; Two spaces between ELSE and SET
QUIT
```

Note the following about the previous example:

- The first line is a comment line (everything after the semicolon is ignored).
- The second line contains **IF** and **SET** commands with arguments and ends with a comment.
- The third line begins with an argumentless command, **ELSE**, which is separated by two spaces from the **SET** command.
- The final line has an argumentless command, **QUIT**, which does not require any spaces before the end of the line.

### Labels and Label Keywords - Introduction

Formal lines and code lines can have labels as shown by the following syntax:

**codeline ::= { label } ls { li ... } linebody**

The following example shows a line containing a label, followed by a line-start indicator and a comment:

```
DISPLAY ; Display the entries in the object's element array.
```

A formal line begins with a label, followed by a formal parameter list, a line start indicator (space or tab), and the body of the line. The syntax is as follows:

formalline ::= label formallist ls linebody

A formal parameter list is a list of local variable names enclosed in parentheses. Sometimes there are no local names in the parameter list (for example, in the case of an extrinsic variable). The syntax for a formal parameter list is as follows:

formallist ::= ( {L localname} )

The following example shows a line containing a label and a formal parameter list, followed by a line-start indicator and a comment.

```
GETITEM(N,OBJECT) ; Return element N of OBJECT
```

### Labels in EsiObjects

An EsiObjects label consists of a label name, which optionally can be preceded by a list of label keywords, enclosed in parentheses and separated by commas. Label use declaration (keywords) is provided in EsiObjects to support:

- Callbacks
- Event processing
- Label inheritance

If no label keywords are specified, the default keywords Local and Private are used. The syntax of a label in EsiObjects is as follows:

label ::= {(L labelkeyword)} labelname

### Label Keywords

| Keyword | Description |
|---|---|
| Local | Local to this code body (not inherited). |
| Common | Inheritable by implementations of this method found at ancestor and descendant classes. |
| Private | Cannot be seen from outside this method. |
| Public | Can be found by the external lookup mechanism. Another method for this object could define the label as a callback entry point with the **$EXTCALLBACK** function. |
| Handler | Callbacks can be made to this label from outside the context of any object when they come in with a jacketed nonobject context. The **$ASSOCIATE** function must be used to associate to an object before instance variables can be accessed. Also, you can use **$SELF**. |
| Open | Label can be found and advertised to external routines. It is not jacketed like handler methods. The label can reveal some piece of functionality (for example, generate a random legal file name), but cannot associate to an object. This keyword is not recommended for general use. |

### Label Inheritance

EsiObjects supports label inheritance, which is indicated by preceding the line label with an asterisk (*). The following is an example of label inheritance:

```
DO *TEST
```

The previous example accesses the label TEST that is implemented within the same method at the superclass. TEST must be a public label for this to work.

## Introduction to Preprocessor Directives

Preprocessor directives are used in EsiObjects to affect the compilation of a method. All directives are positional and are in effect once the directive is compiled. See the table below for a list of the preprocessor directives that are supported by EsiObjects.

| Directive | Description |
|---|---|
| **#define** symbol | Defines a symbol. |
| **#undef** symbol | Kills a symbol. |
| **#ifdef** symbol | Compiles the code lines that follow the directive if the symbol is defined. |
| **#ifndef** symbol | Compiles the code lines that follow the directive if the symbol is not defined. |
| **#endif** | Ends the conditional block. |
| **#m** | Compiles the block as standard M code. |

| | |
|---|---|
| **#endm** | Ends M compilation. |
| **#ifver** version | Will compile the code if the current implementation version is greater than or equal to the requested version number. The requested number is the full number such as 4.0.2.8. |
| **#ifnver** version | Will compile the code if the current implementation version is less than the requested number. The requested number is the full number such as 4.0.2.8. |
| **#static** variable | This directive has two meanings dependent upon the type of variable.<br>**For Instance and Class Variables**: If the specified variable is declared as dynamically initialized, the #static directive can be used in the code body to prevent the check to see if it is defined every time it is referenced. When using this directive, the variable must be looked up at least once before the directive is in effect. If it is not, referencing the variable will result in an undefined error.<br>**For Name Pool Variables**: Name Pools are objects that contain N% arrays. They can be linked into hierarchical structures so that variable nodes in super objects will be inherited. For performance purposes, if the #static directive is used on a Name Pool variable, the inheritance check will not be made. |
| **#const** name value | Directs the compiler to create a variable CN%name=value. That is, the value is bound to a CN%name variable. This variable is then available within the context of the methods execution context. |

The **#ifdef** and **#endif** directives are used to compile a section of code only if a compiler symbol has been defined. Some directives are bounded, which means that the affect of the directive ends when some form of the **#end** directive is encountered (for example, **#ifndef**). Other directives affect the compilation of all lines that follow the directive in the method (for example, **#define**).

The following directives are specific to the underlying M platform.

- #ifdef directive
- #static directive

- #const directive

**Example #ifdef directive**

In the following example, the first line contains the **#ifdef** directive. The **#ifdef** directive specifies that the lines that follow are compiled only if the symbol MSM was defined using **#define**. The next two code lines contain the **OPEN** and **USE** commands and the **#endif** directive. The **#endif** directive ends the section that is compiled conditionally.

```
#ifdef     MSM
    OPEN S1:(T%File:"W")
    USE S1
#endif
```

By default, EsiObjects supports the following symbols for the various MUMPS systems:

| | |
|---|---|
| DSM | Digital Standard MUMPS system. |
| MSM | Micronetics Standard MUMPS system. |
| DTM | DataTree MUMPS system. |
| GTM | Greystone Technology MUMPS system. |
| CACHE | InterSystems Cache system. |

The **#static** directive guarantees that a variable reference is static, instead of sparse. A sparse variable reference runs slower because it must determine whether the variable exists before returning its value, and inherit and/or calculate the value if it does not. A static variable can be compiled down to a direct variable reference.

**Example #static directive**

The following examples illustrate the **#static** directive for Instance (I%) and Class (C%) variables as well as Name Pool (N%) variables.

**Instance and Class Variables**

```
    SET I%Height=100
```

In the previous example, assume the I%Height variable has been declared to be dynamically initialized. When the variable is accessed, it will be created and then set equal to 100.

```
#static I%Height

    Set T%H=I%Height
```

In this example, the **#static** directive tells the compiler not too generate the typical lookup code of a dynamically initialized variable I%Height. The variable is expected to be present.

The above behavior applies to Class variables as well.

### Name Pool Variables

```
#static N%Name

    Set T%X=N%Name
```

Name Pools are objects that contain N% arrays. These objects can be linked into hierarchies. When linked, the sub objects inherit N% variables in the super objects. Sometimes it is desirable for performance purposes to eliminate that search.

In the example above, the N%Name variable is declared static. The compiler will not generate inheritance code for it. When reference, it must be defined or it will generate an undefined error.

### Example #const directive

The **#const** directive sets a constant value. This value is substituted throughout the remainder of the method whenever the constant is encountered. Note that constant-type symbols begin with the code CN when used within a method.

The following example illustrates the use of the **#const** directive to substitute the uppercase alphabet.

```
#const UpperCase "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
#const LowerCase "abcdefghijklmnopqrstuvwyxz"
.
.
.
SET L%String=$TRANSLATE(L%String,CN%LowerCase,CN%UpperCase)
```

Alternative coding using literals for the previous example has the following disadvantages:

- Accessor variables could be used, but this runs slower because a literal is faster than a variable access.
- Literals could be used instead of constant references, but this does not change run-time performance, requires more typing, and is harder to read. Also, the overuse of literals increases the maintenance burden because if a change must be made, then all instances of the literals would have to be changed.

### Example #ifver and #ifnver directives

These directives are used to direct the compiler to compile a code block based on the systems internal version number. For example, from version 4.1.0.0 on beyond, EsiObjects supports Static methods.

Prior to version 4.1, it would have been necessary to create an object and bind it to an an O% as follows:

```
Create O%Database=MyLibrary$Database
```

Reference to an Initialize method within the Initialization interface would have had to look like this.

```
Do O%Database.Initialization::Initialize
```

However, from version 4.1 and on, static methods can be accessed directly via the library$classname reference as follows:

```
Do MyLibrary$Database.Initialization::Initialize
```

If you want to accomdate installing code in older and new versions, you would use the directives as follows:

```
#ifver 4.1.0.0

Do MyLibrary$Database.Initialization::Initialize

#endif

#ifnver 4.1.0.0

Do O%Database.Initialization::Initialize

#endif
```

## Message Syntax

This section describes the EsiObjects messaging syntax. A message is the way you access a method or a property of an object. In EsiObjects, you can use the message syntax alone or as an argument to the **DO** command.

The following is the syntax for methods and properties:

```
ows ::=   oref.          {(L keyword)} {*} service
          fullclassname
```

An object-with-service (**ows**) reference includes the following:

| | |
|---|---|
| **Oref** | is an object reference, or the name of a class preceded by an underscore. |
| **fullclassname** | Is the full library$class specification. Using the fullclassname in lieu of an oref forces the system to message the method of a class as static. The Static property of the method must be turned on. |
| **a period (.)** | is a delimiter |
| **Keyword** | is a list of optional keywords as described in the table Method and Delivery Keywords. |
| * | is an asterisk (*) that indicates an ancestor-method call |
| **Service** | is the following:<br>**servicename{methodactualist}** |

The following is the syntax of a service name (**servicename**):

```
servicename ::=     {interface::} method
                    @name@
```

where:

| | |
|---|---|
| **Interface** | is an interface that explicitly specifies the interface containing the method (If an interface is not specified, the method's primary interface is used.) |
| **Method** | is a method or property name |
| **@name@** | is class element indirection that evaluates to a valid service name |

The following is the syntax for a method actual list (**methodactuallist**):

```
Methodactuallist ::=   ( methparams )
                       @( expratom V methparams )
```

A method's actual parameter list consists of the following:

- Method parameters (methparams)
- An expression atom in parentheses whose value are method parameters (@( expratom V methparams))

A method actual list is not allowed in a **GOTO** command.

Method parameters can be passed positionally as expressions, or by keyword using the format keyword:expression.

methparams ::= L { keyword : } expr

When passing parameters in a method actual list, all parameters are passed positionally until the first keyword parameter is encountered, then all remaining parameters must be passed by keyword. If the first parameter is passed by keyword, then all remaining parameters must be passed by keyword.

## Message Delivery Keywords

The following table describes the message delivery keywords that affect the process of delivering a message to an object.

| Keyword | Description |
|---|---|
| EXISTENCE | If the object does not exist, the method returns NULL (""). |
| FILTER | Unknown parameters are ignored. |
| KNOWS | The method returns NULL ("") if the object does not implement the method. |
| PRIVILEGED | Calls the method in a privileged mode. (Privileges are required.) |

## Reserved $Unknown Method and Interface Name

If a method specified within a message does not exist within the specified interface and the **$Unknown** method exists, it will be executed. This is a convenient way to provide default execution for a service if it does not exist.

The same concept applies to the interface specified within a message. If the specified interface does not exist and a **$Unknown** interface does exist, it will be accessed.

The reserved name must be spelled exactly as **$Unknown**.

## How to Tell Methods and Properties Apart

The only reliable way to tell methods and properties apart is by their names. Methods and properties can be chained together when they return object references. For example, the following command refers to the property Elements(T%Loop) of the object T%Object12, where T%Loop can be thought of as an array subscript value.

```
SET T%Object12.Elements(T%Loop).Text="EsiObjects Control"
```

This property's Value accessor returns an object reference, and the Text property of that object is referred to next. The Assign accessor of this property is activated, with the value "EsiObjects Control" as a method input parameter.

For more information about using accessors, see the Using Accessors section of the EsiObjects Programmer's Reference Guide.

## Examples

The following example references the GetBaseItem service of **$SELF**. This service is probably a method rather than a property because its name implies a procedure rather than an attribute.

```
SET T%EntryNumber=$SELF.GetBaseItem
```

The following example calls the IssueError method in the FMUtil interface of **$SELF**. Two parameters are passed positionally to the method.

```
DO $SELF.FMUtil::IssueError("FM_BADIDX","Bad index in file.")
```

The following example calls the Assign accessor to copy the contents of array node T%Field of the instance variable I%RequiredFld into the array node T%Field of the RequiredFields property in the T%NewObj object's FMUtil interface.

```
S T%NewObj.FMUtil::RequiredFields(T%Field)=I%RequiredFld(T%Field)
```

The following example references the Name and Library properties of the class Collection.  Note that the class name is preceded by an underscore.  (Normally class references include the library, as well.)

```
SET T%Library=_Collection.Library

SET T%Class=_Collection.Name

DO $ENV.Assert("Class: "_T%Library_"$"_T%Class)
```

## Callback Syntax

The **$CALLBACK** function returns a callback frame identifier used in calling back to a public label in the current method. This function passes in callback and type information.

A method uses this function to create such an identifier, which is then passed to an external object. The external object can then call the label directly without incurring the overhead of a full-blown method call. The callback runs in the context of the object that created the callback.

The syntax of a callback can be one of the following:

```
D{O} { postcond }  SP < expratom > { ( L expr )} { postcond }

G{OTO}{ postcond } SP  < expratom > { postcond }

$$ < expratom > { ( L expr ) }
```

The expression atom enclosed in angle brackets must evaluate to a callback frame identifier. Inside the callback, the value of **$QUIT** varies according to the type of callback that is made.

## Callback Types and Options

When a callback is created, certain other information can be specified in addition to the callback label. The type of callback determines the stack frame where the callback occurs and the state of its method-related symbols.

| Type | Description |
|---|---|
| Original | Callback to creator's stack frame. |
| Capture | Callback capturing creator's method-related symbols. |
| Initialized | Callback that starts with a clean variable context. |

Callback Options are true or false flags that can be specified on some types of callback. They are not mutually exclusive — it is possible to have a callback that is persistent and additive, as shown in the following table:

| Option | Description | Applicable Types |
|---|---|---|
| Persistent | Survive for the duration of the creating object. | Capture, Initialized |
| Additive | Preserve variable state between calls. | Capture, Initialized |

Original callbacks are the most common. They dispatch directly to the actual EsiObjects method stack frame that created the callback. (Note that the EsiObjects method stack is not the same as the underlying M process stack.) The callback is automatically freed when that stacks frame exits. Therefore, the callback can only be made from methods called before that stack frame has exited.

Original callbacks are often used for enumeration and searching operations. The method-related symbols of the method that created the callback are always visible during this kind of callback. Such callbacks are always additive and are never persistent.

The following example illustrates a simple enumeration operation.

```
    SET T%CallBack=$CALLBACK(UPDATE,0)
    SET T%Result=0
    DO T%Object12.CountItems(T%CallBack)
    DO $Env.Output("Total items found: "_T%Result)
    QUIT
    ;
UPDATE    ;Callback handler
    SET T%Result=T%Result+1
    QUIT
```

The method creates an original (type 0) callback to the label UPDATE in the variable T%CallBack, and sets an accessor variable T%Result to the initial value 0. It then passes the callback as a parameter of the CountItems method for two separate objects. Whenever CountItems detects a problem it performs the callback. Each time the callback is made, the subroutine in UPDATE increments T%Result. When CountItems returns, T%Result is therefore equal to the number of callbacks made during the two method calls.

Capture callbacks record the callback creator's method-related variables. If the callback is additive, then changes to those variables are preserved between calls. If it is not additive, then the variables always are reset to the value at the time the call was made. If the callback is persistent, then it survives for the lifetime of the creating object, or until it is explicitly freed with the **$FREECB** function.

The following example enumerates all items in collection T%Collection. The variable T%Sum is a cumulative sum of the value properties of each item. As the items are added, the output window (T%OutWind) is updated with the current sum.

```
    SET T%Callback=$CALLBACK(Sum:1,1)
    SET T%Sum=0,T%OutWind=I%
    DO T%Collection.Enum(T%Callback)
    QUIT
Sum(Obj) ; Sum each object
    SET T%Sum=T%Sum+Obj.Value
    SET T%OutWind.Text=T%Sum
    QUIT
```

Initialized callbacks are used internally as the backbone of events and watches. The callback starts with an entirely clean variable context. However, if it is additive then any changes to those variables are carried over to succeeding callbacks.

## Callback Ownership and Lifetime

The creator of a callback is usually considered to be its owner. Only the owner should free the callback.

The lifetime of a callback never extends beyond the lifetime of the creating object. If the object dies, the callback is automatically freed. Original callbacks continue to exist until the creating stack frame terminates, when they are automatically freed. Nonpersistent callbacks are freed whenever the incarnation of **$ENVIRONMENT** changes (in other words, whenever EsiObjects is shut down or restarted). In general, Capture and Initialized callbacks should be freed explicitly with **$FREECB** when their usefulness has ended.

## Documenting Callbacks

All methods that create callbacks or call them should clearly document the callback interface they assume. This includes the following information:

- The number of parameters, if any
- The purpose and use of each parameter
- Whether the callback is to be made as a subroutine or extrinsic function
- The expectations of callback ownership

Note that the validity of callback parameters is determined only at run time. This adds to the importance of adequate documentation.

## Extrinsic Functions - Introduction

Extrinsic functions are user-written functions, which use a parameter passing mechanism. The following is the syntax of an extrinsic function:

$$labelref({argument}{,...})

where:

| | | |
|---|---|---|
| **$$** | identifies an extrinsic function call | |
| **labelref** | is a line label or entry reference | |
| **argument** | is an actual argument list that can be one of the following: | |
| | **value_expr** | |
| | **.name_expr** | |
| | where: | |
| | **value_expr** | is a valid expression |
| | **.name_expr** | Unsubscripted local variable name, or an indirect reference that evaluates to an unsubscripted local variable name |

Entries in an actual list can be passed by reference or can be passed by value. Pass by reference occurs when an entry in the actual list has a period before it. Pass by reference evaluates the argument as a pointer to a local symbol and passes the pointer to the called subroutine. Pass by value evaluates the argument as a value expression and passes the value to the called subroutine.

Note the following:

- An error occurs if a formal list is shorter than the actual list. An actual list can be shorter than the formal list.
- An actual list can be an empty list. An empty list is defined with open and closed parentheses and no actual list arguments.
- A formal list can also be an empty list. An empty list is defined with open and closed parentheses and no formal list arguments
- Indirection is not allowed in the formal list.
- Names in formal lists must be unique.
- Only the length of a line restricts the length of an actual list and a formal list.
- An actual list that is passed by value can be any valid expression and does not have to be a local variable name.
- An actual list argument that is passed by reference must be a local variable name, or an indirect expression that evaluates to a local variable name.
- When you use a **DO**, **GOTO**, or an extrinsic function with an entry reference (**Label^Routinename**), you are leaving the context of the object. This violates the concept of encapsulation and generally is not recommended.

For more information about using extrinsic functions, see the DO, NEW, and QUIT command sections.

## Extrinsic Function Examples

The following example shows how to find the square root of a number using an extrinsic function.

```
SQRT(X) QUIT X**.5
```

The following example shows an error output function that returns the level of the error and the text associated with the error.

```
ERROR(Level,Text) ; Error output function
    IF Level<3 QUIT 0
    DO $ENV.ReportError(Text)
    QUIT 1
```

## Syntax of an Extrinsic Function Callback

The following is the syntax of an extrinsic function callback:

$$<cbref>[(cbactlist)]

where:

| | |
|---|---|
| **Cbref** | is an expratom V as a callback frame ID string |
| **Cbactlist** | is a list of parameters to be passed positionally the callback by value. |

For example:

```
SET T%Next=$$<T%Srch>
SET T%Key=$$<T%Lkup>(T%Key)
```

# Using Expressions

## Expressions

Arguments of EsiObjects commands are made up of expressions. Expressions, which are character strings, can contain one or more elements and are connected by operators. An expression yields a value when it is interpreted. An expression must contain at least one element, which is called an expression atom.

Expression atoms can be one of the following:

- Literal
- Variable
- Function
- Expression atom preceded by a unary operator
- Messages

There are three types of operators:

- Indirection (only the INDIRECTION operator)
- Binary
- Unary

Expressions can be composed of one expression atom or can be made up of a series of expression atoms separated by binary operators. Binary operators test the relationship between two expression atoms or expression and return a result.

Unary operators perform an operation on a single expression atom or expression to the right of the operator.

For more information about messages, see the Message Syntax section of this guide..

### Literals

There are two types of literals:

- Numeric
- String

Numeric literals are strings that get evaluated as numbers. A numeric literal contains only the following:

- Digits 0 to 9
- Unary MINUS ( – ) operator
- Unary PLUS ( + ) operator
- Period or decimal point character
- Letter E (for exponential notation)

The following is the format of exponential notation:

{–}mantissa{–}exponent

Exponential notation lets you enter very large or very small decimal numeric literals. The expanded result cannot exceed 31 characters.

String literals are sets of zero or more of the 128 ASCII characters. You must always enclose strings in quotation marks. String literals can consist of the following:

- Numbers
- Uppercase and lowercase letters
- Punctuation characters (for example, $, !, or &)
- Control characters

The only limitation on the length of a string is on the length of a line (511 bytes).

## Evaluating Expressions

In EsiObjects, all binary operators share the same precedence. Statements are evaluated from left to right in the following sequence:

- All occurrences of indirection
- All unary operators
- All expressions in parentheses
- All expressions with binary operators

You can change the order of evaluation with parentheses. Expressions in parentheses are evaluated (in a left-to-right order) before the entire argument gets evaluated.

You can also concatenate expressions with the binary CONCATENATE ( _ ) operator

# Variables

## Syntax of a Variable Name

A variable is a reference to a storage location. A variable can be unsubscripted or subscripted. A subscript can contain integers (positive or negative), decimals, numeric (positive or negative), or string literals.

The syntax of a variable in EsiObjects, which can be used on some or all of the references to the variable, is as follows:

Code%Name{subscript1, . . . subscriptn}

where:

| | |
|---|---|
| Code | is one of the codes defined in the table Variable Names and Scoping Codes, which determine explicitly the scope of the variable |
| % | is a one-character delimiter that separates the code from the name |
| Name | is the name of the referenced symbol. Symbol names have a permitted length of 1-31 alphanumeric characters. The first character must be alphabetic. The are case sensitive, that is, the symbol ABC is different from AbC. |
| Subscript | is an expression that uniquely identifies a node in an array |

## Variable Names and Scoping Codes

The following table describes the different kinds of variable names supported by EsiObjects. Note that the codes in the table are not case sensitive.

| **Variable** | **Code** | **Scope** | **Description** |
|---|---|---|---|
| Parameter | **P** or **p** | Single method call | A variable whose value is assigned when it's associated parameter is supplied with the message. If the associated parameter is not specified, the parameter variable is undefined. The variable's lifetime ends when the method terminates execution. |
| Temporary | **T** or **t** | Single method call | A variable whose value is created and modified as needed throughout the method's code. The variable's lifetime ends when the message terminates execution. If a code is not specified when setting a variable, the default is generally an temporary variable. |
| Accessor | **A** or **a** | Single method call | The same as a T% variable. Use T% in place of A%. |

| | | | |
|---|---|---|---|
| Instance | **I** or **i** | Object | A variable that can be accessed by any code that is executed inside the context of its object. When the object dies, the instance variable's lifetime also ends. |
| Constant | **CN** or **cn** | Between directives | A variable that is a constant value. It cannot be set directly. It can only be set through the #const compile directive. |
| System | **S** or **s** | Life of the environment | A variable that is accessible to all methods. Only privileged code can alter system variables. |
| Global | **G** or **g** | All contexts | A variable that can be shared by all users. A global variable exists until it is specifically killed with the **KILL** command.<br><br>Note that the circumflex (^) syntax for globals is supported for backward compatibility. |
| Class | **C** or **c** | Class and all instances of it | A variable whose scope is limited to the context of a class object and all instances of that class. It can be accessed by any code that is executed inside the context of the class or one of its instances. Like its class, a class variable exhibits persistence. It continues to exist until it or its class is explicitly destroyed. |
| Local | **L** or **l** | Partition or NEW | A standard M local variable whose scope is global within its M partition. Local variables can be referenced from anywhere within the partition in which they exist. Their lifetime ends when they are explicitly killed, their job terminates, or their **NEW** context expires.<br><br>Local variables generally are not used in EsiObjects. |
| NamePool | **N** or **n** | Global | A symbol table that can contain a variety of symbols. The symbols can be created, referenced, and destroyed in any context. NamePools objects can exhibit persistence. Additionally, NamePool objects can be used as Domains. |
| Universal | **U** or **u** | Same as a M Global with translations. | A variable that is equivalent to a M global. Not used that often. |

| | | | |
|---|---|---|---|
| Domain | **O** or **o** | Global across processes – same as a traditional UCI. | A variable that is available to any object that resides in a specific domain. It disappears when the domain is destroyed or it is explicitly killed. |

The variable reference T%Accum refers to the accessor variable Accum, and the variable reference I%Height refers to the instance variable Height.

## Value of Variables

A variable can evaluate to two types of values in EsiObjects:

- Built-in
- Object identifier (OID)

Built-in values are the standard values supported by M. An OID is a special EsiObjects handle to an object. Built-in objects are normally treated the same way as M symbols, although messages can be sent to them if necessary.

The following table describes the different types of built-in classes.

| **Built-In Class** | **Description** |
|---|---|
| BuiltInString | Any M string value. Restrictions of length or the ASCII characters that can be used usually depend on the native M system. A built-in string can contain an OID, but it generally treats the OID as a string rather than as an object. |
| BuiltInNumeric | An M string that is interpreted numerically. There is nothing to prevent string values from being assigned to a BuiltInNumeric symbol, but it generally is interpreted as a numeric value. |
| BuiltInBoolean | An M string that is interpreted logically. Like BuiltInNumerics, any value can be assigned to a BuiltInBoolean symbol, but the value generally is interpreted logically. |
| BuiltInArray | A standard M array that can contain any number of nodes. The limitations on BuiltInArray subscripts and values are imposed by the native M system. In evaluating these limits, keep in mind that a BuiltInArray instance variable can have its own root at an M array node. |

In contrast to built-in symbols, an OID is an encoded string that is used as a handle to an object. Because a built-in string can contain any characters in the ASCII character set, it is difficult to devise a reliable test between built-ins and OIDs. You can use the **$EXIST** function to make this distinction.

The following table describes the value returned by **$EXIST** in a variety of cases:

| Type of $EXIST Argument | Value Returned |
|---|---|
| Built-in string, numeric or Boolean | B |
| Built-in array | B |
| OID handle of nonexistent object | 0 |
| OID handle of existing object | 1 |

## Variable Scoping

### Variable Scoping - Introduction

The scope of a variable defines the set of messaging contexts in which that variable can be referenced. In M, the scope of a local variable restricts its accessibility in such a way that it cannot be referenced outside the context of the partition in which it is contained.

The lifetime of a local variable ends when its job terminates and the partition dies. Similarly, an instance variable of an object cannot be referenced outside the context of that object and is destroyed when its object dies. For this reason, instance variables are hidden inside their objects and cannot be directly accessed by other objects. This process of hiding is known as encapsulation.

For more information about encapsulation, see the What is an Object? section of the EsiObjects Programmer's Reference Guide.

### Why Is Scoping Important?

Scoping is important for the following reasons:

- Allows you to refer to the instance variable X and always get an instance variable of the current object, without having to specify the object identifier (OID) of the object whose instance variable is being referenced. This makes it easier to write generic code.
- Allows you to call other methods inside the context of a single method call without having to worry that the scoped variable is going to get overwritten or modified by a poorly behaved method that gets called. If a bug causes such a variable to contain the wrong value, scoping of variables makes it much easier to locate that particular bug.
- Prevents you from accidentally modifying variables that are outside the scope and explicitly prevents scoped variables from being tampered with externally.
- Allows you to define short, concise, simple names without needing to worry that those names are already being used outside the scope of a given variable. For example, it is possible to define a temporary variable Data, scoped inside a single method call, without having to worry that some other method is using a temporary variable of the same name.

## Implicit Scoping

Scoping considerations are taken into account whenever a variable is referenced in EsiObjects. If a variable is referenced by its name without scoping information, then EsiObjects must determine the scope of the variable.

For example, suppose a method refers to the variable Text. How does EsiObjects know to which kind of variable the symbol Text refers? EsiObjects uses five criteria to determine the scope of a symbol, in the following order of precedence:

- If Text is defined as an instance variable by the class, then the symbol Text is an instance variable.
- If Text is defined as a class variable of the class, then the symbol Text is a class variable.
- If the symbol Text is defined as a parameter of the method, then the symbol Text is the associated parameter variable. If Text is defined as some variable into which a parameter is mapped, then the symbol Text is the mapped variable.
- If the symbol Text has been encountered higher up in the method, then this symbol Text is the same symbol as the symbol Text encountered nearest the top of the method.
- If none of the preceding four conditions is satisfied, then the symbol Text is a universal variable.

In compiling a method, EsiObjects records the default scope of each variable encountered in a top-to-bottom scan of the method's code. This information is used in number 4 in the previous list. The first time a variable is encountered in the scan, it defines the default scope for that variable whenever it is encountered in the method. The scanning process proceeds in a strict top-to-bottom, left-to-right fashion without regard to order of evaluation or order of execution.

Because the first reference to a variable encountered in the method determines the default scope of that variable for the entire method, be aware that reorganizing the code in a method can occasionally result in changing which reference to the symbol is encountered nearest the top of the method. As a result, the default scope of the variable can be changed inadvertently if both explicit and implicit scoping is used in the same method.

These rules for determining the scope of a variable also apply to the arguments of the **NEW** command or to the symbols included in a label's formal parameter list, which only allow local variable names.

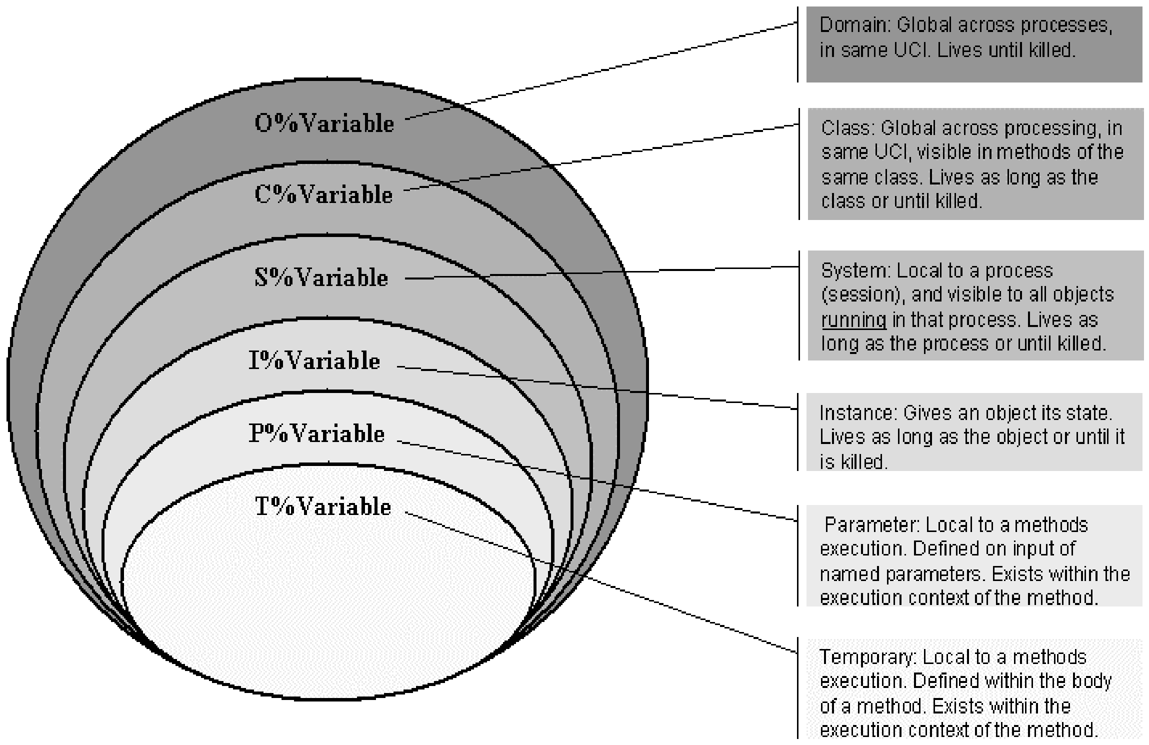
## Explicit Scoping

EsiObjects provides syntax for explicitly defining the scope of a variable. It is possible to have two symbols with the same names but different scopes in the same method. In such cases the first symbol encountered sets the default scope for implicitly scoped symbols of that name in the entire method.

For example, if T%Sym is encountered near the top of a method and L%Sym is encountered later on, followed by a number of different references to the symbol Sym, then the references to Sym are scoped as T%Sym because it occurred nearest the top of the method. However, if the reference to L%Sym is moved higher up in the method than T%Sym, then it defines the scope of the Sym references.

## Scope Hierarchy

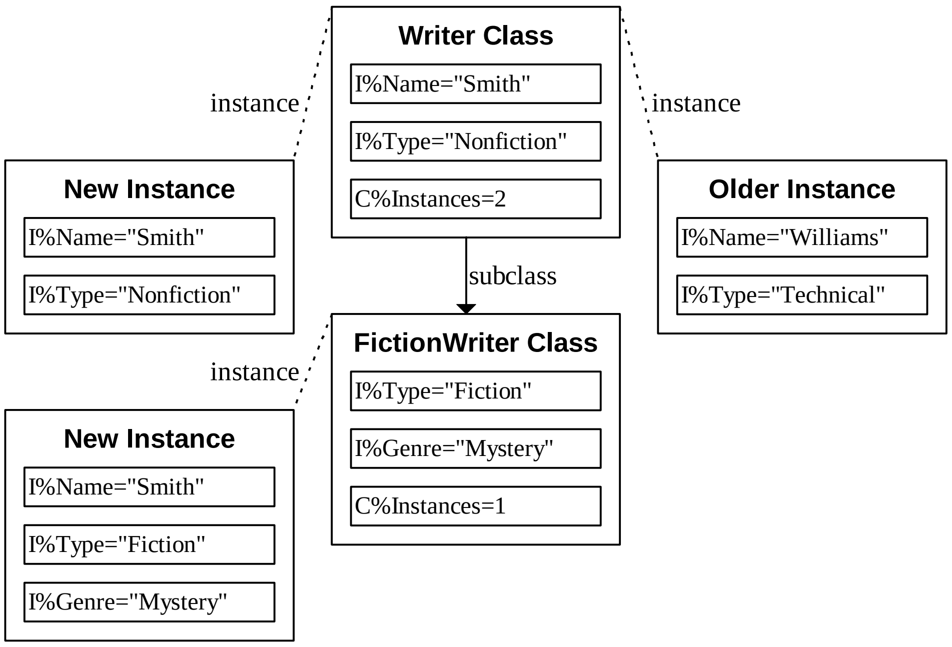
EsiObjects variables are scoped according to a specific hierarchy. That hierarcy is illustrated in the diagram below.


O%Variable
C%Variable
S%Variable
I%Variable
P%Variable
T%Variable
Domain: Global across processes, in same UCI. Lives until killed.
Class: Global across processing, in same UCI, visible in methods of the same class. Lives as long as the class or until killed.
System: Local to a process (session), and visible to all objects running in that process. Lives as long as the process or until killed.
Instance: Gives an object its state. Lives as long as the object or until it is killed.
Parameter: Local to a methods execution. Defined on input of named parameters. Exists within the execution context of the method.
Temporary: Local to a methods execution. Defined within the body of a method. Exists within the execution context of the method.

**Variable Scoping Hierarchy**

## Variable Inheritance

The following diagram shows the difference between class and instance variables:



In this diagram, the class **Writer**, which has two instances, is a superclass of **FictionWriter**, which has one.

Writer implements a class variable called **Instances**, which is common to all instances of that class. (Perhaps this variable would keep track of the total number of currently existing instances of the class.) In other words, a reference to the variable **C%Instances** from within either instance of **Writer** would return the value **2**. Because **FictionWriter** is a subclass of **Writer**, it inherits the definition for Instances (but not its value.) Any instance of **FictionWriter** that refers to its Instances class variable will thus get the value **1**. Changes to this value will not affect **Writer**'s class variable, and vice versa.

**Writer** implements two instance variables, called **Name** and **Type**. All instances of **Writer** and its subclasses will be created with the default values of **"Smith"** for **Name** and **"Nonfiction"** for **Type**. (Note that the newly created instance of **Writer** has those values.) Since each instance of the class has its own instance variables, those values can change without affecting other instances. (Note that the older instance of **Writer** has acquired different values since it was created.) Subclasses of **Writer** will generally inherit the same instance variable definitions (note that the new instance of **FictionWriter** also has **Name** and **Type** variables.) However, a subclass may override those variable definitions, by changing the particulars. Note that **FictionWriter**'s **Type** variable is defined with a default value of **"Fiction"**, and the new instance is created with that value.

Subclasses may also extend the superclass variables by defining additional variables that are not known to the superclass. **FictionWriter** defines an instance variable called **Genre**, which defaults to **"Mystery"**. The newly created instance reflects this, but the superclass and its instances are unaffected.

**Variables** are not inherited if they are marked as ***Private***. For example, if the **Name** instance variable had been marked as private at the class **Writer**, then FictionWriter would not have inherited it. Most instance variables are ***Public***. A private variable (or method, property, etc.) is only defined when there is some compelling reason why it would be inappropriate for subclasses to inherit it.

### *Example: Class Variable Inheritance*

As stated above, class variables are only accessible to the class that implements them. The definition of a class variable is inherited by subclasses, but any methods compiled at the level of the subclass will access another copy of the class variable, stored with the subclass. If a subclass inherits a method from a superclass, then any class variable references in that method will access the class variable at the superclass, not the class variable at the inheriting subclass. The following example illustrates the proper use of class variable inheritance.

In this example, the **superclass** has a list of errors in a class variable **C%Errors**. The **subclass** has an entirely different copy of **C%Errors**. Since the subclass cannot access the contents of the superclass variable, it calls code at the superclass, designed to accomplish this same task.

The method **InitializeError** table initially sets up the class variable. It is only run once, as part of class initialization. The method **FindError** is called to return a descriptive string of an error, whenever one is needed.

Superclass method "InitializeErrorTable"

```
Input: () ; Initialize Error Table

    SET C%Errors("NOERR")="No error"

    SET C%Errors("GENERAL")="General Error"

    SET C%Errors("BADINP")="Bad Input"

    Q
```

Superclass method "FindError"

```
Input: (T%Err="NOERR") ; Find Error

    IF '$data(C%Errors(T%Err)) SET $RETURN="Unknown Error "_T%Error QUIT

    SET $RETURN=C%Errors(T%Err)
```

Subclass method "InitializeErrorTable"

```
Input: () ; Initialize Error Table
    SET C%Errors("NOERR")="OK"
    SET C%Errors("OUTRNG")="Out of Range"
    QUIT
```

Subclass method "FindError"

```
Input: (P%Err="NOERR") ; Find Error
    IF $data(C%Errors(P%Err)) SET $RETURN=C%Errors(P%Err) QUIT
    SET $RETURN=$SUPER.FindError(P%Err)
    QUIT
```

When the **FindError** method is invoked for the subclass, it examines its own class variable **C%Errors** to determine if it contains an entry for the error. If so, it returns a description of the error. However, if it does not know about the error, it calls **FindError** for the superclass, in hopes that perhaps the superclass understands the error.

## InitClassVars and InitSysVars Methods

### *InitSysVars*

The **InitSysVars** method is called whenever a new object is created. For example, if class Patient implements an **InitSysVars** method in its Factory interface, and a new instance of the Patient class is created, then the **InitSysVars** method is invoked to set up some of the instance variables of that class.

The **InitSysVars** method is the primary means for defining special-purpose, inheritable code to set up instance variables. Part of this method is generated automatically whenever instance variable definitions are added in the variable definition editor. This part occurs between the **EOAUTO START** and **END** lines.

### *InitClassVars*

The **InitClassVars** method is called whenever a new class is added to the system. For example, if a brand new class called **Argyle** is added as a new subclass of **Plaid**, and the **Plaid** class implements the **InitClassVars** method in its **Factory** interface, then this method will be executed to set up the class variables

The **InitClassVars** method is the primary means that EsiObjects uses to implement class variable inheritance. Part of this method is generated automatically whenever class variable definitions are added in the variable definition editor. This part occurs between the **EOAUTO START** and **END** lines.

### *General Considerations*

You must ***never*** insert or modify code between the **EOAUTO START** and **END** lines, because that code is liable to be overwritten by EsiObjects the next time that the method source is automatically generated from the variable definition, which can occur under a variety of different circumstances.

The programmer who wishes to add setup code for class variables can use the **InitClassVars** method to do so, provided that the setup code does not fall between the **EOAUTO START** and **END** lines. Note that the **InitClassVars** method contains a **$SUPER** call allowing the same method to be executed at the parent class. Code is usually added after the **$SUPER** class.

## NamePool Variables

NamePool variables reside in an object created from the Base$NewNamePool class. This object is essentially a symbol table that is sharable by other objects, that is, if an object owns the name pool objects OID, it can access all name pool variables contained within the pool object.

An additional feature of name pools is that they can be linked into hierarchies through methods available in the NewNamePool class. When name pools are linked into hierarchical structures, the variables are inheritable.

Explicit reference to a variable within a name pool is as follows:

N%(OID)VarName

where OID is the name pool objects OID and VarName is the name of the variable within the pool. When used in this format, the EsiObjects compiler builds in the required support needed to produce a value.

An alternative implicit access is available that avoid specifying the (OID) specification on every variable reference. By setting the $Pool special variable to the OID prior to using the name pool variable, the (OID) explicit reference to the pool object can be ignored. For example:

```
      Set $Pool=OID
      Set N%VarName="Value"
```

The value of $Pool is stacked and exists within the execution scope of the current method. As in the case of the explicit syntax, the compiler resolves the reference to the proper name pool object.

NamePool variables are very powerful. When a need arises for symbol tables that are sharable and optionally inheritable, this variable scope should be used.

### *Name Pool Inheritance*

NamePool variables reside in an object created from the Base$NewNamePool class. This object is essentially a symbol table that is sharable by other objects, that is, if an object owns the name pool objects OID, it can access all name pool variables contained within the pool object.

Name Pool inheritance occurs when name pool objects are linked into hierarchies. In such cases, the child name pool will inherit those variables in the parent name pool that are not explicitly overridden. The following method call is used to create a name pool in the variable T%ChildPool. This new name pool is a child of the I%ParentPool instance variable.

