Accsoft Internet Services

BEE Script

User Reference

Version 1.0 20/11/2003 9:31

Table of Content

1 Introduction

BEE stands for Business Electronic Enterprise. It is a server-side scripting language for commercial web application development. The web server extracts the program script code sections from the web page, runs them, and substitutes the code sections with their corresponding execution output, then send the whole page to the client's browser.

Bearing the word "Business" in its name, BEE was designed to be business oriented. The syntax is simple and flexible, and comes with two different forms: BEE Script and BEE Tag. This design makes it easy for BEE to mix with HTML Tags and other scripting languages in the same page.

BEE has been developed by *OnMyWeb Internet Services*, which is a web hosting provider. Naturally, the design of BEE has taken into considerations of multiple websites running on the same physical server with shared resources. Ownership, management right, resource allocation and security are among the primary design criteria. As a result, these features are intimately integrated into a web hosting environment instead of as add-on patches.

The administration effort is centralised and kept to minimal; most of the settings are automated and done once upon website creation. No unnecessary initialisations in the program codes and most operations (especially for database access) are one-liner. This architecture makes BEE ideal for hosting firms for SMEs (Small to Medium Enterprises).

Although the initial version of BEE Script is implemented on PHP and Apache, the syntax and design is platform independent, and therefore can be implemented on other platforms that support server-side scripting.

1.1 How to use this document

This document is a plain description of the BEE Script system. It is a "brain-dump" of the creator's head. The presentation and layout may not be suitable for the purpose of learning how to use BEE Script. However, it is aiming at a complete interface that is visible to BEE developers. If some information is missing, it is because the information is not required at all to develop BEE Script applications (or not yet written as this document is still evolving on a daily basis.)

If you need a learning guide that shows only the commonly used features and the technique and examples to use them, please consult the BEE Script User Guide.

1.2 Background

The development of BEE starts from an experimental technology called OLDPAL (OnLine Database Programming Accelerator Language). OLDPAL was a set of non-procedural database access and display script aiming at simplifying the mSQL coding (and subsequently MySQL coding). As the name implies, it severs only database access, which happened to be the dominant process in the commercial web applications that *Accsoft* (the developer of BEE) writes and hosts for its customers.

OLDPAL was nothing like the BEE Script that we see today. However, OLDPAL's core concept and the design principles have been inherited by BEE, which has taken the idea to a new level since. Even now, some low level OLDPAL objects are still being used within BEE.

1.3 Design Principles

The original design principles were documented in "Initial OLDPAL Design". They include the Black-box Principle, the Generosity Principle and the Evolution Principle.

1.3.1 Black-box Principle

BEE was designed to run on a shared hosted environment. That means websites from different owners may run on the same network or even the same machine. It is therefore crucial to make sure the hosted websites can only access the resources they are designated to and nothing else, so that they will not intervene with each other and no privacy will be breached. To the website, the platform and the resources are contained in a "black-box", and no knowledge of them needs to be of concern to the codes.

Here are the main design criteria derived from the Black-box Principle:

- A web page cannot access any databases and authentication data except for the ones designated.
- Database operations must not be affected by the choice of the database platform and its location. Subsequent changes of such choices must not require any code changes.
- Authentication server location and arrangement are transparent to the code. Subsequent changes of the server mechanism must not require any code changes.

1.3.2 Generosity Principle

Sharing of code is commonly required but not commonly archived. The main obstacles towards code sharing is that too much knowledge about the environment is contained in every piece of codes, and when such environment changes, the piece of code needs to be rewritten. As long as there require adaptations, there is no true sharing.

Object-orientedness solves this issue to a certain extent in separating the atomic pieces of codes. In a web portal environment, we need to separate the business logics (the codes) and the appearance (the artwork), so that each client have their own "skin" and preferences, so that sharing can be archived beyond the code level. We share applications, generosely:

- HTML pages can be copied to or shared by different websites without changes. All client-dependent information including settings, content, authentication and resource allocation are separated from the code (the CROFT and Scheme to be discussed later).
- Similar pages on the same website can use the same HTML codes without copy-and-modify. The "template" page is simply reloaded with different contents or functions.
- Object-orientedness with polymorphism is required to encapsulate business logics and should be reused via an application-wide or system-wide common library.

1.3.3 Evolution Principle

Software is a living thing. Applications change to suit their environment and requirements. Changes cause pain, especially among component interfaces. Some how, the structure of position-independent name-value pair gives flexibility in interface design, because:

- The caller is allowed to omit arguments, so that it will not fail when new arguments are introduced. (The callee function will take the default for missing arguments.)
- The caller can omit all arguments and let the callee function to find out from the environment (e.g. the web form – This point was in OLDPAL. In BEE Script, form variable is only one of many ways to pass arguments).

Future development of BEE will aim at distributed processing and in particular focus on Web Services technology. Software components may run on remote machines and common standards like XML and SOAP are being used across the net.

1.4 Characterising BEE

The architecture of BEE Script is characterised by the followings:

- Pre-allocation of resources (CROFT – Customer Resource Online Facility Tables) based on the URL of the web page: Resource handles are completely concealed from the program code. This design simplifies the coding for finding and binding the resources, and allows portability across servers and platforms. As no resources are specified in the program code, different web pages can share the same piece of code or template with absolutely no modification. (Please see "Virtual Page" for details.)
- Security: Because the program code has no access to the resource handles, a web page could never be able to access anything outside of its predefined Scope. Even if an intruder with complete knowledge of the source code runs a modified script on the same physical server, he or she would not be able to access the same resources because the resources are allocated to the URL, not the page. The binding of resources to the URL protects the web page from copy-modify-execute attacks.

- Built-in multi-access-level authentication: Authentication settings and user data (login, passwords and other attributes) are bound to the URL and are subject to the same resource allocation protection under CROFT. The authentication process is transparent to the program code, separating the role of the designer/developers and the security administrators.

- Rich functionality: full cross-platform SQL database support, asynchronous remote function call, intrinsic support for e-mail generation, complete off-the-code program parameters registry (the "scheme" class), automatic web administrator's site, online content editing and "Virtual Page" generation, to name a few.

- Adaptive syntax: Dual form of coding (BEE Script and BEE Tag) makes it flexible to mix with other web page constructs. Attribute Name-Value Pair syntax provides non-positional parameter passing and makes the code robust and easy to extend.

- Non-typed, multi-value and macro-style variable structure: All BEE Variables are implemented in array. Value strings are evaluated recursively for BEE Variables (like string macro substitution), allowing straightforward nested-addressing and simplifying complicated symbolic referencing.

- Well-defined variable hierarchy: The "class%name:element" structure provides a three-tier hierarchy. "System Class" standardises variable usage and removes ambiguity resulted from various programming styles. "Object Classes" provides an object-oriented programming environment for function encapsulation.

- User extendable command set: There are only few dozen intrinsic commands and over half of them is common flow controls like declarations, conditionals, looping and debugging. The rest are authentication, system control, database and specific functions like mail and text control. Programmers can extend the functionality by defining functions, BEE Conversions and objects, which use the same calling syntax as the intrinsic features.

1.5 How BEE runs

BEE is built on three pillars:

1) CROFT for resource allocation, authentication and permanent settings

2) BEE Script/Tag Compiler

3) BEE Run-time Library

The web designer and programmer create the website and upload the web pages onto the BEE enabled web server. The server upon receiving the web pages will scan for BEE Script statements and convert them into BEE Tags. The web pages are then compiled into the target implementation codes (PHP at the moment) and stored in the web storage.

At run-time, the implementation codes will be called on to perform the intended function of the BEE commands. Most of these functions however are implemented as in the BEE Run-time Library, which access CROFT implicitly if resources are required. That is the resource will be identified and opened upon its first use.

1.6 What is a BEE Website

The common idea of a website is a name that starts with "www" and ends with "com" or else with some dots in between, and when punched into the computer that name will bring up a lot of potentially wonderful things. This vague definition is not sufficient for the sake of technical discussion. We need a more specific definition of a website under the context of the BEE technology.

A BEE Website is a collection of web pages running on BEE enabled web server or servers. A BEE Website contains one or many BEE Web Paths and all web pages under all such paths form the BEE Website.

Here is a summary of the hierarchy and the identifier in each level:

Constructs	**Has at least one**	**Belongs to a**	**Identified by**
BEE Website	BEE Web Path	n/a	Owner-Server Duple
BEE Web Path	BEE Web Page	BEE Website	idURL
BEE Web Page	n/a	BEE Web Path	URL starting with the idURL of the BEE Web Path after the protocol specification (http://)

While Authentication (username/password logins) and scheme settings are bound to the BEE Website, BEE Resources (e.g. database) are allocated at the BEE Web Path level. In another word, the idURL defines a set of resources which all the web pages with URLs derived from it have equal right to share.

An idURL starts with the full web server name without the protocol specification (http://). The path after the server name is optional but if exists must be defined as a full path (e.g. /abc/defx and /abc/defy must be defined as two separate idURLs, not just /abc/def).

Please note that a BEE Web Path can be a web page URL. In that case, there is only one BEE Web Page under that BEE Web Path.

Theoretically, a BEE Website may contain BEE Web Paths under different web server names (different "www....com" names). However, BEE Session variables are implemented with cookies and therefore cannot be retain their values across pages on two different web servers. For example, if a shopping cart is implemented with session variables defined on a web page under one web server name, the cart content will not be accessible from web pages under another web server name.

To simplify the language, when "website" or "site" is mentioned in this document, it means BEE Website; "web path" or "path" means BEE Web Path and "web page" or "page" means BEE Web Page, unless explicitly stated otherwise.

2 CROFT – Customer Resource Online Facility Tables

CROFT stands for Customer Resource Online Facility Tables. It consists of three parts: Resource Allocation, Authentication and Scheme settings.

- Resource Allocation is called Owner-Service internally. BEE Resources (mainly database access) are allocated based on the URL of the web page. This design serves two purposes: 1. programmers cannot access resources except for those designated by the administrator to the web page's URL, and 2. the same application without any modification can run on different websites accessing different resources.

- Auth is short for Authentication. Unlike in other scripting language that programmers need to implement their own username/password encoding/decoding/matching, in BEE, the entire user login mechanism is a native feature, and is administered by the administrator (not by the programmer).

- Scheme has two parts, the Application Scheme and Administration Scheme. Application Scheme is for the application process to retrieve and save parameters. It can be taken as persistent settings or data structure that the application codes interpret. Administration Scheme on the other hand are not accessible by the code. It is for the BEE Administrator to structure the website (e.g. to overwrite default directories).