```
SET T%ChildPool=I%ParentPool.Factory::CreateDescendant
```

Note that you can only link two name pools together when creating a new one, and that the newly-created name pool must be a descendant of the previously-existing one.

## Grouping Code into Interfaces

In EsiObjects, program code implemented by any class is generally inherited by its subclasses. Program code is grouped into interfaces, with the **Primary** interface being used for messaging by general users of the object, and all other interfaces being special-purpose in nature.

### *Interfaces*

Each class implements at least one interface, the **Primary** interface. This interface is used for messaging by general users of the object. A large percentage of classes implement a **Factory** interface, which is reserved for the details of object creation and destruction. Many classes also implement other, special-purpose interfaces. Knowing whether a class requires one or more special-purpose interfaces is by no means an exact science. However, the following guidelines constitute useful general principles:

- Any methods or properties (such as its **CREATE** and **DESTROY** methods) that are used to define the object's initial state when it is created, or to clean up the object as it is being destroyed, usually go in the **Factory** interface.
- (**Exception:** the **Create** and **Kill** accessors of a property, if they exist, are most often defined in the **Primary** interface.)
- Any methods or properties that are suitable for use by any external object should go in the **Primary** interface. For example, if a **Dictionary** collection contains a group of objects arranged by the value of a common property, then that property is usually contained in the **Primary** interface.
- Any methods of properties that require a special relationship to the object should go in a special-purpose interface. Such special relationships are described below.

### *Special Relationships*

Sometimes, different instances of a same class will have a special-purpose interface, or a factory object dedicated to producing instances of a specific class might communicate with those instances using a special-purpose interface. In both these cases, it would be inappropriate for "the average object" to make this type of communication.

The name of a special-purpose interface should reflect the purpose, and where possible should not conflict with special-purpose interfaces having different meanings in other groups of classes.

Sometimes, when special relationships between classes are required, the objects will implement "challenge code" as part of their special-purpose interfaces. For example, if a certain method can only be invoked by one particular object, then the program code for that method might check **$CALLER** before going ahead with its task. In other cases, more elaborate challenge code might be appropriate. (This is conceptually similar to a bank refusing to give out information about an account to someone who is not an account holder.)

## Interfaces and Inheritance

Interfaces are groupings of services, that is, methods, properties, events and relationships within a class. A number of related services are usually part of the same interface. Every class implements at least one interface, the **Primary** interface.

Assume there exists a **Person** class that implements a **Primary** interface containing a **Name** property. Now assume there is a **Physician** class that is a sub-class of **Person** (meaning that a physician object is a kind of person object.) **Physician** automatically inherits the **Name** property from **Person**.

The subclasses of a class will always inherit any interfaces that it defines. (Of course, all the methods and properties in that interface will be part of the inherited interface.) If a subclass wishes to extend a special-purpose interface, then the entire interface must first be overridden. The subclasses of a class will always override the **Primary** and **Factory** interfaces.

Within an interface, elements such as methods, properties, and event templates are individually inherited by subclasses, except when defined as private. Any inherited element may be overridden at a lower level. If the overriding element is private, then subclasses will inherit the parent implementation. (For example, if **Pediatrician**'s **Specialty** property had been private, then any subclasses of **Pediatrician** would inherit the version of **Specialty** defined by **Physician**.)

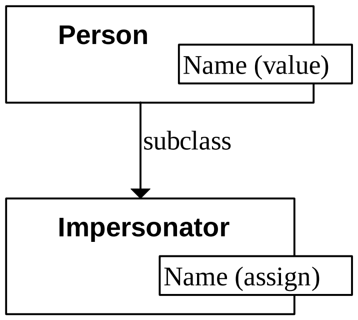
### *Code Inheritance in Projects*

From a language perspective, each EsiObjects project is treated as a subclass of **Base$Application**. Thus, all of the **Procedures**, **Command Handlers** and **Event Handlers** are treated as methods in the project's **Application** class. They can be inherited between projects by defining a new project as a subclass of an existing project.

An alternative is to create a subclass of **Base$Application**, and to define multiple projects as children of your new **Application** subclass.  That way, some of the code can be promoted to the common parent, and shared among multiple projects.

### *Properties and Accessors*

Properties are inherited by subclasses, and each accessor is inherited separately. When a subclass overrides a property, it will still inherit all its accessors. It may then explicitly override one accessor while continuing to inherit another.



In this diagram, the class **Person** implements a **Name** property with a value accessor, allowing external objects to obtain the value of the Person's name. However, external objects cannot assign the value of the **Name** property (just as no one else can change your name for you.) **Impersonator** is a subclass of **Person**. Let's suppose that an Impersonator object can assume a different Name. In that case, it might sometimes be inappropriate for an external object to attempt to assign the **Impersonator** object's **Name** property. Thus it implements an assignment accessor.

In this example, **Impersonator** has overridden the **Name** property and implemented an ***assignment*** accessor, while continuing to inherit the ***value*** accessor from **Person**. Any subclasses of **Impersonator** would actually inherit both accessors.

## Mix-in Classes and Multiple Inheritance

There are two forms of multiple inheritances supported by EsiObjects.

1. **Supertyping** - a subclass has two fully specified parents, both of which are intended to represent full-bodied classes in their own right.

2. **Mix-ins** - one of the superclasses is a special kind of abstract superclass designed to be used only with multiple inheritance.

**Supertyping** represents a break with the classic, taxonomic philosophy of building the class hierarchy. It means that one kind of object (the subclass), is a fusion of its two superclasses. Supertyping is a controversial approach to use, because it violates the strict descendant-tree structure of a taxonomy. More practically, it leaves the door open to multiple inheritance conflicts.

**Multiple inheritance conflicts** occur when more than one superclass implements an element of ***the same name, in the same interface***, and this element is not explicitly overridden by the child class. EsiObjects automatically resolves multiple inheritance conflicts using proximity: an immediate parent will always take precedence over a less immediate one. However, if the two competing superclasses are the same distance away in the tree, then a conflict will result. In such cases, the child class must override the conflicting element. It can then use the syntax for explicit vectoring to generate a call to the appropriate superclass.

**Mix-ins** are special-purpose classes designed to facilitate strategies for avoiding multiple inheritance. Mix-in classes are always abstract, never concrete, and they generally deal with one specific aspect of an object's state or behavior. Multiple inheritance conflicts can also occur with mix-ins, but they are less likely because of the general design strategies generally associated with this type of class.

The difference between these two types is more philosophical than structural. While EsiObjects has a special **mix-in** class type, the biggest difference between the mix-in and supertyping forms of multiple inheritances is in the class design process, rather than in any specific feature of the classes themselves. In other words, it is largely the ***connotations*** that the two class types have in the minds of programmers (similar to the difference between club soda and sparkling water).

# Commands

A command is a name for an action that is performed. Most commands take arguments. An argument can encompass a variety of syntactic elements (such as numeric expressions, variable names, **SET** arguments) that define and control the action of the command. Some commands are argumentless and some commands take arguments only in certain circumstances.

Each command is labeled as to its ANSI Standard status as described in the following table:

| Status | Description |
|---|---|
| Standard | Indicates that the language element is part of the M ANSI Standard. |
| Proposed | Indicates that the language element is being proposed as an addition to the M ANSI Standard. |
| Extended | Indicates that the Standard language element has been modified for use in EsiObjects. |
| EsiObjects | Indicates that the language element is not part of the Standard and is an extension of EsiObjects. |
| Vendor | Indicates that the language element is M vendor-specific. |

## BREAK

The **BREAK** command interrupts the normal flow of execution, invoking the EsiObjects interactive debugger.

### Format

B{REAK} postcond

postcond ::= { : tvexpr }

A truth valued expression. Execution of the command is conditional based on the truth-value of this expression.

### Explanation

The debugger is an important tool and can be used frequently during the debugging process. EsiObjects programmers can use this interactive approach to spot problems quickly and verify that code is working as intended.

### Comments

To use the interactive debugger, the following must be accomplished:

The debugger must be activated.

**BREAK** commands must be inserted in the code to activate the interactive debugger at execution time.

A debug version of the code must be compiled.

## CLOSE

The **CLOSE** command releases ownership of one or more devices owned by the current process. In some cases, certain device-dependent operations can be performed as part of this process.

### Format

C{LOSE} postcond SP L closeargument

postcond ::= { : tvexpr }

A truth valued expression. Execution of the command is conditional based on the truth-value of this expression.

closeargument

Can be one of the following:

expr { : deviceparameters }
@ expratom V L closeargument

where:

| | |
|---|---|
| **Expr** | is any expression whose value is a device identifier of a device owned by the current process |
| **Deviceparameters ::=** | **deviceparam** |
| | **(deviceparam {:deviceparam }...)** |
| **Deviceparam ::=** | **expr** |
| | **devicekeyword** |
| | **deviceattribute = expr** |

### Explanation

The **CLOSE** command gives up ownership of the devices specified in its arguments, making them available to other processes. The format of a valid device specifier, the range of available devices, and the valid device parameters are all M platform-dependent.

After a **CLOSE** operation closes the current device, **$IO** becomes some other device identifier, usually the principal device **$PRINCIPAL**.

## Comments

Keep the following points in mind when you use the **CLOSE** command:

- The kinds of device parameters specified with **CLOSE** are generally related to terminating use of the device.
- Multiple device parameters are enclosed in parentheses, separated by commas.
- Closing an unowned device has no effect.
- There is no argumentless form of the **CLOSE** command.
- **HALT** automatically closes all devices.

## Related

OPEN command

READ command

USE command

WRITE command

$IO special variable

$PRINCIPAL special variable

## DSM and MSM Examples

The following example relinquishes ownership of the device whose identifier is in the symbol T%InputDevice.

```
CLOSE T%InputDevice
```

The following DSM example reads lines of text from a file whose identifier is in the variable T%File and echoes these lines to the principal device (**$PRINCIPAL**) until a blank line is encountered, when the file is closed.

```
OPEN T%File::10
ELSE DO $Env.Output("Device "_T%File_" is unavailable.") QUIT
FOR  DO  QUIT:T%Line=""
. USE T%File
. READ T%Line
. IF T%Line="" QUIT
. USE $PRINCIPAL
. WRITE T%Line,!
CLOSE T%File
QUIT
```

The following MSM example reads lines of text from a file whose identifier is in the variable T%File and echoes these lines to the principal device (**$PRINCIPAL**) until a blank line is encountered, when the file is closed.

```
SET T%Dev=51
OPEN T%Dev:(T%File,"R")::10
ELSE DO $Env.Output("Unable to access HFS") QUIT
FOR  DO  QUIT:T%Line=""
. USE T%Dev
. READ T%Line
. IF T%Line="" QUIT
. USE $PRINCIPAL
. WRITE T%Line,!
CLOSE T%Dev
QUIT
```

## DSM Examples

In the following example, the same device T%InputFile is closed and the device attribute RENAME is specified with its value in the variable T%NewFileName.

```
CLOSE T%InputFile:RENAME=T%NewFileName
```

The following example specifies a list of device parameters, enclosed in parentheses and separated by colons.

```
CLOSE I%Printer:(FORMFEED:SPOOL)
```

## CREATE

The **CREATE** command creates a new object of a given class or nested class. Various kinds of information about the object can be specified when it is created.

### Format

CR{EATE} postcond SP L createargument

postcond ::= { : tvexpr }

tvexpr evaluates to a truth-valued expression. Execution of the command is conditional based on the truth-value of this expression.

createargument

Can be one of the following:

crarg
@ expratom V L createargument

where **crarg** is defined as follows:

glvn ={library$}class{(methodactlist)}{:{keywords}{: (L propname=expr}) }

The following additional definitions also apply:

| | | |
|---|---|---|
| **class ::=** | **libraryname$classname** | |
| | **classname** | |
| | **classname>classname>…** | |
| | **@expr@ V classname** | |
| **keywords ::=** | **crkey** | |
| | **(L crkey)** | |
| **crkey ::=** | **Base** | |
| | **Child** | |
| | **Class** | **= expr** |
| | **Fixed** | |
| | **Share** | |
| | **Name** | |
| | **Domain** | |

## Explanation

The **CREATE** command is used to create an object. A three-step process is used to create the object:

1. A primitive instance of the class is stamped out.
2. If the class implements a CREATE method, it is invoked.
3. Instance variable values are applied to the object as though with the **ZAPPLY** command.

In addition to the required class name, the following information optionally can be specified:

- A method parameter list to be used when the CREATE method is called
- A list of special object creation keywords and their values, as appropriate
- A list of property names and their values

When property values are assigned with the **CREATE** command, each property value assignment can have one of three outcomes:

- If a Create accessor exists for the property, then it is invoked.
- Otherwise, if an Assign accessor exists for the property, then it is invoked.
- Otherwise, a warning is generated.

This procedure is important, because for some properties the Create accessor and Assign accessor do not both exist. The implications are as follows:

- Create accessor only
- Assign accessor only
- The property's value can never change, once the object has been created.

The same accessor is used by the **SET** and **CREATE** commands.

*Note: When creating an instance of a nested class, the classname must be extended to provide the path to the nested class. This is done by using the > character between the class names. For example, Base$List>Iterator specifies the path to a nested class Iterator within the class List which is in the Base Library.*

The following table describes the keywords that you can use to create an object.

| **Keyword** | **Description** |
|---|---|
| **Share** | A true or false value that determines whether the object is private or shareable. If not specified, the object's shareability is the same as that of the creating object. Child objects are not affected by this keyword. |

| | |
|---|---|
| **Child** | If true or not specified, the new object is a child of the object that created it. The object uses the parent's location to determine its own location. If false, it is an independent, freestanding object. Specifying a Domain can override this keyword. If Share is used, this keyword must be false. |
| **Base** | Allows objects to be created at an explicit base location. The Base keyword value is an expression that evaluates to a glvn. When Base is specified, EsiObjects will generate an internal number and create an OID from it and the Base values supplied. For example: If Base=“^ESIBR(“”Data””)”, EsiObjects will create an OID for an object by adding a generated number as the next level subscript producing “^ESIBR(“”Data””,3)”. |
| **Fixed** | Allows objects to be created at a fixed location. This value must be a glvn. Where the keyword Base provides a base glvn for EsiObjects to create a unique OID from, the Fixed keyword lets you specify a glvn that represents the absolute value used for the OID. It is the OID and the created object is mapped to that location. . For example: If Base=“^ESIBR(“”Data””)”, EsiObjects will create an OID for an object without any alterations being made. You are responsible for its uniqueness. |
| **Stack** | This keyword instructs EsiObjects to place the object within the current call frame. This keyword overrides all other keywords when used. Placing an object on the current call frame insures the automatic destruction of the object when the call frame is popped from the stack |
| **Class** | Insures that the variable is class persistent. This is used when creating objects within the context of a class (class variables). |
| **Domain** | Not Implemented Yet |

## Comments

Keep the following points in mind when you use the **CREATE** command:

- The use of the **ZAPPLY** command is legal only in the CREATE method, enabling that method to validate or modify instance variable values before the **CREATE** command has finished executing.
- Two **INDIRECTION** operators (**@Name@**) are used for class name indirection, which prevents ambiguity with other forms of indirection. For more information about class name indirection, see the INDIRECTION operator.
- When a method parameter list is specified, there are two kinds of parameters:
    - keyword (keyword:expression)
    - positional (expression)

- After the first keyword parameter is specified, all remaining parameters must be keyword parameters.
- When the object is created, its internal reference count is initialized to 1 (one). If the **DESTROY** command is applied immediately after the object is created, the object will be destroyed. However, if the **PRESERVE** command is applied n times after the object is created, incrementing the internal reference count, the **DESTROY** command must be applied an equivalent number of times to decrement the counter. Only after that will another destroy action actually delete the object from the system.

## Related

DESTROY command

PRESERVE command

ZAPPLY command

## Examples

In the following example, a child object (to the creating context) of class Address is created. Its CREATE method is passed the positional parameters "Boston" and "MA". A handle to the new object is stored in the temporary variable T%CustAddr.

```
CREATE T%CustAddr=Framework$Address("Boston","MA"):Child=1
```

The results of the following example are the same as those of the first example. The only difference is that class name indirection is used to specify the name of the class.

```
SET T%ClassName="Framework$Address"
CREATE T%CustAddr=@T%ClassName@("Boston","MA")
```

In the following example, a shared (persistent) List object is created. The object will be stored under the ^UTILITY($J) node.

```
CREATE I%Set=Base$List:(Base="^UTILITY($J)",Share=1)
```

The following example illustrates how to create a nested class object.

```
CREATE I%AdmitDate=HIS$Patient>AdmitDate
```

The `HIS$Patient>AdmitDate` syntax provides a path to the Patient classes nested class AdmitDate.

## DESTROY

The **DESTROY** command attempts to destroy an object, setting the value of **$TEST** based on the success of the attempt.

### Format

DE{STROY} postcond SP L destroyargument

postcond ::= { : tvexpr }

A truth valued expression. Execution of the command is conditional based on the truth-value of this expression.

|  |  |
|---|---|
| **destroyargument ::=** | **expr V oref** |
|  | **@ expr V destroyargument** |

The argument of **DESTROY** is an object reference.

### Explanation

The **DESTROY** command does the following:

- Before proceeding, checks to see if the OID of the object has been protected (See the $Protect function). If it has, the object cannot be destroyed.
- Decrements the internal reference-count by one. This command works in concert with the **PRESERVE** command which increments the count. Only when the count goes below 1 will the destroy action continue.
- If the argument is not an **oref**, or references an object that does not exist, no action occurs and **$TEST** is set to 1.
- If the argument contains an oref of an existing object, that object's DESTROY method is invoked and its return value is interpreted as true or false.
  - If the DESTROY method returned a false value (**$RETURN**=0), the object is not destroyed. The **DESTROY** command sets **$TEST** to 0.
  - Otherwise, the DESTROY method returned a true value (**$RETURN**=1); the object is automatically destroyed and **$TEST** is set to 1.

### Comments

Keep the following points in mind when you use the **DESTROY** command:

- The very first action of the **DESTROY** command is to decrement the internal reference counter and check if the counter went below 1. If it did, the destroy action will proceed, calling the DESTROY method is it exists. If it did not, the destroy action quits at this point, leaving the object alive.
- An object's DESTROY method does not need to remove instance variables if it determines that the object can be destroyed. This is automatically done by the **DESTROY** command.

- The default return value of the DESTROY method is 1. It returns 1 if it does not explicitly define another return value. (Other functions return NULL ("") by default.)

Please note that the **DESTROY** object has no effect on a virtual object, because virtual objects have no symbol table to be removed. (Of course, the virtual object can implement a **DESTROY** method that will destroy its target data.) The only way to remove a virtual object is to **KILL** the variable containing the handle to the virtual object.

***However,*** it may be inappropriate for one object to make assumptions about whether another is actual or virtual. For example, a certain class that is declared virtual today may become an actual class in the future. So a reasonable precaution, when eliminating an object, is to both **DESTROY** it and **KILL** the variable containing the handle to the object. If you want to eliminate the handle but have no intention of destroying object's encapsulated data, then simply **KILL** the variable containing the handle to the object.

Finally, note that if a variable containing the handle to an object is scoped within the current method (i.e. **A%**, **T%** or **P%** variables), then the ***variable*** will be destroyed automatically when the method terminates. However, this may ***not*** result in the automatic destruction of any actual ***object*** being referenced by it.

### Related

CREATE command

KILL command

$REFERENCE special variable

$RETURN special variable

$TEST special variable

$PROTECT function

### Examples

The following example destroys the Window object referenced by the symbol T%Window, causing the window to disappear from the display and all of its instance variables to be removed.

```
DESTROY T%Window
ELSE  DO $Env.Assert("DESTROY failed!")
```

## DO Command - Introduction

The **DO** command calls a subroutine or block. When the called code terminates, control reverts to the point immediately following the **DO** command.

### Format

D{O} postcond SP {L doargument}

postcond ::= { : tvexpr }

A truth valued expression. Execution of the command is conditional based on the truth-value of this expression.

doargument

Can be one of the following:

doarg
@expratom V L doargument

where:

| | | |
|---|---|---|
| **doarg ::=** | **dlabel {+offset} {^routineref}** | **postcond** |
| | **^routineref** | |
| | **label {^rname} (L actualparam)** | |
| | **^rname (L actualparam)** | |
| | **& package.externalrtn{(L actualparam)}** | |
| | **ows {(methodactuallist)}< expratom > (L expr)** | |
| | ***publiclabel** | |

### Explanation

The **DO** command calls some body of code as a subroutine. Its behavior is distinct from **GOTO** in that **DO** execution subsequently returns to the point immediately following the **DO** argument from which the subroutine was called. In the case of **GOTO**, execution never returns to that point.

Like **GOTO**, **XECUTE** and **DO** commands allow a postconditional to be applied to the command or to any of its arguments. The following table summarizes the results when the postconditional in either location is true or false:

| **Result** | **Postconditional on Command** | **Postconditional on Argument** |
|---|---|---|
| True | Execute the command and its arguments. | Execute that argument before going on to the next argument or command. |
| False | Skip the command and all its arguments. | Skip that argument and go on to the next argument or command. |

**DO** supports a variety of different forms:

- The argumentless form calls a block beginning on the line following the line where the argumentless **DO** command occurs. This form places **$TEST** on the process stack, which causes its value to be restored when the block is exited.
- The entry reference form (label+offset^routine without parameters) calls a subroutine without an actual parameter list. This form allows an **INDIRECTION** operator ( @ ) before the label name and/or the routine name. This form does not place **$TEST** on the process stack.
- The label reference form (label^routine with parameters) allows an actual parameter list, providing for a certain level of independence between the calling code and the subroutine. It does not support label name or routine name indirection. This form does not place **$TEST** on the process stack.
- The external reference form (&package.externalroutine) allows routines external to M and EsiObjects to be called. This form does not modify **$TEST**.
- The object-with-service form (**object.service**) accepts references to object methods and properties. This form guarantees that **$TEST** is restored when the service is exited. If the asterisk (*) is present, the method is called at the ancestor class. The syntax is as follows:

**ows ::= Oref.{(L keyword)} {*} {interface::} service**

  - An object-with-service reference includes an object reference, a period, and a service. Optionally you can add a list of delivery keywords, an asterisk (*) for an ancestor-method call, and an interface to explicitly specify the interface containing the method. If no interface is specified, the method's primary interface is used. For more information, see the Message Syntax section in this guide.
  - The label inheritance form (***publiclabel**) accepts a reference to a public label inside the current method. The implementation of this method at the ancestor class is called, rather than the implementation at the current class. In this way, functionality can be overridden and inherited by storing it in public subroutines implemented within a method.

## Comments

Keep the following points in mind when you use the DO command:

- Any form of **DO** that specifies a label and/or routine name does not place **$TEST** on the process stack. When execution returns from the subroutine, any changes to **$TEST** are still in effect.
- The following lines of code are hard to evaluate. Without looking at the subroutine MODIFY it is impossible to determine under what circumstances the **ELSE** command on the second line will be executed.

```
IF I%Height'>I%Width DO MODIFY
ELSE  DO $Env.Output("Greater")
```

The following are possibilities:

- I%Height is greater than I%Width. The **IF** on the first line sets **$TEST** to 0, and execution drops down to the second line. Because **$TEST** is 0, the **ELSE** send 'Greater' to the output window..
- I%Height is not greater than I%Width. The **IF** on the first line sets **$TEST** to 1 and executes the **DO**. Inside the subroutine, three things might happen:
- The subroutine MODIFY does not modify **$TEST**. When execution returns, **$TEST** still equals 1 from the **IF** on the first line and the **ELSE** does nothing.
- The subroutine MODIFY does modify **$TEST**, and when it exits **$TEST** equals 1. The **ELSE** on the second line does nothing, based on the most recent **$TEST** operation.
- The subroutine MODIFY does modify **$TEST**, and when it exits **$TEST** equals 0. The **ELSE** on the second line sends 'Greater' to the output window, based on the most recent **$TEST** operation.
- Clearly this situation contains the potential for unexpected results. The examples in this section present a specific solution to this problem based on argumentless **DO.**
- In EsiObjects, **GOTO** is primarily useful for delegation; otherwise, the use of **GOTO** is discouraged. Inside a block, the **GOTO** command is illegal unless it accesses another line in the same body of code and that line's level is the same as the current execution level. The line accessed by **GOTO** need not be connected to the current block. The examples in this section show how to avoid **GOTO** in blocks.

Keep the following points in mind when you use argumentless **DO** blocks:

- Execution skips past any lines of code that are at too high a level. If EsiObjects encounters lines at a higher level than the current execution level, it skips past those lines. Accidental failure to place an argumentless **DO** on the line before a block or subblock causes the block of code to be skipped.
- Execution quits when a line is encountered at too low a level. Therefore, an implied **QUIT** occurs automatically at the end of a block and it is not necessary to place an explicit **QUIT** on the last line of the block. A comment line inside a block must begin with the appropriate number of periods, or it causes an implied **QUIT** to occur and the rest of the block is ignored. If a deeper block does not begin on the line immediately following an argumentless **DO**, then an implied **QUIT** immediately occurs.

Keep the following points in mind when passing parameters with the **DO** command (**DO LABEL^ROUTINE(...)**):

- With pass by value, only a single value is sent along to the formal variable. Its **$DATA** value is therefore guaranteed to equal 1. If an attempt is made to specify an undefined variable in the actual parameter list, then an error occurs.
- With pass by reference, the formal variable becomes an alias for the actual variable. Both symbols must be local variables. If the actual variable is not defined, the formal variable is also undefined and its **$DATA** value is 0. If the actual variable is a local array, the formal variable is an identical array having the **$DATA** value 10 or 11.
- If the formal parameter list is longer than the actual parameter list, the omitted formal parameters are undefined, and has **$DATA** values of 0. If the actual parameter list is longer than the formal list, then an error occurs.

When passing parameters with **DO object.service(...)**and a method parameter list is specified, there are two kinds of parameters: keyword (having the format keyword:expression) and positional (having the format expression). After the first keyword parameter is specified, all remaining parameters must be keyword parameters.

## Related

Message Syntax

Method structure

GOTO command

QUIT command

$TEST special variable

## Examples

Note that in some cases **DO** does not stack **$TEST** and that **$TEST** is likely to change between the **IF** and the **ELSE**. The following example illustrates a typical programming error.

```
    IF I%Height'>I%Width DO TEST
    ELSE DO $Env.Output("Greater")
    .
    .
    .
    QUIT
    ;
TEST       ; Subroutine containing IF and ELSE
    IF I%Height=I%Width DO $Env.Output("Equal")
    ELSE DO $Env.Output("Not Greater")
    QUIT
```

Assuming that I%Height=5 and I%Width=10, the **IF** command on the first line sets **$TEST** to 1 and the **DO** calls TEST. Inside TEST, the **IF** sets **$TEST** to 0, and the **ELSE** executes the environments Output method. The **QUIT** then exits TEST. The **ELSE** on the second line checks **$TEST** (which is now 0) and executes the environments Output method. The first line of output is "Not Greater" and the second line is "Greater". This is probably not what the programmer intended.

A number of language elements (for example, object-with-service references, extrinsic functions, and the argumentless **DO**) place **$TEST** on the process stack. The following example solves the problem shown in the previous example with the argumentless **DO**:

```
IF I%Height'>I%Width DO
. IF I%Height=I%Width DO $Env.Output("Equal") QUIT
. DO $Env.Output("Not Greater")
ELSE DO $Env.Output("Greater")
QUIT
```

When passing parameters by value with the syntax **DO LABEL^ROUTINE(...)**, remember that only a single value can be passed in. The symbol in the formal parameter list takes on the value specified by the expression in the actual parameter list.

```
    DO MODIFY(T%Child,T%ChildHeight+25,T%ChildWidth+50,200,300)
    DO MODIFY(T%Parent,T%ParentHeight,T%ParentWidth)
    .
    .
    .
    QUIT
    ;
MODIFY(L%Object,L%Height,L%Width,L%X,L%Y) ; Modify size/position
    IF $GET(L%X)'="" SET L%Object.X=L%X
    IF $GET(L%Y)'="" SET L%Object.Y=L%Y
    SET L%Object.Height=L%Height
    SET L%Object.Width=L%Width
    QUIT
```

In pass by reference, the formal variable becomes an alias for the actual variable until the subroutine exits, when the formal variable is restored to its previous state. In the following example, a subroutine SWAP is called three times to exchange the values of three pairs of local variables. Note that the variables L%Temp and L%First in the calling code are never confused with L%First and L%Temp in the subroutine.

```
    DO SWAP(.L%Third,.L%First)
    DO SWAP(.L%X,.L%Temp)
    DO SWAP(.L%Width,.L%Height)
    QUIT
SWAP(L%First,L%Second) ;Exchange values of two local variables
    NEW L%Temp
    SET L%Temp=L%First,L%First=L%Second,L%Second=L%Temp
    QUIT
```

The following two statements are equivalent. Both access the method Update, implemented within the same method at the superclass.

```
DO $SUPER.Update
DO $SELF.*Update
```

## DO Command - Parameters

A **DO label^routine** reference with parameters accepts two forms of parameter passing (pass by reference and pass by value).

The following terms are useful when discussing parameter passing:

| Term | Description |
|---|---|
| **Actual parameter list** | The parameter list specified on a **DO** command (or extrinsic function call), specifying the actual values to be associated with each parameter variable. |
| **Formal parameter list** | The parameter list specified on a subroutine or function's initial label line, formally specifying the local variable names to be used for each parameter. |
| **Pass by value** | The parameter is any expression. Its value is assigned to the local variable named in the formal parameter list. Nothing that happens to the formal variable in the parameter list has any effect on the actual parameter. |
| **Pass by reference** | The actual parameter is a local variable name preceded by a period. In the subroutine, the formal variable temporarily becomes an alias for this local variable. Therefore, changes to the formal variable are immediately reflected in the actual variable. |

## DO Command - Argumentless

When the argumentless **DO** is encountered, EsiObjects does the following:

- Adds a new frame to the process stack and records the current execution location, execution level, and **$TEST value.**
- Adds 1 to the current execution level (the number of periods it expects to find after the line start indicator on each line of code it encounters).
- Transfers control to the next line of code.

The execution level *(EL)* initially begins at 0. Only the argumentless **DO** command increases the execution level. Other code calls (**XECUTE**, object-with-service references, the other forms of **DO**, and extrinsic functions) place the EL on the stack but set it to 0 internally. Each line of code has its own level (the line level (LL)), which refers to the number of periods following the line-start indicator.

Whenever EsiObjects encounters a line of code, its behavior is based on a comparison of these two values as follows:

- If LL>EL, skip this line and go on to the next line of code.
- If LL=EL, execute this line of code before going on.
- If LL<EL, issue a **QUIT**, remove the top frame from the process stack, and restore the state recorded on that frame.

The practical implications of these rules are summarized in the Comments discussion.

## DO Command - Callbacks

The format for a callback is as follows:

DO <cbref>[(cbactlist)]

where:

cbref

Is an expratom V as a callback frame ID string

cbactlist

Is a list of parameters to be passed positionally through the callback by value.

The **$CALLBACK** function returns a callback frame identifier used in calling back to a label or public label in the current method. A method uses this function to create such an identifier, which is then passed to an external object. The external object can then call the label directly without incurring the overhead of a full-blown method call. The callback runs in the context to the object that created the callback.

**Original** callbacks are the most common callbacks used. They dispatch directly to the actual EsiObjects method stack frame that created the callback. (Note that the EsiObjects method stack is not the same as the underlying M process stack.) The callback is automatically freed when the stack frame exits, so the callback can only be made from methods called before that stack frame has exited.

**Capture** callbacks record the callback creator's method-related variables. If the callback is additive, then changes to those variables are preserved between calls. If it is not additive, then the variables are always reset to their values at the time the call was made. If the callback is persistent, then it survives for the lifetime of the creating object, or until it is explicitly freed with the **$FREECB** function.

**Initialized** callbacks are used internally as the backbone of events and watches. The callback starts with an entirely clean variable context. However, if it is additive, then any changes to those variables are carried over to succeeding callbacks.

The creator of a callback is usually considered to be its owner. Only the owner should free the callback. The lifetime of a callback never extends beyond the lifetime of the creating object. If the object dies, the callback is automatically freed. Original callbacks continue to exist until the creating stack frame terminates, when they are automatically freed. Nonpersistent callbacks are freed whenever the incarnation of **$ENVIRONMENT** changes (in other words, whenever EsiObjects is shut down or restarted). In general, Capture and Initialized callbacks should be explicitly freed with **$FREECB** when their usefulness has ended.

All methods that create callbacks or call them should clearly document the callback interface they assume. The documentation includes the following information:

- The number of parameters, if any
- The purpose and use of each parameter
- Whether the callback is to be made as a subroutine or extrinsic function
- The expectations of callback ownership

Note that the validity of callback parameters is determined only at run time. This adds to the importance of adequate documentation.

# ELSE

The **ELSE** command causes the remaining statements on the line to be executed if **$TEST** is 0, or to be ignored if **$TEST** is 1.

### Format

E{LSE} SP

### Explanation

The **ELSE** command, like **IF** and **FOR**, has a line scope. This means that it conditionalizes the execution of all the remaining commands on the line.

- If the value of **$TEST** is 1 when **ELSE** is encountered, then all the remaining commands on the line are ignored and execution proceeds to the next line of code.
- If the value of **$TEST** is 0 when **ELSE** is encountered, then execution continues on to the remaining commands on the line.

The **$TEST** special variable is affected by the **IF** and **DESTROY** commands and any time a timeout is encountered. Other conditional operations, such as postconditionals and **$SELECT**, do not affect **$TEST**. The **ELSE** command does not modify the value of **$TEST**.

### Comments

Keep the following points in mind when you use the **ELSE** command:

- Because **ELSE** is always argumentless, at least two spaces must separate it from anything else on the same line.
- **ELSE** does not allow a postconditional. The two other commands with line scope, **IF** and **FOR**, also do not allow postconditionals.
- Because all commands conditionalized by **ELSE** must fit on a single line, argumentless **DO** is sometimes used in the scope of **ELSE** to extend its reach beyond a single line.

### Related

IF command

DO command

$SELECT function

$TEST special variable

### Examples

It is extremely rare to have an **ELSE** command on the same line as an **IF**. This is because **IF** sets **$TEST** to 1 if it executes the rest of the line, and **ELSE** prevents execution unless **$TEST** is 0. The following example is clearly a mistake:

```
IF I%Height>I%Width DO $Env.Output("Greater") ELSE DO $Env.Output("Not
Greater")
```

If I%Height is not greater than I%Width, the **IF** sets **$TEST** to 0 and transfer execution to the next line. Nothing after the **ELSE** command is ever encountered. By contrast, the following example works properly:

```
IF I%Height>I%Width DO $Env.Output("Greater")
ELSE DO $Env.Output("Not Greater")
```

It can be valid to put **ELSE** on the same line as **IF** in those rare cases where something happens between **IF** and **ELSE** to change **$TEST**, such as a timeout or a **DESTROY** command.

```
IF T%Remove DESTROY T%Object12 ELSE DO $Env.Output("Object not destroyed.")
QUIT
```

In the previous example, if the service variable T%Remove does not contain a true value, the entire rest of the line is ignored. But if  T%Remove is true, the **DESTROY** command is invoked to remove the object whose object reference is in T%Object12. If the object is destroyed successfully, then **$TEST** is 1 and everything after the **ELSE** is ignored. But if **DESTROY** fails to destroy the object, **$TEST** is set to 1 and the **ELSE** lets execution pass on to the environments output window and **QUIT**.

# EVENT

The **EVENT** command triggers an event that can be handled by any concerned objects.

## Format

EV{ENT} postcond SP L earg

postcond ::= { : tvexpr }

A truth valued expression. Execution of the command is conditional based on the truth-value of this expression.

| | |
|---|---|
| **earg ::=** | **eventname {(L expr)}**<br>**$PROPERTIES**<br>**propertyname {(L expr)}**<br>**@ expratom V L earg** |

## Explanation

The following expression values are to be used for property events:

| **Value** | **Meaning** |
|---|---|
| "PRESET" | The property is about to be set. |
| "SET" | The property's value has just been assigned (interested objects can query the property for its current value). |
| "SETREJECT" | The assignment has been rejected by setting $Return to zero (false). |
| "PREKILL" | The property is about to be killed. |
| "KILL" | The property has just been killed. |
| "KILLREJECT" | The kill has been rejected by setting $Return to zero (false). |
| "DEAD" | The object implementing the property has just died (interested objects can no longer refer to the property.) |

See the examples section below for more details.

If a single event or property name is specified, parameters can be sent with the event. The **$PROPERTIES** special name contains a list of all properties of the triggering object. Event delivery is asynchronous. The order in which events are delivered to a given object is guaranteed, but the timing of events is not.

## Comments

Keep the following points in mind when you use the **EVENT** command:

- If a concerned object is interested in watching for all properties to change, the only way to trigger an event for that object is with **EVENT $PROPERTIES**. Individually triggering events for each property does not have this effect.
- If a concerned object is watching **$PROPERTIES** and a single property, it is informed twice when an event for that property is triggered.

- The label used to handle an event must be public. It must be able to handle at least two parameters:
  - OBJECT - An object reference to the watched object.
  - MESSAGE - The name of the event or property that was triggered by the **EVENT** command.
- However, many events send additional parameters, and the handler's formal parameter list must not declare fewer parameters than are sent with the event.
- Properties generally initiate an event when they are assigned or killed. This is the main case when property events occur.

## Related

Method structure

IGNORE command

WATCH command

## Examples

The following example triggers the Renamed event for any concerned objects.

```
EVENT Renamed(T%OldName,T%NewName)
```

The following example triggers a property event, indicating that the Name property has been set.

```
EVENT Name("SET")
```

The following example triggers an event for concerned objects watching all properties of the current object and triggers a separate event for the individual properties being watched by concerned objects.

```
EVENT $PROPERTIES
```

## FOR

The **FOR** command causes the remaining statements on the line to be executed repeatedly.

### Format

F{OR} SP {lvn = L forparameter}

| | |
|---|---|
| **Forparameter ::=** | **expr** |
| | **numexpr1 : numexpr1 {: numexpr1}** |

where:

| | |
|---|---|
| **Expr** | is an explicit string or numeric value for the variable during a single iteration of the loop |
| **numexpr1** | is the numeric starting value of the variable during the first loop iteration |
| **numexpr2** | is the numeric incremental value to be added to the variable before each loop iteration other than the first |
| **numexpr3** | is the numeric boundary value that must not be crossed by the variable |

### Explanation

The **FOR** command, like **IF** and **ELSE**, has a line scope. This means that it iterates the execution of all the remaining commands on the line. Note the following:

- For each forparameter, the variable's value is either incremented through a range of values or set to the single explicit value of that forparameter.
- Execution of a **FOR** loop terminates when the variable's value is already beyond the boundary value (in the direction indicated by the sign of the incremental value), or when adding the increment to the variable would cause it to cross that boundary.
- If more than one forparameter is specified, they are processed in order from left to right.
- Execution of a **QUIT** command terminates the innermost **FOR** loop, causing all remaining forparameters in that loop to be skipped.
- Execution of a **GOTO** command in the scope of a **FOR** loop terminates all the loops on that line, from the innermost to the outermost.
- Once the **FOR** loop has started executing, changes to the looping variable can have an impact on the number of iterations. However, changes to any variables originally used to specify the starting, incrementing, and ending values cannot affect the number of iterations.

The argumentless **FOR** command does not specify any variable to iterate. This is the most flexible type of **FOR** loop, and possibly the most common. Two spaces must separate the argumentless **FOR** command from anything else on the line. Iteration continues until a **QUIT** or **GOTO** in the scope of the **FOR** command terminates the loop. If no **QUIT** or **GOTO** is executed, an infinite loop results.

The **FOR** command does not allow argument indirection. The following is illegal:

```
SET T%Illegal="X=1:1:10"
FOR @T%Illegal DO $Env.Output(T%X)
```

However, you can use the **XECUTE** command to achieve the desired results as follows:

```
SET T%Loop="X=1:1:10"
XECUTE "FOR "_T%Loop_" DO $Env.Output(T%X)"
```

### Comments

Keep the following points in mind when you use the **FOR** command:

- If **FOR** is argumentless, at least two spaces must separate it from anything else on the same line.
- **FOR** does not allow a postconditional. The two other commands with line scope, **IF** and **ELSE**, also do not allow postconditionals.
- Because all the commands iterated by **FOR** must fit on a single line, argumentless **DO** is sometimes used in the scope of **FOR** to extend its reach beyond a single line.

### Related

DO command

GOTO command

QUIT command

### Examples

The following example shows a simple **FOR** loop:

```
SET T%String=""

FOR T%Loop=1:1:10 SET T%String=T%String_T%Loop_" "

Results: 1 2 3 4 5 6 7 8 9 10
```

The following example illustrates multiple forparameters. Note that the **QUIT** command terminates during the third forparameter, causing the fourth forparameter to be skipped:

```
SET T%String=""

FOR T%Loop="Hello",50:-7:20,444,100:222 QUIT:T%Loop>300  DO
. S T%String=T%String_T%Loop_" "

Results: Hello 50 43 36 29 22 444
```

The following example illustrates the repeated use of incremental lock with a timeout to provide feedback to the user that an attempt to lock the node is in progress. After 30 seconds, the **FOR** loop gives up and the process is abandoned.

```
DO $Env.Output("Locking")
FOR T%Loop=1:1:10 DO $Env.Output(".") LOCK +^XYZ(0):3 IF  QUIT
ELSE DO $Env.Output("Node is busy. Aborting.") QUIT
SET (T%EntryNumber,^XYZ(0))=^XYZ(0)+1
LOCK -I%List(0)
SET ^XYZ(T%EntryNumber)=T%EntryValue
```

The following example attempts to open a device for up to 30 seconds before it gives up. If the operation is successful, the following occurs:

- An argumentless **FOR** loop reads lines of text from a file whose identifier is in the variable T%File.
- The lines that are read are echoed to the principal device (**$PRINCIPAL**) until a blank line is encountered.
- The file is closed.

```
DO $Env.Output("Opening Device "_T%File)
FOR T%Loop=1:1:10 DO $Env.Output(".") OPEN T%File::3 IF  QUIT
ELSE DO $Env.Output("Device ",T%File," is unavailable.") QUIT
FOR  DO  QUIT:T%Line=""
. USE T%File
. READ T%Line
. IF T%Line="" QUIT
. USE $PRINCIPAL
. WRITE T%Line,!
CLOSE T%File
QUIT
```

The following example, the WALK subroutine, traverses all the descendants of the specified array node, displaying the nodes and their values on the environment output window. It uses a **FOR** loop with **$ORDER** to traverse the nodes, **$DATA** to determine whether a given node contains data, and **$NAME** to convert a subnode into a name value. This name value is then used in name indirection, as the argument of **$DATA**, and is passed as a parameter.