BEE is designed with a shared server in mind, typically in an environment of a service bureau or web hosting company with many SME customers sharing the same physical web server. Therefore, a two-tier structure that identifies the "owner" (the customer) and its "service" (the website or online application) is built into the design concept of BEE.

A BEE Website is identified by a dual-key called the Owner-Service Duple. Authentication and Scheme settings are site-wide. That is, if you can login to a page, you can login to any pages of the website. If a scheme setting is applied to a page, it applies to all pages of the website. (There is nothing to stop you from using the same settings for multiple sites, for example, to share the same password file. Even so, we still need separate Owner-Service Duple because otherwise, the two sites will become one.)

Resources such as database access right are bound to BEE Web Path (or BEE Web Page if the idURL is a page address), as described in the following.

2.1 Resource Allocation

All database specific information is kept in CROFT separating from the program code. A Database Access Specification (DAS) contains information for the BEE system to gain access to the data, parameters like Database Type, Database Host, Database Name, and the Username and password. This information is stored within CROFT and can be uniquely identified by the idURL and an identifier called DBID.

For example:

idURL	DAS: Database Name – Database Type
www.mywebsite.com	Marketing – MySQL database
www.mywebsite.com/order.htm	Ordering – Microsoft access database
www.mywebsite.com/checkacct	Accounting – Oracle database
www.mywebsite.com/checkacct/news	Marketing – MySQL database

If a page can be derived from multiple idURL, the longer one will be used to identify the DAS. For example, http://www.mywebsite.com.au/checkacct/billing.htm can be derived from idURL www.mywebsite.com or www.mywebsite.com/checkacct. In that case, the longer one (www.mywebsite.com/checkacct) will prevail. i.e. the billing.htm page will access the Oracle database "Accounting" (not the MySQL database "Marketing").

In the actual coding, there is no preparation of a database access. You simply launch a "database" BEE Command with the SQL query and/or some other parameters, CROFT will then automatically find the corresponding database on the host specified by the DAS, open it with the DAS username and password through the mechanism of the specified type, and the result is passed back via a database result variable. This simplifies the web page coding to a single command, and makes it portable across platform.

In rare occasions (usually when combining results from heterogeneous systems) which needs to access several DASes in one single page, you can use the DBID identifier to uniquely identified the DAS for the SQL.

For example:

http://www.mywebsite.com/newsheadline.htm	DBID=	database – advertising
http://www.mywebsite.com/newsheadline.htm	DBID=world	database – worldnews
http://www.mywebsite.com/newsheadline.htm	DBID=sports	database – sportsnews

The DBID is only an identifier and is not related to the name of the database or server parameters. Therefore, specifying the DBID in the program code does not make it any less platform-independent or more difficult to port across platform. The code is still separated from the knowledge of the DAS even with DBID.

2.2 Authentication

When a visitor login to a website by providing a username and password, all the web page program needs to do to authenticate the user is one single command: "login". CROFT will automatically search for the Owner-Service Duple by matching the longest idURL with the URL of the web page. Then use the found Owner-Service Duple to look up the Authentication Mechanism Specification (AMS) with the Owner-Server Duple to find the user table for a password check.

An AMS contains the user table name and its database access specification such as the type, host, database and table name (similar to DAS previously mentioned). The AMS also contains the password encryption method, the Administrator Access Level, and a mapping of the six essential fields: UserName, Password, Realm, AccessLevel, Active and Expiry.

The field names of the six essential fields in the user table needs to be entered into their corresponding AMS mapping field, unless their names are exactly as the default (UserName, Password, Realm, AccessLevel, Active and Expiry). If a field is missing from the user table, a value needs to be entered in the AMS mapping field with a leading '#' to tell CROFT to use the value for that field instead of retrieving it from the user table.

A username acceptable by the BEE Website is in the format of either *username* (e.g. john) or *username@realm* (e.g. john@warehouse). This allows the web application to group users into different name spaces or use different user tables (identified by the *realm*), even one from another "friendly" website. (Please see Affiliate access level in the BEE Variable section under "sys" class.)

Upon authentication, the system will first find the Owner-Service Duple by matching the longest idURL as described before. Secondly, the AMS is identified using the Owner-Service Duple and the "realm" specification in the "username" parameter (either passed in through the @ suffix of the username, or as a separate parameter. If not found, the "*" realm will be used (wild-card realm).

Once the AMS is identified, the system will access the user table based on the information in the AMS. The user record will be identified by the username and the realm field if the Realm field mapping exists. (The Realm field is a field in the user table that indicates the Realm that the user belongs to. You can have different realms in different user tables, or one user table for different Realms identified in the realm field of the user table.)

One important point to note about using "realm": You must specify the "RealmField" explicitly in the AMS and specify "*" in "Realm" in order to make use of "realm" in the authentication process. Simply adding a field called Realm in your user record would not make it effective.

Next, the password obtained from the user record will be matched based on the encryption mechanism specified in the AMS.

If the user login is successful (password match), then all the fields in the user record will be loaded into the sys%auth array, which may contains site specific attributes like sys%auth:Name, sys%auth:Tel, sys%auth:TaxCode, etc.

2.3 Scheme

There are two "scheme" setting mechanisms in BEE. One is for programmers for keeping application settings, and will be discussed in the "scheme" class. Another is used internally by CROFT to control the operation of the website.

Scheme in CROFT is for permanent system settings, usually to identify directories in the BEE Hosting Server. The CROFT Scheme settings are set up by the BEE Hosting Administrator, and is not accessible from the program code. Settings for the application program should be put under the scheme class. (Please see BEE Variable under scheme%.)