```
WALK(Node) ; Recursive traversal
    NEW Sub,DataVal,NodeName
    IF $DATA(@Node)#10 DO $Env.Output(Node_" =<”_@Node_">")
    SET Sub=""
    FOR  SET Sub=$ORDER(@Node@(Sub)) QUIT:Sub=""  DO
    . SET NodeName=$NAME(@Node@(Sub))
    . SET DataVal=$DATA(@NodeName)
    . IF DataVal'[0 DO $Env.Output(NodeName_" =<”_@Node_">")
    . IF DataVal>9 DO WALK(NodeName)
    QUIT
```

The following example provides an alternative implementation of WALK. It uses **$DATA** to display the root node if necessary, uses **$LENGTH** and **$EXTRACT** to build an array root, a **FOR** loop with **$QUERY** to traverse the array, and **$EXTRACT** to determine the exiting condition. Inside the **FOR** loop there is only a single instance of indirection with no recursive call.

```
WALK(Node) ; Nonrecursive traversal
    NEW Root,Len
    IF 11[$DATA(@Node) DO $Env.Output(Node_" =<”_@Node_">")
    SET Len=$LENGTH(Node),Root=Node
    IF $EXTRACT(Root,Len)=")" SET $EXTRACT(Root,Len)=","
    ELSE  SET Root=Root_"(",Len=Len+1
    FOR  S Node=$QUERY(@Node) Q:$EXTRACT(Node,1,Len)'=Root  DO
    . DO $Env.Output(Node_" =<”_@Node_">")
    QUIT
```

## GOTO

The **GOTO** command transfers control to the specified execution location without adding a frame to the process stack. Execution does not return to the point following the **GOTO**.

### Format

G{OTO} postcond SP L gotoargument

postcond ::= { : tvexpr }

txexpr evaluates to a truth-valued expression. Execution of the command is conditional based on the truth-value of this expression.

gotoargument

Can be one of the following:

| | |
|---|---|
| **dlabel {+offset} {^routineref}** | **postcond** |
| **^routineref** | |
| **object.service** | |
| **@ expratom V L gotoargument** | |

### Explanation

The primary use of **GOTO** in EsiObjects is for delegation to another object and/or service. In delegation, the current method uses **GOTO** to make an object-with-service call that does not return to the current execution context. The return value of the delegated service is treated as the return value of the calling code body.

**GOTO** can be used in other contexts such as subroutines and extrinsic functions, but its general-purpose use is not recommended. **GOTO** is illegal inside a block unless it accesses another line in the same code body and that line's execution level is the same as the execution level of the line containing the **GOTO**.

**GOTO** has a special function inside the scope of the **FOR** command. It transfers control at the current stack level, terminating execution of the **FOR** loop. An alternative is to terminate the loop with **QUIT**, transferring control externally.

Like **DO** and **XECUTE**, **GOTO** allows a postconditional to be applied to the command, or to any of its arguments. The following table summarizes the results when the postconditional in either location is true or false.

| Result | Postconditional on Command | Postconditional on Argument |
|---|---|---|
| True | Execute the command and its arguments. | Execute that argument, never returning to process any additional arguments. |
| False | Skip the command and all its arguments. | Skip that argument and go on to the next argument or command. |

## Callbacks

The format for a callback is as follows:

GOTO <cbref>

where:

cbref evaluates to an expratom V as a callback frame ID string.

For example:

GOTO <T%Revector>

GOTO <T%Error>

Control is transferred to the callback. The callback must not require any parameters.

## Comments

Keep the following points in mind when you use the **GOTO** command:

- In EsiObjects the **GOTO** command, though legal, is not recommended for general use except in cases of delegation. There is no task in EsiObjects that requires the use of **GOTO**.
- Some programmers are reluctant to use **GOTO** in any context, but in EsiObjects it is an important tool when explicit delegation is called for.

## Related

Message Syntax

Method structure

DO command

FOR command

XECUTE command**Examples**

In the following example, the object delegates the current task to its parent object, whose handle is in the instance variable I%Parent. The return value of the called service Shutdown becomes the return value of the current method doing the delegation.

```
GOTO I%Parent.Shutdown(T%Status)
```

## HALT

The **HALT** command ends the M session.

### Format

H{ALT} postcond

### Parameters

postcond ::= { : tvexpr }

A truth valued expression. Execution of the command is conditional based on the truth-value of this expression.

### Explanation

An unconditional **HALT** exits from the M session. It unlocks all local and global nodes that were locked and closes all devices that you own. A process is deleted if it was started with the **JOB** command. Otherwise, the process remains active. All unshared objects cease to exist.

The **HALT** command has no effect when executed within the EsiObjects Xecute Shell.

### Related

CLOSE command

HANG command

LOCK command

QUIT command

## HANG

The **HANG** command suspends execution for the specified number of seconds.

### Format

H{ANG} postcond SP L hangargument

### Parameters

postcond ::= { : tvexpr }

A truth valued expression. Execution of the command is conditional based on the truth-value of this expression.

hangargument

Can be one of the following:

numexpr
@ expratom V L hangargument

where:

| | |
|---|---|
| **numexpr** | is the number of seconds for which execution is suspended (This expression's numeric interpretation is used.) |

### Explanation

The **HANG** command suspends execution for the specified number of seconds. If the hang value is 0 or less, execution is not suspended.

### Comments

Keep the following points in mind when you use the **HANG** command:

- The abbreviation H also applies to the **HALT** command, which is always argumentless. Therefore, the following statement is interpreted as **HANG 5**:

```
H 5
```

- Some underlying platforms round fractional values down to the nearest integer. Decimal precision can vary on these platforms that do support fractional amounts.
- The pause between **HANG** operations can be slightly longer than the number of seconds specified.

### Related

HALT command

### Examples

The following example illustrates the conventional use of the **HANG** command to pause execution for an integer number of seconds (one second in this case).

```
FOR T%Loop=1:1:10 DO $Env.Output(T%Loop) HANG 1
```

In the following example, execution pauses for 0.25 seconds between **WRITE** operations. Some underlying M platforms round fractional amounts down to the nearest integer (0 in this case). If the underlying M platform does not support fractional arguments, no suspension occurs.

```
FOR T%Loop=1:1:10 DO $Env.Output(T%Loop) HANG 0.25
```

# IF

The **IF** command executes or ignores the remaining statements on the line based on the true or false value of some conditions.

## Format

I{F} SP {L ifargument}

ifargument

Can be one of the following:

tvexpr
@ expratom V L argument

where:

| | |
|---|---|
| **Tvexpr** | is an expression whose value is interpreted as either true or false |

## Explanation

The **IF** command with defined with no arguments is the opposite of the **ELSE** command. Argumentless **IF** lets execution pass to the rest of the commands on the line only if **$TEST** is 1. This form is most commonly used after language elements (other than **IF**) that modify **$TEST**, such as timeouts or the **DESTROY** command.

With one or more arguments, **IF** begins to evaluate each of its arguments from left to right as true or false. If a true argument is encountered, **IF** sets **$TEST** to 1 and execution continues with the rest of the line (including any remaining **IF** arguments). If a false argument is encountered, **$TEST** is set to 0 and the rest of the line (including any remaining **IF** arguments) is skipped.

## Comments

Keep the following points in mind when you use the **IF** command:

- **IF** does not allow a postconditional. The two other commands with line scope, **ELSE** and **FOR**, also do not allow postconditionals.
- Because all the commands conditionalized by **IF** must fit on a single line, argumentless **DO** is sometimes used in the scope of **IF** to extend its reach beyond a single line.
- The **$TEST** special variable is affected by the **IF** and **DESTROY** commands and any time a timeout is encountered. Other conditional operations, such as postconditionals and **$SELECT**, do not affect **$TEST**.

## Related

DO command

ELSE command

$SELECT function

$TEST special variable

## Examples

In the following example, the **WRITE** and **QUIT** commands are performed only if **DESTROY** set **$TEST** to 1 (in other words, the object was successfully destroyed).

```
DESTROY T%Object12
IF  DO $Env.Output("Object was destroyed.") QUIT
```

In the following example, the single-argument **IF** command can be used with the **AND (&)** operator to perform some commands only if one or more conditions are true:

```
IF X>Y&(X>Z) DO $Env.Output("X is higher than Y or Z.")
```

The multiple-argument **IF** command is usually equivalent to the use of **AND**. For example, the following line of code is equivalent to the preceding example.

```
IF X>Y,X>Z DO $Env.Output("X is higher than Y or Z.")
```

The multiple-argument **IF** does not examine any arguments after the first false one. In some cases it runs faster than the and (&) form shown previously, and in some cases its behavior is distinct (in other words, the ignored arguments have side-effects such as changing the naked indicator, calling extrinsic functions, invoking object methods, properties, accessors, and so on).

For example, the following line of code always invokes the $Order accessor of the property Element(T%Loop) of object T%Window, which could have functional side effects, even if the first part of the condition T%Window.Size>300 is false:

```
IF T%Window.Size>300&($ORDER(T%Window.Element(T%Loop))'="") DO
$Env.Output("Not finished.")
```

The previous code is not functionally equivalent to the following line of code because the second IF argument is not encountered if the first argument is not true:

```
IF T%Window.Size>300,$ORDER(T%Window.Element(T%Loop))'="" DO $Env.Output("Not
finished.")
```

## IGNORE

The **IGNORE** command specifies that one or more events are to be ignored for one or more objects.

### Format

IG{NORE} postcond SP {L iarg}

postcond ::= { : tvexpr }

A truth valued expression. Execution of the command is conditional based on the truth-value of this expression.

| | |
|---|---|
| **iarg ::=** | **target {.eleref}**<br>**@ expratom V L iarg** |
| **eleref ::=** | **Ielem**<br>**(L ielem)** |
| **ielem ::=** | **Rielem**<br>**@ expratom V rielem** |
| **rielem ::=** | **$PROPERTIES**<br>**$EVENTS**<br>**event**<br>**property** |

### Explanation

The argumentless form specifies that the object in whose context it executes is no longer concerned about any events or properties for any objects.

The target, if specified, indicates an object for which events are to be ignored. If only the target is specified, then all events and properties are ignored for that object.

It is possible to ignore specific events or properties by naming them, to ignore all events by specifying the special name (not a special variable) **$EVENTS**, or to ignore all properties by specifying the special name **$PROPERTIES**.

### Comments

Keep the following points in mind when you use the **IGNORE** command:

- If a concerned object wants to watch all events except for one about the watched object, it is not possible to use **WATCH $EVENTS** and then use **IGNORE** to ignore the specific event. Instead, it is necessary to specifically watch those events about which the object is concerned.
- The concerned object detaches itself from the watched object using the **IGNORE** command. There is no detachment mechanism for the watched object to disassociate itself with one or more concerned objects, but the watched object can choose not to generate some or all events.
- If the ignored object has used the **$WATCHDETECT** function, it can be informed of the fact that it is being ignored.

### Related

EVENT command

WATCH command

$WATCHDETECT function

### Examples

The following example causes the current object to ignore all events and properties.

```
IGNORE
```

The following example causes the current object to ignore all events and properties for the object T%Object12.

```
IGNORE T%Object12
```

The following example causes the current object to ignore all events for the object T%Object12.

```
IGNORE T%Object12.$EVENTS
```

The following example causes the current object to ignore all properties for the object T%Object12.

```
IGNORE T%Object12.$PROPERTIES
```

The following example causes the current object to ignore the Renamed event for the object T%Object12.

```
IGNORE T%Object12.Renamed
```

The following example causes the current object to ignore the Renamed and ObjectDeleted events for the object T%Object12.

```
IGNORE T%Object12.(Renamed,ObjectDeleted)
```

## JOB

The **JOB** command calls a body of code to be executed in the context of a separate, newly created process.

### Format

J{OB} postcond SP L jobargument

postcond ::= { : tvexpr }

A truth valued expression. Execution of the command is conditional based on the truth-value of this expression.

| | | |
|---|---|---|
| | **dlabel {+offset} {^routineref}** | |
| **jobargument ::=** | **^routineref** | **{: jobparameters}** |
| | **label {^rname} (L expr)** | |
| | **^rname (L expr)** | |
| | **@expratom V L jobargument** | |

### Explanation

Execution of the current process continues in parallel. When the called code terminates, the new process is also terminated.

At some point, a final **QUIT** command occurs in the called code, removing the last remaining frame from the process stack. When the called code terminates, the new process also terminates.

An actual parameter list can be specified with the **JOB** command. The rules are the same as for parameter passing on the **DO** command, except that the new process has its own local variable table and pass by reference is not allowed. Parameters can only be passed by value.

Special job parameters, determined by the underlying M platform, optionally can be specified. These affect the creation of the new process. Examples might include setting the priority of the job or setting the total amount of local variable memory.

In some cases, the system may not have enough slots or memory available to create a new process. The calling process suspends execution until the new process can be created. It is possible to specify a timeout on the **JOB** command, in which case the attempt to create a new job aborts after the specified number of seconds. Whenever a timeout is specified, **$TEST** is always equal to 1 if the operation succeeded, or 0 if it times out.

## Comments

Keep the following points in mind when passing parameters with the **JOB** command (**JOB LABEL^ROUTINE(...)**):

- Only pass by value can be used when passing parameters. Only a single value is sent to the formal variable. Therefore, its **$DATA** value is guaranteed to be 1. If an attempt is made to specify an undefined variable in the actual parameter list, then an error occurs.
- There is no way to pass in a local array. The only way to provide an array is to use subscript indirection to pass a pointer to a global array to be accessed by the job.
- If the formal parameter list is longer than the actual parameter list, the omitted formal parameters are undefined and have **$DATA** values of 0. If the actual parameter list is longer than the formal list, then an error occurs.

## Related

DO command

QUIT command

$JOB special variable

$TEST special variable

## Examples

The following example creates a new job with a call to the label START in the routine PQUEUE.

```
JOB START^PQUEUE
```

The following example shows how to use **JOB** with a parameter list to specify important values for the newly created process.

```
JOB START^PQUEUE(T%File,I%Printer)
```

# KILL

The **KILL** command destroys a variable or an array node and all its descendants.

## Format

K{ILL} postcond SP {L killargument}

postcond ::= { : tvexpr }

A truth valued expression. Execution of the command is conditional based on the truth-value of this expression.

killargument

Can be one of the following:

glvn
(L local)
object . property
@ expratom V L argument

where:

| | |
|---|---|
| **glvn** | is a global or local variable name or array node included in the kill |
| **local** | is a local variable name to be excluded from the kill |
| **object** | is the object reference of an object |
| **property** | is the name of a property to be killed for that object |

## Explanation

There are three forms of the **KILL** command:

- Inclusive - Removes the specified variable names, array nodes, or both. If the target has descendant array nodes, these are also removed. This is the most common form and the one most consistent with the paradigm of EsiObjects.
- Exclusive - Removes all local (L%) variables except those enclosed in parentheses. Use of the exclusive form is discouraged.
- Argumentless - Removes all local (L%) variables. Not currently supported in EsiObjects.

When a variable or array node is killed, all descendant array nodes are also removed, for example:

```
KILL ^MYGLO(22,1)
```

When the argument of **KILL** is of the form object.property, the Kill accessor for that property is automatically invoked in an attempt to kill the property. This is a method in whose context the **$RETURN** special variable's value equals 1 by default. If the method returns 1 or any other true value, an event is triggered by the dispatch mechanism. If the method returns a 0, it means that the property was not killed and no event is triggered.

Handlers of the event automatically triggered by a successful return of the Kill accessor must accept four parameters in their formal parameter list. These parameters are described in the following table.

| Parameter | Description |
|---|---|
| Object | The object reference of the object containing the property. |
| Property | The full property name, of the form interface::name. |
| Callframe Object | An object that can optionally be used to inquire into the parameters that were passed in. |
| Operation | **SET** or **KILL** commands. |

## Comments

Keep the following points in mind when you use the **KILL** command:

- Killing a handle to an object does not destroy the object. Use the **DESTROY** command instead.
- Only local variable names are allowed inside the parentheses of an exclusive **KILL** command. Local array nodes are not allowed.
- There is never a need to use the exclusive and argumentless forms of KILL in EsiObjects. Local (L%) variables can be made temporary by using the **NEW** command. They automatically are destroyed when a **QUIT** occurs at the same stack frame level. Accessor (T%) variables can be used in place of local variables, causing them to be scoped within a single code body.
- Any time a variable or array node is killed, all descendant array nodes are also removed. If the target did not exist in the first place, **KILL** has no effect. Therefore, using the **$DATA** function on a symbol immediately after a **KILL** command has been applied to it always yields the return value 0.

## Related

DESTROY command

NEW command

## Examples

The following example uses the **KILL** command to explicitly destroy the instance variables I%Height and I%Width.

```
KILL I%Title,I%Name
```

The following example kills the symbol T%Customer, a handle to a window object. It has no effect on the object referenced by that handle.

```
KILL T%Customer
```

The following example uses **DESTROY**, instead of **KILL**, which attempts to kill the symbol T%Employee and destroy the object it references. Afterwards, argumentless **IF** is used to test the results of the **DESTROY** operation. If **$TEST** is true, it means that the object was destroyed and T%Employee was killed. Note that unlike **KILL**, the **DESTROY** command does not kill any descendant array nodes of T%Employee.

```
DESTROY T%Employee
ELSE DO $Env.Output("Warning: "_T%Employee.Name_" not destroyed.") Q
.
.
.
```

Killing a property invokes the Kill accessor. For example, the following code invokes the Kill accessor for the Visit property of the object T%Object12.

```
KILL T%Object12.Visit($HOROLOG,T%Txnum)
```

Assuming that this method is able to successfully accomplish its task, it returns some true value such as 1. In that case, an event is triggered. This event includes the following parameters:

- Object
- Full property name
- Callframe object
- Operation

The following code can be used to handle this event. If the **$HOROLOG** value specified in the first parameter indicates today's date, the handler exits. Otherwise, it gets the remaining parameters and calls the LOG subroutine to make note of the event.

```
HANDLE(T%Object,T%Property,T%CFO,T%Oper) ; Handle assign/kill
    IF +(T%CFO.Parameter(1))=+$HOROLOG QUIT  ; Exit if it is today
    SET T%Value=T%CFO.Parameter(2)
    DO LOG(T%Value,T%Oper)
    QUIT
```

# LOCK

The **LOCK** command provides a convention whereby concurrent processes can avoid conflicts caused by simultaneous attempts to update the same variable.

## Format

L{OCK} postcond SP {L lockargument}

postcond ::= { : tvexpr }

tvexpr evaluates to a truth-valued expression. Execution of the command is conditional based on the truth of this expression.

```
                      +      nref
lockargument ::=      -      (L nref)     {: timeout}
@ expratom V L lockargument
```

The syntax element **nref** is a name reference, frequently the name of a global array node. Name indirection is allowed.

## Explanation

The **LOCK** command is a tool for establishing conventions where concurrently executing processes can avoid coming into conflict when they attempt to modify the same data. **LOCK** makes an entry in a system lock table, preventing other processes from locking the same data until it is unlocked. **LOCK** does not prevent other processes from modifying or destroying data; it only prevents them from locking the same data. As such, it provides no guarantee that other processes are not coming into conflict with the locked data.

When a name is locked, an entry for that name is made in the lock table. This entry prevents other processes from locking the same name or any ancestor or descendant of that name.

There are several different forms of the **LOCK** command. Use only the incremental and decremental forms in EsiObjects. The older forms of **LOCK** can violate object encapsulation by affecting locks made internally by other objects.

- Incremental - Incremental **LOCK** prefixes the name (or list of names in parentheses) with the plus (+) sign. It adds a lock to each specified name without removing any locks. If the locked name is not already in the lock table for the current process, it attempts to lock that name. If the current process already locks the name, it adds 1 to the number of locks of that name.
- Decremental - Decremental **LOCK** prefixes the name (or list of names in parentheses) with the minus (-) sign. If a locked name is in the lock table for the current process, it decrements the number of locks for that name without affecting other locks. When the number of locks of a name reaches 0, that name is unlocked.
- Argumentless - Argumentless **LOCK** removes all lock table entries for the current process. It is not recommended for use in EsiObjects.

- Single-name - Single-name **LOCK** removes all lock table entries for the current process before attempting to lock the specified name. It is not recommended for use in EsiObjects.
- Multiple-name - Multiple-name **LOCK** is expressed as a list of names enclosed in parentheses to lock. It removes all lock table entries for the current process before attempting to lock these names, one after another, from left to right. It is not recommended for use in EsiObjects.

## Comments

Keep the following points in mind when you use the **LOCK** command:

- Only the incremental and decremental forms of **LOCK** are recommended for use in EsiObjects.
- **LOCK** can optionally include a timeout. When present, the timeout causes **LOCK** to abort after waiting at least the specified number of seconds for the names to become available. **$TEST** is set to 1 if all the specified names were successfully locked, or to 0 if not all the locks were successfully completed. **ELSE** or argumentless **IF** can be used to check whether the timeout period expired.
- There is never any reason to use a timeout on a decremental lock.
- Note that the following form of **LOCK** is dangerous. If the timeout period expires, it is impossible to tell which globals were locked and which were not. For example, it is possible that ^X and ^Y were locked, but not ^Z. Alternatively, none of the globals may have been locked. This makes it impossible to know which globals to decrementally unlock.

```
LOCK +(^X,^Y,^Z):3
```

- If each code body is responsible for incrementally locking and then decrementally unlocking the names with which it is concerned, then it is easy to isolate locking responsibility. A dangerous situation arises when subroutines, functions, and especially object services begin to create a web of lock dependencies.

## Related

ELSE command

IF command

## Examples

The following example illustrates the use of incremental and decremental lock to add a new entry to a list in global ^XYZ. The list's last-entry pointer is located at array node ^XYZ(0). The programmer who wrote this code has decided that this node is the only place where processes come into conflict. For this reason the array node ^XYZ(T%EntryNumber) is not locked, though it would not be incorrect to do so.

The question of when to lock versus when not to lock is an important programming issue. Also note that the incremental and decremental locks have been placed as close together as possible to minimize the length of time that other processes have to wait for ^XYZ(0) to become available.

```
LOCK +^XYZ(0)
SET (T%EntryNumber,^XYZ(0))=^XYZ(0)+1
LOCK -I%List(0)
SET ^XYZ(T%EntryNumber)=T%EntryValue
```

The following example is a modification of the previous example. It illustrates the repeated use of incremental lock with a timeout to provide feedback to the user that an attempt to lock the node is in progress. After 30 seconds, the process is abandoned.

```
DO $Env.Output("Locking")
FOR T%Loop=1:1:10 DO $Env.Output(".") LOCK +^XYZ(0):3 IF  QUIT
ELSE DO $Env.Output("Node is busy. Aborting.") QUIT
SET (T%EntryNumber,^XYZ(0))=^XYZ(0)+1
LOCK -I%List(0)
SET ^XYZ(T%EntryNumber)=T%EntryValue
```

# MERGE

The **MERGE** command performs a nondestructive array-copy operation, copying an array node and all its descendants to a new location.

## Format

M{ERGE} postcond SP L mergeargument

postcond ::= { : tvexpr }

A truth valued expression. Execution of the command is conditional based on the truth-value of this expression.

mergeargument

Can be one of the following:

$\text{glvn}_{\text{dest}}$ = $\text{glvn}_{\text{source}}$

@ expratom V L argument

where:

| | |
|---|---|
| **$\text{glvn}_{\text{dest}}$** | is the destination array node (The contents of the source array node are copied into this location without automatically destroying any information that may be contained there or in descendant array nodes.) |
| **$\text{glvn}_{\text{source}}$** | is the source array node (The contents of this array node are copied to the destination array subtree.) |

## Explanation

If array nodes already exist in the destination location, then those not explicitly overwritten by source nodes are not affected. If the source location's **$DATA** value is 0, then **MERGE** has no effect. If its **$DATA** value is 1, then **MERGE** is identical to **SET**.

An error results if the destination array is a subtree of the source array.

In the following example, note the starting state of the two arrays I%Elements and T%Temp. In the following diagrams, filled circles represent array nodes containing values (have **$DATA** values of 1 or 11) and open circles do not contain values (have $DATA values of 10).

```
MERGE T%Temp=I%Elements(22)
```

Following the **MERGE** command, I%Elements remains unchanged.

## Comments

Keep the following points in mind when you use the **MERGE** command:

- The **MERGE** command can copy a large number of array nodes. However, it can be expensive to execute and should not be used gratuitously. Note that **MERGE** is much faster than copying all descendant array nodes with a **FOR** loop and repeated **SET** commands.

- Some features of EsiObjects and the underlying M platform have not been implemented fully. When certain SSVNs or logical object structures are copied with **MERGE**, the destination array can contain only those nodes physically present in the source structure at the moment the **MERGE** operation occurs. Some sparsely allocated information cannot show up in the destination array (this behavior is likely to change in the future).

## Related

KILL command

SET command

## Examples

In the following example, the contents of the array T%SRC(22,17) are merged into T%DEST. Therefore, if the node T%SRC(22,17,0) exists, it is copied into T%DEST(0), and so on.

```
MERGE T%DEST=T%SRC(22,17)
```

The following example guarantees that the destination array T%DEST("Name") is an exact copy of the source array T%SRC("Info") by first killing the destination array.

```
KILL T%DEST("Name")
MERGE T%DEST("Name")=T%SRC("Info")
```

# NEW

The **NEW** command preserves the states of certain local variables to be restored when a **QUIT** pops the current stack frame level.

## Format

N{EW} postcond SP {L newargument}

postcond ::= { : tvexpr }

A truth valued expression. Execution of the command is conditional based on the truth of this expression.

newargument

Can be one of the following:

local
(L local)
@ expratom V L newargument

where:

| | |
|---|---|
| **local** | is a local (L%) variable name (array nodes are not allowed) whose current state is to be copied onto the process stack |

## Explanation

There are three forms of the **NEW** command:

- Inclusive - The most common form and the one most consistent with the paradigm of EsiObjects.
- Exclusive - A rare form of the **NEW** command whose use is discouraged. The purpose of this form is to remove all local (L%) variables except those enclosed in parentheses. The only time to use this form is when calling volatile external M code that is likely to use the exclusive **KILL** command, or to otherwise accidentally interfere with internal symbols required by EsiObjects.
- Argumentless - Not currently supported in EsiObjects, the purpose of the argumentless **NEW** is to remove all local (L%) variables.

The **NEW** command preserves the state of one or more local variables by copying them onto the stack so they can be restored later. When the **NEW** command acts on a local variable, it is as though the following actions were performed:

- 1. An entry was made on the top frame of the process stack, recording the name of the local variable.
- 2. If the variable exists, its current value (and the values of all descendant array nodes) is copied onto the top frame of the process stack.
- 3. The variable is killed.

For all local variables to be restored, when **QUIT** terminates execution at the current stack frame level before returning to the calling code context, it checks the top stack frame to see if their values (and the values of any descendant array nodes) are recorded there. If the values are recorded, then these values are restored.

If the **NEW** command operates on an undefined variable, that variable's value is undefined again after the **QUIT**, regardless of any operations that have been performed on it in between.

## Comments

Keep the following points in mind when you use the **NEW** command:

- When a **NEW** command is encountered, some underlying M platforms physically copy the entire contents of the affected local variables onto the stack, while other platforms do not. Some platforms temporarily experience an increase in local variable memory, while other platforms see a decrease. In either case, the result is a net loss of space. Overuse of the **NEW** command can cause an error by overstepping the limits of available memory.
- Some implementations of error processing clear the stack, thereby restoring the states of any local variables stacked with **NEW**. The 1995 ANSI M Standard error processing constructs do not have this effect.

## Related

KILL command

QUIT command

## Examples

The following example uses the **NEW** command to preserve the states of the local variables X, Y, and Z.

```
NEW L%X,L%Y,L%Z
```

# OPEN

The **OPEN** command gains exclusive ownership of a device, allowing the current process to send output to, and/or to read input from that device.

## Format

O{PEN} postcond SP L openargument

postcond ::= { : tvexpr }

A truth valued expression. Execution of the command is conditional based on the truth of this expression.

openargument

Can be one of the following:

expr {: {deviceparameters} {: {timeout} {: mnemonicspec}}}
@ expratom V L argument

where:

| | |
|---|---|
| **deviceparameters ::=** | **deviceparam** |
| | **(deviceparam {:deviceparam }...)** |
| **deviceparam ::=** | **expr** |
| | **devicekeyword** |
| | **deviceattribute = expr** |
| **Timeout** | is the number of seconds to wait before giving up if the device does not become available (Whenever a timeout expires, **$TEST** is set to 1 if the operation was successful, or to 0 if it failed.) |
| **Mnemonicspec ::=** | **mnemonicname** |
| | **(L mnemonicname)** |

## Explanation

No other process can open the same device until ownership is relinquished with the **CLOSE** command. If an attempt is made to open a device owned by another process, the current process hangs until the device is released, or until a specified timeout period expires. Whenever a timeout expires, **$TEST** is set to 1 if the operation is successful, or to 0 if it fails.

A variety of device parameters and mnemonic specifiers can be specified. These depend on the capabilities of the device in question and on the underlying M platform.

## Comments

Keep the following points in mind when you use the **OPEN** command:

Device names and their meaning are highly dependent on the specific M platform. Consult your Programmers Reference Guide for further details.

- Mnemonic names (mnemonicname) are specific to the underlying M platform.
- When attempting to open a device that may be owned by another user, it is often a good idea to use a timeout in case the device is unavailable for an extended period of time. In some cases, the user may be asked whether to give up the process or to wait until the device becomes unavailable.
- After ownership of a device has been established with **OPEN**, it is still necessary to make the device current before attempting any input/output (I/O) operations on that device. The **USE** command sets the current device.
- Multiple device parameters are enclosed in parentheses, separated by colons. Multiple mnemonic specifiers are enclosed in parentheses, separated by commas.

## Related

CLOSE command

READ command

USE command

WRITE command

$TEST special variable

## Examples

The following DSM example attempts to open a device for up to 30 seconds before giving up. If the operation is successful, the following occurs:

- Lines of text are read from a file whose identifier is in the variable T%File.
- The lines are echoed to the principal device (**$PRINCIPAL**) until a blank line is encountered.
- The file is closed.

```
DO $Env.Output("Opening Device "_T%File)
FOR T%Loop=1:1:10 DO $Env.Output(".")OPEN T%File::30 IF  QUIT
ELSE DO $Env.Output("Device "_%File_ is unavailable.") QUIT
FOR  DO  QUIT:T%Line=""
. USE T%File
. READ T%Line
. IF T%Line="" QUIT
. USE $PRINCIPAL
. WRITE T%Line,!
CLOSE T%File
QUIT
```

The following MSM example attempts to open the HFS file server device for up to 30 seconds before giving up. If the operation is successful, the following occurs:

- Lines of text are read from a file whose identifier is in the variable T%File.
- The lines are echoed to the principal device (**$PRINCIPAL**) until a blank line is encountered.
- The file is closed.

```
SET T%Dev=51
OPEN T%Dev:(T%File,"R")::30
ELSE DO $Env.Output("Unable to access HFS") QUIT

FOR  DO  QUIT:T%Line=""
. USE T%Dev
. READ T%Line
. IF T%Line="" QUIT
. USE $PRINCIPAL
. WRITE T%Line,!
CLOSE T%Dev
QUIT
```

## PRESERVE

The **PRESERVE** command increments the internal reference count of an object reserving it from destruction by the **DESTROY** command.

### Format

PRE{SERVE} postcond SP L preserveargument

postcond ::= { : tvexpr }

tvexpr evaluates to a truth-valued expression. Execution of the command is conditional based on the truth-value of this expression.

| | |
|---|---|
| **preserveargument ::=** | **expr V oref** |
| | **@ expr V preserveargument** |

The argument of **PRESERVE** is an object reference.

### Explanation

The **PRESERVE** command does the following:

- If the argument is not an **oref**, or references an object that does not exist, no action occurs.
- If the argument contains an oref of an existing object, that object's internal reference count is incremented by one.

Within the context of an object, the objects internal reference count can be incremented or retrieved using the **$REFERENCE** special variable.

### Comments

Keep the following points in mind when you use the **PRESERVE** command:

- When using the **DESTROY** command to destroy an object, the following occurs:
  - First the object’s internal reference count is decremented.
  - Next, the destroy action quits if the reference count is one or greater. No further action is taken.
  - If the reference count goes to zero, the destruction of the object proceeds normally and the DESTROY method will be called if it exists.
- The **PRESERVE** command simply increments the objects internal counter and has no other side effects.
- For every application of the **PRESERVE** command to increment an object’s internal reference count, a cooresponding **DESTROY** command must be applied to decrement that count. Only on the destroy action that decrements the count to less than one will the actual destroy take place.
- When the **CREATE** command creates an object, the internal reference count is set to one. You do not have to apply the **PRESERVE** command at object creation.

Please note that the **PRESERVE** command has no effect on a virtual object, because virtual objects have no state.

### Related

CREATE command

KILL command

$REFERENCE special variable

DESTROY command

$TEST special variable

### Examples

The following example increments the internal reference count of the object referenced by the symbol T%SharedObject.

```
PRESERVE T%SharedObject
```

## QUIT

The **QUIT** command terminates the stack level, recording the execution context as recorded on the top process stack frame.

### Format

Q{UIT} postcond SP {quitargument}

postcond ::= { : tvexpr }

A truth valued expression. Execution of the command is conditional based on the truth of this expression.

quitargument

Can be one of the following:

expr
@ expratom V expr

where:

| | |
|---|---|
| **expr** | Is an expression (The expression's value determines the return value of the current extrinsic function.) |

### Explanation

The **QUIT** command terminates execution of the current subroutine or extrinsic function. Inside the scope of the **FOR** command, **QUIT** terminates iteration and exits the **FOR** loop. Inside a block, **QUIT** exits the current dot indent level, returning control to the calling argumentless **DO** level.

The argument of a **QUIT** command that terminates an EsiObjects method, property accessor or event handler is automatically assigned to the special variable **$RETURN**. If no argument is specified, then $RETURN is used as the return value.

When **QUIT** terminates a stack frame created by **DO** or **XECUTE**, execution resumes immediately to the right of the **DO** or **XECUTE** argument that created the stack frame. If the **DO** was argumentless, execution resumes with the next command following the **DO**. In the case of an extrinsic function or value-returning object-with-service call, a value is returned into the context of an expression, and the evaluation of that expression is resumed. When **QUIT** is encountered in the scope of (on the same line and to the right of) **FOR**, execution of the innermost **FOR** loop is terminated.

If the current process is an application-mode process or a background process created by the **JOB** command, execution of the final **QUIT**, removing the last frame from the process stack, terminates the process.

The **QUIT** command does not allow the conventional form of argument indirection. Note that the syntax is **@expr**, not **@quitargument**. The implication is that the following example, which intends to return the value 2, is legal:

```
SET Result="Y-2",Y=4
QUIT @Result
```

But the following example, which intends to return the value 5, may not be legal for some underlying M platforms:

```
SET Result="Y_Z",Y="X+2",Z="-1",X=4
QUIT @@Result
```

## Comments

Keep the following points in mind when you use the **QUIT** command:

- In the case of an extrinsic function or value-returning object-with-service call, a value is returned into the context of an expression, and the evaluation of that expression is resumed. One difference between the two cases is that **QUIT** has an argument in an extrinsic function, and is argumentless in the code body that implements an object service.
- The argumentless **QUIT** must be followed by two spaces before a comment or other command that follows it on the same line. Do not confuse a postconditional with the argument of the **QUIT** command.

## Related

Extrinsic functions

DO command

FOR command

XECUTE command

## Examples

The argument of a **QUIT** command that terminates an EsiObjects method, property accessor or event handler is automatically assigned to the special variable $RETURN.  If no argument is specified, then $RETURN is used as the return value.  Thus the example below…

```
SET $RETURN=T%Value

QUIT
```

…could be expressed more succinctly by the following example…

```
QUIT T%Value
```

The following example illustrates the use of **QUIT** in a recursive extrinsic function (in other words, one that calls itself). This example implements the mathematical factorial operation. For example, 5 factorial (written 5!) equals 5*4*3*2*1. It is generally true that n! equals n*(n-1)!, and that 0! equals 1.

```
FACT(T%N) ; Return L%N factorial
IF T%N<0!(T%N\1'=T%N) QUIT ""
IF T%N=0 QUIT 1
QUIT T%N*$$FACT(T%N-1)
```

The factorial operation only applies to nonnegative integers, so the first **IF** command causes NULL ("") to be returned when T%N is not a nonnegative integer. The second **IF** command handles the special case where T%N equals 0. The **QUIT** on the third line handles the general case where T%N! equals T%N*(L%N-1)!.

# READ

The **READ** command reads input from the current device and can send simple output. Traditionally, the **READ** command has been used with dumb terminal devices. EsiObjects does not support these devices. When the **READ** command is used, it is usually used to read from an external device other than a dumb terminal.

## Format

R{EAD} postcond SP L readargument

postcond ::= { : tvexpr }

A truth valued expression. Execution of the command is conditional based on the truth-value of this expression.

| | |
|---|---|
| **Readargument ::=** | **strlit** |
| | **format** |
| | **glvn {# intexpr} {: timeout}** |
| | *** glvn {: timeout}** |
| | **@ expratom V L readargument** |

## Explanation

There are a variety of different forms of the **READ** command.

| | | |
|---|---|---|
| String literal | A string or numeric value to be sent as output. Note that, in contrast to **WRITE**, an expression cannot be specified in a **READ** argument. | |
| Format control | A format control parameter, whose exact behavior can be device dependent, and can have any one of the following formats: | |
| | **#** | Issues a form feed. |
| | **!** | Issues a return (CR+LF combination). |
| | **?integer** | Attempts to move the input position forward on the current device by writing spaces until **$X** equals the specified integer. If **$X** is not less than this value, no action occurs. |
| | **/controlmnemonic** | Performs a special device operation defined by the specified control mnemonic. |
| Variable name | A variable name. The input value is stored in the specified variable. The read operation can optionally be modified by the following constructs: | |

| | |
|---|---|
| **# intexpr** | The maximum number of characters to read. The read process terminates when this many characters are read, guaranteeing that the variable does not contain more characters. If a timeout occurs or a return is encountered, the variable contains fewer characters. |
| **:timeout** | The maximum number of seconds to pause between characters. **$TEST** is 1 if no timeout occurred, 0 if a timeout occurs. |
| ***** | A single character is read from the current device. The variable contains the ASCII value of that character, or 1 if a timeout occurs. |

## Comments

If specified, the maximum number of characters to read must be an integer greater than 0. Negative values can cause errors. Noninteger values are interpreted as integers. For example, the following two lines are equivalent:

```
READ T%X:"5.6Hello"
```

```
READ T%X:5
```

## Related

WRITE command

$IO special variable

$X special variable

## Examples

If a **READ** argument is a string or format control parameter, it is used for output purposes. If the argument is a variable name, it is used as a place to store input. These two forms are used in combination in the following example, which asks the user for an entry, storing the result in the variable T%EntryNumber and issuing a return after the user has typed a response:

```
READ "Entry: ",T%EntryNumber,!
Entry: Sockets
```

The following code reads lines of text from a file whose identifier is in the variable T%File and echoes those lines to the principal device (**$PRINCIPAL**) until a blank line is encountered. Then the file is closed.

```
OPEN T%File::10
ELSE  DO $Env.Output("Device "_T%File_" is unavailable.") QUIT
FOR  DO  QUIT:T%Line=""
. USE T%File
. READ T%Line
. IF T%Line="" QUIT
. USE $PRINCIPAL
. WRITE T%Line,!
CLOSE T%File
QUIT
```

## SET

The **SET** command assigns the value of a variable (or some other **SET** destination).

### Format

S{ET} postcond SP L setargument

postcond ::= { : tvexpr }

A truth valued expression. Execution of the command is conditional based on the truth-value of this expression.

| | | |
|---|---|---|
| **setargument ::=** | **setleft** | **= expr** |
| | **(L setleft)** | |
| | **@ expratom V L setargument** | |

where:

| | |
|---|---|
| **setleft ::=** | **glvn** |
| | **$D{EVICE}** |
| | **$EC{ODE}** |
| | **$ET{RAP}** |
| | **$K{EY}** |
| | **$X** |
| | **$Y** |
| | **$E{XTRACT} (glvn{,intexpr1{,intexpr2}})** |
| | **$P{IECE} (glvn,expr1{,intexpr1{,intexpr2}})** |
| | **object.property** |

### Explanation

The following is the most common form of **SET**:

SET variable=value

In this format, the variable (or array node) is dynamically created if it did not exist yet. If it already existed, its old value is overwritten. Existing array descendants are not affected.

The first line in the following example assigns the value of variable T%Done and the second line assigns the value of the subscript named by the variable T%Loop inside the array I%Elements.

```
SET T%Done=1
SET I%Elements(T%Loop)=""
```

It is also possible to set the values of certain special variables. For example, the constructs **SET $Y** and **SET $X** are used to modify EsiObjects' notion of the current row and column positions, respectively.

```
DO PLOT(T%Row,T%Column) ; Set actual cursor position
SET $X=T%Column,$Y=T%Row
```

**SET $EXTRACT** is used to replace one or more character positions of a variable's contents without affecting the rest of the string. If the variable does not yet exist, it is given a starting value of NULL (""). If the number of characters in the existing string is less then the starting character position, extra space characters are added as necessary.

```
SET T%String="ABCDEFG"
SET $EXTRACT(T%String,3,5)="*"
DO $Env.Output(T%String)


Results: AB*FG
```

**SET $PIECE** is used to replace one or more delimited pieces of a variable's contents without affecting the rest of the string. If the variable does not yet exist, it is given a starting value of NULL (""). If the number of pieces in the existing string is less then the starting piece position, extra delimiters are added as necessary.

```
SET T%String="one/two/three/four/five/six/seven"
SET $PIECE(T%String,"/",3,5)="*"
DO $Env.Output(T%String)

Results: one/two/*/six/seven
```

Entire **SET** arguments are evaluated one after another in left-to-right order. Within one **SET** argument, the following order of evaluation applies:

1. If array subscripts or indirect references are found to the left of the equals sign (except in **SET $PIECE** and **SET $EXTRACT** arguments other than the first argument), those array subscripts and indirect references are evaluated.
2. The expression to the right of the equals sign is evaluated.
3. The resulting value is assigned to the destinations on the left side of the equals sign. If there is more than one destination, they are assigned in left-to-right order.

When the **SET** destination on the left side of the equals sign is of the form object.property, the Assign accessor for that object is invoked. This is a method in whose context **$RETURN** defaults to 1. If the method returns 1 or some other true value, it means that the assignment operation was successful; an event is then triggered. If the method returns 0 or some other false value, it means that the assignment operation was not successful; no event is triggered.

Handlers of the event automatically triggered by a successful return of the Assign accessor must accept four parameters in their formal parameter list as follows:

- Object - The object reference of the object containing the property.
- Property - The full property name, of the form interface::name.
- Callframe object - An object that can optionally be used to inquire into the parameters that were passed in.
- Operation - **SET** or **KILL** commands.

### Comments

In assigning values to variables, the variable need not be defined prior to setting it. The process of assignment dynamically creates the variable if necessary, or overwrites any existing value. This is also true when using the **SET $PIECE** and **SET $EXTRACT** constructs, which both assume a starting value of NULL ("") if the variable did not exist.

### Related

$EXTRACT function

$PIECE function

$DEVICE special variable

$ECODE special variable

$ETRAP special variable

$KEY special variable

$X special variable

$Y special variable

### Examples

The first line in the following example assigns the value 1 to the variable T%Done.