The reason that we need similar mechanism in CROFT is that some settings are required even before the scheme class is initialised, e.g. the setting of the name of the scheme directory itself. Also, CROFT Scheme settings can only be changed by the BEE Hosting Administrator, making the CROFT Scheme ideal for security system settings.

Most of the CROFT Scheme settings have a default value and a CROFT Scheme entry is required only if the value is different from the default (e.g. if you want to put the VirtualBase directory to be somewhere else.)

3 BEE Variables

BEE Variables are simple but powerful. They are simple because there is only one data structure – array. No pointers and data type constraints, everything is purely string macros.

For example, after `var abc = "myvalue";`, variable **abc** contains the literal "myvalue". Then after we do `var xyz = "abc";`, we got **xyz** containing "abc" and **{xyz}** containing "myvalue". Now we create an array out of thin air (as it's already there): `var abc:april = "fool"; var abc:may = "flower";`, then **{abc|list}** will give "''=>'myvalue','april'=>'fool','may'=>'flower'", and so will **{{xyz}|list}** because {xyz} evaluates to abc, and {{xyz}|list} evaluates to {abc|list}. We've just changed the language from "containing" to "evaluating". Now you got the idea. (Please note that "list" used in the above example is called a BEE Conversion to convert an array into a displayable string.)

Curly bracketed variables always evaluate to a string. To specify the variable as a whole with all its array elements, you need to use the **(var)** cast.

In the following sample code, the comment lines (the ones starting with "//") indicate the possible evaluations, where the arrow "->" means "evaluates to".

```
var abc = "myvalue";
       // {abc} -> myvalue
       // {abc:} -> myvalue
var xyz = "abc";
       // {xyz} -> abc
       // {{xyz}} -> {abc} -> myvalue
var abc:april = "fool";
       // {abc:april} -> fool
var abc:may = "flower";
       // {abc:may} -> flower
       // {abc|list} -> ''=>'myvalue','april'=>'fool','may'=>'flower'
       // {{xyz}|list} -> {abc|list} -> (ditto)
var yourval = "{abc}";
       // {yourval} -> {yourval:} -> myvalue
       // {yourval|list} -> =>myvalue
var def = (var)abc;
       // {def} -> {def:} -> myvalue
       // {def|list} -> =>myvalue,april=>fool,may=>flower
```

Please note that the double quotation marks surrounding the values (the right-hand-side of the assignment) are optional, and can be replaced by single quotation marks. If there is no spaces in the value string, the quotation marks can be omitted altogether.

3.1 BEE Variable Name

This is the general form of a BEE Variable Name:

```
[class%][file&]name[:[#]element]
```

Note 1: The File Specifier '*file*&' is considered part of *name* and is for "scheme", "file" and "upload" classes only.

Note 2: The element positioner '#' before *element* is used to address the element by position in the array (e.g. #2 for the 3rd element). (Without the element positioner, *element* will be an alphanumeric index.) Accessing with the positioner (even reading) will automatically extend the array if necessary to reach that position. e.g. If array "a" has only 3 elements, display "{a:#8}"; or even if ({a:#8|isset}) ... will extend the array "a" to have 9 elements, with 6 null elements appended after this original three.

The three-tier hierarchy of class-name-element is central to the BEE Variable architecture.

Class indicates the type of the variable and often implies the usage and operations of it. (The meaning of "class" is different from that in the Object-Oriented context. In OO, class is a template of the object; In BEE, "class" is the object. For details, please see Chapter "Classes and Objects")

Name is the name of the variable. BEE Variables are in fact arrays. A single-value scalar variable is in fact the array element with the blank index, which is called the default element.

Element is the index to the BEE Variable array. If the Element part of a BEE Variable name is omitted, it will be evaluated to either the whole array (*class*%*name*) or just the default element (the one with the blank index – *class*%*name*:), depending on the context (e.g. based on the BEE Conversion, the assignment target, the Class it belongs to, or explicit specification.)

3.1.1 General Classes (Object Classes)

The concept of "class" is closely related to the one of "object". However, in BEE, the terms "class" and "object" are used interchangeably. They both mean a "super variable" that holds multiple array variables (like a two-dimensional variable).

General Classes are used as common variables. They are only accessible within the local context (e.g. inside the function they are defined) unless declared as "global" or "parent".

Variables that does not have a class part (*class*%) in their name belongs to the general class "value". That is why "value%" is commonly omitted to increase readability.

When a General Class is used as an object (created by a constructor using the "new" syntax), it is called an Object Class. The two do not have any semantic differences.

3.1.2 Special Classes

These are classes that operate in exactly the same way as General Classes. What "special" about "Special Classes" is that they are used by the system to store values used to communicate with the application.

These classes are:

`arg`	Value already passed into the function's Context
`message`	Text error message returned from a function or command (The name part is the function name and therefore is case insensitive.)
`param`	Value to be passed into a function or command; cleared automatically after the function call. (The name part is the function name and therefore is case insensitive. However, the Element part is the argument name and is case sensitive. If an argument is passed via the function calling line, BEE will convert all argument names to lowercase. The param% class provides a way to pass non-lowercase argument name to a function.)
`result`	Value passed out from a function or command (The name part is the function name and therefore is case insensitive.)
`status`	Numeric error code returned from a function or command (The name part is the function name and therefore is case insensitive.)
`systemp`	Temporary values used by the system. (Please do not use this class.)

arg is an argument available from within a user-defined function. It is in the form of arg%function:*argname* and is equivalent to the ones on the argument list of the user-defined function. However, it is recommended to access function arguments via arg%*argname* for array access and case insensitive argument name (*argname* is always in lowercase from within the function).

arg%arg refers to the "default" argument, which is the one without an argument name. e.g. fn "def"; will see "def" in arg%arg from inside the function.

Please note that param% is prepared by the caller before calling the function, and arg% is used from within the callee function to retrieve the passed in values.

message is a string (or error message) a BEE Command passes out. It is blank unless there is an error. It is usually in the form of message%*tagname*. (message%*tagname* usually does not take an array.)

param is a value to be passed into a BEE Command or user-defined function. It is in the form of param%*function*:*argument* and is equivalent to the ones on the argument list of the tag or function. Explicitly set "param" take precedence over the ones on the argument list. Array can be passed in via the array type (e.g. "(array)..."). Please note that param% is prepared by the caller before calling the function, and arg% is used from within the callee function to retrieve the passed in values.

result is a value made available as a result of a BEE Command operation. It is usually in the form of result%*tagname*, and therefore can store an array.

status is a number (or error code) a BEE Command passes out. It is zero unless there is an error. It is usually in the form of status%*tagname* (status%*tagname* usually does not take an array.)

systemp is a temporary variable used by the system. Please do not use this class.

3.1.3 System Classes

The following classes are used by the system. They are globally accessible and their operations are governed by the system in a restrictive way. For example, some are read-only and some can not be accessed as an array etc:

Class	**Get**	**Set**	**Clear**	**Array access**	***file*& acceptable**
`class`	Yes	No	No	No	No
`const`	Yes	No	No	No	No
`cookie`	Yes	Yes	Yes	Yes	No
`debug`	Yes	Yes	No	No	No
`file`	Yes	"text" and "action" only	"text" only	Yes	Yes
`form`	Yes	Yes	Yes	Yes	No
`function`	Yes	No	No	No	No
`scheme`	Yes	Yes	Yes	Yes	Yes
`session`	Yes	Yes	Yes	Yes	No
`sys`	Yes	some only	some only	some only	No
`text`	Yes	Yes	No	Yes	Yes
`upload`	Yes	"saveas" only	No	No	Yes

class is for properties of user-defined classes. It is read-only.

Variable	Value/meaning	Access
class%*className* (*className* is not "list")	An array of variable names in the class *className*. This is a "virtual" array and access to individual elements is not allowed. e.g. you cannot use class%myclass:#0 to access the first element of thg class%myclass array. However, you can "foreach" the variable or assign it to another variable	get (array only)

	for access to its elements.	
class%*className*:varcount	The number of variables in *className*	get
class%*className*:varlist	A comma-delimited list of variable names in the class *className*.	get
class%*className*:list	Same as class%*className*:varlist	get
class%list:*className*	Same as class%*className*:varlist	get
class%list	An array of all class names in the context. It is a "virtual" array (like class%*className*, no access to individual ones is allowed.)	get (array only)

const is internal constant for some special functions. It is read-only.

Variable	Value/meaning	Access
const%socket:*constant*	Constants related to socket operation. They include: AF_INET AF_UNIX SOCK_STREAM SOCK_DGRAM SOCK_SEQPACKET SOCK_RAW SOCK_RDM	get
const%mail:HTML_HEADER	For the "header" parameter in the "mailto" command to turn the message into HTML format.	get
const%true const%false	A boolean value of "true" A boolean value of "false"	get

cookie is a persistent value that can be kept on the client's machine across multiple client sessions. You can set an expiry time on a cookie so its value will persist until the time is up, regardless of whether the user has closed the browser or not.

Variable	Value/meaning	Access
cookie%*name*: or cookie%*name*:value	The value of the cookie being stored in the client's machine for later retrieval. (Although cookie values are supposed to be encoded, this is done implicitly in the system so you are free to use even white-spaces, semi-colon and comma. Accessing the "value" element of cookie%*name* will trigger the cookie operation. (Other cookie elements are no more than those of an "ordinary" class, except that their values are used	get set

	in the cookie operation when the "value" element is being accessed). You can only set or clear a cookie's value before the first display (even before the <html> tag). Also, the change does not take effect until after the page run. (For the very page run, you need to refer to the source of the new value direct.)	
cookie%*name*:expire	The expiry time (on the client's machine) in timestamp format (seconds since 1 Jan, 1970 GMT)	get set
cookie%*name*:path cookie%*name*:domain cookie%*name*:secure	The "path" parameter of the cookie The "domain" parameter of the cookie non-zero if SSL is required	get set

"session" class is implemented with a cookie internally but the "session" class and the "cookie" class are completely independent. They only happen to be implemented by a common technology called "cookie".

debug is for accessing the online debugger information. It is read-only.

Variable	Value/meaning	Access

debug%active	The debug functions will be active only if debug%active is set to non-zero. Default is 1.
	
	This variable is usually used to turn off all debug features in one go. (The operation will speed up marginally.)
		get(no array)
		set(no array)

debug%display	The string set to this variable will be displayed on the current debug window.	get(no array)
		set(no array)

debug%file debug%file:name debug%line debug%line:number	The name of the current script file The line number of the command currently being executed.	get(no array)

debug%function:time	If set to *n*, the execution time of the function down to *n*th level will be displayed upon its return. Default is 0.	get(no array)
		set(no array)

debug%trace	If set to non-zero, display [*file*:*line*] before executing a line. Default is 0.	get(no array)
		set(no array)

debug%trace:function
debug%trace:functions
	Comma-delimited list of actual function names of functions only in which the debug%trace setting will be effective.
	