```
SET T%Done=1
```

The multiple-destination **SET** command can be used to assign the same value to more than one destination in a single operation. A list of destinations on the left side of the equals sign is enclosed in parentheses. The following example sets the instance variables I%Height and I%Width to 0.

```
SET (I%Height,I%Width)=0
```

In the following example, the construct **SET $ETRAP** is used to specify a line of code to be invoked in the event of an error.

```
SET $ETRAP="DO ERRHNDL^MYRTN(""READFILE"")"
```

**SET $EXTRACT** is used to replace one or more character positions of a variable's contents without affecting the rest of the string. In this example, characters 3 through 5 of the string in T%String are replaced with an asterisk (*).

```
SET T%String="ABCDEFG"
SET $EXTRACT(T%String,3,5)="*"
DO $Env.Output(T%String)
```

```
Results:AB*FG
```

If the variable does not yet exist, it is given a starting value of NULL (""). If the number of characters in the existing string is less then the starting character position, extra space characters are added as necessary. In the following example, the variable T%String is undefined, and character position 5 is replaced with the string "Text". To achieve this, four spaces are automatically placed at the start of the string.

```
KILL T%String
SET $EXTRACT(T%String,5)="Text"
DO $Env.Output(T%String)
```

```
Results:    Text
```

**SET $PIECE** is used to replace one or more delimited pieces of a variable's contents without affecting the rest of the string. In this example, pieces 3 through 5 of the string in T%String are replaced with an asterisk (*).

```
SET T%String="one/two/three/four/five/six/seven"
SET $PIECE(T%String,"/",3,5)="*"
DO $Env.Output(T%String)
```

```
Results: one/two/*/six/seven
```

If the variable does not exist yet, it is given a starting value of NULL (""). If the number of pieces in the existing string is less then the starting piece position, extra delimiters are added as necessary. In this example, the variable T%String is undefined, and "." piece 5 is replaced with the string Text. To achieve this, four periods are automatically placed at the start of the string.

```
KILL T%String
SET $PIECE(T%String,".",5)="Text"
DO $Env.Output(T%String)
```

```
Results: ....Text
```

The following example uses a **FOR** loop with the one-argument **$LENGTH**, **SET $PIECE**, and **$EXTRACT** to produce a string in which the individual characters of the string EsiObjects become comma-delimited pieces in the variable T%String. After these lines have been executed, T%Result should contain the string "E,s,i,O,b,j,e,c,t,s".

```
SET T%Result="",T%String="EsiObjects"
FOR T%Loop=1:1:$LENGTH(T%String) DO
. SET $PIECE(T%Result,",",T%Loop)=$EXTRACT(T%String,T%Loop)
```

The following extrinsic function performs a search-and-replace operation on a string, sending back the transformed string as its return value. It uses the two-argument **$LENGTH** to measure the number of pieces in the source string, and uses **$PIECE** and **SET $PIECE** to do the replacement operation.

```
REPL(L%String,L%From,L%To) ; Replace L%From with L%To in L%String
  NEW L%Iter,L%Result,L%Length
  IF L%From="" QUIT ""
  SET L%Length=$LENGTH(L%String,L%From)
  IF L%To="" SET L%Result="" FOR L%Iter=1:1:L%Length DO
  . SET L%Result=L%Result_$PIECE(L%String,L%From,L%Iter)
  ELSE  FOR L%Iter=L%Length:-1:1 DO
  . S $P(L%Result,L%To,L%Iter)=$P(L%String,L%From,L%Iter)
  QUIT L%Result
```

The following expression should return the string EsiObjects Language for EsiObjects Programming:

```
$$REPL("M Language for M Programming","M","EsiObjects")
```

Setting a property's value invokes the Assign accessor method. For example, the following code invokes the Assign accessor method for the Visit property of the object T%Object12:

```
SET T%Object12.Visit($HOROLOG,T%Txnum)=10
```

Assuming that the Assign accessor method is able to successfully assign the value 10, it returns some true value such as 1. In this case, an event is triggered. The event includes the following parameters:

- Object
- Full property name
- Callframe object
- Operation

The following code might be used to handle this event. If the **$HOROLOG** value specified in the first parameter indicates today's date, the handler exits. Otherwise, it gets the remaining parameters and calls the LOG subroutine to make note of the event.

```
HANDLE(T%Object,T%Property,T%CFO,L%Oper) ; Handle assign/kill
    IF +(T%CFO.Parameter(1))=+$H Q  ; Exit if it is today
    SET T%Value=CFO.Parameter(2)
    DO LOG(T%Value,T%Oper)
    QUIT
```

## USE

The **USE** command is used to set the current input device to some owned device.

### Format

U{SE} postcond SP L useargument

postcond ::= { : tvexpr }

tvexpr evaluates to a truth-valued expression. Execution of the command is conditional based on the truth-value of this expression.

useargument

Can be one of the following:

expr {: {deviceparameters} {: mnemonicname}}
@ expratom V L argument

where:

| | |
|---|---|
| **deviceparameters ::=** | **Deviceparam** |
| | **(deviceparam{:deviceparam}...)** |
| **deviceparam ::=** | **Expr** |
| | **Devicekeyword** |
| | **Deviceattribute = expr** |

### Explanation

The **USE** command is used to set the current input device **$IO** to some owned device. The argument of the **USE** command must be a device that is owned by the current process, optionally followed by device parameters and/or a mnemonic name appropriate to that device. If this optional information is not specified, default values are assigned by the underlying M platform.

### Comments

Keep the following points in mind when you use the **USE** command:

- Only a device already owned by the current process can be specified in the argument of **USE**. The **OPEN** command is used to gain ownership of a device. The principal device (**$PRINCIPAL**) is automatically owned at login (except in the case of certain background processes not tied to any device).
- Multiple device parameters are enclosed in parentheses, separated by colons.
- Device 0 is synonymous with **$PRINCIPAL**. Therefore, note the following command:
- USE 0

The previous command is equivalent to the following:

- USE $PRINCIPAL
- The **USE** command affects the special variables **$IO**, **$KEY**, **$X**, **$Y**, and **$DEVICE**, whose values are all determined by the current device.

## Related

CLOSE command

OPEN command

READ command

WRITE command

$DEVICE special variable

$IO special variable

$KEY special variable

$PRINCIPAL special variable

$X special variable

$Y special variable

## Examples

The following DSM example reads lines of text from a file whose identifier is in the variable T%File. It echoes these lines to the principal device (**$PRINCIPAL**) until a blank line is encountered. Then the file is closed. The **USE** command is invoked repeatedly to set the current device so that line after line can be read first from the file and then can be displayed on the principal device.

```
OPEN T%File::30
ELSE DO $Env.Output("Device "_T%File_" is unavailable.") QUIT
FOR  DO  QUIT:T%Line=""
. USE T%File
. READ T%Line
. IF T%Line="" QUIT
. USE $PRINCIPAL
. WRITE T%Line,!
CLOSE T%File
QUIT
```

The following MSM example attempts to use the HFS file server device for up to 30 seconds before giving up. If the operation is successful, the following occurs:

- Lines of text are read from a file whose identifier is in the variable T%File.
- The lines are echoed to the principal device (**$PRINCIPAL**) until a blank line is encountered.
- The file is closed.

```
SET T%Dev=51
OPEN T%Dev:(T%File,"R")::30
ELSE DO $Env.Output("Unable to access HFS")
FOR  DO  QUIT:T%Line=""
. USE T%Dev
. READ T%Line
. IF T%Line="" QUIT
. USE $PRINCIPAL
. WRITE T%Line,!
CLOSE T%Dev
QUIT
```

## WATCH

The **WATCH** command specifies that one or more events are to be watched for one or more objects.

### Format

WA{TCH} postcond SP {L warg}

postcond ::= { : tvexpr }

A truth valued expression. Execution of the command is conditional based on the truth-value of this expression.

| | |
|---|---|
| **warg ::=** | **target.weleref@expratom V L warg** |
| **weleref ::=** | **ielem:vector** |
| | **(L ielem:vector)** |
| **ielem ::=** | **Rielem**<br>**@expratom V rielem** |
| **rielem ::=** | **$PROPERTIES**<br>**$EVENTS**<br>**event**<br>**property** |
| **vector ::=** | **label{^{interface::}method}** |

### Explanation

It is possible for the **WATCH** command to watch specific events or properties by naming them, to watch all events by specifying the special name (not a special variable) **$EVENTS**, or to watch all properties by specifying the special name **$PROPERTIES**. The target argument indicates an object for which events and properties are to be watched.

Each time a watch is specified, a callback vector must be named. This vector must be a public label in the specified method (or the current method if none is specified). The method is assumed to be in the primary interface for the current class, unless another interface is specified.

### Property Watches

When watching $PROPERTY or a specific property, the format for the associated label within the specified method is as follows:

```
(Public)LABEL(Object,Property,CallFrame,Action,Value) ;
```

The following table describes the valid parameters. These values are passed into the method when the event is fired.

| Parameter | Description |
|---|---|
| **Object** | The object reference of the object containing the property. |
| **Property** | The full property name in the form interface::propertyname. |
| **Callframe** | An object that can optionally be used to inquire into the parameters that were passed in. This object is the callframe object. It is always null when the DEAD event type is specified. |
| **Action** | "PRESET", "SET", "SETREJECT", "PREKILL", "KILL", "KILLREJECT" or "DEAD" identifies the event type. |
| **Value** | When a property value is actually changed through a SET command, the value it is set to will be passed back. This applies to the PRESET and SET actions. |

One useful property of the Callframe object is **CancelAction**. If this property is set to 1 (Set obj.CancelAction=1) for the Action types PRESET and PREKILL, the setting or killing of the property will be aborted.

## Event Watches

When watching $EVENTS or a specific event, the format for the associated label within the specified method is as follows:

```
(Public)LABEL(Object,Event,…) ;
```

The first two parameters always present by default and are described below.

| Parameter | Description |
|---|---|
| **Object** | The object reference of the object firing the event. |
| **Event** | The full name of the event in the form interface::eventname. |

Any additional parameters passed into the handler are those specified on the **EVENT** command.

When a watched object is destroyed via the **DESTROY** command, the system will automatically generate a **ObjectDead** event. This is useful in cases where it is important to know when objects are destroyed so that some action can be taken. This event can be watched in the normal fashion using the **WATCH** command.

## Comments

Keep the following points in mind when you use the **WATCH** command:

- If a concerned object wants to watch all events except for one about the watched object, it is not possible to use **WATCH $EVENTS** and then use the **IGNORE** command to the specific event. Instead, it is necessary to specifically watch those events about which the object is concerned.
- If a concerned object is watching **$PROPERTIES** and a single property, it is informed twice when an event for that property is triggered.
- If the watched object has used the **$WATCHDETECT** function, it can be informed that it is being watched.
- The label used to handle an event must be public. It must be able to handle at least two parameters:
- OBJECT - An object reference to the watched object.
- MESSAGE - The name of the event or property that was triggered by the **EVENT** command.
- An example of a public label would be:

  (PUBLIC)BEVENT(OBJ,MSG) ;
- However, many events send additional parameters, and the handler's formal parameter list must not declare fewer parameters then are sent with the event.

## Related

Method structure

EVENT command

IGNORE command

$WATCHDETECT function

## Examples

The following example causes the current object to watch all events for the object T%Object12. The handler for an event from this object is the label MODIFY in the method Update that is part of the primary (default) interface.

```
WATCH T%Object12.$EVENTS:MODIFY^Update
```

The following example causes the current object to watch all properties for the object T%Object12. The handler for a property from this object is the label MODIFY in the method Update.

```
WATCH T%Object12.$PROPERTIES:MODIFY^Update
```

The following example causes the current object to watch the Renamed event for the object T%Object12. The handler for this property is the label MODIFY in the method Update.

```
WATCH T%Object12.Renamed:MODIFY^Update
```

The following example causes the current object to watch the Renamed and ObjectDeleted events for the object T%Object12. The handlers for these methods are the public labels MODIFY and DELETE, respectively, in the method Update.

```
WATCH T%Object12.(Renamed:MODIFY^Update,ObjectDeleted:DELETE^Update)
```

# WRITE

The **WRITE** command is used to write to the current device. Traditionally, the **WRITE** command has been used with dumb terminals. EsiObjects does not support dumb terminals, however, the Output Window of the EsiObjects Visual Development Environment serves as an output device for the **WRITE** command.  The **WRITE** command can be used for all devices supported by the underlying M implementation.

## Format

W{RITE} postcond SP L writeargument

postcond ::= { : tvexpr }

tvexpr evaluates to a truth-valued expression. Execution of the command is conditional based on the truth-value of this expression.

| | |
|---|---|
| **writeargument ::=** | **expr** |
| | **format** |
| | *** intexpr** |
| | **@ expratom V L readargument** |

## Explanation

There are several different forms of the **WRITE** command.

**Expression** An expression whose value is sent to the current device as output.

| | | |
|---|---|---|
| **Format Control** | | A format control parameter, whose exact behavior can be device dependent and can have any one of the following formats: |
| | **#** | issues a form feed. |
| | **!** | issues a return. (CR+LF) |
| | **? integer** | writes spaces while **$X** is less than the specified integer. If **$X** is not less than this value, nothing happens. |
| | **/controlmnemonic** | performs a special device operation defined by the specified control mnemonic. |
| **Integer Expression** | | A number denoting an ASCII character code value. The corresponding ASCII character is sent as output to the current device. This is useful in generating escape sequences and control characters. The **$CHAR** function is an alternative approach. |

## Comments

Keep the following points when you use the **WRITE** command:

- ASCII character codes can be specified with the **WRITE *integer** syntax or with the **$CHAR** function. For example, the following examples are the same:

```
WRITE *27,*96,*65
```

```
WRITE $CHAR(27,96,65)
```

- The argument of a **WRITE** command can be any expression. For example, you can call an extrinsic function in an expression. This extrinsic function can issue a **USE** command, which changes the current device. When the function returns, the **WRITE** command's argument are directed to the new current device instead of the device that was current when the **WRITE** command began to evaluate its argument.

## Related

$IO special variable

$X special variable

## Examples

The **WRITE** command can specify format control parameters that modify the output position on the current device. These parameters can be combined in any way in a single **WRITE** argument, as long as the **?** parameter is the last parameter specified. In the following example, the **WRITE** command issues a form feed and two returns, moves the cursor to the 10th column position, and displays the height in variable H. It then issues another return, moves the cursor to the 11th column position, and displays the width in variable W.

```
WRITE #!!?10,"Height: ",H,!?11,"Width: ",W
```

The following example reads lines of text from a file whose identifier is in the variable T%File, echoing those lines to the principal device (**$PRINCIPAL**) until a blank line is encountered. Then the file is closed.

```
OPEN T%File::10
ELSE DO $Env.Output("Device "_T%File_" is unavailable.") QUIT
FOR  DO  QUIT:T%Line=""
. USE T%File
. READ T%Line
. IF T%Line="" QUIT
. USE $PRINCIPAL
. WRITE T%Line,!
CLOSE T%File
QUIT
```

Many M programmers are used to using the WRITE command to write information out to the principle device while programming and in particular, while debugging. You can use the WRITE command in the same manner in EsiObjects. When executing the WRITE command in an execute shell, the output will be directed to the Output tab sheet of the Output Window. For example:

```
W "Test" ;Create a line in the Output Window with the text "Test"

W !,"Test" ;Create an empty Line, followed by "Test"

W # ;Clear the Output tab sheet of the Output Window

W ?10,"Test” ;Create a line, 10 spaces, then another line with "Test"

W !! ;This is not supported and will generate an error.
```

```
W !,! ;Adds two lines to the output window.
```

# XECUTE

The **XECUTE** command's argument is a string of EsiObjects code to be executed as though it were called as a subroutine from within the current code body. The Xecute command violates encapsulation and should not be used unless necessary.

### Format

X{ECUTE} postcond SP L xargument

postcond ::= { : tvexpr }

A truth valued expression. Execution of the command is conditional based on the truth-value of this expression.

xargument

Can be one of the following:

expr postcond

@ expratom V L xecuteargument

where:

| | |
|---|---|
| **expr** | is an expression whose value is a single line of M code |

### Explanation

In general, the following example:

```
XECUTE T%CodeToExecute
```

Is equivalent to the following:

```
DO SUBRTN99
...
SUBRTN99 ; Imaginary Subroutine
    (Contents of T%CodeToExecute)
    QUIT
```

Note that neither a label nor a line-start indicator should be present at the beginning of the xecute string, nor should a return be present at the end.

It is useful to keep this analogy in mind when thinking about the effect of certain language elements, such as **QUIT** and **$TEST**, inside the scope of the **XECUTE** command. **XECUTE** does not place $TEST on the stack, and a **QUIT** inside its context exits the **XECUTE**, not the calling line of code. Because the contents of the expression are confined to a single line of code, it is not possible to place any commands after the scope of a **FOR**, **IF** or **ELSE**.

Nearly all EsiObjects language elements can be used inside the xecute string, as long as they are appropriate in a subroutine called from the current code body. The xecute string can contain extrinsic function calls, subroutines invoked by **DO**, and recursive calls to the

**XECUTE** command. Because it is confined to a single line, there is no reason to place the argumentless **DO** inside an xecute string.

Like **GOTO** and **DO**, **XECUTE** allows a postconditional to be applied to the command, or to any of its arguments. The following table summarizes the results when the postconditional in either location is true or false.

| Result | Postconditional on Command | Postconditional on Argument |
|---|---|---|
| True | Execute the command and its arguments. | Execute that argument before going on to the next argument or command. |
| False | Skip the command and all its arguments. | Skip that argument and go on to the next argument or command. |

## Comments

Keep the following points in mind when you use the **XECUTE** command:

- **XECUTE** creates a process stack frame to remember the execution location where it was called just like a subroutine call invoked with **DO**. The value of the **$TEST** special variable is not recorded on this stack frame. As a result, the value of **$TEST** can be changed by operations inside the xecute string, or by code invoked from **DO** or **XECUTE** commands in the xecute string.
- A new stack frame is created to run the code in the **XECUTE** string. A **QUIT** command inside the xecute string removes this stack frame, exiting the xecute string and returning to the immediate right of the calling **XECUTE** argument. (This does not apply to a **QUIT** command that is inside the scope of a **FOR** loop in the xecute string, which only terminates the loop as usual.)

## Related

DO command

QUIT command

## Examples

Sometimes, when using the **XECUTE** command, it is necessary to specify quotation marks inside a string literal. In such cases, two quotation marks are used for every quotation mark desired in the target string. This is applied recursively for strings inside strings. For example, consider the following line of code:

```
DO $Env.Output("Hello")
```

This line of code uses quotation marks to delimit the string literal Hello. It can be embedded inside an **XECUTE** command as follows:

```
XECUTE "DO $Env.Output(""Hello"")"
```

## ZAPPLY

The **ZAPPLY** command applies some or all of the object's instance variables in the context of an object's CREATE method. This allows access to these instance variables inside the context of the CREATE method.

### Format

ZAP{PLY} postcond SP {L zapargument}

postcond ::= { : tvexpr }

A truth valued expression. Execution of the command is conditional based on the truth-value of this expression.

zapargument

Can be one of the following:

property

@ expratom V L zapargument

where:

| | |
|---|---|
| **property** | is the name of a property to be applied |

### Explanation

The **ZAPPLY** command applies the values of instance variables to an object inside the CREATE method. The CREATE method, if defined, is called by the **CREATE** command when creating a new instance of a class. The instance variable values are not normally applied until after the CREATE method has finished executing. Therefore, **ZAPPLY** allows instance variable values to be accessed inside the context of the -CREATE method.

For a full description about creating an object, see the CREATE command.

### Comments

Keep the following points in mind when you use the **ZAPPLY** command:

- Argumentless **ZAPPLY** applies all instance variable values to the object.
- The argument of the **ZAPPLY** command is a property name, not an instance variable name.
- Use of the **ZAPPLY** command is valid only inside the CREATE method and in no other context.

### Related

CREATE command

## Examples

The following example applies all instance variables to the object being created. This is currently the most common use of the **ZAPPLY** command.

```
ZAPPLY
```

In the following example, only the Height and Width properties are applied.

```
ZAPPLY Height,Width
```

# Special Variables

Special variables are system defined and maintained variables that contain information about various values or processes in the operating environment. If you need to know or use the information stored in the special variable, you can access the information by using the special variable name.

Each special variable is labeled as to its ANSI Standard status as described in the following table:

| **Status** | **Description** |
|---|---|
| Standard | Indicates that the language element is part of the M ANSI Standard. |
| Proposed | Indicates that the language element is being proposed as an addition to the M ANSI Standard. |
| Extended | Indicates that the Standard language element has been modified for use in EsiObjects. |
| EsiObjects | Indicates that the language element is not part of the Standard and is an extension of EsiObjects. |
| Vendor | Indicates that the language element is M vendor-specific. |

## $CALLER

The **$CALLER** special variable returns an object reference to the object that sent the current message.

### Format

$CALLER

### Explanation

The **$CALLER** return value is an object reference (**oref**) or NULL ("") if no object sent the message. This special variable generally is used to send messages back to the calling object as part of a dialog, or to interact with properties of the calling object.

### Comments

Callbacks are an alternative to using the **$CALLER** special variable. For more information about using callbacks see Callback Syntax and the Using Events in the EsiObjects Programmer's Reference Guide.

### Related

Message Syntax

### Examples

The following example assigns the Text property of **$CALLER**.

```
SET $CALLER.Text=T%Text
```

## $CALLFRAME

The **$CALLFRAME** special variable returns the OID of the current call frame.

### Format

$CALLFRAME

### Explanation

**$CALLFRAME** returns the OID of the current call frame context. The OID can be used to provide object-oriented access to that objects interface. It is typically used in conjunction with the $DELEGATE function. $DELEGATE is used to delegate the call frame to another method. $CALLFRAME is used to access that call frame object as an object through its services.

### Related

Message Syntax

$DELEGATE Function

$UNKOWN special method.

### Examples

The following example saves the value of **$CALLMETH** to a temporary variable.

```
  SET T%Name=$CALLMETH
```

## $CHILDCNT

The **$CHILDCNT** special variable returns the number of children an object has. It is the cardinality of children collection pointed to by **$PEERS**.

### Format

**$CHILDCNT**

### Explanation

The $CHILDCNT special variable returns the number of children objects that exist within the context of the current object. The children are stored in a collection and the count is the cardinality of that collection. Child objects are created throught the Child=1 keyword on the CREATE command.

### Comments

The $CHILDREN special variable holds the pointer to the collection of child objects. The $CHILDCNT special variable returns the cardinality of that collection. Typically, these special variables are used to find leaks in an application, that is, objects that are not acconted for. Typically, it is used as a debugging tool.

The EsiObjects Object Browser makes use of these special variables. A special tool bar exists that permits the display of child objects.

### Related

$CHILDREN special variable

$LASTCHILDID special variable

$PEERS special variable

$SELF special variable

CREATE command

## $CHILDREN

The $CHILDREN special variable provides a browsing context for finding the children of the current object.

### Format

**$CHILDREN**

### Explanation

The $CHILDREN special variable contains a pointer to the collection that holds the children of the current object created by specifying the Child=1 on the CREATE command. Access to the collection is typically used by the programmer to determine memory leaks. That is, are there objects that exist that are not accounted for by the application.

### Comments

$CHILDREN is used primarily as a debugging tool to determine orphaned child objects.

The EsiObjects Object Browser uses this variable to access the child objects if the Child tool bar button is pressed.

### Related

$CHILDCNT special variable

$LASTCHILDID special variable

$PEERS special variable

$SELF special variable

CREATE command

## $CLASS

The **$CLASS** special variable contains the class of the current object.

### Format

$CLASS

### Explanation

The **$CLASS** special variable contains an object reference (**oref**) to the class of the current object.

### Comments

The name of the class can be obtained by examining its Name property. The class name is not always known because method implementations are often inherited by subclasses.

### Related

Message Syntax

$CALLMETH

### Example

The following example gets the name of the object's class.

```
SET T%ClassName=$CLASS.Name
```

## $DEVICE

The **$DEVICE** special variable returns the status of the current device

### Format

$D{EVICE}

### Explanation

The **$DEVICE** special variable returns the status of the current device in the following format:

status {,info {,text}}

where:

| | |
|---|---|
| **status** | is normally 0, but can be nonzero in case of an error |
| **Info** | contains other device information |
| **Text** | is a text message explaining the current status |

Only **text** can contain commas.

### Comments

Keep the following points in mind when you use the **$DEVICE** special variable:

- **$DEVICE** evaluates to a true value if a device error has occurred. Therefore, it is convenient to use **$DEVICE** as an argument of the **IF** command.
- **$DEVICE** is sensitive to the current device. Therefore, whenever the **USE** command is issued, the value of **$DEVICE** is likely to change as a result.
- The text portion of **$DEVICE** can contain commas. Therefore, when using **$PIECE** to obtain this portion, it is not sufficient to get only the third comma-piece.

### Related

IF command

USE command

$IO special variable

## Examples

The following example uses the truth value of **$DEVICE** to determine whether an error has occurred on the current device. If an error occurs, it uses **$IO** to determine the device identifier, **$PIECE** to obtain the three components of **$DEVICE** and displays them in the output window.

```
IF $DEVICE DO  ; Display error text
. SET T%Code=$PIECE($DEVICE,",")
. SET T%=Status$PIECE($DEVICE,",",2)
. SET T%=Text$PIECE($DEVICE,",",3,999)
. DO $ENV.Output("Error on device "_T%Device)
. DO $ENV.Output("Error code="_T%Code_", Status="_T%Status)
. DO $ENV.Output("Error text="_T%Text)
```

## $DOMAIN

The $DOMAIN special variable produces the current domain name.

### Format

**$DOMAIN**

### Explanation

Domains are objects for creating a common naming structure and storage structure.

### Comments

O% variables reside in domains.

$Domain can be set to the name of a valid domain name or a NewNamePool object.

The domain must exist.

### Related

CREATE Command

### Examples

```
Set $Domain=”Test” ;Switch to the Test domain.
```

## $ECODE

The **$ECODE** special variable contains the current error condition.

### Format

$EC{ODE}

### Explanation

The **$ECODE** special variable contains the current error condition in the following format:

, ecode {, ecode ...} ,

Each ecode cannot contain any commas. Values of ecode beginning with M are part of the ANSI M Standard, values beginning with U are user-defined, and values beginning with Z are implementation-specific. If no error occurs, **$ECODE** equals NULL ("").

### Comments

The error codes in **$ECODE** are surrounded by commas (they are not comma-delimited). Therefore, the number of comma-pieces in the string is always two greater than the number of error codes.

EsiObjects adds the following special codes to those generated by ANSI Standard M, and by the underlying M implementation.

ZOREJECT
ZOCALL
ZOUNDEL
ZODEAD
ZOSUBSCR
ZOPOOL

All codes are in the form of

<Code-EsiObjectsCode>

### Related

$ESTACK special variable

$ETRAP special variable

## Examples

The following example displays the current error trap, error stack level, and each of the error codes in **$ECODE**. Note that because the argumentless **DO** creates an additional stack level, it is necessary to record some of the values before entering the block.

```
IF $ECODE'="" SET T%ECode=$ECODE,T%EStack=$ESTACK DO
. SET T%ETrap=$ETRAP
. DO $ENV.Output("Error Trap: "_T%ETrap)
. DO $ENV.Output("Error Stack level: "_T%EStack)
. DO $ENV.Output("Error Codes:")
. FOR T%Loop=2:1:$LENGTH(T%ECode,",")-1 DO
. . SET T%ThisECode=$PIECE(T%ECode,T%Loop)
. . DO $ENV.Output("     "_T%ThisEcode_" "_$$ERRLKUP(T%ThisECode))
QUIT
```

A special extrinsic function at the ERRLKUP label is not shown.

## $ENVIRONMENT

The **$ENVIRONMENT** special variable contains an object reference to the EsiObjects environment.

### Format

$ENV{IRONMENT}

### Explanation

The environment can handle a variety of useful general messages. Error reporting is one of many applications of this feature.

### Comments

The environment always exists when EsiObjects is running. Every time the system is restarted, a new incarnation of the environment is created. Therefore, object references to the environment are no longer valid during the next EsiObjects session. Checking whether the incarnation has changed is a useful way of determining whether a new session has been started.

### Examples

The following example displays a warning message. Note the use of **$ENVIRONMENT** to produce the error message.

```
DO $ENVIRONMENT.Error(Text:"Invalid input sent to ")
```

The following example uses **$ENVIRONMENT** to find the value of the WindowHeight preference.

```
SET T%Height=$ENVIRONMENT.FindPreference(Name:"WindowHeight")
```

## $ESTACK

The **$ESTACK** special variable contains the number of stack levels since the stack frame that last contained an error condition.

### Format

$ES{TACK}

### Explanation

The **$ESTACK** special variable contains the number of stack levels since the stack frame that last contained an error condition. If no error condition has occurred, **$ESTACK** is empty.

### Comments

Every subroutine call, argumentless **DO**, and so on, creates a new stack frame entry. If an error occurs, then this adds 1 to the value of **$ESTACK**.

### Related

$ECODE special variable

$ETRAP special variable

### Examples

The following example displays the current error trap, error stack level, and each of the error codes in **$ECODE**. Note that because the argumentless **DO** creates an additional stack level, it is necessary to record some of the values before entering the block.

```
IF $ECODE'="" SET T%ECode=$ECODE,T%EStack=$ESTACK DO
. SET T%ETrap=$ETRAP
. DO $ENV.Output("Error Trap: "_T%ETrap)
. DO $ENV.Output("Error Stack level: "_T%EStack)
. DO $ENV.Output("Error Codes:")
. FOR T%Loop=2:1:$LENGTH(T%ECode,",")-1 DO
. . SET T%ThisECode=$PIECE(T%ECode,T%Loop)
. . DO $ENV.Output("     "_T%ThisECode_" "_$$ERRLKUP(T%ThisECode))
QUIT
```

## $ETRAP

The **$ETRAP** special variable equals a string of code to be invoked at the current M process stack level if an error occurs.

### Format

$ET{RAP}

### Explanation

The **$ETRAP** special variable equals a string of code to be invoked at the current M process stack level if an error occurs.

### Comments

The M process stack is not equivalent to the EsiObjects method process stack.

### Related

$ECODE special variable

$ESTACK special variable

### Examples

The following example displays the current error trap, error stack level, and each of the error codes in **$ECODE**. Note that because the argumentless **DO** creates an additional stack level, it is necessary to record some of the values before entering the block.

```
IF $ECODE'="" SET T%ECode=$ECODE,T%EStack=$ESTACK DO
. SET T%ETrap=$ETRAP
. DO $ENV.Output("Error Trap: "_T%ETrap)
. DO $ENV.Output("Error Stack level: "_T%EStack)
. DO $ENV.Output("Error Codes:")
. FOR T%Loop=2:1:$LENGTH(T%ECode,",")-1 DO
. . SET T%ThisECode=$PIECE(T%ECode,T%Loop)
. . DO $ENV.Output("     "_T%ThisECode_" "_$$ERRLKUP(T%ThisECode))
QUIT
```

## $HOROLOG

The **$HOROLOG** special variable contains an internal numeric representation of the current date and time.

### Format

$H{OROLOG}

### Explanation

The **$HOROLOG** special variable contains an internal numeric representation of the current date and time. This special variable contains two numeric values, separated by a comma. The first value is the number of days since December 31, 1840. The second value is the number of seconds since midnight.

### Comments

Keep the following points in mind when you use the **$HOROLOG** special variable:

- The first comma-piece contains the number of days since December 31, 1840. Remember that there is a leap year every four years. Also, every 100 years there is a centennial (no leap year).
- MODULO division and integer division are important tools to keep in mind when calculating the time based on the second comma-piece of **$HOROLOG.**

### Examples

The following example produces a string containing the approximate year, based on the first comma-piece of **$HOROLOG**.

```
SET T%Year=$HOROLOG\365.25+1841
```

The following extrinsic variable returns a string containing the approximate time, based on the second comma-piece of **$HOROLOG**. Note the use of **$SELECT**, **$JUSTIFY**, and **$TRANSLATE** in this function.

```
TIME() ; TIME extrinsic variable
    NEW L%Time,L%Hour,L%Minute,L%Meridian
    SET L%Time=$PIECE($HOROLOG,",",2)
    IF L%Time#43200=0 QUIT "12:00"_$SELECT(L%Time:"pm",1:"am")
    SET L%Hour=L%Time\3600
    SET L%Meridian=$SELECT(L%Hour>11:"pm",1:"am")
    SET L%Hour=$JUSTIFY(L%Hour#12,2)
    IF L%Hour=" 0" SET L%Hour=12
    SET L%Minute=$JUSTIFY(L%Time\60#60,2)
    SET L%Time=$TR(L%Hour_":"_L%Minute_L%Meridian," ",0)
    QUIT L%Time
```

## $INTERFACE

The **$INTERFACE** special variable contains the name of the current default interface which is normally Primary.

### Format

**$IN{TERFACE}**

### Explanation

Within the execution context of a method, the current interface is always available within the **$INTERFACE** special variable. Its scope is the currently executing method.

### Related

Message Syntax

### Example

The following line sets a temporary variable to the name of the interface the execution context.

```
Set T%Name=$INTERFACE
```

## $IO

The **$IO** special variable returns the device identifier of the current device.

### Format

$I{O}

### Explanation

The **$IO** special variable returns the device identifier of the current device.

### Comments

Keep the following points in mind when you use the **$IO** special variable:

- The value of **$IO** can change whenever the **USE** command is issued.
- The values of the **$DEVICE**, **$KEY**, **$X**, and **$Y** special variables are also sensitive to the current device.

### Related

USE command

$DEVICE special variable

$KEY special variable

$X special variable

$Y special variable

### Examples

The following example uses the truth-value of **$DEVICE** to determine whether an error has occurred on the current device. If so, it uses **$IO** to determine the device identifier, **$PIECE** to obtain the three components of **$DEVICE**, and displays them on the principal login device **$PRINCIPAL**.

```
IF $DEVICE DO  ; Display error text
. SET T%Device=$IO
. SET T%Code=$PIECE($DEVICE,",")
. SET T%=Status$PIECE($DEVICE,",",2)
. SET T%=Text$PIECE($DEVICE,",",3,999)
. USE $PRINCIPAL
. DO $ENV.Output("Error on device "_T%Device)
. DO $ENV.Output("Error code="_T%Code_", Status="_T%Status)
. DO $ENV.Output("Error text="_T%Text)
```

## $JOB

The **$JOB** special variable contains the job number of the current process.

### Format

$J{OB}

### Explanation

The job number of the current process is a positive integer that is unique to a single process. This number is useful for operations that may need to be executed simultaneously by more than one M process.

### Comments

It is common to use **$JOB** as an array subscript when it is necessary to maintain separate information for a variety of different processes. Because there is sometimes a possibility that **$JOB** values can be reused later by processes not related to the current process, it is useful to initialize process-related subtrees after startup.

### Related

JOB command

### Examples

The following example uses **$JOB** as an array subscript to prevent information from other process from coming into conflict with information from the current process.

```
SET ^GLO($JOB,T%Entry)=T%Value
```

The following example uses **$JOB** as an array subscript in initializing a subtree, to prevent information from former processes that shared the same job number from coming into conflict with information from the current process.

```
KILL ^GLO($JOB)
```

## $KEY

The **$KEY** special variable contains the control sequence that terminated the last **READ** operation on the current device.

### Format

$K{EY}

### Explanation

If the last **READ** was not terminated by a control sequence, **$KEY** is NULL (""). Examples of read operations that cannot terminate with a control sequence are fixed-width reads, ASCII code reads, and reads with a timeout.

### Comments

Keep the following points in mind when you use the **$KEY** special variable:

- In some forms of the **READ** command, information is lost whenever the user presses the return key or uses some other control string to terminate the **READ** operation. The **$KEY** special variable allows this information to be retained.
- The value of **$KEY** varies according to the current device. Therefore, it is likely to change whenever a **USE** command is executed. If you want to change devices before using the value in **$KEY** (such as for error reporting), record this value in a temporary variable before issuing the **USE** command.

### Related

USE command

$IO special variable

### Examples

The following example uses **$KEY** to determine whether or not the last **READ** was terminated by a control string. Note that the values of **$KEY** and **$IO** are recorded before the **USE** command is issued because the **USE** command changes the current device. **$ASCII** is used to convert control characters back into numeric codes.

```
IF $KEY'="" DO
. SET T%List="(no terminator)"
. FOR T%Loop=1:1:$LENGTH($KEY) SET
$PIECE(T%List,"+",T%Loop)=$ASCII($KEY,T%Loop))
. DO $ENV.Output("The last read on device "_$IO_" was terminated by:
"_T%List)
QUIT
```

## $LASTCHILDID

The **$LASTCHILD** special variable returns the internal number of the last child in the current objects child collection.

### Format

**$LASTCHILD**

### Explanation

The $LASTCHILDID is used to get the last child of a collection.

### Comments

This is used primarily as a debugging aid to determine orphan children of an object.

### Related

$CHILDREN special variable

$CHILDCNT special variable

$PEERS special variable

$SELF special variable

CREATE command

## Examples $LIBRARY

The **$LIBRARY** special variable contains an object reference to the current library.

### Format

$LIB{RARY}

### Explanation

This current library is the one used for naked lookups and is always used unless another library is specified explicitly. **$LIBRARY** is useful in sending messages to the current library.

**$LIBRARY** can be set to another library OID or name.  The scope of the set is the duration of the current method execution context.

### Comments

In many cases, the Main library is the default library. However, it is possible to change libraries from EsiObjects, which changes the value of **$LIBRARY**.

### Related

Message Syntax

$Library function

The following code sends a message to the library object OID bound to **$LIBRARY** to get an object reference to a class **Writer** if one is implemented in the current library. If not, the current code context is exited by **QUIT**.

```
SET T%WriterClass=$LIBRARY.FindClass(Name:"Writer")
QUIT:T%WriterClass=""
```

The following code sets the Special Variable **$LIBRARY** to the "TestLibrary" name.

```
SET $LIBRARY="TestLibrary"
```

## $LOCALOBJECTS

The $LOCALOBJECTS special variable returns a pointer to a collection that contains session specific objects stored as semi-persistent, that is, stored in globals that should be killed by system.

### Format

**$LOCALOBJECTS**

### Explanation

$LOCALOBJECTS is provides access to those objects that should be destroyed by the system.

### Comments

$LOCALOBJECTS is used for debugging purposes.

It is used by the EsiObjects Object Browser to expose local objects.

### Related

## $MAXNUM

The **$MAXNUM** special variable contains the highest numeric value that can be represented safely by the underlying M platform.

### Format

$MAXNUM

### Explanation

Attempts to handle numeric values greater than the highest numeric value supported on the underlying M platform are likely to result in an error.

### Comments

Generally the underlying platform represents the number contained in **$MAXNUM** using exponential notation. Any number of significant digits can therefore be lost in the process. Adding or subtracting 1 to **$MAXNUM** can simply return **$MAXNUM**.

### Related

$MINNUM special variable

### Examples

The following example creates a numeric integer T%Middle that is directly between 0 and **$MAXNUM**.

```
SET T%Middle=$MAXNUM/2
```

## $MEMORYOBJECTS

The $MEMORYOBJECTS special variable produces an object pointer that provides a root context for all top-level objects stored within local memory for a session.

### Format

$MEMORYOBJECTS

### Explanation

### Comments

These objects are created when the share=0,child=0 keywords are used on the CREATE command.

The objects reside in the local store.

Used by the EsiObjects Object Browser to display the objects.

### Related

CREATE command

## $MAXSTR

The **$MAXSTR** special variable returns the maximum length of a string that can be stored at a global node for the underlying M platform currently connected to.

### Format

$MAXSTR

### Explanation

Attempts to create strings that exceed the value of $MAXSTR will result in an error.

### Examples

The following example checks to see if a string created by concatenating the contents of two variables exceed the maximu allowed length.

```
IF $Length(T%Part1)+$Length(T%Part2)>$MAXSTR
```

## $MESSAGE

The **$MESSAGE** special variable contains a string naming the current method that was specified when the message was sent.

### Format

$MESSAGE

### Explanation

The **$MESSAGE** special variable contains a string naming the current method, as specified when the message was sent. This is the same as the name of the current method unless an alias was used or interception has occurred. This special variable can be useful in the context of intercepting code, code referenced by an alias, or code executed with the **XECUTE** command that needs to behave differently for different methods.

### Comments

The value of **$MESSAGE** is not guaranteed to be the same as the name of the current method. For example, it is possible for a method to be known by an alias.

### Examples

The following example sends the FindAll message to **$SELF** if the current method was referenced as Find by the caller.

```
IF $MESSAGE="Find" DO $SELF.FindAll QUIT
```

## $MINNUM

The **$MINNUM** special variable contains the highest numeric value that can be represented safely by the underlying M platform.

### Format

$MINNUM

### Explanation

The **$MINNUM** special variable contains the lowest numeric value that can be represented by the underlying M platform without equaling 0. Attempts to handle numeric values between this amount and 0 are likely to be rounded down to 0.

### Comments

Generally the underlying platform represents the number contained in **$MINNUM** using exponential notation. Therefore, any number of significant digits can be lost in the process. Multiplying **$MINNUM** by a positive number between 1 and 2 can simply return **$MAXNUM**.

### Related

$MAXNUM special variable

### Examples

The following example creates a numeric integer T%Middle that is equal to the product of **$MAXNUM** and **$MINNUM**.

```
SET T%Middle=$MINNUM*$MAXNUM
```

## $PARAMETERS

The **$PARAMETERS** special variable contains the compiled parameter list for the current method call.

### Format

$PARAM{ETERS}

### Explanation

The compiled parameter list cannot be displayed because it might contain control characters. The **$PARAMETERS** special variable is primarily useful in parameter list indirection.

### Related

Indirection

$PARAMETERLIST special variable

### Examples

In the following example, the parameter list sent to this method is sent along on a call to the Revoke method and represents delegation.

```
GOTO T%Object22.Revoke@$PARAMETERS
```

## $PARAMETERLIST

The **$PARAMETERLIST** special variable returns a string containing the parameter list as specified when the current method was called.

### Format

$PARAMETERLIST

$PARALIS

$PRMLIS

### Explanation

In contrast with the string returned in the **$PARAMETERS** special variable, this string is returned in programmer-readable form.

### Comments

This variable contains a programmer-readable string copy of the parameter list used to invoke the current method. It is useful in testing and debugging only and is not to be used for parameter list indirection.

### Related

$PARAMETERS special variable

### Examples

The following example displays the parameter list used to invoke the current method.

```
  DO Assert.$ENVIRONMENT("Param. list: "_$PARAMETERLIST)
```

## $PEERS

The **$PEERS** special variable produces a pointer to the collection of peer (sibling) objects of the current objects execution context.

### Format

**$PEERS**

### Explanation

The $PEERS collection contains all the peers or siblings to the current objects context the special variable is executed within.

### Comments

The $PEERS variable is used to access the peers objects.

It is used by the EsiObjects Object Browser to display all peers of the current object context.

### Related

$CHILDREN special variable

$CHILDCNT special variable

$LASTCHILDID special variable

$SELF special variable

CREATE command

### Examples

## $POINTER

The **$POINTER** special variable contains a pointer to the location of the current object.

### Format

$POINTER

### Explanation

The **$POINTER** special variable returns a namevalue appropriate to use in variable name indirection or to use with the **$QLENGTH** and **$QSUBSCRIPT** functions. Special privileges are required to compile code that contains this special variable.

### Comments

Keep the following points in mind when using the **$POINTER** special variable.

- Directly accessing the global structure of an object is a serious violation of its encapsulation.
- You must have the proper privileges to use the **$POINTER** special variable.
- This special variable is not recommended for general use in EsiObjects.

### Related

$NAME function

$OIDPTR function

$PTROID function

$QLENGTH function

$QSUBSCRIPT function

### Examples

The following code uses **$POINTER** and the **LOCK** command to lock an object.

```
LOCK +@$POINTER
```

## $POOL

The **$POOL** special variable contains the name of the default name pool.

### Format

$POOL

### Explanation

The value contained in **$POOL** is used in a NamePool reference where no name pool is explicitly defined. The construct **SET $POOL** is used to change the default NamePool.

### Comments

A name pool reference that does not explicitly specify the pool is identical to one that specified **$POOL**. One purpose of **$POOL** is to compare it to some known pool to see if that pool is the default.

### Related

$SYSPOOL special variable

### Examples

The following example references a value ESI$MainDirectory in the default name pool **$POOL**.

```
SET T%File=N%($POOL)ESI$MainDirectory_"OUTPUT.TXT"
```

## $PRINCIPAL

The **$PRINCIPAL** special variable returns the device identifier of the principal (or login) device.

### Format

$P{RINCIPAL}

### Explanation

If the current process is a background process not tied to any device, **$PRINCIPAL** returns NULL ("").

### Comments

The special device identifier (0) can also be used to refer to the principal device. However, **$PRINCIPAL** is more informative.

### Related

$DEVICE special variable

$IO special variable

### Examples

The following example uses the truth-value of **$DEVICE** to determine whether an error has occurred on the current device. If so, it uses **$IO** to determine the device identifier, uses **$PIECE** to get the three components of **$DEVICE**, and displays these values on the principal login device **$PRINCIPAL**.

```
IF $DEVICE DO  ; Display error text
. SET T%Code=$PIECE($DEVICE,",")
. SET T%=Status$PIECE($DEVICE,",",2)
. SET T%=Text$PIECE($DEVICE,",",3,999)
. DO $ENV.Output("Error on device "_$IO)
. DO $ENV.Output("Error code="_T%Code_", Status="_T%Status)
. DO $ENV.Output("Error text="_T%Text)
```

## $PRIVILEGED

The **$PRIVILEGED** special variable returns 1 if the current message is privileged.

### Format

$PRIV{ILEGED}

### Explanation

Compilation of the method determines this value. It can be changed locally by requesting privileges from the environment. If privileges are granted, the value of **$PRIVILEGED** is 1 (true). Privileged methods can use various lower-level internal functions.

### Comments

Keep the following points in mind when you use the **$PRIVILEGED** special variable:

- Privileges are not required to use this special variable.
- Privileged methods are allowed to perform certain lower-level operations that are unavailable to nonprivileged functions. However, many of these operations are not recommended for general used by EsiObjects programmers.

### Examples

The following example executes a subroutine and exits if the current method does not have any privileges.

```
IF '$PRIVILEGED DO NoPriv QUIT
```

## $QUIT

The **$QUIT** special variable returns 1 if the current code context is a subroutine, or 0 if the current code context is an extrinsic function.

### Format

$Q{UIT}

### Explanation

The **$QUIT** special variable is useful for testing and debugging, or for creating a body of code that can be accessed either as an extrinsic function or as a return value.

### Comments

Keep the following points in mind when you use the **$QUIT** special variable:

- In general, it is not recommended to create a single body of code that can be called either as a subroutine or as an extrinsic function. Create two bodies of code that share their common functionality in one or more common subroutines.
- In some programming contexts such as error trapping, it may be necessary to use **$QUIT.**

### Related

QUIT command

### Examples

The following example uses **$QUIT** to exit with a return value if the context is an extrinsic function, or to exit without a return value if the context is an extrinsic function.

```
IF $QUIT QUIT T%ReturnValue
ELSE  QUIT
```

## $REFERENCE

The **$REFERENCE** special variable exposes the internal reference count on an object.

### Format

$REFERENCE

### Explanation

Occasionally an object may collaborate with one or more other objects, providing a service for as long as it is needed. Each user of the service may try to destroy the service object after it has finished with it. Under these circumstances, if an object succeeded in destroying the object, subsequent objects making reference to it would fail if they did not constantly check for its existence. To make sure that an object remains alive until the last destroy action is applied; EsiObjects implements an internal reference counter. When the object is created by the **CREATE** command, the reference count is initialized to one (1). When a **DESTROY** command is applied, it will decrement the counter. When the count goes below 1, the object will be destroyed. To make sure the object stays around, each using object would apply the **PRESERVE** command to the object. This command increments the count by one. Once it has finished with the object, it would apply the **DESTROY** command which will decrement the count by one. This preserves the object so that it is available to all other objects and makes it available to the creating object for destruction.

### Comments

Keep the following points in mind when you use the **$REFERENCE** special variable:

- The value of **$REFERENCE** should always be greater and or equal to one.
- Within an objects method or property, the **$REFERENCE** can be set to another value.

### Related

CREATE command

DESTROY command

PRESERVE command

### Examples

The following example show how the **$REFERENCE** can be used.

```
Do $Env.Assert(“Object “_T%Object.Name_” has “_$Reference_” references to
it.”)
```

This example show how the **$REFERENCE** can be set

```
Set $Reference=$Reference+1
```

## $RETURN

The **$RETURN** special variable contains the value that the current method returns on exit.

### Format

$RET{URN}

### Explanation

Initially when a method is called, the value of **$RETURN** is generally set to NULL (""). The **SET $RETURN** construct is used to change the return value from its default value. Its value is scoped inside the current method call.

### Comments

Keep the following points in mind when you use the **$RETURN** special variable:

- The value of **$RETURN** always defaults to NULL (""), except in an object's special Destroy method, when it defaults to 1.
- In a value-returning method, the construct **SET $RETURN** is used to specify a different return value.

### Related

Method structure

DESTROY command

Message Syntax

### Examples

The following example sets the value of **$RETURN** to 1. The current method returns 1 unless the value of **$RETURN** is changed inside this method before its final **QUIT** occurs.

```
SET $RETURN=1
```

## $ROOTOBJECTS

The **$ROOTOBJECTS** special variable points to a collection of objects that are outside the context of the session.

### Format

**$ROOTOBJECTS**

### Explanation

The $ROOTOBJECTS collection contains objects that are created outside of the sessions context.

### Comments

Most root objects are protected and to see the objects you must have the proper privileges. Privileges are determined by the EsiObjects startup command qualifier used (/ESI, /ADMIN or /DEBUG) or the Security levels assigned to you logon username.

### Related

CREATE command

### Examples

## $SELF

The **$SELF** special variable returns an object reference to the current object.

### Format

$SELF

### Explanation

Primary use of the **$SELF** special variable is for an object to send a message to itself or to refer to its own property. It can also give other objects a handle to themselves, which can be used in future messaging dialog.

### Comments

Keep the following points in mind when you use the **$SELF** special variable:

- It is possible for an object to destroy itself using the **DESTROY** command. However, any subsequent references to **$SELF** or instance variables before the method exits with **QUIT** causes an error.
- The syntax **DO $SUPER.Label** is equivalent to **DO $SELF.*Label**.

### Related

Message Syntax

$SUPER special variable

### Examples

The following example creates a new object of class Button whose Owner equals **$SELF**.

```
CREATE T%Button=Button(Text:"OK",Owner:$SELF)
```

## $SHAREDOBJECTS

The $SHAREDOBJECTS special variable that points to a collection of all shared objects.

### Format

**$SHAREDOBJECTS**

### Explanation

Shared objects that are created with the CREATE command keyword Share=1 (as opposed to Base, Fixed or Domain).

### Comments

Shared objects reside in the ^VESoshob global.

The EsiObjects Object Browser uses this variable to access the shared objects.

### Related

CREATE command

### Examples

## $STACK

The **$STACK** special variable returns the number of stack frames currently on the M process stack.

### Format

$ST{ACK}

### Explanation

The **$STACK** special variable always contains an integer value of 0 or greater.

### Comments

Keep the following points in mind when you use the **$STACK** special variable:

- It is equivalent to the function call **$STACK(-1)**.
- **$STACK** function calls return information about the current M process stack level if the **$STACK** special variable is the first argument of **$STACK**.

### Related

$STACK function

### Examples

The following example displays information about the current error condition for every stack frame in **$STACK** that contains error codes. Note the use of the **$STACK** special variable on the third line to determine the total number of stack frames.

```
DO $ENV.Output("Process Type: "_$STACK(0))
DO $ENV.Output("Frames on Stack: "_$STACK)
FOR T%Loop=1:1:$STACK IF $STACK(T%Loop,"ECODE")'="" DO
. DO $ENV.Output("")
. SET T%Code=$STACK(T%Loop,"ECODE")
. SET T%Line=$STACK(T%Loop,"PLACE")
. SET T%Text=$STACK(T%Loop,"MCODE")
. DO $ENV.Output("     Errors at Frame "_T%Loop_": "_T%Code)
. DO $ENV.Output("          Execution Location: "_T%Line)
. IF T%Text'="" DO $ENV.Output("          "_T%Text)
QUIT
```

## $STORAGE

The **$STORAGE** special variable contains the number of free characters available for use in the partition of the current process.

### Format

$S{TORAGE}

### Explanation

The method of calculating this value depends on the underlying M platform. Its behavior is even less clear in EsiObjects because some transient objects are stored in the partition. However, many transient objects are not stored in the partition.

### Comments

Keep the following points in mind when you use the **$STORAGE** special variable:

- Its behavioral characteristics vary depending on the underlying M platform.
- EsiObjects stores some nonpersistent objects globally and stores others in the partition.

### Related

KILL command

SET command

### Examples

The following example calls the LOADARR subroutine if the value of **$STORAGE** is greater than 10240.

```
IF $STORAGE>10240 DO LOADARR
```

## $SUPER

The **$SUPER** special variable contains an object reference to the superclass method and property implementations of the current object.

### Format

$SUPER

### Explanation

The **$SUPER** special variable is used generally as a messaging target to inherit the superclass implementation of the current method or public label.

### Comments

The syntax DO $SUPER.Method is equivalent to DO $SELF.*Method.

### Related

Message Syntax

$SELF special variable

Labels in EsiObjects

Label Inheritance

### Examples

The following example delegates to the method ThisMethod as implemented by the superclass.

```
GOTO $SUPER.ThisMethod
```

The following example delegates to the public label ThisLab as implemented by the superclass implementation of the current method.

```
GOTO *ThisLab^$SUPER
```

## $SYSPOOL

The **$SYSPOOL** special variable contains an object reference to the system name pool.

### Format

$SYSPOOL

### Explanation

The **$SYSPOOL** special variable is used in name pool references to names in the system name pool where various system objects are shared.

### Comments

The **$SYSPOOL** special variable returns the system name pool and the **$POOL** special variable returns the current default name pool. The two are not always identical.

### Related

$POOL special variable

### Examples

The following example references a value ESI$MainDirectory in the system name pool **$SYSPOOL**.

```
SET T%File=N%($SYSPOOL)ESI$MainDirectory_"OUTPUT.TXT"
```

## $SYSTEM

The **$SYSTEM** special variable returns a value that uniquely represents the system.

### Format

$SY{STEM}

### Explanation

The **$SYSTEM** special variable returns a value that uniquely reprents the underlying M system, which represents the domain of concurrent processes for which **$JOB** is unique.

### Comments

**$SYSTEM** is of the form

V,S

where V is a vendor Id (**48** = ESI Technology Corp), and S is a globally unique system ID.

### Related

$JOB special variable

## $TEST

The **$TEST** special variable returns the value of the test flag.

### Format

$T{EST}

### Explanation

The **$TEST** special variable returns the value of the test flag. The test flag is set by an IF command, a timeout, or by the **DESTROY** command.

### Comments

Keep the following points in mind when you use the **$TEST** special variable:

- The argumentless **DO** command places **$TEST** on the process stack before invoking a block, causing its value to be restored when the block is exited.
- The **IF** and **DESTROY** commands affect the **$TEST** special variable, and any time a timeout is encountered in the **JOB**, **LOCK**, **OPEN**, or **READ** commands. Other conditional operations, such as postconditionals and **$SELECT**, do not affect **$TEST**.
- **$TEST** is scoped inside a method context. Calls to other objects or methods never modify **$TEST**. However, subroutine calls and the **XECUTE** command can modify **$TEST**.
- Any form of **DO** that specifies a label and/or routine name does not place **$TEST** on the process stack. When execution returns from the subroutine, any changes to **$TEST** are still in effect.

```
IF I%Height'>I%Width DO MODIFY
ELSE  DO $ENV.Output("Greater")
```

The previous example is extremely risky, and hard to evaluate. Without looking at the subroutine MODIFY, it is impossible to determine under what circumstances the **ELSE** command on the second line will be executed. Note the following:

- I%Height is greater than I%Width. The **IF** on the first line sets **$TEST** to 0, and execution drops down to the second line. Because **$TEST** is 0, the **ELSE** executes the **$ENV.Output**.
- I%Height is not greater than I%Width. The **IF** on the first line sets **$TEST** to 1 and executes the **DO**. The following can occur inside the subroutine:
- The subroutine MODIFY does not modify **$TEST**. When execution returns, **$TEST** still equals 1 from the IF on the first line, and the **ELSE** does nothing.
- The subroutine MODIFY does modify **$TEST**, and when it exits **$TEST** equals 1. The **ELSE** on the second line does nothing, based on the most recent **$TEST** operation.

- The subroutine MODIFY does modify **$TEST**, and when it exits **$TEST** equals 0. The **ELSE** on the second line executes the **$ENV.Output**, based on the most recent **$TEST** operation.

Clearly this situation can lead to unexpected results. The examples section presents a specific solution to this problem using an argumentless **DO.**

## Related

DESTROY command

DO command

ELSE command

IF command

JOB command

LOCK command

OPEN command

READ command

$SELECT function

## Examples

The following example illustrates a typical programming error because **$TEST** is likely to change between the **IF** and the **ELSE**. Note that in some cases **DO** does not stack **$TEST**.

```
    IF I%Height'>I%Width DO TEST
    ELSE  DO $ENV.Output("Greater")
    . . .
    QUIT
    ;
TEST       ; Subroutine containing IF and ELSE
    IF I%Height=I%Width DO $ENV.Output("Equal")
    ELSE  DO $ENV.Output("Not Greater")
    QUIT
```

In the previous example, assume that I%Height=5 and I%Width=10, the IF command on the first line sets **$TEST** to 1 and the **DO** calls TEST. Inside TEST, the **IF** sets **$TEST** to 0, and the **ELSE** executes the **$ENV.Output**. The **QUIT** then exits TEST. The **ELSE** on the second line checks **$TEST** (which is now 0) and executes the **$ENV.Output**. The first line of output is Not Greater and the second line is Greater. This is probably not what the programmer intended.

A number of language elements (for example, object-with-service references, extrinsic functions, and the argumentless **DO**) place **$TEST** on the process stack. The following example solves the problem shown in the previous example with the argumentless **DO**:

```
IF I%Height'>I%Width DO
. IF I%Height=I%Width DO $ENV.Output("Equal") QUIT
. DO $ENV.Output("Not Greater")
ELSE  DO $ENV.Output("Greater")
QUIT
```

The following example uses **$SELECT** and is functionally equivalent to the previous example, except that it does not modify **$TEST**.

```
DO
$ENV.Output($SELECT(I%Height>I%Width:"Greater",I%Height=I%Width:"Equal",1:"No
t Greater"))
```

The following example uses the **DESTROY** command to destroy the Window object referenced by the symbol T%Window, causing the window to disappear from the display and all of its instance variables to be removed. The **ELSE** command references **$TEST** to determine whether the attempt was successful.

```
DESTROY T%Window
ELSE  DO $ENV.Assert("DESTROY Failed!")
```

The **IF** command with no arguments is the opposite of **ELSE** because it lets execution pass to the rest of the commands on the line only if **$TEST** is 1. This form is most commonly used after language elements (other than **IF**) that modify **$TEST** (for example, timeouts or **DESTROY**).

```
DESTROY T%Object12
IF  DO $ENV.Output("Object was destroyed.") QUIT
```

In the previous example, the **$ENV.Output** and **QUIT** commands are performed only if **DESTROY** set **$TEST** to 1 (in other words, the object was successfully destroyed).

The following example illustrates the repeated use of incremental **LOCK** with a timeout to provide feedback to the user that an attempt to lock the node is in progress. If the lock does not complete normally within 30 seconds, the process is abandoned.

```
DO $ENV.Output("Locking...")
FOR T%Loop=1:1:10 LOCK +^XYZ(0):3 IF  QUIT
ELSE  DO $ENV.Output("Node is busy. Aborting.") QUIT
SET (T%EntryNumber,^XYZ(0))=^XYZ(0)+1
LOCK -I%List(0)
SET ^XYZ(T%EntryNumber)=T%EntryValue
```

## $X

The **$X** special variable returns the output column position of the current device.

### Format

$X

### Explanation

The **$X** special variable returns the output column position of the current device. Control sequences that affect the output position cannot accurately update **$X**. However, you can use the construct **SET $X** to correct such problems.

### Comments

Keep the following points in mind when you use the **$X** special variable:

- Control sequences that affect the output position cannot accurately update **$X**. Therefore, it is advisable to use caution when interpreting the contents of this special variable. However, you can use the construct **SET $X** to correct such problems.
- **$X** is sensitive to the current device. Therefore, its value is likely to change whenever the **USE** command is issued.

### Related

USE command

$Y special variable

### Examples

The following example sets the value of **$X** to reflect the current column position and $Y to reflect the current row position.

```
SET $X=T%Column,$Y=T%Row
```

## $Y

The **$Y** special variable returns the output row position of the current device.

### Format

$Y

### Explanation

The **$Y** special variable returns the output row position of the current device. Control sequences that affect the output position cannot accurately update **$Y**. However, you can use the construct **SET $Y** to correct such problems.

### Comments

Keep the following points in mind when you use the **$Y** special variable:

- Control sequences that affect the output position cannot accurately update **$Y**. Therefore, it is advisable to use caution when interpreting the contents of this special variable. However, you can use the construct **SET $Y** to correct such problems.
- **$Y** is sensitive to the current device. Therefore, its value is likely to change whenever the **USE** command is issued.

### Related

USE command

$X special variable

### Examples

The following example sets the value of **$X** to reflect the current column position and **$Y** to reflect the current row position.

```
SET $X=T%Column,$Y=T%Row
```

## $ZVIRDATA

The **$ZVIRDATA** special variable contains the substantive data represented by a virtual object, which must consist of a single string value.

### Format

$ZVIRDATA

### Explanation

The **$ZVIRDATA** special variable is used to establish a basic object context. The CREATE method can set **$ZVIRDATA** to establish specific context information. **$ZVIRDATA** cannot be set outside the context of the **CREATE** command.

### Comments

Although it is possible to send messages to virtual objects, they are not really objects as such and do not have instance variables.  They are exceptionally lightweight, however, and useful for representing external data as objects.

### Related

CREATE command

DESTROY command

### Examples

The following example sets the object's data portion to 1.

```
SET $ZVIRDATA=1
```

## Functions

## $ASCII

The **$ASCII** function returns the ASCII code number of a single character inside a string.

### Format

$A{SCII} ( expr {,intexpr} )

### Arguments

expr - a string expression containing the character whose ASCII code is to be returned.

intexpr - an integer defining the position of the target character in the string.

### Explanation

The **$ASCII** function returns the ASCII code value of a single character inside a string. If no character position is specified, the ASCII code of the first character in the string is returned. The character position is always interpreted as an integer. Decimal values are truncated.

By convention in EsiObjects, the special value –1 is used as the ASCII code of an empty string. If a position beyond the string's length or a number less than 1 is specified for the character position, then –1 is also returned.

### Comments

Keep the following points in mind when you use the $ASCII function:

- **$ASCII** is the opposite of **$CHAR**. For any integer T%Integer in the range –1 to 255, the following expression should always return the value of T%Integer:

```
$ASCII($CHAR(T%Integer))
```

- **$ASCII** is related to **$EXTRACT**. Note the following:

```
$ASCII(T%String,T%Integer)
```

- The previous example is functionally equivalent to the following example:

```
$ASCII($EXTRACT(T%String,T%Integer))
```

- **$ASCII** is often useful with the **$KEY** special variable because it contains control characters that may need to be converted back to their ASCII code values.

### Related

$CHAR function

$EXTRACT function

$KEY special variable

## Examples

The following example displays the code of the first character in the string T%String to the Output window:

```
SET T%String="EsiObjects"
DO $Env.Output($ASCII(T%String))


Results: 69
```

The following example displays the code of the fifth character in the string T%String to the Output window:

```
SET T%String="EsiObjects"
DO $Env.Output($ASCII(T%String,5))


Results: 98
```

The following example displays the code of the 12th character in T%String to the Output window, but because there are only 11 characters the return value is –1:

```
SET T%String="EsiObjects"
DO $Env.Output($ASCII(T%String,12))


Results: -1
```

The following example uses **$ASCII**, **$LENGTH**, and **SET $PIECE** to create an indirectible string to generate the contents of T%String:

```
SET T%Result
FOR T%Loop=1:1:$LENGTH(T%String) DO
. SET $PIECE(T%Result,T%Loop)=$ASCII(T%String,T%Loop)
SET T%Result="$CHAR("_T%Result_")"
```

***Note: The following comments and code examples assume IO to traditional M Input/Output devices. It is assumed that a command window is present for this exercise since traditional M I/O commands are being used.***

The following example uses $KEY to determine whether or not the last READ was terminated by a control string. Note that the values of $KEY and $IO are recorded before the USE command is issued because this command changes the current device. $ASCII is used to convert control characters back into numeric codes.

```
IF $KEY'="" DO
. SET T%ControlString=$KEY
. SET T%Device=$IO
. USE $PRINCIPAL
. WRITE "The last read on ",T%Device," was terminated by ASCII"
. FOR T%Loop=1:1:$LENGTH(T%ControlString) DO
. . IF T%Loop>1 WRITE "+"
. . WRITE $ASCII(T%ControlString,T%Loop)

. USE T%Device
QUIT
```

## $ASNVECTOR

The **$ASNVECTOR** function returns the assignment vector of a variable.

### Format

$ASN{VECTOR} ( namexpr {,subexpr {,typexpr}} )

### Arguments

namexpr ::= expr V name

The first argument is a string containing the name of the variable whose assignment vector is to be returned.

subexpr ::= expr V subscriptlist

The second argument is either the null string ("") or a string containing a complete subscript list, including the surrounding parentheses.

| | | |
|---|---|---|
| | **A** | Accessor |
| | **C** | Class |
| | **CN** | Constant |
| | **G** | Global |
| | **I** | Instance |
| **typexpr ::= expr V** | **L** | Local |
| | **N** | NamePool |
| | **O** | Object Name |
| | **P** | Parameter |
| | **R** | Relative/Region |
| | **S** | System |
| | **U** | Universal |

The third argument is a special code indicating the type of the variable whose assignment vector is to be returned.

### Explanation

***Note: The $ASNVECTOR function is a privileged function and is not recommended for general use.***

The assignment vector is an indirectible string that can be used to directly access a symbol. The behavior of lookup and assignment operations performed on assignment vectors vary according to object internals. Special privileges are required to compile code that uses **$ASNVECTOR**.

## Comments

Keep the following points in mind when you use the **$ASNVECTOR** function:

- Privileges are required to compile EsiObjects code that contains **$ASNVECTOR**.
- Many operations on assignment vectors yield different results than the same operations performed on the actual symbols. For example, a **SET** command performed on an instance variable can invoke the target object's Assign accessor, while a **SET** command performed with name indirection on that instance variable's assignment vector does not.

## Related

$LOOKUP function

$OIDPTR function

$PTROID function

$WALK function

## Examples

The following example gets the assignments vector to the instance variable I%Height and sets its value to 0 using indirection. The preferred method for doing this is to instead set the instance variable's value directly.

```
SET T%Handle=$ASNVECTOR("Height","","I")
SET @T%Handle=0
```

## $ASSOCIATE

The **$ASSOCIATE** function establishes the current object context.

### Format

$AS{SSOCIATE} ( oref )

### Arguments

oref - an object reference to the object to be tested for class membership.

### Explanation

The **$ASSOCIATE** function is used to associate an object context when none exists. This allows access to the instance variables of the object.

You can use the **$ASSOCIATE** function with the **$GETENTRYREF** function to allow nonobject code to invoke object methods.

### Comments

Keep the following points in mind when you use the **$ASSOCIATE** function:

- Can only be used when no context exists.
- Must be used in a method of the target object.

### Related

Method structure

Message Syntax

$GETENTRYREF function

### Examples

The following example shows a callback function that translates an ID number to an object associated to that object and fires a click event.

```
CLICK(ID) ;
    SET T%Object.A=^Reg(ID)
    IF '$ASSOCIATE(ID) DO $Env.Output("Error")
    Event CLICK
```

## $CALLBACK

The **$CALLBACK** function returns a callback frame identifier used to call back to a label within the current method and object.

### Format

$CALL{BACK} ( label {,typecode {,optionscode}} )

### Arguments

label - a label or public label as it would be referenced from within the current code context.

typecode - a numeric code value interpreted as follows:

| Type | Description | Code |
|---|---|---|
| Original | Callback to creator's stack frame. | 0 |
| Capture | Callback capturing creator's method-related symbols. | 1 |
| Initialized | Callback that starts with a clean variable context. | 2 |

optionscode - an integer interpreted according to bit value

| Type | Description | Bit | Types |
|---|---|---|---|
| Persistent | Survive for the duration of the creating object. | 0 | 1,2 |
| Additive | Preserve variable state between calls. | 1 | 1,2 |

### Explanation

Objects can create callbacks to specific labels inside the current method. This allows external objects to directly invoke a specific label in a specific method. The code being invoked runs in the context of the object that created the object. Callbacks can be invoked as **DO** or **GOTO** arguments or as extrinsic functions.

For more information about callbacks, see Callback Syntax section of this guide.

### Comments

Keep the following points in mind when you use the **$CALLBACK** function:

- The callback frame identifier can be used to call externally into the specified context.
- Use the **DO** form of a callback for output, looping, and update functions.
- Use the **GOTO** form for delegation and error handling.
- Use the extrinsic function form of a callback for searches and property lookup.

### Related

Message Syntax

$EXTCALLBACK function

$FREECB function

### Examples

The following example returns a callback to MODIFY.

```
SET T%CallBack=$CALLBACK(MODIFY)
```

The following example returns a callback to DELETE. The callback starts with a clean variable context and is persistent and additive.

```
SET T%CallBack=$CALLBACK(DELETE,2,3)
```

## $CHAR

The **$CHAR** function returns a string containing the characters specified by its arguments.

### Format

$C{HAR} ( L code )

### Arguments

code

An expression interpreted as an integer whose value ranges between –1 and 255, used to specify the value of a single character in the string.

### Explanation

The **$CHAR** function returns a string containing the characters specified by a series of numeric ASCII codes. The value –1 represents the null string (""), which contains no characters.

### Comments

Keep the following points in mind when you use the **$CHAR** function:

- It is not legal to store control characters directly in the text of EsiObjects code. In many cases, you can use the **$CHAR** function to overcome this limitation.
- **$CHAR** is the opposite of **$ASCII**. If the string in T%Char is 0 or 1 characters, the following expression always returns the value of T%Char:

```
$CHAR($ASCII(T%Char))
```

***Note: The following comments and code examples assume IO to traditional M Input/Output devices.***

- The syntax **WRITE** *code is sometimes an alternative to using **$CHAR**. The following two lines of code are functionally equivalent:

```
WRITE $CHAR(13,10)
WRITE *13,*10
```

- In some cases, using the **WRITE** command to send control strings with **$CHAR** can result in changes to the current output position that can render the values of **$X** and **$Y** inaccurate. Control mnemonics and format control parameters do not have this limitation. For example, the following **WRITE** command is equivalent to the two previous examples except for its effect on **$X** and **$Y**.

```
WRITE !
```

- The **READ *** variable form of the **READ** command produces an ASCII code number that can be converted back into an ASCII character with **$CHAR**.

### Related

WRITE command

$ASCII function

$X special variable

$Y special variable

### Examples

The following example generates the string ABC by specifying the ASCII codes 65, 66, and 67.

```
DO $Env.Output($CHAR(65,66,67))
```

```
Result: ABC
```

The following example sets the value of the service variable T%Code to the ASCII Tab character (ASCII code 9).

```
SET T%Code=$CHAR(9)
```

***Note: The following comments and code examples assume IO to traditional M Input/Output devices.***

The following example accepts input from the current device as an ASCII code and converts it to a character value. Both values are used when calling the subroutine HANDLE.

```
READ *T%Code
SET T%Char=$CHAR(T%Code)
DO HANDLE(T%Code,T%Char)
```

## $CLASSOID

The **$CLASSOID** function returns the full object reference of a class, given its name as input.

### Format

$CLASSOID(expr)

### Arguments

expr ::= expr V FullClassName – An expression whose value is the full library$class name.

### Explanation

This function is useful whenever it is necessary to transform a string containing the name of a class into an object reference. The function will always return a valid OID even if the class does not exist. The return value is not validated. It is up to the programmer to validate the existence of the class.

### Related

$OIDPTR

$PTROID

### Examples

The following chained example first transforms a full class reference into the class object reference. The OID is then used to get the classes parent OID. The parent OID is used to retrieve the parent classes name.

```
SET T%Parent=$ClassOID("Base$Set").Parent.Name
```

## $COPY

The **$COPY** function copies the current objects instance table of the current object into the destination array specified by its argument.

### Format

$COPY ( glvn )

### Arguments

glvn - the array into which the current object's instance table is to be copied.

### Explanation

***Note: Because special privileges are required to compile code containing the $COPY function, it is not recommended for general use.***

The **$COPY** function initializes the destination array prior to copying. Sparse lookups are not performed as part of this process.

Because **$COPY** initializes the destination array prior to copying the object's instance table, its effect can be viewed as a combination of the **KILL** and **MERGE** commands (except that the Kill accessor and Assign accessor methods are never invoked).

The **$COPY** function returns 1 if the copying operation was successful, 0 if it was not.

### Comments

Keep the following points in mind when you use the **$COPY** function:

- Copying an object's instance table into an array is risky, and the results are not guaranteed to be consistent. For example, object references to internal objects cannot be updated to point into the destination array.
- Sparse lookups are not performed by **$COPY**. This means that instance variables that theoretically exist but have not been specifically created in the object's instance table cannot be found in the destination array following **$COPY**.

### Related

KILL command

MERGE command

### Examples

The following example copies the current object's instance table into T%Result, exiting from the current context if the operation was not successful.

```
IF '$COPY(T%Result) QUIT
```

## $DATA

The **$DATA** function checks the structural existence of a variable name or array node.

### Format

$D{ATA} ( glvn )

### Arguments

glvn - a variable name or array node whose structural existence is to be checked.

### Explanation

The **$DATA** function checks the structural existence of a variable name or array node. It answers two questions at once:

- Does the symbol contain a value (is it safe to reference the symbol without **$GET**)?
- Does the symbol have array children below it?

The argument of **$DATA** is a variable name or array node. It always produces one of four return values:

| | |
|---|---|
| 0 | The symbol is undefined. It has no value and no array descendants. |
| 1 | The symbol contains a value, but has no array descendants. |
| 10 | The symbol contains no value, but has array descendants. |
| 11 | The symbol contains a value, and has array descendants. |

## Comments

Keep the following points in mind when you use the **$DATA** function:

- The **$DATA** function determines the structural existence of a symbol or array node. It does not determine whether a variable contains a handle to an object that currently exists. However, the **$EXIST** function does have this capability.
- Often the **$DATA** function is used to ask a more specific question about the structural existence of an object. Using the example variable L%X, the following questions can be asked in the following different ways:

| **General Question** | **Examples** |
|---|---|
| Is L%X defined in any way? | IF $DATA(L%X) |
| Is L%X completely undefined? | IF '$DATA(L%X) |
| Does L%X contain no value? | IF $DATA(L%X)#10=0<br>IF $DATA(L%X)[0 |
| Does L%X contain a value? | IF $DATA(L%X)#10<br>IF $DATA(L%X)'[0<br>IF 11[$DATA(L%X) |
| Does L%X Have Array Descendants? | IF $DATA(L%X)>9<br>IF $LENGTH($DATA(L%X))=2 |
| Does L%X not have descendants? | IF $DATA(L%X)<10<br>IF $LENGTH($DATA(L%X))=1 |

- Use the **$DATA** and **$GET** functions to interact with symbols that may or may not contain a value. In some cases, it is better to use **$GET** instead of **$DATA**, while in other cases **$DATA** is preferable.

## Related

CONTAINS ( [ ) operator

EQUALS ( = ) operator

GREATER THAN ( > ) operator

LESS THAN ( < ) operator

MODULO ( # ) division operator

NOT ( ' ) operator

$EXIST function

$GET function

## Examples

The following example assigns the value 100 to the variable T%Size if the instance variable I%Height contains no value, or assigns the value of I%Height to T%Size if it does contain a value.

```
IF $DATA(I%Height)[0 SET T%Size=100
ELSE  SET T%Size=I%Height
```

The following simpler example accomplishes exactly the same task (except that **$TEST** is not modified):

```
SET T%Size=$GET(I%Height,100)
```

In the following example, the current code body is exited if I%Elements does not have array children.

```
IF $DATA(I%Elements)<10 QUIT
```

The following example, the WALK subroutine, traverses all the descendants of the specified array node, displaying the nodes and their values on the output window. It uses a **FOR** loop with **$ORDER** to traverse the nodes, uses **$DATA** to determine whether a given node contains data, and uses **$NAME** to convert a subnode into a name value. This name value is then used in name indirection as the argument of **$DATA** and is passed as a parameter.

```
WALK(Node) ; Recursive traversal
    NEW Sub,DataVal,NodeName
    IF $DATA(@Node)#10 DO $Env.Output(Node_" =<”_@Node_">")
    SET Sub=""
    FOR  SET Sub=$ORDER(@Node@(Sub)) QUIT:Sub=""  DO
    . SET NodeName=$NAME(@Node@(Sub))
    . SET DataVal=$DATA(@NodeName)
    . IF DataVal'[0 DO $Env.Output(NodeName_" =<”_@Node_">")
    . IF DataVal>9 DO WALK(NodeName)
    QUIT
```

The following example provides an alternative implementation of WALK. It uses **$DATA** to display the root node if necessary, uses **$LENGTH** and **$EXTRACT** to build an array root, uses a **FOR** loop with **$QUERY** to traverse the array, and uses **$EXTRACT** to determine the exiting condition. Inside the **FOR** loop there is only a single instance of indirection with no a recursive call. Therefore, this example may run faster.

```
WALK(Node) ; Nonrecursive traversal
    NEW Root,Len
    IF 11[$DATA(@Node) DO $Env.Output(Node_" =<”_@Node_">")
    SET Len=$LENGTH(Node),Root=Node
    IF $EXTRACT(Root,Len)=")" SET $EXTRACT(Root,Len)=","
    ELSE  SET Root=Root_"(",Len=Len+1
    FOR  S Node=$QUERY(@Node) Q:$EXTRACT(Node,1,Len)'=Root  DO
    . DO $Env.Output(Node_" =<”_@Node_">")
    QUIT
```

## $DELEGATE

The **$DELEGATE** function is used to delegate the execution of a method or accessor to another object that has the same protocol. The system automatically changes context to the delegated object and all input parameters are pass.

### Format

$DE{LEGATE} ( oref )

### Arguments

oref - an object reference to an object that the call will be delegated to.

### Explanation

The **$DELEGATE** function is used to pass all callframe information to another object that has the same input specification. It is a very fast way to delegate responsibility to another object. The receiving objects must have the same call interface. IN, INOUT and OUT parameter keywords are honored.

### Related

QUIT Command

SET Command

$CALLFRAME special variable

### Examples

Assume a dispatcher object recieves a message to send a truck to deliver a package. The input parameters on the message call contain all the information needed to dispatch a truck to deliver a package. The Dispatcher object has a list of all driver objects. The dispatcher simply identifies the dirver that is available and delegates responsibility to him by issuing a:

```
   QUIT $DELEGATE(T%DriverOID)
```

The system will switch context to the object identified by T%DriverOID and redirect all input specifications to that object’s method.

$Delegate will invoke the current service with the current parameters on the requested object. Should the delegating function need to alter the parameters being delegated it may do so by first manipulating the current call frame using the $Callframe object.

## $EXIST

The **$EXIST** function determines whether or not an object exists.

### Format

$EX{IST} ( oref[, cnexpr] )

### Arguments

oref ::= expr V OID - object reference to an object that may or may not exist.

cnexpr ::= expr V classname - The optional name of the class of which the object may or may not be a member.

### Explanation

For the one-argument form, if the argument of **$EXIST** contains an object reference to an object that currently exists, the function returns 1. If the object no longer exists, the function returns 0. If the argument of **$EXIST** is not an object reference, it returns B (built-in data type), which is interpreted as 0 in a truth-value context.

For the two-argument form, the second argument must evaluate to a class. The full class reference should be specified, that is, “library$class”. If the object is an instance of the class, the $Exist function will return true (1).

### Comments

Keep the following points in mind when you use the **$EXIST** function:

- The **$DATA** and **$EXIST** functions are related but are different. **$DATA** checks for the structural existence of a symbol without referencing its value and **$EXIST** checks the value it is passed to determine whether it is an object reference to an existing object.
- If the argument is a symbol that is not structurally defined (in other words, its **$DATA** value is not 1 or 11), then an undefined variable error can occur.

### Related

$DATA function

### Examples

The following example makes sure that the object T%Window exists before sending it an Activate message.

```
IF $EXIST(T%Window) DO T%Window.Activate
```

The following example shows how to use **$EXIST** to determine whether the object in the temporary symbol Unknown can be sent a Browse message (in other words, if it refers to an existing object).

```
IF $EXIST(T%Unknown) DO T%Unknown.Browse
```

The previous code does not make any distinction between built-ins and invalid OIDs. The behavior is identical if the **$EXIST** function returns B or 0, which is in keeping with the intent of the example.

The two argument form of the $**EXIST** is used to determine whether an OID is associated with a particular class. In the following example, if the object bound to the T%OID  variable is not a “Base$Set” instance, then assert an error to the client.

```
IF ‘$EXIST(T%OID,”Base$Set”) DO $Env.Assert(“Object a Base$Set object.”)
```

## $EXTCALLBACK

The **$EXTCALLBACK** function returns a callback frame identifier used in calling back to a label within any method of the current object.

### Format

$EXTCALL{BACK} (methexp, labelexp, typecode {, optionscode})

### Arguments

methexp ::= expr V method

The first argument is a method name that is valid for the current object.

labelexp ::= expr V label

The second argument is a label or public label, as it would be referenced from within the specified method's code context.

typecode - the second argument is a numeric code value interpreted as follows:

| Type | Description | Code |
|---|---|---|
| Capture | Callback capturing creator's method-related symbols. | 1 |
| Initialized | Callback that starts with a clean variable context. | 2 |

optionscode

An integer interpreted according to bit values:

| Option | Description | Bit |
|---|---|---|
| Persistent | Survive for the duration of the creating object. | 0 |
| Additive | Preserve variable state between calls. | 1 |

### Explanation

Objects can create callbacks to specific labels inside their methods. This allows external objects to directly invoke a specific label in a specific method. The code being invoked runs in the context of the object that created the object. Callbacks can be invoked as **DO** or **GOTO** arguments or as extrinsic functions.

The **$EXTCALLBACK** function is a privileged function that returns a callback frame identifier used in calling back to a label within any method of the current object. This identifier can be used to call externally into the specified context. Special privileges are required to compile code using this function, and its general use in EsiObjects is not recommended. In most cases you would use the **$CALLBACK** function to create callbacks.

### Comments

Keep the following points in mind when you use the **$EXTCALLBACK** function:

- Use the **DO** form of a callback for output, looping, and update functions.
- Use the **GOTO** form for delegation and error handling.
- Use the extrinsic function form of a callback for searches and property lookup.

### Related

Message Syntax

$CALLBACK function

### Examples

In the following example, the **$EXTCALLBACK** function is used to return a callback to the MODIFY label within the Activate method of the current object. The call frame starts with a clean variable context.

```
SET T%CallBack=$EXTCALLBACK("Activate","MODIFY",2)
```

The following example returns a callback to DELETE. The callback starts with a clean variable context, and is persistent and additive.

```
SET T%CallBack=$EXTCALLBACK(DELETE,2,3)
```

## $EXTRACT

The **$EXTRACT** function returns some part of a string based on character cell positions. A special **SET** form can be used to modify portions of a string stored in a variable.

### Format

$E{XTRACT} ( expr {,intexpr1 {,intexpr2}} )

### Arguments

expr - An expression whose value is interpreted as a string, some portion of which is to be returned back.

intexpr1 - An integer value indicating the starting position of the substring to be returned.

intexpr2 - An integer value indicating the ending position of the substring to be returned.

### Explanation

**$EXTRACT** accepts three arguments:

- String
- Starting position
- Ending position

The ending position is an absolute character position, not the number of characters after the start. The second and third arguments are similar to the third and fourth arguments of **$PIECE**, for example:

```
SET T%String="ABCDEFG"
DO $Env.Output($EXTRACT(T%String,3,5))
```

```
Result: CDE
```

Only the first argument is required. If the ending position is omitted, the starting position is used as a default value. If the starting position is also omitted, the first position in the string is used. The first argument of **$EXTRACT** is interpreted as a string. The second and third arguments are interpreted as integers. Decimal values are truncated.

A special **SET** form can be used to modify portions of a string stored in a variable. The special construct **SET $EXTRACT** is used to replace one or more character positions of a variable's contents without affecting the rest of the string.