	Setting debug%trace:function to blank
		get(no array)
		set(no array)
		clear(no array)

	means to trace the main program only. If debug%trace:function is not set (or cleared), all lines will be traced.	
debug%trace:showfunction	If set to non-zero with debug%trace, show [*file*:*line*-*actual_function_name*]. Default is 0.	get(no array) set(no array)
debug%trace:showinfo	If set to non-zero with debug%trace, show more information (e.g. the BEE command). Default is 0.	get(no array) set(no array)
debug%trace:showtime	If set to non-zero with debug%trace, show *HH:MM:SS*:[*file*:*line*]. Default is 0.	get(no array) set(no array)
debug%trace:count debug%trace:maxline debug%trace:maxlines	If set to a positive number, debug%trace will automatically switched off after the specified number of lines have been traced. Default is null, which means not to switch off debug%trace until explicitly done (i.e. set to 0.)	get(no array) set(no array)
debug%trap[:*file*:*line*]	If set to a positive number, the operation will stop before the "number"th time executing the line specified as *file*:*line*. If debug%watch is set, the "watched" value will be displayed before the operation stops. If *file*: part is not specified, the operation will stop before that line regardless of the file being in. If both file and line is not specified, the trap will apply to every line. e.g. setting debug%trap to 10 will cause the operation to execute 9 more lines and stop before the 10th. Setting debug%trap to 1 will stop the operation before executing the next line.	get(no array) set(no array)
debug%trapfunction[:*file*:*line*]	If set to a function name, when the operation is trapped at *file*:*line* the specified function will be executed before the operation stops. (The debug%trapfunction and debug%trap of the same element index (the :*file*:*line* part) always come together.) The trap-function will be given the context of the trap point so that it can show or even reprocess any variables upon the trap. (Yes, the trap-function may "pollute the environment".) The trap-function may resume the	get(no array) set(no array)

	trapped operation by assigning a positive integer to "debug%trapresume". Upon returning, the system will re-initialise the "trap count" (the debug%trap[:*file*:*line*] variable) by the value of "debug%trapresume". Note: debug%active is set to 0 (all debug features off) before the trap-function is entered. If you want the debug features to be active again after resuming, please set debug%active to 1 before returning.	

debug%trapresume	To be set by the "trapfunction".
	
	If set to a positive integer, the operation will resume upon the return of the trapfunction, with the "trap count" for that trap point re-initialised to that positive integer.
	
	If not set or set to a non-positive integer, the operation will stop upon the return of the trapfunction.
		get(no array)
		set(no array)

debug%watch[:*file*:*line*]	Comma-delimited list of values to show before executing the line specified as *file*:*line*.
	
	If *file*: part is not specified, the values will be displayed on that line regardless of the file being in.
	
	If both file and line is not specified, the values will be displayed whenever a line is "traced".
	
	Note: The values are interpreted under the local context. Please use absolute attribute to avoid pre-evaluation:
	var debug%watch:*file*:*line* =! ...
		get(no array)
		set(no array)

debug%watchformat	The format of the "watch" with the name represented by @name and value by @value. Default is [@name=@value].	get(no array)
		set(no array)

debug%window	The name of the debug window that you want the debug message to display in.
	
	The debug window will be opened at the time of setting debug%window regardless of debug%active, but no message will be displayed until debug features are set and enabled.
		get(no array)
		set(no array)

	The debug window will remain open until it is closed manually. Default is blank (the same browser window of the web page.)	

debug%window:display	Same as debug%display	get(no array)
		set(no array)

file is contains information about files in your directory. The "name" part of a file class variable is taken as the name of a file under the FileDir – {sys%croft:filedir}. You may specify files under the CROFT directories (those specified in {sys%croft:wwwdir}, {sys%croft:textdir}, {sys%croft:schemedir} and {sys%croft:vbdir}), but in this case you need to specify them in full path (e.g. file%{sys%croft:wwwdir}/images/logo.gif&size}).

Variable	Value/meaning	Access
file%*file*&name file%*file*&basename file%*file*&dirname file%*file*&size file%*file*&mimetype file%*file*&atime file%*file*&ctime file%*file*&mtime file%*file*&exists file%*file*&readable file%*file*&writable file%*file*&isdir file%*file*&isfile file%*file*&islink file%*file*&content file%*file*&contents	The name of the file The file name without directory path The directory name of the file The file size in bytes The MIME content type of the file The last access time of the file The last inode change time of the file The last modification time of the file 1 if the file exists, or 0 if otherwise 1 if the file is readable, or 0 if otherwise 1 if the file is writable, or 0 if otherwise 1 if *file* is a directory, or 0 if otherwise 1 if *file* is a file, or 0 if otherwise 1 if *file* is a link, or 0 if otherwise The content of the file in a string Same as file%*file*&content	get
file%*file*&text or file%*file*	The array that holds all the text lines in the file, indexed numerically starting from 0 (the first line is file%*file*&text:0 or file%*file*:0). If *file* is a directory, the array contains a list of file names in that directory in "natural" orders returned from the operating system. The text lines are held in cache. Once accessed, it will be available even if the file is renamed or even deleted.	get
file%*file*&action:copy	Copy the file named by the variable to the file named by the set value (file%*fromfile*&action:copy = "{*tofile*}"; The {*tofile*} is in FileDir – {sys%croft:filedir}, or {sys%croft:wwwdir} or {sys%croft:textdir} provided the full	set