```
SET T%String="ABCDEFG"
SET $EXTRACT(T%String,3,5)="*"
DO $Env.Output(T%String)

Results: AB*FG
```

If the variable does not exist yet, it is given a starting value of NULL (""). If the number of characters in the existing string is less then the starting character position, extra space characters are added as necessary.

### Comments

Keep the following points in mind when you use the **$EXTRACT** function:

- If the starting position is a value less than 0, then 0 is used.
- If the ending position is greater than the number of characters in the string, then the string's actual length is used.
- If the starting portion of the string comes after its end, or if the starting position is greater than the ending position, then NULL ("") is returned.
- The $EXTRACT function generally can be used in any expression context. However, the special construct **SET $EXTRACT** also can be used to modify a substring of a variable's contents.

### Related

SET command

$LENGTH function

$PIECE function

### Examples

The following example displays the third through fifth characters of the string ABCDEFG.

```
DO $Env.Output($EXTRACT("ABCDEFG",3,5))

Results: CDE
```

The following example displays the 5th through 10th characters of the string ABCDEFG. Because there are only seven characters, the fifth through seventh characters are displayed.

```
DO $Env.Output($EXTRACT("ABCDEFG",5,10))


Results: EFG
```

The following example displays characters negative three through two of the string ABCDEFG. Because negative character positions are empty, the first two characters are displayed.

```
DO $Env.Output($EXTRACT("ABCDEFG",-3,2))


Results: AB
```

The following example displays characters 10 through 12 of the string ABCDEFG. Because the starting position is beyond the end, nothing is displayed.

```
DO $Env.Output($EXTRACT("ABCDEFG",10,12))
```

The following example displays characters five through three of the string ABCDEFG. Because the starting position greater than the ending position, NULL ("")  is displayed.

```
DO $Env.Output($EXTRACT("ABCDEFG",5,3))
```

The following example displays character four of the string ABCDEFG. Because the ending position is not specified, the character D is returned.

```
DO $Env.Output($EXTRACT("ABCDEFG",4))


Results: D
```

The following example displays the first character of the string ABCDEFG. Because the starting and ending positions are not specified, only the first character is returned.

```
DO $Env.Output($EXTRACT("ABCDEFG"))


Results: A
```

The following example uses the special construct **SET $EXTRACT** to modify the third through fifth characters of the variable T%String. Note that the replacement string is simply substituted for those characters. Because the replacement string's length is not the same as the length of the substring being replaced, this operation changes the length of the string in T%String.

```
SET T%String="ABCDEFG"
SET $EXTRACT(T%String,3,5)="*"
DO $Env.Output(T%String)


Results: AB*FG
```

The following example uses **SET $EXTRACT** to modify the tenth character of the variable T%String. However, because there are only seven characters in T%String at the time, two spaces are added first.

```
SET T%String="ABCDEFG"
SET $EXTRACT(T%String,10)="*"
DO $Env.Output(T%String)


Results: ABCDEFG  *
```

If the variable does not yet exist, it is given a starting value of NULL (""). In the following example, the variable T%String is undefined, and character position 5 is replaced with the string Text. To achieve this, four spaces are automatically placed at the start of the string.

```
KILL T%String
SET $EXTRACT(T%String,5)="Text"
DO $Env.Output(T%String)


Results:    Text
```

The **SET $EXTRACT** construct is one way to generate a string containing only spaces. This example creates a string containing 80 spaces.

```
KILL T%String
SET $EXTRACT(T%String,81)=""
```

The following example uses a **FOR** loop with the one-argument **$LENGTH**, **SET $PIECE**, and **$EXTRACT** functions to produce a string in which the individual characters of the string "EsiObjects" become comma-delimited pieces in the variable T%String. After these lines have been executed, T%Result should contain the string E,s,i,O,b,j,e,c,t,s.

```
SET T%Result="",T%String="EsiObjects"
FOR T%Loop=1:1:$LENGTH(T%String) DO
. SET $PIECE(T%Result,",",T%Loop)=$EXTRACT(T%String,T%Loop)
```

The following example, the WALK subroutine, traverses all the descendants of the specified array node, displaying the nodes and their values on the output window. It uses **$DATA** to display the root node if necessary, uses **$LENGTH** and **$EXTRACT** to build an array root, uses a **FOR** loop with **$QUERY** to traverse the array, and uses **$EXTRACT** to determine the exiting condition.

```
WALK(Node) ; Nonrecursive traversal
    NEW Root,Len
    IF 11[$DATA(@Node) DO $Env.Output(Node_" =<”_@Node_">")
    SET Len=$LENGTH(Node),Root=Node
    IF $EXTRACT(Root,Len)=")" SET $EXTRACT(Root,Len)=","
    ELSE  SET Root=Root_"(",Len=Len+1
    FOR  S Node=$QUERY(@Node) Q:$EXTRACT(Node,1,Len)'=Root  DO
    . DO $Env.Output(Node_" =<”_@Node_">")
    QUIT
```

## $FIND

The **$FIND** function finds the next location of a substring within another string after a specified starting position.

### Format

$F{IND} ( expr1, expr2 {, intexpr} )

### Arguments

expr1 - The larger string to be searched.

expr2 - The substring to look for.

Intexpr - The starting character position from which to begin the search.

### Explanation

The **$FIND** arguments are as follows:

- A string to be searched
- The substring to look for
- The starting position for the search

**$FIND** begins at the starting position, scanning forward until the substring is found. If the search position is not specified, the search begins at the start of the string. If it is found, the character position immediately after the end of the substring is returned. (This is the character position from which the next search might begin.) If it is not found, 0 is returned.

### Comments

Keep the following points in mind when you use the **$FIND** function:

- The starting position is interpreted as an integer value greater than zero. If a decimal value is specified, it is truncated to an integer. If a value less than 1 is specified, then 1 is used.
- If a search position greater than the end of the string is specified, **$FIND** returns 0.

### Related

$EXTRACT function

$LENGTH function

$PIECE function

## Examples

The following example uses **$FIND** to return the position of the substring “ssi” inside the string Mississippi, starting at search position 1.

```
DO $Env.Output($FIND("Mississippi","ssi",1))


Results: 6
```

The following example uses **$FIND** to return the position of the substring “ssi” inside the string Mississippi, starting at search position 6.

```
DO $Env.Output($FIND("Mississippi","ssi",6))


Results: 9
```

The following example uses **$FIND** to return the position of the substring “ssi” inside the string Mississippi, starting at search position 9.

```
DO $Env.Output($FIND("Mississippi","ssi",6))


Results: 0
```

The following example uses **$FIND** to return the position of the substring “ssi” inside the string Mississippi, starting at the first character in the string.

```
DO $Env.Output($FIND("Mississippi","ssi"))


Results: 6
```

The following example uses a **FOR** loop to make a string containing pointers to all the substring positions in the target string.

```
HITLIST(L%String,L%Sub) ; Display strings with hits
    NEW L%Pos,L%Hits
    SET L%Pos=1
    FOR  DO  QUIT:'L%Pos
    . SET L%Pos=$FIND(L%String,L%Sub,L%Pos)
    . IF 'L%Pos QUIT
    . SET $EXTRACT(L%Hits,L%Pos)="^"
    DO $Env.Output("Searching for '"_L%Sub_"' in '"_L%String_"'.")
    DO $Env.Output(L%String)

DO $Env.Output(L%Hits)
    QUIT
```

## $FNUMBER

The **$FNUMBER** function returns an edited form of a numeric expression.

### Format

$FN{UMBER} ( numexpr , fncodexpr {, intexpr} )

### Arguments

numexpr - A numeric value to be formatted.

fncodexpr - A code string containing a series of characters specifying the formatting operations to be performed on the number. The code strings are interpreted as follows:

| **Code String** | **Description** |
|---|---|
| **P** or **p** | Places parentheses around negative values, or enters spaces if the number is positive (prefix and suffix). |
| **,** | Inserts a comma every three places to the left of the decimal point. |
| **+** | Inserts a plus sign (+) before numbers greater than 0. |
| - | Suppresses the minus sign for negative values. |
| **T** or **t** | Places signs after the string. If sign suppression is enabled, then using this code string results in a trailing space. |

intexpr - an integer value specifying the number of decimal places to which the number is to be rounded.

### Explanation

The **$FNUMBER** function performs a variety of different formatting operations on numeric values. For any of the formatting possibilities listed, the number can be rounded to any number of decimal places. Trailing zeros are left when necessary by the rounding operation. If only two arguments are specified, no rounding occurs. If the second argument is NULL (""), no special formatting occurs other than the rounding operation.

Note the following:

- The **P code string** places parentheses around negative values and suppresses the minus sign (–). If the string is positive, it is surrounded by spaces. This code can be used with the comma code string (**,**) only. It is not legal with any other code strings.
- The **comma code string** (**,**) inserts a comma every three places to the left of the decimal point. Digits to the right of the decimal point are not affected. If the number is between 1000 and –1000, this code string has no effect.
- The **+ code string** inserts a plus sign on numbers greater than 0. If the number is negative, this code string has no effect.

- The **– code string** suppresses the minus sign on negative values. If the number is positive, this code string has no effect. Use this code string to find the absolute value of a number.
- The **T code string** places any sign after the string. If a numeric sign (positive or negative) is to be displayed, then the sign appears at the end of the number instead of at the beginning. If sign suppression is enabled, a trailing space results.

## Comments

Keep the following points in mind when you use the **$FNUMBER** function:

- All code string combinations are allowed except you can only use the **P code string** with the comma code string (,).
- If the second argument is NULL (""), no special numeric formatting occurs other than rounding.
- If the third argument is omitted, no rounding occurs.
- The return value of **$FNUMBER** is a string that can be subjected to further formatting operations (for example using other functions such as **$JUSTIFY** and **$TRANSLATE**).

## Related

PLUS (+) operator

$JUSTIFY function

$TRANSLATE function

## Examples

The following example uses **$FNUMBER** to place parentheses around negative values and commas before the thousands places.

```
DO $Env.Output($FNUMBER(-9876543.219,"P,",2))
```

```
Results: (9,876,543.22)
```

The following example uses **$FNUMBER** to place parentheses around negative values and commas before the thousands places. Because the number is positive, it is surrounded by spaces. Asterisks are displayed to show the spaces in the output.

```
DO $Env.Output("*"_$FNUMBER(9876543.219,"P,",2)_"*")


Results: * 9,876,543.22 *
```

The following example uses **$FNUMBER** with no formatting codes and two decimal places to round the number. Note the use of the numeric interpretation operator + to strip any trailing zeros.

```
DO $Env.Output(+$FNUMBER(-9876543.995,"",2))


Results: -9876544
```

The following example uses **$FNUMBER** with the minus code (-) to remove the minus sign on negative values. Note that the result is the absolute value of the number.

```
DO $Env.Output($FNUMBER(-9876543.219,"-"))


Results: 9876543,219
```

The following example uses **$FNUMBER** to display the plus sign (+) on positive values and places the sign at the end of the value.

```
DO $Env.Output($FNUMBER(53647,"T+"))


Results: 53647+
```

The following example uses **$FNUMBER** to suppress the minus sign on negative values and place the minus sign (–) at the end of the value. The asterisks are used to show the space character. The T- combination always results in a trailing space.

```
DO $Env.Output("*"_$FNUMBER(-764318.84,"T-")_"*")


Results: *764318.84 *
```

The following example uses **$FNUMBER** with **$JUSTIFY** to format a number with commas and parentheses, rounded to two decimal places and right justified in a field of fifteen spaces. Asterisks are used to bring out the spaces. Note that the rounding occurs in the innermost function.

```
DO $Env.Output("*"_$JUSTIFY($FNUMBER(-764318.84321,"P,",2),15)_"*")


Results: *  (764,318.84)*
```

The following example uses **$FNUMBER** with **$TRANSLATE** to format a number with periods instead of commas in the thousands places, and a comma instead of a period as the decimal indicator.

```
DO $Env.Output($TRANSLATE($FNUMBER(6543210.987,",",2),".,",",."))


Results: 6.543.210,99
```

The following example uses **$FNUMBER** with **$TRANSLATE** to format a number with negative values surrounded by square brackets.

```
DO $Env.Output($TRANSLATE($FNUMBER(-43210,"P"),"()","[]"))
```

```
Results: [43210]
```

## $FREECB

The **$FREECB** function frees a callback.

### Format

$FREECB ( cbref )

### Arguments

cbref ::= expratom V callbackframe

The callback frame identifier string used to invoke the callback.

### Explanation

Objects can create callbacks to specific labels inside their methods. This allows external objects to directly invoke a specific label in a specific method. The code being invoked runs in the context of the object that created the object. Callbacks can be invoked as **DO** or **GOTO** arguments, or as extrinsic functions.

Once a callback is no longer required it should be freed. Under certain conditions, EsiObjects automatically frees callbacks. Original type callbacks are automatically destroyed when the process stack frame that created them exits. Nonpersistent callbacks are freed automatically when their target object is destroyed, and when there is a new incarnation of the environment (in other words, at process shutdown or startup). Persistent callbacks are automatically freed only when their target object is destroyed.

Despite these considerations, it is recommended that all callback types other than Original should be freed explicitly when they are no longer needed. This is done using the **$FREECB** function. The **$FREECB** function's argument is a callback reference. It frees this callback, returning a true value if the callback existed.

### Comments

Keep the following points in mind when you use the **$FREECB** function:

- Use the **DO** form of a callback for output, looping, and update functions.
- Use the **GOTO** format for delegation and error handling.
- Use the extrinsic function form of a callback for searches and property lookup.

### Related

Message Syntax

$CALLBACK function

$EXTCALLBACK function

### Examples

The following example frees the callback in T%CallBack. If the callback was already free, it displays a message.

```
IF '$FREECB(T%CallBack) DO $Env.Output("Callback was already freed!")
```

## $GET

The **$GET** function references a variable whose existence is in doubt, without the danger of getting an undefined variable error.

### Format

$G{ET} ( glvn {, expr} )

### Arguments

glvn - A variable whose value is to be referenced.

expr - The default value to be returned instead, if the variable does not contain a value.

### Explanation

The second argument of **$GET** specifies a default value to be returned if the variable is undefined. If no second argument is specified, NULL ("") is used as the default.

If the second argument is present, the value of the argument is always evaluated (even if the variable is defined).

### Comments

Keep the following points in mind when you use the **$GET** function:

- **$GET** is related to **$DATA** because both are used to interact with symbols that cannot have a value. But unlike **$DATA**, **$GET** is insensitive to the difference between a variable that is not defined and one that is defined with the specified default value. In certain cases such insensitivity is desired, while in other cases it is not.
- If the value of the second argument is present, it is always evaluated (even if the variable is defined). This means that compute-expensive operations should not be placed in the second argument. In certain cases, it is useful to use **$DATA** and **$SELECT** together.

### Related

$DATA function

$SELECT function

### Examples

The following example executes the **QUIT** if L%N is not defined (**$GET** returns NULL ("") as the default value), or if L%N is defined and its value is NULL ("") (**$GET** returns its value).

```
IF $GET(L%N)="" QUIT
```

The following example uses **$DATA**. It assigns the value 100 to the variable T%Size if the instance variable I%Height contains no value, or assigns the value of I%Height to T%Size if it does contain a value.

```
IF $DATA(I%Height)[0 SET T%Size=100
ELSE  SET T%Size=I%Height
```

The following simpler example uses **$GET** to do the same task as shown in the previous example (except that $TEST is not modified):

```
SET T%Size=$GET(I%Height,100)
```

In the following example, the following two lines of code are not equivalent. The first line uses **$GET**, causing the **$CALLBACK** function in the second argument to be called and its value to be ignored if T%CallBack is undefined. The second line uses **$SELECT**, causing the **$CALLBACK** function to be evaluated only if the variable is undefined.

```
SET T%CallBack=$GET(T%CallBack,$CALLBACK(MODIFY))

S T%CallBack=$S($D(T%CallBack)#10:T%CallBack,1:$CALLBACK(MODIFY))
```

## $GETENTRYREF

The **$GETENTRYREF** function returns an entry reference to a handler label that can be called from any M context external to EsiObjects.

### Format

$GETENT{RYREF} ( geterefarg )

### Arguments

geterefarg ::= expr V MethEnt

MethEnt ::= label ^ method

The argument is an expression whose value is of the form **label^method**.

### Explanation

When the handler is called from outside EsiObjects, it is not immediately in any object context. To use instance variables, **$SELF**, and other object-sensitive language elements, the code must associate itself with an object using the **$ASSOCIATE** function. Method inheritance uses the normal inheritance path of the class that implements the method.

The external M code that executes the external callback does not have the benefit of EsiObjects language elements. It can use simple **DO** argument indirection to perform the external callback.

### Comments

Keep the following points in mind when you use the **$GETENTRYREF** function:

- The label in the argument must be declared as Open or Handler.
- The handler is not immediately executed in any object context and must use **$ASSOCIATE** to associate itself with an object if it needs to use **$SELF**, instance variables, and other EsiObjects language elements.

### Related

Method structure

Message Syntax

$ASSOCIATE function

### Examples

The following example gets a handler to the label HANDLE in the method Update.

```
SET T%CallBack=$GETENTRYREF("HANDLE^Update")
```

## $INFO

The $INFO function returns information about an object.

### Format

$INFO( oref, item)

### Arguments

oref - An object reference to the object about which information is being requested.

item - An informational item number or name, denoting the kind of information desired about the object. Possible values are summarized in the following table:

| Item | Name | Description | Returns |
|---|---|---|---|
| **1** | **ClassPointer** | Pointer to object's class. | OID of the object’s class. |
| **2** | **ClassName** or **Class** | Name of the object's library and class (and nested classes). | A string in the form: Lib$Class>NestedClass>… |
| **3** | **Existence** | True if the object exists. | 1 if the object exists, 0 if it does not exist. |
| **4** | **Persistence** | True if the object is persistent. | 1 if object is persistent, 0 if it is not persistent. |
| **5** | **Domain** | The domain in which the object resides. | Name of the Domain (string). |
| **6** | **Parent** | The object's parent. | Object reference (OID) to the parent. |
| **7** | **Name** | The object's primary name. | Name of the object if it exists (string). |
| **8** | **Reference** | The object’s internal reference count. | The actual reference count value of the object. |
| **9** | **Virtual** | The object is a virtual object, not a real object. | 0 if not virtual, non-zero if it is virtual. |
| **10** | **ExternalClass** | The objects external class name (What Java Proxy is used). | Name of the external class. |
| **11** | **Self** | The object handle which is equivalent to $Self. It allows various remote agents to get at the information. Only useful within a Java proxy. | The objects handle. |
| **12** | **Protected** | The object is protected from general debugging access. (An object is protected using the $OSR Function) | 1 if object is protected, 0 if it is not. |

### Explanation

**$INFO** provides a general mechanism to be used in obtaining status information about an object. Objects could implement properties to return these values, but **$INFO** is automatically available for all objects without placing any constraints or burdens upon the programmer.

### Comments

Keep the following points in mind when you use the **$INFO** function:

- The **$INFO** and **$EXIST** functions are related but are different. **$EXIST** only checks for the existence of an object where **$INFO** checks for a number of object related characteristics.

## Related

$EXIST function

$DATA function

## Examples

The following example makes sure that the object T%Employee exists before sending it an Promote message:

```
IF $INFO(T%Employee,3) DO T%Employee.Promote("Supervisor")
```

The following is equivalent to the last example:

```
IF $INFO(T%Employee,”Existence”) DO T%Employee.Promote("Supervisor")
```

The following example makes sure the object bound to the T%Class temporary variable points to a class befor it gets the classes name.

```
IF $INFO(T%Class,1) S T%ClassName=T%Class.Name
```

The example below returns the class path name of an object when it is a nested class. The value placed in T%Path would be HIS$Patient>AdmitDate if the object inn T%AdmitDate is an instance of the nested Patient class AdmitDate.

```
Set T%Path=$INFO(T%AdmitDate,2)
```

## $ISA

The **$ISA** function performs two kinds of checks.

- It checks an object as a member of a certain class or one of its descendants
- Additionally, if the object is of a certain type. Types supported are String, Integer, Numeric, Boolean, Object, Variant or Any.

### Format

$ISA ( oref , cnexpr )

$ISA ( oref , type )

### Arguments

oref - An object reference to the object to be tested for class membership.

cnexpr ::= expr V classname - The name of the class of which the object may or may not be a member.

Type ::= expr V itype – The name of an internal type.

### Explanation

The $ISA function is used to insure that an instance of a class belongs to that classes parentage or of a certain internal type.

### Examples

This example checks an object bound to the T%Obj variable as belonging to the Collection class. If not, it issues a dialog box with an error message.

```
  If ‘$ISA(T%Obj,”Collection”) Do $Env.Assert(“Not a Collection object.”)
```

The following examples can be used to test for a internal type:

```
$IsA(x,"String")    ;Matches any none object

$IsA(x,"Integer")   ;Matches any positive or negative interger

$IsA(x,"Numeric")   ;Matches a numeric value (Excluding scientific notation)

$IsA(x,"Boolean")   ;Matched 0 or 1

$IsA(x,"Object")    ;Matches any object

$IsA(x,"Variant")   ;Matches anythng

$IsA(x,"Any")       ;Matched anything
```

## $JUSTIFY

The **$JUSTIFY** function right-justifies a string or number in a field containing a certain number of spaces, and rounds numeric values to a specified number of decimal places.

### Format

$J{USTIFY} ( expr, numexpr2 )

$J{USTIFY} ( numexpr1, numexpr2, numexpr3 )

### Arguments

expr - A string to be right-justified by adding spaces to the left (two-argument form).

numexpr1 - A numeric value to be rounded to the specified number of decimal places and right-justified by adding spaces to its left (three-argument form).

numexpr2 - The minimum width of the return value.

numexpr3 - The number of decimal places to which the rounding is carried out (three-argument form).

### Explanation

The first argument of **$JUSTIFY** is interpreted as a string in the two-argument form, or as a number in the three-argument form. This is because the three-argument form needs to perform an inherently numeric (rounding) operation.

The second argument is the minimum width of the return value. If the string or rounded number contains fewer characters than this value, the appropriate numbers of spaces are added to the left until its width equals this value. If the string or rounded number contains more characters than the second argument, it is longer than the minimum width indicated by the second argument.

The third argument, if specified, is the number of decimal places to which rounding is carried out. Trailing zeros are added if necessary. If not specified, the first argument is not interpreted numerically and no rounding occurs.

### Comments

Keep the following points in mind when you use the **$JUSTIFY** function:

- The third argument is interpreted as an integer and cannot be a negative number.
- In the three-argument form, if the first argument is a value greater than –1 but less than 1, then the return value has a zero digit to the left of the decimal.

### Related

Unary PLUS ( + ) operator

$FNUMBER function

$TRANSLATE function

### Examples

The following example uses **$JUSTIFY** to right-justify a string in a field of twenty spaces. Asterisks are used to show where the spaces are added.

```
DO $Env.Output("*"_$JUSTIFY("Jane Q. Public",20)_"*")
```

```
Results: *   Jane Q. Public*
```

The following example uses **$JUSTIFY** to round a number to two decimal places, right-justifying it in a field of 10 spaces. Asterisks are used to show where the spaces are added.

```
DO $Env.Output("*",$JUSTIFY(1234.5678,10,2),"*")
```

```
Results: *  1234.57*
```

The following example uses the unary **PLUS (+)** operator with **$JUSTIFY** to round a number to two decimal places. Note the use of the value 0 in the second argument.

```
DO $Env.Output(+$JUSTIFY(1234.9995,0,2))
```

```
Results: 1235
```

The following example uses **$JUSTIFY** and **$TRANSLATE** to pad a number with leading zeros so that it is 5 characters wide.

```
DO $Env.Output($TRANSLATE($JUSTIFY(123,5),0," "))
```

```
Results: 00123
```

The following example uses **$FNUMBER** with **$JUSTIFY** to format a number with commas and parentheses, rounded to two decimal places and right justified in a field of fifteen spaces. Asterisks are used to bring out the spaces. Note that the rounding occurs in the innermost function.

```
DO $Env.Output("*"_$JUSTIFY($FNUMBER(-764318.84321,"P,",2),15)_"*")
```

```
Results: *  (764,318.84)*
```

The following extrinsic variable returns a string containing the approximate time, based on the second comma-piece of **$HOROLOG**. Note the use of **$SELECT**, **$JUSTIFY**, and **$TRANSLATE** in this function.

```
TIME() ; TIME extrinsic variable
    NEW L%Time,L%Hour,L%Minute,L%Meridian
    SET L%Time=$PIECE($HOROLOG,",",2)
    IF L%Time#43200=0 Q "12:00"_$SELECT(L%Time:"pm",1:"am")
    SET L%Hour=L%Time\3600
    SET L%Meridian=$SELECT(L%Hour>11:"pm",1:"am")
    SET L%Hour=$JUSTIFY(L%Hour#12,2)
    IF L%Hour=" 0" SET L%Hour=12
    SET L%Minute=$JUSTIFY(L%Time\60#60,2)
    SET L%Time=$TR(L%Hour_":"_L%Minute_L%Meridian," ",0)
    QUIT L%Time
```

## $LENGTH

The **$LENGTH** function measures the length of a string in terms of character cells or delimited pieces.

### Format

$L{ENGTH} ( expr1 {, expr2} )

### Arguments

expr1 - A string whose length is to be measured in terms of character cells or delimited pieces.

expr2 - If present, a delimiter into which the string is to be broken up.

### Explanation

The one-argument form of **$LENGTH** measures the number of characters in the string. In the two-argument form, the second argument is a delimiter used to divide the string into pieces. This form of **$LENGTH** returns the number of pieces in the string. Delimiters are usually one character in length, but the only limit to their length is the maximum string size.

The number of pieces is similar to the number of words in a sentence. For example, in the following string, using a space as a delimiter, the total number of pieces equals 4:

```
  "John dropped the ball."
```

The number of pieces is always equal to the number of nonoverlapping instances of the delimiter, plus 1. The following table shows additional examples.

| String | Delimiter | Number of Pieces |
|---|---|---|
| "first, second,third, fourth" | , | 4 |
| "ABCBBDABE" | B | 5 |
| "^^^^^" | ^ | 6 |
| "Hello" | ? | 1 |
| "xxxxxx" | Xx | 4 |
| "xxxxxxx" | Xx | 4 |
| "" |  | 1 |

### Comments

Keep the following points in mind when you use the **$LENGTH** function:

- In the one-argument format, if the string is NULL ("") then **$LENGTH** returns 0.
- In the two-argument form, if the second argument is specified as NULL (""), then the return value is always 0.
- The one-argument form of **$LENGTH** often is used with **$EXTRACT**. The two-argument form often is used with **$PIECE**.

### Related

SET command

$EXTRACT function

$PIECE function

$ZLENGTH function

## Examples

The following example uses a **FOR** loop with the one-argument **$LENGTH**, **SET $PIECE**, and **$EXTRACT** to produce a string in which the individual characters of the string "EsiObjects" become comma-delimited pieces in the variable T%String. After these lines have been executed, T%Result should contain the string "E,s,i,O,b,j,e,c,t,s".

```
SET T%Result="",T%String="EsiObjects"
FOR T%Loop=1:1:$LENGTH(T%String) DO
. SET $PIECE(T%Result,",",T%Loop)=$EXTRACT(T%String,T%Loop)
```

The following extrinsic function performs a search-and-replace operation on a string, sending back the transformed string as its return value. It uses the two-argument **$LENGTH** to count the number of pieces in the source string, and uses **$PIECE** and **SET $PIECE** to do the replacement operation.

```
REPL(L%String,L%From,L%To) ; Replace L%From with L%To in L%String
      NEW L%Iter,L%Result,L%Length
      IF L%From="" QUIT ""
      SET L%Length=$LENGTH(L%String,L%From)
      IF L%To="" SET L%Result="" FOR L%Iter=1:1:L%Length DO
      . SET L%Result=L%Result_$PIECE(L%String,L%From,L%Iter)
      ELSE  FOR L%Iter=L%Length:-1:1 DO
      . SET $P(L%Result,L%To,L%Iter)=$P(L%String,L%From,L%Iter)
      QUIT L%Result
```

The following expression returns the string EsiObjects Language for EsiObjects Programming.

```
$$REPL("M Language for M Programming","M","EsiObjects")
```

The following example, the WALK subroutine, traverses all the descendants of the specified array node, displaying the nodes and their values on the output window. It uses **$DATA** to display the root node if necessary, uses **$LENGTH** and **$EXTRACT** to build an array root, a **FOR** loop with **$QUERY** to traverse the array, and **$EXTRACT** to determine the exiting condition.

```
WALK(Node) ; Recursive traversal
    NEW Sub,DataVal,NodeName
    IF $DATA(@Node)#10 DO $Env.Output(Node_" =<”_@Node_">")
    SET Sub=""
    FOR  SET Sub=$ORDER(@Node@(Sub)) QUIT:Sub=""  DO
    . SET NodeName=$NAME(@Node@(Sub))
    . SET DataVal=$DATA(@NodeName)
    . IF DataVal'[0 DO $Env.Output(NodeName_" =<”_@Node_">")
    . IF DataVal>9 DO WALK(NodeName)
    QUIT
```

## $LIBRARY

The **$LIBRARY** function returns the object reference of a library, given its name as input.

### Format

$LIB{RARY} ( libexpr )

### Arguments

libexpr ::= expr V libraryname - An expression whose value is the name of a library.

### Explanation

This function is useful whenever it is necessary to transform a string containing the name of a library into an object reference. The function returns NULL ("") if the specified library does not exist.

### Related

$LIBRARY special variable

### Examples

The following example transforms a library name that the user has selected from a list box into an object reference that can be used to communicate with the library. It then asks the library to copy all its class names into a list box.

```
SET T%Library=$LIBRARY("MyLibrary")
IF T%Library="" QUIT
DO T%Library.CopyClassList(Target:I%ListBox,Names)
```

## $LOOKUP

The **$LOOKUP** function allows privileged code to access the values in a symbol table, optionally including subscript levels.

### Format

$LOOKUP (name, subscripts, typexpr)

### Arguments

name - The name of the variable.

subscripts - An expression whose value is an entire subscript list, including the parentheses.

| | | |
|---|---|---|
| | **A** | Accessor |
| | **C** | Class |
| | **CN** | Constant |
| | **G** | Global |
| | **I** | Instance |
| **Typexpr ::= expr V** | **L** | Local |
| | **N** | NamePool |
| | **O** | Object Name |
| | **P** | Parameter |
| | **R** | Relative/Region |
| | **S** | System |
| | **U** | Universal |

### Explanation

You can use the **$LOOKUP** function with any EsiObjects symbol, including templates and name pools.

### Comments

Keep the following points in mind when you use the **$LOOKUP** function:

- You can use the **$LOOKUP** function to get the values of EsiObjects variables instead of using name or subscript indirection.
- Once the names of the variables are obtained with **$WALK**, their values can be referenced with **$LOOKUP**.
- Because special privileges are required, general use of **$LOOKUP** in EsiObjects is not recommended.

### Related

INDIRECTION (@) operator

$WALK function

### Examples

The following example contains a **FOR** loop used to traverse the names of an object's instance variables with **$WALK** and **$LOOKUP** is used to obtain their values (for those that have simple values).

```
SET T%Loop=""
FOR  SET T%Loop=$WALK(T%Loop,"","I") QUIT:T%Loop=""  DO
. DO T%Window.AddLine("Var: "_T%Loop_", Val: "_$LOOKUP(T%Loop,"","I"))
QUIT
```

## $NAME

The **$NAME** function converts a variable name or array reference to a string representation in which the subscripts are expressed as literals.

### Format

$NA{ME} ( glvn, intexpr )

### Arguments

glvn - A variable name or array reference to be converted to a string representation.

intexpr - The maximum number of subscript levels for the return value.

### Explanation

If the maximum number of subscript levels is specified, then any extra subscripts in the specified array node are not present in the return value. If the number is 0, then only the array root node is returned. If the number is not specified, then all the subscripts in the array node are present in the return value.

### Comments

Keep the following points in mind when you use the **$NAME** function:

- The **$NAME** function has many uses, but one of the most common is to convert an array node reference using subscript indirection into a string that can be used in name indirection, or can itself be used as the base location for deeper levels of subscript indirection.
- **$NAME** is also useful in generating strings containing array names to be stored in variables or passed as parameters.
- The return value of **$NAME** is a namevalue appropriate to be used with name indirection, as a root in subscript indirection, or as a parameter of **$QLENGTH** and **$QSUBSCRIPT**.

### Related

INDIRECTION ( @ ) operator

$QLENGTH function

$QSUBSCRIPT function

$QUERY function

### Examples

The following example uses **$NAME** to convert an array node reference, accessed through subscript indirection, into a string representation that can be stored in a variable. Variable names and expressions in the subscript values are simplified to literal values in the target string.

```
SET T%Handle=$NAME(@T%Target@(T%Loop,T%Line+1))
```

The following example uses **$NAME** to return a string containing only the first three subscript levels of the specified array node.

```
SET T%Handle=$NAME(@T%Target,3)
```

The following example uses **$NAME** to return a string containing only the root node of the specified array.

```
SET T%Handle=$NAME(@T%Target,0)
```

The following example, the WALK subroutine, traverses all the descendants of the specified array node, displaying the nodes and their values on the output window. It uses a **FOR** loop with **$ORDER** to traverse the nodes, uses **$DATA** to determine whether a given node contains data, and uses **$NAME** to convert a subnode into a name value. This name value is then used in name indirection as the argument of **$DATA** and is passed as a parameter.

```
WALK(Node) ; Recursive traversal
    NEW Sub,DataVal,NodeName
    IF $DATA(@Node)#10 DO $Env.Output(Node_" =<”_@Node_">")
    SET Sub=""
    FOR  SET Sub=$ORDER(@Node@(Sub)) QUIT:Sub=""  DO
    . SET NodeName=$NAME(@Node@(Sub))
    . SET DataVal=$DATA(@NodeName)
    . IF DataVal'[0 DO $Env.Output(NodeName_" =<”_@Node_">")
    . IF DataVal>9 DO WALK(NodeName)
    QUIT
```

The following example is an alternative implementation of the WALK subroutine. It uses **$DATA** to display the root node if necessary, uses **$LENGTH** and **$EXTRACT** to build an array root, uses a **FOR** loop with **$QUERY** to traverse the array, and uses **$EXTRACT** to determine the exiting condition. Inside the **FOR** loop there is only a single instance of indirection with no a recursive call.

```
WALK(Node) ; Nonrecursive traversal
    NEW Root,Len
    IF 11[$DATA(@Node) DO $Env.Output(Node_" =<”_@Node_">")
    SET Len=$LENGTH(Node),Root=Node
    IF $EXTRACT(Root,Len)=")" SET $EXTRACT(Root,Len)=","
    ELSE  SET Root=Root_"(",Len=Len+1
    FOR  S Node=$QUERY(@Node) Q:$EXTRACT(Node,1,Len)'=Root  DO
    . DO $Env.Output(Node_" =<”_@Node_">")
    QUIT
```

# $NORMALIZE

The **$NORMALIZE** function is generally used to convert an external value into an internal value for strorage. For example, the external values Yes and No can be normalized to the values 1 and 0 respectively as can True and False. The **$NORMALIZE** function is used in concert with the $Normalize property accessor. When a property is messaged within the context of the **$NORMALIZE** function, the $Normalize accessor is executed

## Format

$NORMALIZE ( Normmpr, expr)

## Arguments

| | |
|---|---|
| **Normmpr ::=** | **Object . service** |
| | **Expr** |

Normmpr - A reference to an objects property service that contains the $Normalize accessor that will be executed to normalize the value.

expr - An expression that evaluates to a string to be normalized.

## Explanation

If the first argument of **$NORMALIZE** is an object service reference to the property that contains the $Normalize accessor. The return value is based on the validity of the service or property assignment value. The second argument is an expression that must evaluate to a string. It is the value to be normalized.

## Comments

The **$NORMALIZE** function should always return the normalized value to the caller. The input value to the function should always be checked for validity before passing it in. The **$NORMALIZE** function should only normalize the value. No attempt should be made to validate or save the value within this function - use the $Valid and Assign accessors for this respecirively.

## Related

$VALID function

Message Syntax

### Examples

The following example illustrates how the $Normalize function would normalize the external value "Yes" to the internal form 1. First an instance of Employee is created and bound to the T%Employee temporary variable. Next, the Veteran property of the Employee object is accessed within the context of the $Normalize function. The $Normalize function returns the normalized value (1) and binds it to the T%Vet temporary variable.

```
CREATE T%Employee=Framework$Employee

S T%Vet=$NORMALIZE(T%Employee.Veteran,"Yes")
```

## $OIDPTR

The **$OIDPTR** function is a privileged function that transforms an object reference into an M pointer that can be used with name indirection.

### Format

$OIDPTR ( oref )

### Arguments

oref - The object reference of the object whose pointer is to be returned.

### Explanation

The **$OIDPTR** function is a privileged function that transforms an object reference into an M pointer that can be used with name indirection. Some of the object's contents are stored under its base pointer, but others are not.

If the argument is not a valid object reference or the operation otherwise fails, the function returns NULL ("").

### Comments

Keep the following points in mind when you use the **$OIDPTR** function:

- It is impossible to tell which of an object's structures fall under its base pointer and which do not.
- Use of **$OIDPTR** easily can result in violations of object encapsulation. Therefore, it is not recommended for general use in EsiObjects.

### Related

$PTROID function

### Examples

The following example locks the root node of the external object whose **oref** is contained in the temporary variable ExtObj, thereby effectively locking the object and its subcomponents. However, any object structures that do not fall underneath this pointer are not locked.

```
LOCK +@$OIDPTR(T%Object12)
```

## $ORDER

The **$ORDER** function returns the next or prior subscript, using the specified array reference as a starting point.

### Format

$O{RDER} ( glvn {, direction} )

### Arguments

glvn - The array node from which the search should begin.

```
direction ::= expr V          1
                             -1
```

The value 1 indicates a forward search and –1 indicates a backward search. If not specified, 1 is the default.

### Explanation

The search takes place at the deepest subscript level specified. In other words, if an n-level array node is specified as the argument, the next or prior nth-level subscript is returned. The return value is always a subscript in the subtree having the same first n-1 subscripts. If no such subscript exists in the specified direction, NULL ("") is returned.

The order in which subscripts are returned is the array's logical collating sequence. In most cases standard ASCII collating order is used:

- 1. NULL ("") comes first.
- 2. All purely numeric values come next, in numeric order. A value X is considered to be purely numeric if the expression +X=X is true (in other words, if its numeric interpretation equals its actual value). Therefore, 2 is numeric, and 2.0 and 2 installations are not numeric.
- 3. All other string values come next, in order of the ASCII code values of their characters. Therefore, A comes before Armadillo and Z, but a comes after all these values.
- 4. NULL ("") comes last.

Subscripts are grouped together in this order underneath their common ancestor. Because NULL ("") comes both first and last in this sequence but is not itself a legal subscript value, it is common to use NULL ("") as both the starting and ending values when using **$ORDER**.

The following example illustrates the traversal of the immediate descendants of a global array node `^MYGLO(22,1).`

```
SET T%Loop=""
FOR  SET T%Loop=$ORDER(^MYGLO(22,1,T%Loop)) QUIT:T%Loop=""  DO
. DO $Env.Output(T%Loop_" = "_^MYGLO(22,1,T%Loop))
QUIT
```

This loop begins the traversal from the subscript position NULL (""), ending it when the iterating variable T%Loop equals NULL ("").

### Comments

Keep the following points in mind when you use the **$ORDER** function:

- Because NULL ("") is the starting and ending value, it is important to test for this value in the terminal condition. Otherwise, **$ORDER** uses it as a starting value, often causing an infinite loop.
- It is more complex to traverse descendant array nodes with **$ORDER** than with **$QUERY**. Often a recursive call is required.
- In some cases where it is desired to visit descendant array nodes, the **$QUERY** function is an alternative to **$ORDER**. However, **$ORDER** is used in the majority of application-programming cases.
- Because **$ORDER** only visits nodes at a single subscript level, it visits all nodes at that level. This means it can visit any array node whose **$DATA** value is 1, 10, or 11.
- The relational **SORTS AFTER ( ]] )** operator is used to determine whether one subscript follows another in the subscript collating sequence. Therefore, often it is used with **$ORDER**.

### Related

SORTS AFTER ( ]] ) operator

FOR command

$DATA function

$QUERY function

## Examples

The following example, the WALK subroutine, traverses all the descendants of the specified array node, displaying the nodes and their values on the output window. It uses a **FOR** loop with **$ORDER** to traverse the nodes, uses **$DATA** to determine whether a given node contains data, and uses **$NAME** to convert a subnode into a name value. This name value is then used in name indirection as the argument of **$DATA** and is passed as a parameter.

```
WALK(Node) ; Recursive traversal
    NEW Sub,DataVal,NodeName
    IF $DATA(@Node)#10 DO $Env.Output(Node_" =<”_@Node_">")
    SET Sub=""
    FOR  SET Sub=$ORDER(@Node@(Sub)) QUIT:Sub=""  DO
    . SET NodeName=$NAME(@Node@(Sub))
    . SET DataVal=$DATA(@NodeName)
    . IF DataVal'[0 DO $Env.Output(NodeName_" =<”_@Node_">")
    . IF DataVal>9 DO WALK(NodeName)
    QUIT
```

The following example provides an alternative implementation of the WALK subroutine. It uses **$DATA** to display the root node if necessary, uses **$LENGTH** and **$EXTRACT** to build an array root, uses a **FOR** loop with **$QUERY** to traverse the array, and uses **$EXTRACT** to determine the exiting condition. Inside the **FOR** loop there is only a single instance of indirection with no a recursive call.

```
WALK(Node) ; Nonrecursive traversal
    NEW Root,Len
    IF 11[$DATA(@Node) DO $Env.Output(Node_" =<”_@Node_">")
    SET Len=$LENGTH(Node),Root=Node
    IF $EXTRACT(Root,Len)=")" SET $EXTRACT(Root,Len)=","
    ELSE  SET Root=Root_"(",Len=Len+1
    FOR  S Node=$QUERY(@Node) Q:$EXTRACT(Node,1,Len)'=Root  DO
    . DO $Env.Output(Node_" =<”_@Node_">")
    QUIT
```

## $OSR

The **$OSR** (Object Service Request) function is used as a general-purpose function to provide object services.

### Format

$OSR(expr$_1$,expr$_2$, expr$_n$ …)

where:

expr$_1$ V string or numeric. It is the name or number of the Service Request.

expr$_2$ V OID which is the target object the service is to be applied to.

expr$_n$… V specific to the Service Request

### Explanation

Object Service provides a general approach to implementing services that the programmer can use. The following table contains a list of services.

| **Service Name** | **Service Number** | **Description** | **Parameters** |
|---|---|---|---|
| Protect | 1 | Applies protection to an object so that the object cannot be browsed. This service actually changes the state of the object. | Expr$_3$ that evaluates to 1. |

### Comments

Services will be added to the list as they are implemented.

### Related

$INFO function

### Examples

The following shows how to protect an object using the $OSR function. Assume that a list contains all the financial transactions of a person. Protecting it from browsing is a requirement.

```
Create T%List=Base$List

Do T%FinObj.LoadList(T%List) ;Loads list with transactions.

Do $OSR(“Protect”,T%List,1) ;Protect list from browsing.

If $INFO(T%obj,”Protected”) W “The object is protected”
```

## $PIECE

The **$PIECE** function returns one or more pieces in a delimited string.

### Format

$P{IECE} ( expr1 , expr2 {, intexpr1 {, intexpr2}} )

### Arguments

expr1 - A string to be broken up in terms of delimited pieces.

expr2 - If present, a delimiter into which the string is to be broken up.

intexpr1 - The number of the first piece being specified.

intexpr2 - The number of the second piece being specified.

### Explanation

The first argument of **$PIECE** is a string and the second argument is a delimiter used to divide the string into pieces. The function returns one or more of the pieces in the string. Delimiters are usually one character in length, but the only limit to their length is the maximum string size.

Informally speaking, the pieces in a string are loosely similar to the words in a sentence if the space character is used as a delimiter. In the following example, the space character is used as a delimiter and the total number of pieces equals 4:

"John dropped the ball."

In the previous example, the first piece is “John”, the fourth piece is “ball” and so on.

In a more formal sense, the number of pieces is always equal to the number of nonoverlapping instances of the delimiter, plus 1. The pieces are the portions of the string occurring between the delimiters. If there is nothing between two delimiters, the value of the piece is NULL (""). The following table contains additional examples.

| **String** | **Delimiter** | **Number of Pieces** | **First Piece** | **Last Piece** |
|---|---|---|---|---|
| "first , second, third, fourth" | , | 4 | "first " | " fourth" |
| "ABCBBDABE" | B | 5 | "A" | "E" |
| "^^^^^" | ^ | 6 | "" | "" |
| "Hello" | ? | 1 | "Hello" | "Hello" |
| "axxxxxx" | xx | 4 | "a" | "" |
| "xxxxxxx" | xx | 4 | "" | "x" |
| "" | Space | 1 | "" | "" |

If the delimiter is specified as NULL (""), then the return value of **$PIECE** is always NULL ("")

The third and fourth arguments of **$PIECE** are similar to the second and third arguments of **$EXTRACT**. Both arguments are interpreted as integers. If the third argument is less than 1, the value 0 is used. If the fourth argument is greater than the number of pieces in the string, the actual number of pieces is used. If the third argument is greater than the number of pieces in the string, or greater than the fourth argument, NULL ("") is returned.

If the third and fourth arguments together address more than one of the pieces in the string, then the entire portion of the string from the starting piece to the ending piece is returned, including the intervening delimiters. For example, the following expression returns third-fourth-fifth:

```
$PIECE("first-second-third-fourth-fifth-sixth-seventh","-",3,5)
```

If the fourth argument is omitted, the value of the third argument is used as a default. This causes the single piece referenced by the third argument to be returned. If the third argument is also omitted, the first piece is returned.

### Comments

Keep the following points in mind when you use the **$PIECE** function:

- The special cases involving the third and fourth arguments of **$PIECE** obey the same general principles as the special cases involving the second and third arguments of **$EXTRACT**.
- The two-argument form of **$LENGTH** is often used with **$PIECE** when it is necessary to count the pieces in a string.

- **SET $PIECE** is a useful tool when manipulating strings in terms of any substring they may contain, even when the substring is not explicitly being used as a delimiter.

## Related

SET command

$EXTRACT function

$LENGTH function

$QSUBSCRIPT function

$ZPIECE function

## Examples

The following example sets T%Result to the value “third-fourth-fifth” using the four-argument form of **$PIECE**.

```
SET T%String="first-second-third-fourth-fifth-sixth-seventh"
SET T%Result=$PIECE(T%String,"-",3,5)
```

The following example sets T%Result to the value fourth using the three-argument form of **$PIECE**.

```
SET T%String="first-second-third-fourth-fifth-sixth-seventh"
SET T%Result=$PIECE(T%String,"-",4)
```

The following example sets T%Result to the value "" using the three-argument form of **$PIECE**.

```
SET T%String="first-second-third-fourth-fifth-sixth-seventh"
SET T%Result=$PIECE(T%String,"-",8)
```

The following example sets T%Result to the value first using the two-argument form of **$PIECE**.

```
SET T%String="first-second-third-fourth-fifth-sixth-seventh"
SET T%Result=$PIECE(T%String,"-")
```

The following example sets T%Result to the value "" because the fourth piece is a null piece.

```
SET T%String="A^B^C^^E"
SET T%Result=$PIECE(T%String,"^",4)
```

The **SET $PIECE** is used to replace one or more delimited pieces of a variable's contents without affecting the rest of the string. In this example, pieces 3, 4, and 5 of the string in T%String are replaced with an asterisk (*).

```
SET T%String="one/two/three/four/five/six/seven"
SET $PIECE(T%String,"/",3,5)="*"
DO $Env.Output(T%String)


Results: one/two/*/six/seven
```

If the variable does not exist yet, it is given a starting value of NULL (""). If the number of pieces in the existing string is less then the starting piece position, extra delimiters are added as necessary. In the following example, the variable T%String is undefined, and "." piece 5 is replaced with the string Text. To achieve this, four periods are automatically placed at the start of the string.

```
KILL T%String
SET $PIECE(T%String,".",5)="Text"
DO $Env.Output(T%String)

Results: ....Text
```

The following example uses a **FOR** loop with the one-argument **$LENGTH**, **SET $PIECE**, and **$EXTRACT** to produce a string in which the individual characters of the string EsiObjects become comma-delimited pieces in the variable T%String. After these lines have been executed, T%Result should contain the string E,s,i,O,b,j,e,c,t,s.

```
SET T%Result="",T%String="EsiObjects"
FOR T%Loop=1:1:$LENGTH(T%String) DO
. SET $PIECE(T%Result,",",T%Loop)=$EXTRACT(T%String,T%Loop)
```

The following extrinsic function performs a search-and-replace operation on a string, sending back the transformed string as its return value. It uses the two-argument **$LENGTH** to count the number of pieces in the source string, and uses **$PIECE** and **SET $PIECE** to do the replacement operation.

```
REPL(L%String,L%From,L%To) ; Replace L%From with L%To in L%String
    NEW L%Iter,L%Result,L%Length
    IF L%From="" QUIT ""
    SET L%Length=$LENGTH(L%String,L%From)
    IF L%To="" SET L%Result="" FOR L%Iter=1:1:L%Length DO
    . SET L%Result=L%Result_$PIECE(L%String,L%From,L%Iter)
    ELSE  FOR L%Iter=L%Length:-1:1 DO
    . S $PIECE(L%Result,L%To,L%Iter)=$PIECE(L%String,L%From,L%Iter)
    QUIT L%Result
```

The following expression returns the string EsiObjects Language for EsiObjects Programming:

```
$$REPL("M Language for M Programming","M","EsiObjects")
```

## $PROTECT

The **$PROTECT** function protects an object from being preserved or destroyed.

### Format

$PROTECT(expr)

where:

expr V OID

### Explanation

The $PROTECT function creates a protected pointer (OID) to an object. Often, when an object handle is exposed to a consumer, protecting it from being destroyed or preserved is important.

### Comments

The protected object ignores both the Preserve & Destroy commands.

Used to protect an object when the pointer is exposed.

Changes the OID form $C(31)_N_ptr to $C(31)_T_ptr.

### Related

DESTROY command

$REFERENCE special variable

PRESERVE command

CREATE command

### Examples

Assume that a handle to a patients record must be returned to the consumer and that it must be protected from being destroyed or preserved. The handle can be handed back by the methods Quit command.

```
Quit $Protect(I%PatOid)
```

## $PTROID

Given a string representing the name of an M variable that is the base location for an object, **$PTROID** returns a handle for the object.

### Format

$PTROID ( namevalue , typexpr )

### Arguments

namevalue ::= expr V glvn - An expression whose value is the base array node of the object's location.

typexpr ::= expr V type - An expression whose value is generally N for normal, but can be T for template, L for class or I for instance.

### Explanation

The **$PTROID** function is a privileged function that transforms an M pointer into an object reference. This can cause errors if a valid object is not stored at the specified location.

### Comments

Keep the following points in mind when you use the **$PTROID** function:

- **$PTROID** is only needed in cases where an M pointer to an object exists, but the object reference is not known. It is generally better to use object-level services to interact with objects.
- Use of **$PTROID** can result in violations of object encapsulation. Therefore, it is not recommended for general use.

### Related

$OIDPTR function

### Examples

The following example converts the root node ^OOTEST(10) into a normal object reference.

```
SET T%Object12=$PTROID("^OOTEST(10)","N")
```

## $PTRSTR

The **$PTRSTR** function converts an object reference into a normalized form suitable for use with string operations.

### Format

$PTR{STR} ( oref )

### Arguments

oref - The object reference to be converted.

### Explanation

The **$PTRSTR** function converts an object reference into a normalized form suitable for use with string operations.

### Comments

Keep the following points in mind when you use the **$PTRSTR** function:

- Unlike **$PTROID** and **$OIDPTR**, **$PTRSTR** is not a privileged function.
- The string produced by **$PTROID** can contain control characters, but it is suitable for use in string operations. Usually, the string can be displayed without causing errors, and is interpreted as a literal value rather than an object reference.

### Related

$OIDPTR function

$PTROID function

### Examples

The following example returns the pointer to a database directory object in string form and displays it in the output window.

S T%DbPtr=$PTROID(I%Databases))

```
Results:  N  N^shrobj(35,1)
```

## $QLENGTH

The **$QLENGTH** function returns the number of subscripts in a string containing an array reference.

### Format

$QL{ENGTH} ( namevalue )

### Arguments

namevalue - A string containing the name of an array node.

### Explanation

The **$QLENGTH** function returns the number of subscripts in a string containing an array reference. For example, if the string references an array node with 5 subscripts, then **$QLENGTH** returns 5. If the string references a root array node with no subscripts, then **$QLENGTH** will return 0.

### Comments

The behavior of this function is unspecified in cases where the argument is not a properly formatted **namevalue**.

### Related

$LENGTH function

$QSUBSCRIPT function

$ZLENGTH function

### Examples

The following **FOR** loop displays the root node and all the subscripts of the array referenced in T%Target.

```
FOR T%Loop=0:1:$QLENGTH(T%Target) DO
.  DO $Env.Output($QSUBSCRIPT(T%Target,T%Loop))
```

The following example removes the last subscript from the array node in T%Target:

```
SET T%Target=$NAME(T%Target,$QLENGTH(T%Target)-1)
```

## $QSUBSCRIPT

The **$QSUBSCRIPT** function returns the specified subscript in a string containing an array reference.

### Format

$QS{UBSCRIPT} ( namevalue, intexpr )

### Arguments

namevalue - A string containing an array node reference suitable for use with name indirection.

intexpr - The numeric position of the subscript whose value is to be returned.

### Explanation

The **$QSUBSCRIPT** function returns the specified subscript in a string containing an array reference. If subscript number 0 is requested, the array root node is returned. If subscript number –1 is requested, then the return value is the environment if the array reference includes an environment name, or NULL if it does not. If a subscript number is specified that is greater than the actual number of subscripts in the array reference, the NULL ("") is returned. Subscripts numbered less than -1 are not allowed.

For example, consider a string that references an array node with 5 subscripts. If subscript 3 is requested, then **$QSUBSCRIPT** returns the literal value of the third subscript. If subscript 0 is requested, it returns the value of the array root. If subscript 6 is requested, it returns NULL ("").

### Comments

The behavior of this function is unspecified in cases where the argument is not a properly formatted **namevalue**.

### Related

$NAME function

$PIECE function

$QLENGTH function

$ZPIECE function

### Examples

The following **FOR** loop displays the root node, and all the subscripts of the array referenced in T%Target.

```
FOR T%Loop=0:1:$QLENGTH(T%Target) DO
.   DO $Env.Output($QSUBSCRIPT(T%Target,T%Loop))
```

## $QUERY

The **$QUERY** function returns the full reference of an array node that has a value associated with it.

### Format

$Q{UERY} ( glvn )

### Arguments

glvn - Specifies the array node position from which the search is to begin.

### Explanation

The **$QUERY** function's argument is a **glvn** (array node) and its return value is a **namevalue** (string containing an array node). Name indirection is often used to convert this string back to an array node.

The order in which subscripts are returned is the array's logical collating sequence. In most cases standard collating order is used:

- 1. NULL ("") comes first.
- 2. All purely numeric values sort next in numeric order. A value X is considered to be purely numeric if the expression +X=X is true (in other words, if its numeric interpretation equals its actual value). Therefore, 2 is numeric and 2.0 and 2 installations are not numeric.
- 3. All other string values come next, in order of the ASCII code values of their characters. Therefore, A comes before Armadillo and Z, but a comes after all these values.
- 4. NULL ("") comes last.

Subscripts are grouped together in this order underneath their common ancestor. Because NULL ("") is an illegal subscript value, it is never actually returned by **$QUERY** except when there is no next value to return.

The following example illustrates a **$QUERY** traversal beginning from the array node ^MYGLO(22,1). Note that the traversal does not automatically stop once it has passed the descendants of this node.

```
CREATE I%File=Base$AbsSerializationObject

DO I%File.Open("MYGLO.TXT")

SET T%Loop="^MYGLO(22,1)"
FOR  SET T%Loop=$QUERY(@T%Loop) QUIT:T%Loop=""  DO
. DO I%File.Use("MYGLO.TXT")

. DO I%File.Write(T%Loop,@T%Loop)
DO I%File.Close("MYGLO.TXT")

QUIT
```

This loop begins the traversal from the array node ^MYGLO(22,1), ending it when the iterating variable T%Loop equals NULL (""). A more complicated terminal condition would be required to exit when the subtree has been completed.

### Comments

Keep the following points in mind when you use the **$QUERY** function:

- **$QUERY** only visits array nodes whose **$DATA** values are 1 or 11.
- The root node of the subtree to be traversed should be specified as the starting value.
- It is more complex to traverse descendant array nodes with **$ORDER** than with **$QUERY**. However, determining when the end of a subtree has been reached is easier with **$ORDER**. **$QUERY** returns NULL ("") when it reaches the end of the array, not the end of the subtree being traversed. Therefore, a special test is required to determine when the subtree has been entirely traversed.

### Related

FOR command

$DATA function

$ORDER function

## Examples

The following example, the WALK subroutine, traverses all the descendants of the specified array node, exporting the node name and value to an external serial device. It uses a **FOR** loop with **$ORDER** to traverse the nodes, uses **$DATA** to determine whether a given node contains data, and uses **$NAME** to convert a subnode into a name value. This name value is then used in name indirection as the argument of **$DATA** and is passed as a parameter.

```
WALK(Node) ; Recursive traversal
;Assumes I%File points to a serialization object

DO I%File.Open("MYGLO.TXT")

    IF $DATA(@Node)#10 DO I%File.Write(Node,@Node)
    SET Sub=""
    FOR  SET Sub=$ORDER(@Node@(Sub)) QUIT:Sub=""  DO
    . SET NodeName=$NAME(@Node@(Sub))
    . SET DataVal=$DATA(@NodeName)
    . IF DataVal'[0 DO I%File.Write(NodeName,@NodeName)
    . IF DataVal>9 DO WALK(NodeName)
    QUIT
```

The following example provides an alternative implementation of WALK. It uses **$DATA** to display the root node if necessary, uses **$LENGTH** and **$EXTRACT** to build an array root, uses a **FOR** loop with **$QUERY** to traverse the array, and uses **$EXTRACT** to determine the exiting condition. Recursive calls are unnecessary when using the **$QUERY**.

```
WALK(Node) ; Nonrecursive traversal
    IF 11[$DATA(@T%Node) DO I%File.Write(T%Node,@T%Node)
    SET T%Len=$LENGTH(T%Node),T%Root=T%Node
    IF $EXTRACT(T%Root,T%Len)=")" SET $EXTRACT(T%Root,T%Len)=","
    ELSE  SET T%Root=T%Root_"(",T%Len=T%Len+1
    FOR  SET T%Node=$QUERY(@T%Node) QUIT:$EXTRACT(T%Node,1,T%Len)'=T%Root  DO
    . DO I%File.Write(T%Node,@T%Node)
    QUIT
```

## $QUOTE

The **$QUOTE** function returns a string enclosed in quotation marks.

### Format

$QUO{TE} ( expr )

### Arguments

expr - The string to enclose in quotation marks.

### Explanation

The **$QUOTE** command is useful when the **XECUTE** command or the **INDIRECTION ( @ )** operator are used and quotation marks need to be doubled. It is also useful in simplifying the management of nested quotation marks in cases where multiple levels of indirection are necessary.

### Comments

The entire string is enclosed in quotation marks. Any quotation marks inside the string are replaced by two quotation marks.

### Related

INDIRECTION ( @ ) operator

XECUTE command

### Examples

The following example constructs a command that will extract a substring from the variable T%String, starting with T%Start and ending with T%End, placing it in the T%Value variable.

```
X "Set T%Value=$E("_""""_T%String_""""_",T%Start,T%End)"
```

## $RANDOM

The **$RANDOM** function returns a random integer in the specified range.

### Format

$R{ANDOM} ( intexpr )

### Arguments

intexpr - A positive integer specifying the number of possible return values minus 1.

### Explanation

The **$RANDOM** function returns a random integer in the specified range. The argument, always interpreted as an integer, specifies the number of possible return values. If the argument is a positive integer X, the function returns a number between 0 and X-1. If the argument is less than 1, an error occurs.

### Comments

Keep the following points in mind when you use the **$RANDOM** function:

- The return value of this function is not truly random, only arbitrary. Regular patterns can sometimes be detected in extended sequences of so-called random numbers.
- The return value is always an integer 0 or greater, but mathematical operations can be performed on this value to telescope it into any numeric range with any desired distribution frequency or degree of sensitivity, so this limitation is not truly a handicap.

### Examples

The following example generates a random integer between 0 and 99:

```
SET T%Result=$RANDOM(100)
```

The following example generates a random decimal number between –10 and 10, with a possible return value at every one-hundredth interval.

```
SET T%Result=$RANDOM(2001)/100-10
```

The following example generates a random decimal number between 0 and 10, with 100 possible return values. The values are not evenly distributed: most of them are less than 1, but the number 0 is never returned.

```
SET T%Result=10/($RANDOM(100)+1)
```

## $REVERSE

The **$REVERSE** function reverses a string of characters in the reverse order of the string argument.

### Format

$RE{VERSE} ( expr )

### Arguments

expr - A string to be reversed.

### Explanation

The **$REVERSE** function reverses the input string so that the last character in the input string becomes the first character in the result string. The second from the last character in the input string becomes the second character in the result string, and so on, until all characters have been reversed.

### Comments

Keep the following points in mind when you use the **$REVERSE** function:

- The **$REVERSE** function is rarely used. However, turning a string around can make it easier to manipulate.
- **$REVERSE** returns a result identical to the input string when the input string is a single character or a null string.

### Related

$EXTRACT function

$LENGTH function

### Examples

The following example gets the last character of a string.

```
SET T%Result=$EXTRACT($REVERSE(T%Result))
```

The following example shows an alternative way to get the last character of a string.

```
SET T%Result=$EXTRACT(T%Result,$LENGTH(T%Result))
```

The following example gets the last piece of a string.

```
SET T%Result=$REVERSE($PIECE($REVERSE(T%Result),T%Delim))
```

The following example shows an alternative way to get the last piece of a string.

```
SET T%Result=$PIECE(T%Result,T%Delim,$LENGTH(T%Result,T%Delim))
```

The following example determines whether the array node I%Elements(T%Loop) is undefined (does not contain a value) and, if so, exits.

```
IF '$EXTRACT($REVERSE($DATA(I%Elements(T%Loop)))) QUIT
```

Each of the following lines contains an alternative way to exit if the array node I%Elements(T%Loop) is undefined.

```
IF $DATA(I%Elements(T%Loop))[0 QUIT

IF 11'[$DATA(I%Elements(T%Loop)) QUIT

IF $DATA(I%Elements(T%Loop))#2=0 QUIT

IF $DATA(I%Elements(T%Loop))#10=0 QUIT
```

## $SELECT

The **$SELECT** function returns one of several different values, depending on any number of true or false conditions.

### Format

$S{ELECT} ( L tvexpr : expr )

### Arguments

tvexpr - A condition to be evaluated if none of the **$SELECT** arguments to the left of it have true conditions.

expr - An expression to be evaluated and its value returned only if its associated condition is the first true condition.

### Explanation

The **$SELECT** function evaluates its conditions from left to right until a true condition is encountered. At that point, it evaluates the expression associated with this condition, returning its value. Expressions associated with false conditions are never evaluated. Conditions to the right of the first true condition are not evaluated, nor are their associated expressions.

In the following example, **$SELECT** returns the value even based on the true condition of X>0:

```
SET X=2
DO Env.Output($SELECT(X<0:"MINUS",X#2:"ODD",X>0:"EVEN",1:"ZERO"))


Results: even
```

### Comments

Keep the following points in mind when you use the **$SELECT** function:

- **$SELECT** has no effect on **$TEST**.
- **$SELECT** can sometimes replace several lines of code using **IF** and **ELSE**.
- **$SELECT** can sometimes be used instead of the two-argument form of **$GET** with a performance improvement because the default value need not be evaluated if the variable is defined.
- A return value is required, so the last condition must evaluate to true. An error occurs if the last condition does not evaluate to true.

## Related

IF command

ELSE command

$GET function

## Examples

The following example illustrates a typical programming error that can occur because **$TEST** is likely to change between the **IF** and the **ELSE**.

```
    IF I%Height'>I%Width DO
.  DO TEST
    ELSE  DO $Env.Output("Greater")
    QUIT
TEST ; Subroutine containing IF and ELSE
    IF I%Height=I%Width DO $Env.Output("Equal")
    ELSE DO $Env.Output("Not Greater")
    QUIT
```

Assuming that I%Height=5 and I%Width=10, the **IF** command on the first line sets **$TEST** to 1 and the **DO** calls TEST. Inside TEST, the **IF** sets **$TEST** to 0, and the **ELSE** performs a write to the output window. The **QUIT** then exits TEST. The **ELSE** on the second line checks **$TEST** (which is now 0) and performs a write to the output window. The first line of output is "Not Greater" and the second line is "Greater". This is probably not what the programmer intended.

A number of language elements (method and property calls, extrinsic functions, and the argumentless **DO**) place **$TEST** on the stack, avoiding the problem shown in the previous example. Also, postconditionals and the **$SELECT** function can be used to conditionalize certain operations without affecting **$TEST**. The following example solves the previous problem by using the argumentless **DO**:

```
IF I%Height'>I%Width DO
. IF I%Height=I%Width DO $Env.Output("Equal") QUIT
. DO $Env.Output("Not Greater")
ELSE DO $Env.Output("Greater")
QUIT
```

The following example using **$SELECT** is functionally equivalent to the previous example, except that it does not modify **$TEST**.

```
DO
$Env.Output($SEL(I%Height>I%Width:"Greater",I%Height=I%Width:"Equal",1:"Not
Greater"))
```

In the two-argument form of **$GET**, the value of the second argument is always evaluated, (even if the variable is defined). This means that compute-expensive operations should not be placed in the second argument. The following example uses **$GET**, causing the **$CALLBACK** function in the second argument to be called and its value ignored if T%CallBack is undefined.

```
SET T%CallBack=$GET(T%CallBack,$CALLBACK(MODIFY))
```

The following example might be more efficient because it uses **$SELECT**, causing the **$CALLBACK** function to be evaluated only if the variable is undefined.

```
S T%CallBack=$S($D(T%CallBack)#10:T%CallBack,1:$CALLBACK(MODIFY))
```

The following extrinsic variable returns a string containing the approximate time, based on the second comma-piece of **$HOROLOG**. Note the use of **$SELECT**, **$JUSTIFY**, and **$TRANSLATE** in this function.

```
TIME() ; TIME extrinsic variable
    NEW L%Time,L%Hour,L%Minute,L%Meridian
    SET L%Time=$PIECE($HOROLOG,",",2)
    IF L%Time#43200=0 QUIT "12:00"_$SELECT(L%Time:"pm",1:"am")
    SET L%Hour=L%Time\3600
    SET L%Meridian=$SELECT(L%Hour>11:"pm",1:"am")
    SET L%Hour=$JUSTIFY(L%Hour#12,2)
    IF L%Hour=" 0" SET L%Hour=12
    SET L%Minute=$JUSTIFY(L%Time\60#60,2)
    SET L%Time=$TRANS(L%Hour_":"_L%Minute_L%Meridian," ",0)
    QUIT L%Time
```

## $STACK

The **$STACK** function returns information about the underlying M process stack.

### Format

$ST{ACK} ( intexpr {, stackcodeexpr} )

### Arguments

intexpr - Specifies a process stack frame number about which information is desired, or one of the special values 0 or –1.

stackcodeexpr ::= expr V stackcode - Specifies the kind of information that is requested about a specific stack frame.

### Explanation

The information provided by **$STACK** is principally useful in debugging. However, the EsiObjects process stack does not necessarily have any correspondence to the underlying M process stack.

Internal EsiObjects optimizations can alter the behavior of the M process stack for a given operation, and **$STACK** might change in future releases to explicitly support the EsiObjects process stack. For these reasons, caution is advised when using **$STACK**.

In its one-argument form, the argument of **$STACK** determines the type of return value:

| | |
|---|---|
| 1 | Returns the number of frames on the process stack (equivalent to the **$STACK** special variable). |
| 0 | Returns a platform-specific value indicating the way in which the process was originally invoked. |
| N | Assuming n is a positive integer, the name of the command used to create that stack level (in other words, **DO** or **XECUTE**), the string $$ if it is an extrinsic function, or an error code if it is an error frame. If n is greater than the number of stack levels, NULL ("") is returned. |

In its two-argument form, the first argument is a stack frame number and the second argument specifies the specific kind of information to be returned about that stack frame.

| | |
|---|---|
| PLACE | The location of the code that invoked that stack level. If it is the current stack level, then the location of the currently executing command is used. The location is of the general form:<br>**{label} {+intexpr} {^routinename} SP + eoffset**<br>Where **eoffset** is the character position of the place in the line where the stack level was located, but its exact accuracy is not guaranteed. |
| MCODE | A string containing the actual line of code that invoked that stack level. An empty string if the text is not available. |
| ECODE | A list of error codes added (to the **$ECODE** special variable) at that level. |

### Comments

Keep the following points in mind when you use the **$STACK** function:

- **$STACK** is handled by the underlying M platform. Therefore, use caution when interpreting its return values.
- The construct **$STACK(-1)** is equivalent to the special variable **$STACK** and rarely is used.
- The construct **$STACK($STACK)** always returns information about the current M process stack level.

### Related

DO command

XECUTE command

$STACK special variable

### Examples

The following example displays information about the current error condition for every stack frame in **$STACK** that contains error codes.

```
DO $Env.Output("Process Type: ",$STACK(0))
DO $Env.Output("Frames on Stack: ",$STACK)
FOR  T%Loop=1:1:$STACK IF $STACK(T%Loop,"ECODE")'="" DO
. SET T%Code=$STACK(T%Loop,"ECODE")
. SET T%Line=$STACK(T%Loop,"PLACE")
. SET T%Text=$STACK(T%Loop,"MCODE")
. DO $Env.Output("Errors at Frame "_T%Loop_": "_T%Code
. DO $Env.Output("Execution Location: "_T%Line)
. IF T%Text'="" DO $Env.Output(T%Text)
. DO $Env.Output(" ")
QUIT
```

## $TEXT

The **$TEXT** function returns a single line of code from the specified routine or current code body.

### Format

$T{EXT} ( textarg )

### Arguments

| | |
|---|---|
| **textarg ::=** | **+ intexpr [ ^ routineref ]** |
| | **Entryref** |
| | **@ expratom V textarg** |
| **entryref ::=** | **Dlabel [+ intexpr] [^ routineref]** |
| | **^ routineref** |

### Explanation

In EsiObjects, **$TEXT** is handled entirely by the underlying M platform, and applies to the intermediate M code, not to the EsiObjects source. Therefore, it is only reliable to use as follows:

- On lines having a label and a comment beginning with two semicolons
- On any contiguous following lines that contain a comment beginning with two semicolons, and no commands

Note the following:

- The EsiObjects source code on a line is not guaranteed to be present, but may be present in future releases.
- Intermediate M source can appear, but only unreliably.
- There is no direct correlation between the lines in the EsiObjects source and the lines in the M source, except on label lines containing a comment with two semicolons, and on any contiguous following lines containing only a comment with two semicolons.

Many source editors use the TAB character (ASCII character #9) as the line start indicator. However, **$TEXT** always renders the line start indicator as a space (ASCII #32). If the argument of **$TEXT** addresses a line that does not exist, the function returns NULL ("").

Three forms of indirection are allowed with **$TEXT**. The label and routine names, if present, can be indirected. Also, the entire argument of **$TEXT** can be indirected.

## Comments

Keep the following points in mind when you use the **$TEXT** function:

- **$TEXT** allows its own special form of indirection (the entire **$TEXT** argument can be indirected).
- **$TEXT** applies to the underlying M code, not to the EsiObjects source. Therefore, its reliability is restricted. In a future release, **$TEXT** may apply instead to the EsiObjects source.

## Related

Method structure

IndirectionIntroductiontoIndirection

## Examples

The following example uses **$TEXT** to set up an array in I%Elements.

```
INIT(Year) ; Set up I%Elements
    FOR  T%Loop=1:1 DO  QUIT:T%Number=""
     . SET T%Line=$TEXT(+T%Loop)
     . SET T%Number=$PIECE(T%Line,";",3)
     . SET T%Text=$PIECE(T%Line,";",4)
     . IF T%Number="" QUIT
     . SET I%Elements(T%Loop,"Days")=T%Number
     . SET I%Elements(T%Loop,"Name")=T%Text
     IF Year#4=0,Year#100 SET I%Elements(2)=29
     QUIT
     ;
ELEMENTS ;;
     ;;31;January
     ;;28;February
     ;;31;March
     ;;30;April
     ;;31;May
     ;;30;June
     ;;31;July
     ;;31;August
     ;;30;September
     ;;31;October
     ;;30;November
     ;;31;December
     ;;
```

## $TRANSLATE

The **$TRANSLATE** function converts a string by changing all members in one set of characters to the corresponding members of another set of characters.

### Format

$TR{ANSLATE} ( expr1, expr2 {, expr3} )

### Arguments

expr1 - A source string to be converted in some way.

expr2 - A string containing a set of characters to be changed in the source string.

expr3 - A string containing a set of characters to replace the set in the second argument.

### Explanation

The **$TRANSLATE** function performs a character-for-character replacement within a string. It accepts a source string, a set of characters to change from, and a set of characters to change them to. The translated string is the return value. In the return value, all instances of the first character in the ‘from’ string are changed to the first character in the to string, then this process is repeated for the second character in from, and so on.

```
DO $Env.Output($TRANSLATE("Macdonalds","sandMolc","oE i"))
```

```
Results: Ei Eio
```

Any characters in the ‘from’ string that are not opposed in the to string are translated into null strings (in other words, are stripped). If there are ten characters in the ‘from’ string, and only five in the to string, then the last five characters of from are stripped. Any extra characters in to are ignored. If the third argument is omitted, it defaults to NULL ("") and all characters in the second argument are stripped from the source string.

### Comments

Keep the following points in mind when you use the **$TRANSLATE** function:

- **$TRANSLATE** converts individual characters. It is not a search-and-replace function.
- The primary use of **$TRANSLATE** is for doing case conversions. It can also be used to filter illegal characters

### Related

CONTAINS ( [ ) operator

$FNUMBER function

$JUSTIFY function

### Examples

The following extrinsic function shows how to use **$TRANSLATE**. The label XLATE is called with the target string passed in as String and a U (to upper) or L (to lower) is passed in the parameter To.

```
XLATE(String,To) ;
SET Lower="abcdefghijklmnopqrstuvwxyz"
SET Upper="ABCDEFGHIJKLMNOPQRSTUVWXYZ"
IF To="U" QUIT $TRANSLATE(String,Lower,Upper)
QUIT $TRANSLATE(String,Upper,Lower)
```

Sometimes it is necessary to determine whether a string contains a certain character. For example, user input is often not allowed to contain the database delimiter. The **CONTAINS ( [ )** operator is generally useful in such cases. However, it is occasionally necessary to test whether an input string contains one of several characters. In the following example, **$TRANSLATE** is used to determine whether the input string STR contains one of five illegal delimiter characters.

```
ILLEGDLM(STR) ; Return 1 if STR contains delimiters ~`|\^, 0 if not
    QUIT STR=$TRANSLATE(STR,"~`|\^")
```

The following example uses **$FNUMBER** with **$TRANSLATE** to format a number with periods instead of commas in the thousands places and a comma instead of a period as the decimal indicator.

```
DO $Env.Output($TRANSLATE($FNUMBER(6543210.987,",",2),".,",",."))


Results: 6.543.210,99
```

The following example uses **$FNUMBER** with **$TRANSLATE** to format a negative number with square brackets (instead of a minus sign).

```
DO $Env.Output($TRANSLATE($FNUMBER(-43210,"P"),"()","[]"))


Results: [43210]
```

The following example uses **$JUSTIFY** with **$TRANSLATE** to pad a number with leading zeros so that it is 5 characters wide.

```
DO $Env.Output($TRANSLATE($JUSTIFY(123,5),0," "))


Results: 00123
```

The following extrinsic variable returns a string containing the approximate time, based on the second comma-piece of **$HOROLOG**. Note the use of **$SELECT**, **$JUSTIFY**, and **$TRANSLATE** in this function.

```
TIME() ; TIME extrinsic variable
    NEW L%Time,L%Hour,L%Minute,L%Meridian
    SET L%Time=$PIECE($HOROLOG,",",2)
    IF L%Time#43200=0 QUIT "12:00"_$SELECT(L%Time:"pm",1:"am")
    SET L%Hour=L%Time\3600
    SET L%Meridian=$SELECT(L%Hour>11:"pm",1:"am")
    SET L%Hour=$JUSTIFY(L%Hour#12,2)
    IF L%Hour=" 0" SET L%Hour=12
    SET L%Minute=$JUSTIFY(L%Time\60#60,2)
    SET L%Time=$TRANS(L%Hour_":"_L%Minute_L%Meridian," ",0)
    QUIT L%Time
```

## $VALID

The **$VALID** function determines whether a certain value is valid for a given property, or whether a certain property/method is valid for an object.

### Format

$VALID ( validmpr {, expr} )

### Arguments

| | |
|---|---|
| **validmpr ::=** | **Object . service** |
| | **Expr** |

validmpr - An expression or a reference to an object's method/property that is to be tested.

expr - A value that might possibly be assigned to the property, whose validity is to be tested.

### Explanation

If the first argument of **$VALID** is an expression, then the type is a string and the function returns true. If an object with service is specified, the return value is based on the validity of the service or property assignment value. If the second argument is specified, then the function returns true if its value could be assigned validly to the object.

### Comments

If the first argument is not an expression, then the object's $Valid accessor is invoked to compute the return value of **$VALID**. This means that the return value, while expected to be true or false as appropriate, can vary according to the intent of the programmer who wrote the accessor method.

### Related

$NORMALIZE function

### Examples

The following code attempts to verify that the value in T%SSN constitutes an acceptable value for the Social Security Number property for employee object bound to the T%Employee variable before assigning it that value.

```
IF $VALID(T%Employee.SSN,T%SSN) SET T%Employee.SSN=T%SSN
```

## $WALK

The **$WALK** function allows privileged code to traverse the entries in a symbol table, optionally including their subscripts.

### Format

$WALK ( name, subscripts, typexpr {, direction} )

### Arguments

name - The name of the variable.

subscripts - An expression whose value is an entire subscript list, including the parentheses.

| | | |
|---|---|---|
| | **A** | Accessor |
| | **C** | Class |
| | **CN** | Constant |
| | **G** | Global |
| | **I** | Instance |
| **typexpr ::= expr V** | **L** | Local |
| | **N** | NamePool |
| | **P** | Parameter |
| | **S** | System |
| | **U** | Universal |

direction - The direction of the search. Use 1 for searching forwards (default) and –1 for searching backwards.

### Explanation

The **$WALK** function allows privileged code to traverse the entries in a symbol table, optionally including their subscripts, similar to **$ORDER**. It can be used with any EsiObjects symbol, including templates and name pools. However, this function is not recommended for general use in EsiObjects.

### Comments

Keep the following points in mind when you use the **$WALK** function:

- **$WALK** is used to loop through all the instance variables of an object, all the accessor variables of a method, and so on.
- Once the names of the variables have been obtained with **$WALK**, their values can be referenced with **$LOOKUP**.

### Related

FOR command

$LOOKUP function

$ORDER function

### Examples

The following example contains a **FOR** loop used to traverse the names of an object's instance variables with **$WALK**, while **$LOOKUP** is used to get their values (for those that have simple values).

```
SET T%Loop=""
FOR  SET T%Loop=$WALK(T%Loop,"","I") QUIT:T%Loop=""  DO
. DO $Env.Output("Var: "_T%Loop_", Val: "_$LOOKUP(T%Loop,"","I"))
QUIT
```

## $WATCHDETECT

The **$WATCHDECTECT** function allows an object to detect when it is being watched.

### Format

$WATCH{DETECT} ( on/off, vector )

### Arguments

on/off - Returns 0 or 1.

vector ::= label {^{interface::} method} - The callback label to invoke when a watch is established or ignored.

### Explanation

Because an object can be watched, it allows an object to tailor its behavior. **$WATCHDETECT** returns true if it succeeds. Any watch or ignore causes the callback vector to be invoked, which passes the event or property being watched.

The format of the watch detect callback parameters is as follows:

label(obsrvobj,desc,event,state)

where:

| | |
|---|---|
| **Obsrvobj** | is the object that is observing the current object |
| **Desc** | is a description string that has the following format: |
| | "$SYSTEM>$HOOK" |
| **Event** | is the name of the event of property |
| **State** | is 1 if watched, 0 if ignored |

### Comments

Keep the following points in mind when you use the **$WATCHDETECT** function:

- Any watch established prior to enabling **$WATCHDECT** is not seen.
- The best place to establish a watch detection is in the CREATE method.
- The best place to turn off watch detection is during the DESTROY method.

### Related

WATCH command

### Examples

The following example establishes watch detection during the creation of an object.

```
;CREATE Method
IF $WATCHDETECT(1,Entry^ONWATCH)
; Set up Internal State
QUIT
```

The following example establishes a watch detection during the creation of an object. The OnWatch method is used to track what events are being watched on the current object.

```
;OnWatch Method
ENTRY(Observer,Description,Event,State)
    IF STATE DO
    . SET I%Active(Event,Observer)=""
    ELSE  DO
    . KILL I%Active(Event,Observer)
```

## $ZLENGTH

The **$ZLENGTH** function returns the number of subscripts in a string containing an array reference.

### Format

$ZL{ENGTH} ( expr1, expr2 )

### Arguments

expr1 - A string in which the number of pieces is to be measured (ignores strings enclosed in quotation marks).

expr2 - The delimiter used to break the string apart.

### Explanation

The **$ZLENGTH** function counts the number of pieces in a string. The second argument is a delimiter used to divide the string into pieces. In the string, anything enclosed in balanced quotation marks is ignored when counting pieces. Delimiters are usually one character in length. The only limit to the length of a delimiter is the maximum string size.

The number of pieces in a string is similar to the number of words in a sentence. In the following example, the space character is used as a delimiter and the total number of pieces equals 4:

```
John dropped the ball.
```

Note that in the following string, the parts enclosed in quotation marks are ignored when counting delimiters:

```
"Don't look now," Sally said, "but John dropped the ball."
```

In the previous example (the space character is the delimiter), the total number of pieces equals 4.

The number of pieces is always equal to the number of nonoverlapping instances of the delimiter not found inside quotation marks plus 1. The following table contains additional examples.

| **String** | **Delimiter** | **Number of Pieces** |
|---|---|---|
| one,"two",three | , | 3 |
| ^A("A,B",1,2,3) | , | 4 |
| ^B(2,"Hello, Bye",3,"7,3,2") | , | 4 |
| A/"B/C"/D/"E"/F"/G/H/I" | / | 5 |

If the second argument is specified as NULL (""), then the return value is always 0.

### Comments

The behavior of this function is unspecified in cases where the delimiter contains one or more quotation marks.

### Related

$LENGTH function

$QLENGTH function

$ZPIECE function

### Examples

In the following example, the **FOR** loop displays all the comma-delimited pieces in the string contained in L%String to the output window.

```
FOR T%Loop=1:1:$ZLENGTH(L%String,",") DO
.  DO $Env.Output($ZP(L%String,",",T%Loop))
```

The following example removes the first space-delimited piece from the string contained in L%String:

```
SET L%String=$ZPIECE(L%String," ",2,$QLENGTH(L%String," "))
```

## $ZPIECE

The **$ZPIECE** function returns one or more pieces from a delimited string (ignoring nested strings).

### Format

$ZPI{ECE} ( expr1, expr2 {, intexpr1 {, intexpr2} )

### Arguments

expr1 - A delimited string from which one or more pieces are to be returned.

expr2 - The delimiter used to break the string apart.

intexpr1 - The starting piece position to return.

intexpr2 - The ending piece position to return.

### Explanation

The second argument is a delimiter used to divide the string into pieces. Inside the string, anything enclosed in balanced quotation marks is ignored when identifying pieces. Most delimiters are usually one character in length, but the only limit to their length is the maximum string size.

The number of pieces is similar to the number of words in a sentence. In the following example, the space character is used as a delimiter:

```
John dropped the ball.
```

In the previous example, the total number of pieces is four. The first piece is John and the last piece is Ball. Note that in the following string, the parts enclosed in the quotation marks are ignored when counting delimiters:

```
"Don't look now," Sally said, "but John dropped the ball."
```

In the previous example, using the space as a delimiter, the total number of pieces is four.

The number of pieces is always equal to the number of nonoverlapping instances of the delimiter not found inside quotation marks, plus 1. The following table contains additional examples.

| String | Delimiter | Number of Pieces | First Piece | Last Piece |
|---|---|---|---|---|
| one,"two",three | , | 3 | one | Three |
| ^A("A,B",1,2,3) | , | 4 | ^A("A,B" | 3) |
| ^B(2,"Hello, Bye",3,"7,3,2") | , | 4 | ^B(2 | "7,3,2") |
| A/"B/C"/D/"E"/F"/G/H/I" | / | 5 | A | F"/G/H/I" |

If the second argument is specified as NULL (""), then the return value is always ("").

## Comments

The behavior of this function is unspecified in cases where the delimiter contains one or more quotation marks.

## Related

$PIECE function

$QSUBSCRIPT function

$ZLENGTH function

## Examples

The following **FOR** loop displays the root node and all the subscripts of the array referenced in T%Target.