	path is specified.)	
file%*file*&action:delete	Delete the file unless the value is set to zero or blank	set
file%*file*&action:existence	Delete the file if the value is set to zero or blank. Note: file%*file*&action:existence is the opposite of file%*file*&action:delete.	set
file%*file*&action:rename file%*file*&action:name	Rename the file to the new name in the set value (file%*oldname*&action:rename = "{*newname*}"; The {*newname*} is in FileDir – {sys%croft:filedir}, or {sys%croft:wwwdir} or {sys%croft:textdir} provided the full path is specified.)	set
file%*file*&action:source	Copy the file named by the set value to the file named by the variable (file%*tofile*&property:source = "{*fromfile*}";) Note: file%*file*&action:source is the same as file%*file*&action:copy, except that the direction of the copying is reversed. The set value of the former is the "*fromfile*", while that of the later is the "*tofile*". (The {*tofile*} is in FileDir – {sys%croft:filedir}, or {sys%croft:wwwdir} or {sys%croft:textdir} provided the full path is specified.)	set
file%*file*&action:write	Write the lines in file%*file*&text up to the number of characters specified in the set value (0 or non-numeric for all). Note: file%*file*&text is held in cache. Nothing will be written to the actual file while the variable is being built up, until the write action.	set
file%*file*&action:append	Same as file%*file*&action:write, except that the number of characters specified are appended to the end of the file, instead of from the beginning.	set

Note: Accessing file%*file*&text multiple times (e.g. in a loop) does NOT incur overhead of file opening, seeking and closing. The whole file is loaded into the variable upon the first access of file%*file*%text. Subsequent accesses are directly from the variable itself. (PHP underlying the BEE system handles variable memory in a neat and efficient way.) Operations on file% variables will set status%file:*filename* and status%message:*filename* accordingly.

form is a value passed in via the URL argument list (GET) or a form submission (POST). For single-value Form variables, form%*name* is the same as sys%form:*name*. But for multiple-value Form variables, you can only access them via form%*name*.

Form variables can be set and cleared. That means you can modify the form submission values or even create somes to simulate a form submission.

Please note that all form% values (and sys%form values) are subject to automatic "washval" conversion before returning the value to avoid attack via online forms. To turn this feature off, set scheme%WashForm to only a comma (","). Optionally, you may turn off WashForm for a particular access level and above.

For example, if scheme%WashForm:editor being "," will turn off "washval" for forms if the current login has editor privilege or higher. This is useful if you allow BEE Values in editable content.

function is for properties of user-defined functions. It is read-only.

Variable	Value/meaning	Access
function%*funcName*:exists function%*funcName*:exist	1 if the function exists, or 0 if otherwise Same as function%*funcName*&exists	get

scheme is a setting for the website. It is saved to the website permanently until changed. It is usually used to specify constant settings which is referenced from the program code. Scheme values are loaded into the client session when it starts. Any changes to scheme values only affect the current session, and they will be all discarded after the session finish (e.g. browser close).

Scheme values can be explicitly reloaded (via "scheme reload") and saved (via "scheme save"). Once saved, the scheme values will be stored with the website. However, other sessions will not pick up the changes unless the reload the scheme.

If *file*& (except for "list&") is specified after scheme%, the settings in the specified scheme file will be used instead of the common site scheme file. In this case, the changes will be reloaded every time they are used, and saved immediately upon changes (which means all sessions will pick up the changes.)

There is a special scheme operation: "scheme%*file*&" without the *name* part. This itself returns a list of *names* in the scheme files. In addition, it has a "side-effect" of loading all scheme values in the scheme file into "*file*%*name*:*element*". That way, you can load the whole scheme file into a class under the Context in one go. e.g. "var scheme%*file*&;". (*file* must not be "list".)

There is yet another special scheme variable: "scheme%list&*name*". This array contains the list of all scheme files under the scheme subdirectory *name*. If *name* is missing, it will show all scheme files under the scheme directory itself.

General scheme settings (scheme%*name*) are usually used for common settings that require fast access and local update. Specific scheme files (scheme%*file*&*name*) are usually used to hold design parameters that require little or no updates, to simplify the coding. For example, you can store form field attributes and validation criteria into scheme files for the program to interpret while displaying a form. To reserve scheme name space, some applications create scheme subdirectory for its own scheme files (e.g. DRB use scheme%drb/...&... for its scheme files).

Operations on scheme% variables will set status%scheme and message%scheme accordingly.

session is a persistent value kept through out the client session. A client session is the period starting from the website first being accessed by a particular browser run, until that browser instance closes.

A session is implemented by a 128-bit session key stored at the client browser as a cookie. The session context is stored at the server side and therefore is secured from client-side hacking. The server uses the session key cookie from the client to determine which session context to load. A client cannot tap into a session unless it has the knowledge of the 128-bit key of the targeted session, which is virtually impossible.

Please note that Login and logout do not start and finish a session, even though it can be programmed to have all session variables erased and scheme settings reloaded upon logout to effectively simulate a new session.

sys (mostly read-only) is system data such as authentication attributes and site information.