```
FOR T%Loop=0:1:$QLENGTH(T%Target) DO
.  DO $Env.Output($QSUBSCRIPT(T%Target,T%Loop))
```

The following example removes the last subscript from the array node in T%Target.

```
SET T%Target=$NAME(T%Target,$QLENGTH(T%Target)-1)
```

## Operators

Operators are symbolic characters that specify the operation to be performed and the type of value to be produced from their associated operand or operands. This section describes the following types of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- String Operator
- Indirection Operator

The following table contains a list of the operators supported by EsiObjects.

## Arithmetic Operators

Arithmetic operators perform arithmetic operations. The following table illustrates those arithmetic operators supported by EsiObjects.

| Operator | Syntax |
|---|---|
| Binary ADD | **A+B** |
| Binary DIVIDE | **A/B** |
| Binary EXPONENTIATION | **A******B** |
| Binary INTEGER DIVIDE | **A\B** |
| Unary MINUS | **-A** |
| Binary MODULO | **A#B** |
| Binary MULTIPLY | **A*B** |
| Unary PLUS | **+B** |
| Binary SUBTRACT | **A-B** |

## Binary ADD ( + )

The binary **ADD** operator produces the sum of two numerically interpreted operands.

### Format

operand + operand

### Explanation

Binary **ADD** uses any leading, valid numeric characters (the digits 0 to 9, the decimal point, the unary **MINUS** operator, the unary **PLUS** operator, and the letter E) as the numeric values of operands. Then binary **ADD** produces a value that is the sum of the value of the operands. If an operand has no leading numeric characters, binary **ADD** gives it a value of 0.

### Related

Expression evaluationEvaluatingExpressions

### Examples

The following example shows string arithmetic on two operands that have leading digits.

```
DO $Env.Output("3 APPLES"+"8 ORANGES")


Results: 11
```

The following example performs addition on two real, numeric literals.

```
DO $Env.Output(1044.368+91.36)

Results: 1135.728
```

The following example performs addition on two defined local variables.

```
SET A=3.01,B=92.7 DO $Env.Output(A+B)

Results: 95.71
```

The following example illustrates that leading zeroes on a numerically evaluated operand do not affect the results the operator produces. It also shows operands without leading numerics.

```
DO $Env.Output("007" + 100 + "One" + " 10")

Results: 107
```

The following example returns the total number of elements in two collections (PRQUEUE1 and PRQUEUE2), which hold items that are going to be printed. The method TotalElements returns the number of items in the queue.

```
DO $Env.Output(PRQUEUE1.TotalElements+PRQUEUE2.TotalElements)
```

## Binary DIVIDE ( / )

The binary **DIVIDE** operator produces the quotient that is the result of dividing two numerically interpreted operands.

### Format

operand A/operand B

### Parameters

operand A - the dividend

operand B - the divisor

### Explanation

Binary **DIVIDE** uses any leading, valid numeric characters (the digits 0 to 9, the decimal point, the unary **MINUS** operator, the unary **PLUS** operator, and the letter E) as the numeric values of the operands. Then it produces a quotient that is the result of dividing operand A by operand B. If an operand has no leading numeric characters, binary **DIVIDE** assumes its value to be 0.

### Related

Expression evaluation

Binary MODULO ( # ) operator

### Examples

The following example divides two integer numeric literals.

```
SET QPZ=355/113
DO $Env.Output(QPZ)
3.1415929035398
```

The following example performs division on operands with leading digits.

```
DO $Env.Output("16 PIES"/"4 PEOPLE")

Results: 4

The following example gets the number of errors logged in the Errorlog object
since January 1, 1995. It divides the number of errors by the number of days
since January 1, 1995 to get the average number of errors per day.

SET T%Ave=Errorlog.Count(from:"1/1/95")/Date.DaysSince("1/1/95)
```

## Binary EXPONENTIATION ( ** )

The binary **EXPONENTIATION** operator produces the exponentiated value of operand A raised to the power of operand B.

### Format

operand A**operand B

### Parameters

operand A - the operand designated as the base

operand B - the operand designated as the exponent

### Explanation

Binary **EXPONENTIATION** uses any leading numeric characters (the digits 0 to 9, the unary **MINUS** operator, the decimal point, and the letter E) as the numeric values of the operands. Then it produces a result that is operand A raised to the power of operand B. If an operand has no leading numeric characters, binary **EXPONENTIATION** assigns it a value of 0. If you attempt to raise a negative number to a non-integer power, an error occurs.

### Related

Expression evaluation

### Examples

The following example shows how to use exponentiation to find the square root of a number

```
DO $Env.Output(16**.5)

Results: 4
```

## Binary INTEGER DIVIDE ( \ )

The binary **INTEGER DIVIDE** operator produces the integer result of the division of operand A by operand B.

### Format

operand A\operand B

### Parameters

operand A - The dividend

operand B - The divisor

### Explanation

Binary **INTEGER DIVIDE** uses leading, valid numeric characters (the digits 0 to 9, the unary **MINUS** operator, the unary **PLUS** operator, the decimal point, and the letter E) as the values of the operands. Then binary **INTEGER DIVIDE** produces a result that is the integer portion of the quotient of the division of operand A by operand B. It does not return a remainder and it does not round up the result.

If an operand has no leading numeric characters, its value is assumed to be 0. An error occurs if you perform integer division with a zero-valued divisor.

### Related

Expression evaluation

### Examples

The following example performs integer division on two real numeric operands.

```
DO $Env.Output(27.82\16.39767)
```

```
Results: 1
```

The following example uses binary **ADD** and binary **INTEGER DIVIDE** to perform the rounding up operation to the nearest integer.

```
SET X=9.996
DO $Env.Output(X+.5\1)
```

```
Results: 10
```

The following example uses a function to round a value to a given level of precision.

```
RND(V,Prc)  ; Rounds to a level of decimal point
    SET Prcvl=10**(Prc\1)
    QUIT (V*Prcvl+.5)\1/Prcvl
```

The following example rounds a value to the penny.

```
SET AVG=199.748632
SET ADJAVG=((AVG*100+.5)\1)/100
DO $Env.Output(ADJAVG=199.75)
```

## Unary MINUS ( – )

The unary **MINUS** operator negates an operand's numeric interpretation.

### Format

–operand

### Explanation

Unary **MINUS** uses any leading, valid numeric characters (the digits 0 to 9, the decimal point, the unary **PLUS** operator, other unary **MINUS** operators, and the letter E) as the numeric value of the operand. Unary **MINUS** then returns the additive inverse of this numeric value. If the string has no leading numeric characters, unary **MINUS** assigns the string a numeric value of 0.

Because unary **MINUS** returns a numeric value, it can be used to compare a string expression to a numeric literal or other numeric expression.

### Comments

Unary **MINUS** operator takes precedence over the binary arithmetic operators. A numeric expression is scanned and any unary **MINUS** operator to the operand on its right is applied. Then the expression is evaluated and a result is produced.

### Related

Expression evaluation

$FNUMBER function

### Examples

In the following example, parentheses take precedence over unary operators. The string in the parentheses is treated as one value (12BOATROPES). When the operand is interpreted numerically, the string is scanned, a numeric value of 12 is encountered, and then interpretation stops. The unary **MINUS** operator is applied to this value and returns a value of –12.

```
DO $Env.Output(-("12BOAT"_"ROPES"))
```

```
Results
-12
```

Multiple unary **MINUS** operators with an operand are applied in a right-to-left order. The following examples show the use of binary **SUBTRACT** and multiple unary **MINUS** operators:

```
DO $Env.Output(18---10)


Results: 8
```

```
DO $Env.Output(18----10)


Results: 28
```

The following example reverses the sign of a numeric literal.

```
DO $Env.Output(-+100)
```

```
Results: -100
```

## Binary MODULO ( # )

The binary **MODULO** operator produces the value of an arithmetic modulo operation on two numerically interpreted operands.

### Format

operand A#operand B

### Parameters

operand A - the value on which the modulo operation is to be performed

operand B - the modulus

### Explanation

Binary **MODULO** uses any leading, valid numeric characters (the digits 0 to 9, the decimal point, the unary **MINUS** operator, the unary **PLUS** operator, and the letter E) as the numeric values of the operands. If an operand has no leading numeric characters, binary **MODULO** gives it a value of 0.

When the operands A and B are both positive, then the modulo operation is the remainder of operand A integer divided by operand B.

### Comments

Keep the following points in mind when you use the binary **MODULO** operator:

- The operation is not defined if operand B is zero-valued.
- The operation returns a 0 if operand A is zero-valued.
- When both operands are positive, the modulo operation produces the remainder of the integer division of operand A by operand B. This is not true if either operand is negative.

### Related

Expression evaluation

Binary INTEGER DIVIDE ( \ )

## Examples

The following examples illustrate the modulo operation with two positive operands. The modulo operation produces a value equivalent to the remainder after division of operand A by operand B.

```
DO $Env.Output(47#10)

Results: 7

DO $Env.Output(24#6)

Results: 0

DO $Env.Output(15.76#5.5)


Results: 4.76
```

The following examples illustrate the effect of the modulo operater on two operands preceded with unary **MINUS** operators. The modulo operation is equivalent to the following:

-(operand A#operand B)

```
DO $Env.Output(-47#-10)


Results: -7

DO $Env.Output(-24#-6)


Results: 0
```

The following examples show the effect of a unary **MINUS** on operand B. The expression has the value -(-operand A#operand B).

```
DO $Env.Output(47#-10)


Results: -3

DO $Env.Output(24#-6)

Results: 0
```

The following examples show the effect of a zero-valued operand A. The result is 0 regardless of the sign of operand B. When operand B evaluates to 0, the operation is undefined and results in an error.

```
DO $Env.Output("ALPHA"#10)

Results: 0

DO $Env.Output(0#-10)


Results: 0
```

## Binary MULTIPLY ( * )

The binary **MULTIPLY** operator returns the product of two numerically interpreted operands.

### Format

operand*operand

### Explanation

Binary **MULTIPLY** uses any leading numeric characters (the digits 0 to 9, the unary **MINUS** operator, the decimal point, and the letter E) as the numeric values of the operands. Then it produces a result that is the product of the two operands. If an operand has no leading numeric characters, binary **MULTIPLY** assigns it a value of zero.

### Related

Expression evaluation

### Examples

The following example multiplies two string operands with leading digits.

```
DO $Env.Output("2 Years"*"8 Workers")

Results: 16
```

The following example multiplies one string literal and one numeric literal.

```
DO $Env.Output("7.5"*.5)


Results: 3.75
```

The following example multiplies the values in two local variables.

```
SET H=10,W=12 DO $Env.Output("Area: "_H*W)

Results: 120
```

## Unary PLUS ( + )

The unary **PLUS** operator gives its operand a numeric interpretation.

### Format

+operand

### Explanation

Unary **PLUS** gives its operand a numeric interpretation. It uses any leading, valid numeric characters (the digits 0 to 9, the decimal point, the unary **MINUS** operator, another unary **PLUS** operator, and the letter E) to determine the numeric value of the operand.

### Comments

Because unary **PLUS** returns a canonic representation of the value, you can use it to ensure that a given subscript is stored as a numeric subscript.

### Related

Expression evaluation

$FNUMBER function

### Examples

The following example evaluates a string value as a numeric value.

```
DO $Env.Output(+"0030")
```

```
Results: 30
```

The following example evaluates a string value as a numeric value. Because the string literal does not contain any leading numeric characters, its numeric value is 0.

```
DO $Env.Output(+"ABCDEFG")
```

```
Results: 0
```

## Binary SUBTRACT ( - )

The binary **SUBTRACT** operator produces the difference between two numerically interpreted operands.

### Format

operand A-operand B

### Parameters

operand A - the minuend (the value from which operand B is to be subtracted)

operand B - the subtrahend (the value to be subtracted from the minuend)

### Explanation

Binary **SUBTRACT** interprets any leading, valid numeric characters (the digits 0 to 9, the decimal point, the unary **MINUS** operator, the unary **PLUS** operator, or the letter E) as the numeric values of the operands. Then it produces a value that remains after subtraction. If an operand has no leading numeric characters, binary **SUBTRACT** assumes its value to be 0.

### Related

Expression evaluation

### Examples

The following example subtracts a real numeric literal from an integer numeric literal.

```
DO $Env.Output(12.756-3.75)
```

```
Results: 9.006
```

The following example performs subtraction on two literals with leading digits.

```
DO $Env.Output("12 APPLES"-"4 ORANGES")


Results: 8
```

The following example subtracts quota from the total sales amount in the salesperson object.

```
SET T%Perf=Salesperson.TotalSalesAmount-SalesPerson.Quota
```

## Relational Operators

Relational operators perform relationship operations. The following table illustrates those relational operators supported by EsiObjects.

| **Operator** | **Syntax** |
|---|---|
| Binary CONTAINS | **A[B** |
| Binary EQUALS | **A=B** |
| Binary FOLLOWS | **A]B** |
| Binary GREATER THAN | **A>B** |
| Binary LESS THAN | **A<B** |
| Binary PATTERN MATCH | **A?PATTERN** |
| Binary SORTS AFTER | **A]]B** |

## Binary CONTAINS ( [ )

The binary **CONTAINS** operator tests whether the sequence of characters in the right operand is a substring of the left operand.

### Format

operand A[operand B

### Parameters

operand A - the operand being tested to determine if it contains operand B

operand B - the operand being tested to determine if it is contained in operand A

### Explanation

Binary **CONTAINS** treats operands as string values and gives them no special interpretation. Binary **CONTAINS** returns true if operand A contains the character string represented by operand B. It returns false if operand A does not contain the character string represented by operand B.

To produce a true result, the characters in operand B must be in the same order as the characters in the substring of operand A. If operand B is the null string, binary **CONTAINS** always produces a result of true.

### Comments

Use the unary **NOT** operator with binary **CONTAINS** to produce a negative **CONTAINS** (**DOES NOT CONTAIN**). You can express **DOES NOT CONTAIN** using either of the following equivalent formats:

operand A'[operand B

'(operand A[operand B)

**DOES NOT CONTAIN** reverses the truth value of binary **CONTAINS** applied to both operands. **DOES NOT CONTAIN** produces a true result if operand A does not contain the character string represented by operand B. It produces a false result if operand A does contain the character string represented by operand B.

## Examples

The following example tests whether D contains C. Because D does contain C, the result is true.

```
SET C="MOTOR",D="MOTORCYCLE"
DO $Env.Output(D[C)


Results: 1
```

The following example shows how to use **DOES NOT CONTAIN** to determine if a string is not a substring of another string.

```
SET CODE ="- OK - Operation Successful!"
IF CODE '["- OK -" GOTO ERROR
```

The following examples show that all strings contain the null string, even the null string.

```
DO $Env.Output("CATALOG"["")


Results: 1

DO $Env.Output(""[""))


Results: 1
```

The following example checks if the file object's PrivString property contains a D and if so it destroys the FileObject. If PrivString does not contain a D, the example asserts a message that it cannot delete the object.

```
IF T%Command="Delete" DO
. IF FileObj.PrivString["D" DESTROY FileObj
. ELSE  DO $ENVIRONMENT.Assert:("No privs for delete operation.")
```

## Binary EQUALS

The binary **EQUALS** operator compares two operands for equality.

### Format

operand=operand

### Explanation

Binary **EQUALS** returns a result of true if the two operands are identical strings; otherwise, it returns a result of false.

To produce a true result, the character sequence in both operands must be identical. There can be no intervening characters (including spaces). Binary **EQUALS** does not imply any numeric interpretation of either operand.

You can use binary **EQUALS** to test for numeric equality if both operands have numeric values. The following example produces a result of true:

```
DO $Env.Output(07=7)
```

```
Results: 1
```

If the operands are not automatically converted to numeric values (as in the process of evaluating numeric literals), you can force the conversion by using the unary **PLUS** operator. The following example produces a result of true:

```
DO $Env.Output(+"007"="7")
```

```
Results: 1
```

The following statement does not set both A and B to 7:

```
SET A=B=7
```

The previous statement sets A equal to true if the value of B is 7. A is set to false if B has some other value. If you want to set both A and B equal to 7, do the following:

```
SET (A,B)=7
```

### Comments

Keep the following points in mind when you use the binary **EQUALS** operator:

- You can specify a **NOT EQUALS** operation by using the unary **NOT** operator with binary **EQUALS**. You can express the **NOT EQUALS** operation in two ways:
- operand'=operand
- '(operand=operand)

- **NOT EQUALS** reverses the truth-value of the **EQUALS** operator applied to both operands. If the two operands are not identical, the result is true. If the two operands are identical, the result is false.
- With the **SET** command, the equal sign becomes an assignment operator that indicates the assignment of the value of the right operand to the left operand.
- For example, the following statement sets variable A equal to Channel Islands:

```
SET A="Channel Islands"
```

## Related

Expression evaluation

## Examples

The following example checks a string to see if it is empty.

```
IF STR="" DO $Env.Output("Empty String")
```

The following example illustrates two uses of the equal sign. First, the example uses the equal sign with the **SET** command to give two local variables the value of two strings. Second, the example tests the identify of the strings using binary **EQUALS**. Because the strings are not identical, the result is false.

```
SET A="A56BC",B="ABC" DO $Env.Output(A=B)


Results: 0
```

The following example asserts the Yes/No dialog box and the user can click on a Yes or No button. The Assert method passes back a "Yes" or "No" and it is compared to the literal "Yes". The result of the comparison is 1 if the user selects the Yes button or 0 if the No button is selected. The value is displayed in the environments output window.

Note: The Assert method's parameters are passed in by keyword. The Assert method is found in the ESI library, in the primary interface of the Environment class.

```
DO $Env.Output($Env.Assert(Text:"Save File?",Buttons:"Yes")="Yes")
```

## Binary FOLLOWS ( ] )

The binary **FOLLOWS** operator tests whether the characters in the left operand come after the characters in the right operand in ASCII collating sequence.

### Format

operand A]operand B

### Parameters

operand A - the operand being tested to see if its characters follow the characters in operand B in ascending ASCII collating sequence

operand B - the operand that operand A is being compared to

### Explanation

The binary **FOLLOWS** test compares the ASCII characters in both operands starting with the leftmost character. The test ends when a character is found in operand A that is different from the character at the corresponding position in operand B, or when there are no characters left to compare in either of the operands.

Binary **FOLLOWS** produces a result of true if the first differing in operand A has a higher ASCII value than the corresponding character in operand B (that is, if the character in operand A comes after the character in operand B in ASCII collating sequence). It produces a result of false if the first unique character in operand A has a lower ASCII value than the corresponding character in operand B.

If operand A is identical to operand B, then a truth-value of false is returned. If operand A is shorter than operand B, but otherwise identical, then a truth-value of false is returned. If operand B is shorter than operand A, but otherwise identical, then a truth value of true is returned.

### Comments

You can produce a **NOT FOLLOWS** operation by using the unary **NOT** operator with binary **FOLLOWS**. You can express **NOT FOLLOWS** using either of the following equivalent formats:

operand A']operand B

'(operand A]operand B)

**NOT FOLLOWS** reverses the truth-value of binary **FOLLOWS** applied to both operands. If all characters in the operands are identical, or if the first unique character in operand A has a lower ASCII value than the corresponding character in operand B, **NOT FOLLOWS** returns a result of true. If the first unique character in operand A has a higher ASCII value than the corresponding character in operand B, **NOT FOLLOWS** returns a result of false.

If operand A is shorter than operand B, but otherwise identical, **NOT FOLLOWS** returns a value of true. If operand B is shorter than operand A, but otherwise identical, **NOT FOLLOWS** returns a value of false.

## Related

Expression evaluation

## Examples

The following example tests to determine if the string HARPOON follows the string HARP in ASCII collating order. The result is true.

```
DO $Env.Output("HARPOON"]"HARP")
```

```
Results: 1
```

The following example tests the collating order of numeric literals. Because 3 in 123 follows the corresponding 2 in 122, the result is true.

```
DO $Env.Output(123]122)
```

```
Results: 1
```

The following example also tests numeric literals. Because the numeric literal 12 collates before the numeric literal 2, the result is false.

```
DO $Env.Output(12]2)
```

```
Results: 0
```

The following example tests whether the string CDE follows string ABC in ASCII collating order. Because C in CDE follows A in ABC, the result is true.

```
DO $Env.Output("CDE"]"ABC")
```

```
Results: 1
```

The following uses binary **FOLLOWS** to test for a non-null string. Because any string, except the null string, follows the null, the expression T%Value]"" is true whenever T%Value is non-null.

QUIT:T%Value]""

## Binary LESS THAN ( < )

The binary **LESS THAN** operator tests whether operand A is numerically less than operand B.

### Format

operand A<operand B

### Parameters

operand A - the operand considered the smaller

operand B - the operand considered the larger

### Explanation

The binary **LESS THAN** returns a result of true if operand A has a lesser numeric value than operand B. It returns a result of false if operand A has an equal or greater numeric value than operand B.

### Comments

You can produce a **NOT LESS THAN** operation by using the unary **NOT** operator with binary **LESS THAN** as follows:

operand A'<operand B

'(operand A<operand B)

**NOT LESS THAN** reverses the truth value of binary **LESS THAN** applied to both operands. It produces a result of true when operand A is greater than operand B or when operand A is equal to operand B. It produces a result of false when operand A is less than operand B.

You can use the **NOT LESS THAN** operation to specify greater than or equal to.

### Related

Expression evaluation

## Examples

The following example shows the result of using a series of relational operators. All expressions with binary operators are evaluated left to right. The first operation is 100<X, the result of which is 0. The second operation is 0<10, the result of which is 1.

```
SET X=0
DO $Env.Output(100<X<10)
```

```
Results: 1
```

The following example verifies that the password entered is a least 5 characters in length.

```
IF PwControl.textLength<5 $ENVIRONMENT.Assert("Password must contain at least
5 characters") QUIT
```

## Binary GREATER THAN ( > )

The binary **GREATER THAN** operator tests whether operand A is numerically greater than operand B.

### Format

operand A>operand B

### Parameters

operand A - the operand considered the larger

operand B - the operand considered the smaller

### Explanation

The binary **GREATER THAN** operator evaluates the two operands numerically. Binary **GREATER THAN** produces a result of true if operand A is numerically larger than operand B. It produces a result of false if operand A is numerically equal to or smaller than operand B.

### Comments

Use the unary **NOT** operator with binary **GREATER THAN** to produce a **NOT GREATER THAN** operation. You can express **NOT GREATER THAN** using either of the following equivalent formats:

operand A'>operand B

'(operand A>operand B)

**NOT GREATER THAN** reverses the truth value of binary **GREATER THAN** applied to both operands. You can use it to specify less than or equal to. **NOT GREATER THAN** produces a true result when operand A is less than operand B or operand A is equal to operand B. **NOT GREATER THAN** produces a false result when operand A is greater than operand B.

### Related

Expression evaluation

## Example

The following example tests two numeric literals.

```
DO $Env.Output(1900>1950)

Results: 0
```

The following example tests two variables with the **NOT GREATER THAN** operator. Because both variables have an identical numerical value, the result is true.

```
SET A="99",B="112"

DO $Env.Output(A'>B)

Results: 1
```

The following example tests if the number of elements in the buffer is greater than 5 and if it is, then removes one element.

```
IF Buffer.TotalElements>5 SET T%Obj=Buffer.Remove
```

## Binary PATTERN MATCH (?)

The binary **PATTERN MATCH** operator tests whether the string of characters in the left operand is correctly specified by the pattern in the right operand.

### Format

operand?pattern

### Parameters

operand - The string whose characters you want to test for a pattern

pattern - Can be one of the following:

a sequence of one or more **patatoms**

@expr_atom

where:

| | |
|---|---|
| **Patatom** | can be one of the following:<br>**repcount patcharacter {...}**<br>**repcount string_literal**<br>**repcount alternation**<br>where:<br>**repcount** — is a repeat count<br>**patcharacter** — is a pattern code character (a character that represents a group of ASCII characters)<br>**string_literal** — is a quoted string literal<br>**alternation** — is a set of patatom sequences to choose from to pattern match a segment of the operand string (provides logical **OR** capability in pattern specifications) |
| **@expr_atom** | is an indirect reference that evaluates to a sequence of one or more patatoms |

### Explanation

Binary **PATTERN MATCH** returns true when the pattern correctly specifies the pattern of characters in the operand and returns false if the pattern does not correctly specify the pattern of characters in the operand.

You can produce a **NOT MATCH** operation by using the unary **NOT** operator with binary **PATTERN MATCH** as follows:

operand'?pattern
'(operand?pattern)

**NOT MATCH** reverses the truth-value of binary **PATTERN MATCH**. If the characters in the operand cannot be fully described by the pattern, then **NOT MATCH** returns a result of true. If the pattern matches all of the characters in the operand, then **NOT MATCHES** returns a result of false.

The binary **PATTERN MATCH** operation is performed by comparing the characters in the operand string against their expected values as described by the pattern.

- Repeat count
- Pattern Code Characters
- String literals
- Alternations

An alternation has the following syntax:

(patatom sequence {, patatom sequence} . . .)

## Comments

If a pattern match successfully describes only part of a string, then the pattern match returns a result of false. That is, there cannot be any string left over when the pattern is exhausted. The following expression evaluates to a result of false because the pattern does not match the final R:

```
"SUSHI BAR"?.U1P2U
```

## Related

Expression evaluation

Pattern indirection

INDIRECTION (@) construct

## Examples

The following example produces a result of true. The string tested includes two numeric characters, one punctuation character, two numeric characters, one punctuation character, and two numeric characters.

```
DO $Env.Output("10/26/72"?2N1P2N1P2N)
```

```
Results: 1
```

The following example produces a result of false. The first character in the tested string is 1 of the 52 uppercase and lowercase alphabetics, but the test has no provision for 2 characters.

```
SET B="LA" DO $Env.Output(B?1A)
```

```
Results: 0
```

The following example produces a result of true.

```
DO $Env.Output("3672STK-0067"?2.N.3U1P2.4N)
```

The following example produces a result of false. The first two characters are alphanumeric, but the third character is punctuation.

```
DO $Env.Output("B4*"?3AN)
```

```
Results: 0
```

The following example produces a result of true. The string STK matches the first three characters, and the 1.E matches the remainder.

```
DO $Env.Output("STK-0037"?1"STK"1.E)
```

```
Results: 1
```

The following example checks the text entered in control to ensure that it matches the specified pattern (one or more alphabetic characters, a comma, and one or more alphabetic characters).

```
IF FullUsernmControl.Text'?(1.A1","1.A) DO $Env.Assert("Invalid Format") QUIT
```

## Pattern Code Characters

The following table describes the pattern code characters you can use with binary **PATTERN MATCH**. Note that these codes are summarized by the mnemonic "CLEANUP".

| Character | Specifies |
|---|---|
| C | Any one of the 33 control characters (including DEL) or any of the 128 8-bit characters |
| L | Any one of the 26 lowercase, alphabetic characters from a to z |
| E | Any one of the characters in the ASCII set and all 8-bit characters |
| A | Any one of the 26 uppercase or 26 lowercase, alphabetic characters from A to Z or a to z |
| N | Any one of the 10 ASCII numeric characters from 0 to 9 |
| U | Any one of the 26 uppercase, alphabetic characters from A to Z |
| P | Any one of the 33 punctuation characters (including SP) |

## Binary SORTS AFTER ( ]] )

The binary **SORTS AFTER** operator tests whether the left operand sorts after the right operand in numeric subscript collating sequence.

### Format

operand A]]operand B

### Explanation

Binary **SORTS AFTER** compares two operands to determine if the first operand sorts after the second in the subscript ordering sequence defined by the single argument **$ORDER** function for the numeric collating sequence.

Binary **SORTS AFTER** produces a result of true if the first operand sorts after the second operand. Otherwise, it produces a result of false. If operand A is equal to operand B, then a truth value of false is returned.

In a numeric collating sequence canonic number operands sort according to numeric value, with negative numbers sorting first, followed by zero, then positive numbers. All operands that are not canonic numbers (except the null string, which sorts before all non-null string operands) sort after canonic number operands. If both **SORTS AFTER** operands are not canonic numbers, then they sort in the same way as the binary **FOLLOWS** operator.

### Comments

You can produce a **NOT SORTS AFTER** operation by using the unary **NOT** operator with binary **SORTS AFTER** as follows:

operand A']]operand B

'(operand A]]operand B)

**NOT SORTS AFTER** reverses the truth-value of binary **SORTS AFTER** applied to both operands. If operand A is identical to operand B, or if operand B sorts after operand A, then **NOT SORTS AFTER** returns a result of true. If operand A sorts after operand B, **NOT SORTS AFTER** returns a result of false.

### Related

Expression evaluation

Binary FOLLOWS ( ] )

$ORDER function

## Examples

The following example creates an array and shows how the binary **SORTS AFTER** operator, the **$ORDER** function, and the binary **FOLLOWS** operator sort the array.

```
SET I%NODE(2)=""
SET I%NODE(122)=""
SET I%NODE(123)=""
SET I%NODE("+")=""
SET I%NODE("HARP")=""
SET I%NODE("HARPOON")=""
```

### $ORDER Function

```
SET T%X=""

FOR  SET X=$ORDER(I%NODE(T%X)) QUIT:T%X="" DO $Env.Output(T%X)

Results:
2
122
123
+
LAMP
LAMPOON
```

### Binary SORTS AFTER Operator

```
DO $Env.Output(2]]"" )

Results: 1


DO $Env.Output(122]]2)

Results: 1


DO $Env.Output(123]]122)

Results: 1


DO $Env.Output("+"]]123)

Results: 1


DO $Env.Output("HARP"]]"+")
Results: 1

DO $Env.Output("HARPOON"]]"HARP")
Results: 1
```

## Binary FOLLOWS Operator

```
DO $Env.Output(2]"")
Results: 1


DO $Env.Output(122]2)
Results: 0


DO $Env.Output(123]122)
Results: 1


DO $Env.Output("+"]123)
Results: 0


DO $Env.Output("HARP"]"+")
Results: 1


DO $Env.Output("HARPOON"]"HARP")
Results: 1
```

## Logical Operators

Logical operators perform logical operations. The following table illustrates those logical operators supported by EsiObjects.

| **Operator** | **Syntax** |
|---|---|
| Binary AND | **A&B** |
| Binary INCLUSIVE OR | **A!B** |
| Unary NOT | **'B** |

## Binary AND ( & )

The binary **AND** operator tests whether both of its operands have a truth value of true.

### Format

operand&operand

### Explanation

Binary **AND** produces a value of true only if both operands are true (that is, have non-zero values when evaluated numerically); otherwise, it produces a value of false.

### Comments

You can specify the Boolean operation of **NOT AND** (**NAND**) by using the unary **NOT** operator with binary **AND**. You can express **NOT AND** using either of the following equivalent formats:

operand'&operand

'(operand&operand)

The negative **AND** reverses the truth value of binary **AND** applied to both operands. It produces a value of true when either operand, or both operands, are false. It produces a value of false only when both operands are true.

### Related

Expression evaluation

### Examples

The following example evaluates two non zero-valued operands as true and produces a value of true.

```
SET L= 1,R=-71 DO $Env.Output(L&R)

Results: 1
```

The following example evaluates one true and false operand and produces a value of false.

```
SET L=1,R=0 DO $Env.Output(L&R)


Results: 0
```

The following example evaluates two false operands with a negative **AND**. It produces a value of true.

```
SET A=0,B=0 DO $Env.Output(A'&B)

Results: 1
```

The following example checks user input T%USRNM and T%PW against the user object. If not valid, a message is asserted.

```
IF User.Username=T%USRNM&User.Password=T%PW DO
.  DO INIT
ELSE  $ENVIRONMENT.Assert("Invalid username or password") DO LOGIN
```

## Binary INCLUSIVE OR ( ! )

The binary **INCLUSIVE OR** operator tests whether one or both operands are true.

### Format

operand!operand

### Explanation

Binary **INCLUSIVE OR** produces a result of true if either operand has a value of true, or if both operands have the value of true. It produces a result of false only if both operands are false.

### Comments

You can produce a **NOT OR** (or **NOR**) operation by using unary **NOT** with **INCLUSIVE OR**. To express **NOT OR** use either of the following equivalent forms:

operand'!operand

'(operand!operand)

The **NOT OR** operation reverses the truth-value of binary **OR** applied to both operands. If both operands are false, **NOT OR** produces a result of true. If either operand is true or if both operands are true, it produces a result of false.

### Related

Expression evaluation

### Examples

The following example evaluates one true and one false operand and produces a true result.

```
SET L=1,R=0 DO $Env.Output(L!R)

Results: 1
```

The following example evaluates two false operands and produces a false result.

```
SET L=0,R=0 DO $Env.Output(L!R)

Results: 0
```

The following **NOT OR** example evaluates two false operands and produces a true result.

```
SET L=0,R=0 DO $Env.Output(L'!R)

Results: 1
```

The following example shows how two tests can be combined. The example matches the user name and password entered against the same information for the current uses. If they do not match, then an error message is displayed.

```
IF UsrnmControl.Text'=Usr.Usrnm!(UsrPwControl.Text'=Usr.Passwd)
ELSE  $ENVIRONMENT.Assert("Invalid Login") QUIT
```

## Unary NOT ( ' )

The unary **NOT** operator inverts the truth value of the Boolean operand or operator it modifies.

### Format

'operand

'operator

### Explanation

Unary **NOT** with an operand inverts the truth-value of the operand. If the operand is true, unary **NOT** gives it a value of false. If the operand is false, unary **NOT** gives it a value of true.

Unary **NOT** with an operator inverts the sense of the operation it performs. It effectively inverts the result of the operation. The following table describes how the unary **NOT** operator affects the meaning of the binary operators.

| Operator | Meaning |
|---|---|
| **'&** | The expression is true when one or both operands are false. |
| **'!** | The expression is true only when both operands are false. |
| **'=** | The expression is true only when both operands are not identical strings. |
| **'>** | The expression is true when the left operand is less than or equal to the right operand. |
| **'<** | The expression is true when the left operand is greater than or equal to the right operand. |
| **'[** | The expression is true when the right operand is not contained in the left operand. |
| **']** | The expression is true when the left operand does not follow the right operand in ASCII collating sequence. |
| **'?** | The expression is true when the left operand is not a string in the pattern specified by the right operand. |

### Comments

You can invert the meaning of any relational or logical operator using either of the following equivalent formats:

operand'operator operand

'(operand operator operand)

### Related

Expression evaluation

## Examples

The following example tests the sum of two local variables in the context of a postconditional expression. Because their values do not equal 10, control transfers to the line labeled INV.

```
A   SET A=4,B=3
    .
    .
    .
    DO:A+B'=10 INV
```

The following example tests two variables with the **NOT AND** (**NAND**) operator. Because one variable is true and one is false, the result is true.

```
SET A=0,B=1 DO $Env.Output(A'&B)
```

```
Results: 1
```

The following example tests two variables with the binary **AND** operator. Because the example places unary **NOT** with the false operand, the truth-value is reversed, and the result is true.

```
SET A=1,B=0 DO $Env.Output(A&'B)
```

```
Results: 1
```

## String Operator

String operators perform operations on strings. The following table illustrates the string operator supported by EsiObjects.

| **Operator** | **Syntax** |
|---|---|
| Binary CONCATENATE | **A_B** |

## Binary CONCATENATE

The binary **CONCATENATE** operator produces a result that is a string composed of the right operand appended to the left operand.

### Format

operand_operand

### Explanation

Binary **CONCATENATE** gives its operands no special interpretation. It treats them as string values. If the concatenated string is longer than string length supported on the underlying M platform, an error occurs.

### Examples

The following example concatenates two string literals.

```
DO $Env.Output("OBJECT"_"ORIENTED")
```

```
Results: OBJECTORIENTED
```

The following example concatenates two local variables, NAME and SUBS.

```
SET T%NAME="^DD",T%SUBS="(1,2,3)",T%GLOBAL=T%NAME_T%SUBS
DO $Env.Output(T%GLOBAL)
```

```
Results: ^DD(1,2,3)
```

The following example concatenates two string literals and the null string. The null string has no effect on the length of a string. (you can concatenate an infinite number of null strings to a string.)

```
SET T%V="TECH"_""_"NOLOGY" DO $Env.Output(T%V)
TECHNOLOGY
```

The following example sets the property FULLNAME in object user to the variable Last and First, which are concatenated.

```
SET USER.FULLNAME=T%Last_","_T%First
```

## Indirection (@) Operator

The **INDIRECTION** construct is used to translate string data into executable code at runtime. For example, a string can be translated into the argument of a command, or into certain kinds of variable references. The @ is a language contstruct that is only used in certain restricted contexts. Indirection gives the programmer a language construct that can be used to generalize code, that is, defer execution until runtime based on the execution context.

### Format

@expr_atom

@expr_atom@(subscript{,...})

### Parameters

expr_atom

The expression atom whose value is to be used

@expr_atom@(subscript{,...})

A string used to name an array node that descends from the variable referred to by **expr_atom**

### Explanation

Each occurrence of indirection in a statement is replaced with a corresponding value, which is then used in the statement. All occurrences of indirection must evaluate to:

- One or more command arguments (argument indirection)
- A variable name, routine name or label name (name indirection)
- A subscripted variable name prefix, terminated by a second @ (subscript indirection)
- A pattern (pattern indirection)

### Comments

In the evaluation of subscript indirection, if an expression atom refers to an unsubscripted global or local variable, the value of the indirection is the variable name and all characters to the right of the second **INDIRECTION ( @ )** operator (note that an entire expression may be used, provided that it is surrounded by parentheses).

### Related

Binary PATTERN MATCH ( ? )

### Examples

The following example uses argument indirection to call the routine at the label VALID within the current routine.

```
H1  SET T%HND="VALID"
    DO @T%HND
```

The following example uses routine name indirection to call the routine at the first line of the routine REB.

```
H2  SET DOAR="REB"
    DO ^@DOAR
```

The following example sets T%W to the pattern for one or more digits and T%B to 10 and tests whether T%B meets the specified pattern. Because the operand evaluates to two digits, the result is true and control passes to the label OK.

```
SET T%W="1.N",T%B=10
IF T%B?@T%W DO OK
```

The following example uses subscript indirection to initialize an array with some defaults.

```
          SET NAM="X(A)",A=10
INIT      ; Initializes default values
          SET DEF(1)="",DEF(2)=+$HOROLOG
          SET DEF(3)=$PIECE($HOROLOG,",",2),DEF(4)="Unknown"
          FOR I=1:1:4 SET @NAME@(I)=DEF(I)
          QUIT
```

## Argument Indirection

In this type of indirection, indirection evaluates to one or more command arguments. In the following example, MyNode(10,6) is set to the value 4.

```
SET ND="MyNode(10,6)=12\3"
SET @ND
```

## Name Indirection

In this type of indirection, the indirection evaluates to a name. A name is any language element that contains an uppercase or lowercase alphabetic character or percent sign (%) followed by up to seven alphanumeric characters. You can use name indirection for the following:

- Variable names
- Line labels
- Routine names

When you use indirection to reference a named variable, the value of the indirection must be a complete global or local variable name, including any necessary subscripts. In the following example, the variable DT is set to 10/16/92.

```
SET ND="DT",@ND="10/16/92"
```

When you reference a line label with indirection, the value of the indirection must be a syntactically valid line label.

In the following example, ACT is set to the value of the line label that matches the state of V. Control is transferred to the label selected.

```
SET ACT=$SELECT(V="green":"GO",V="yellow":"ACCL",V="red":"STOP",1:"caution")
```

When you reference a routine name with indirection, the value of the indirection must be a syntactically valid routine name. In the following example, control is transferred to the routine VALID.

```
CHK  SET ROU="VALID"...

      .
      .
      .
     GO ^@ROU
```

## Subscript Indirection

This form of indirection is an extended form of name indirection. The value of the indirection must be the name of a local or global array node. Subscript indirection is syntactically different than the other forms of indirection. Subscript indirection has the following format:

@expr_atom@(subscript,...)

where:

| | |
|---|---|
| **expr_atom** | evaluates to a local or global variable name, defined or undefined |
| **subscript** | is a string of one or more subscripts separated by commas and enclosed in parentheses |

Subscript indirection creates (or references) variables that logically descend from the variable referenced by the expression following the first **INDIRECTION** operator. The actual variable reference created depends on the list of subscripts following the second **INDIRECTION** operator.

In the following example, the variable D(1,1) is set to the minimum of 100 or D(1).

```
SET ND="D(1)"
SET @ND@(1)=$SELECT(@ND>100:100,1:@ND)
```

## Pattern Indirection

Pattern Indirection is a special form of indirection, unrelated to any other kind. The **INDIRECTION** operator and its operand replace a pattern match and the value of the indirection must be a valid pattern. The **INDIRECTION ( @ )** operator follows the **PATTERN MATCH ( ? )** operator. For example:

```
    SET PAT="3N1""-""2N1""-""4N",MSG="Invalid SSN"
    IF VAL'?@PAT DO INVAL
    .
    .
    .
    QUIT
INVAL DO $ENVIRONMENT.Assert(MSG)
```

For more information about pattern matching, see the description of the binary PATTERN MATCH operator.

## Class Element Indirection

Allows indirection to be used when dealing with class defined names. Class element indirection allows the name of a class to be indirected when using the **CREATE** command. This allows the creator of the object to specify the actual class name at run time. For example:

```
;Function to return a shape object based on input
;T%Shape:  0 = Box, 1 = Circle,  2 = Triangle
SET
T%Class=$Select(T%Shape=1:"MyCircleClass",T%Shape=2:"StockTriangle",1:"Box")
CREATE T%Out=@T%Class@
;
SET $RETURN=T%Out
```

The names of properties and methods can also be indirected when making service requests. For example:

```
;Sorts the T%List Object based on the value of T%What
; 0 = A Normal sort, 1 = a ReverseSort
SET T%Action=$SELECT(T%What=0,"Sort",1:"ReverseSort")
DO T%List.@T%Action@
QUIT

; Looks up the value of a property of the linked object
; T%Prop is the name of the property to look up
SET $RETURN=I%Link.@T%Prop@
```

## Parameter List Indirection

Parameter list indirection provides a mechanism whereby the parameter passed on a service request can be indirected. This is useful for delegation, for example:

```
; Forward the current call to the linked object
;
SET $RETURN=I%Link.Method@($PARAMS)
; Construct a parameter list and invoke a method
;
SET T%Params="20,20"
DO I%Window.Move@(T%Params)
```

If the values are stored in temporary variables, rather than specified literally, the following construct must be used to indirect the parameter list correctly. (Note that $LOOKUP is a privileged function.)

If  T%x=20 and T%y=20, then

```
Set T%Params=$QUOTE($LOOKUP("T%x"))_","_$QUOTE($LOOKUP("T%y"))

DO I%Window.Move@(T%Params)
```

# Index