Variable	Value/meaning	Access
sys%accesslevelnames:0 sys%accesslevelnames:1 sys%accesslevelnames:2 sys%accesslevelnames:3 sys%accesslevelnames:6 sys%accesslevelnames:8 sys%accesslevelnames:10	Public: not logged in Affiliate: authenticated at "friendly" site Member: logged in VIP: logged in with privilege Editor: logged in as editor Manager: logged in as manager Admin: administrator login (See "access" command for details.)	get
sys%accesslevelnames:Public sys%accesslevelnames:Affiliate sys%accesslevelnames:Member sys%accesslevelnames:VIP sys%accesslevelnames:Editor sys%accesslevelnames:Manager sys%accesslevelnames:Admin	0 1 2 3 6 8 10	get
sys%argv	The argv variable when running in command line mode	get
sys%auth:username sys%auth:realm sys%auth:accesslevel sys%auth:user	Username logged in Realm logged in Access level number Same as "sys%auth:username"	get set (only if *AllowSetAuth* CROFT Scheme is set
sys%auth:loginname sys%auth:accesslevelname	username@realm Access level name	get
sys%auth:*attr*	User attribute (e.g. Name, Tel)	get set (but not saved until "auth save") clear
sys%client:agent	Browser type	get(no array)

sys%client:ip sys%client:hostname sys%client:host sys%client:referrer	Client machine IP Client machine name Same as sys%client:hostname Referring page URL	
sys%context	The identifier of the current variable context. This value is useful in passing a local context to other scopes via the "link" command.	get(no array)
sys%croft:hostid sys%croft:owner sys%croft:service sys%croft:webserver sys%croft:webpath sys%croft:dbid sys%croft:dbtype sys%croft:dbhost sys%croft:dbname sys%croft:sitedir sys%croft:wwwdir sys%croft:filedir sys%croft:schemedir sys%croft:textdir sys%croft:vbdir	Host ID of the web hosting server Owner Code of the website Service Code of the website Server Name Path where the Scope is defined Database ID Database Type Database Host Database Name Site Directory WWW (Web) Directory File Directory Scheme Directory Text Directory VB (VirtualBase) Directory	get
sys%croftauth:dbtype sys%croftauth:dbhost sys%croftauth:dbname sys%croftauth:dbtable sys%croftauth:passwordtype sys%croftauth:adminaccesslevel sys%croftauth:dbusername sys%croftauth:dbpassword sys%croftauth:usernamefield sys%croftauth:passwordfield sys%croftauth:realmfield sys%croftauth:accesslevelfield sys%croftauth:activefield sys%croftauth:expiryfield	Database Type of Auth Database Host of Auth Database Name of Auth Database Table of Auth Password Type used in Auth The minimum level considered admin Database Username to Auth Database Password to Auth Field name of Username in Auth Field name of Password in Auth Field name of Realm in Auth Field name of Access Level in Auth Field name of Active in Auth Field name of Expiry in Auth	get
sys%default:textfile	The default text-file value for "text name" in the "text" command	get set
sys%form:*name*	Value submitted from a form (same as form%*name* except that form%*name* can have multiple values) (See "form%" about "washval" security.)	get set clear
sys%formescaped	Return 1 if the form variable submitted with '\' escaped '\\' (PHP 4 behaviour), or 0 otherwise.	get(no array)
sys%header	HTTP Header	set(no array)
sys%https	Return 1 if HTTPS is on or 0 otherwise.	get(no array)

sys%ineditor	Return 1 if the page is running in the text editor, or 0 otherwise.	get(no array)
sys%ini:*name*	Return the "ini" variable named by *name* (as defined in the "ini.php" file).	get(no array)
sys%isssl	Same as "sys%https"	get(no array)
sys%mimetype:*file*	MIME Content Type of the specified file, or the current page if no file is specified. (*file* is in FileDir – {sys%croft:filedir}, or a CROFT directories provided the full path is specified.)	set(no array)
sys%output:content sys%output:length	The content of the output buffer The current length of the output buffer	get(no array)
sys%php:*name*	The PHP variable indexed by *name* in $HTTP_SERVER_VARS (PHP 4) or $_SERVER (PHP 5)	get
sys%phpversion	The PHP Version that BEE is running on.	get(no array)
sys%random:*n*	A randomnumber between 1 and *n*	get(no array)
sys%randomnumber:*n*	Same as "sys%random:*n*"	get(no array)
sys%post:data sys%post:type sys%post:charset sys%post:host sys%post:mimetype sys%post:contenttype	Data posted Content Type of the data posted Character Set of the data posted Host posted to Same as "sys%post:type" Same as "sys%post:type"	get(no array)
sys%password:old sys%password:new	The old password and new password in a password changing operation. This is set-only and can only assigned an array with two elements: "old" and "new". The operation will set status%auth:password and message%auth:password accordingly.	set array only
sys%self	Same as "sys%urlpart:pathpage"	get(no array)
sys%scheme:linelastgot	The line number of the last read scheme item in the scheme file. (For array items, the line number of the first element will return.)	get
sys%scheme:linenextset	The line number of the next scheme item to set in the scheme file. (This value affect only the next set-scheme operation only. If the operation is setting an existing item, this value will be ignored. After the operation (regardless the value is ignored or not), it will be reset to zero – which	get set

	means new items are set at the end of the file.)	
sys%session:id sys%session:idlesince	The 128-bit Session ID in hexadecimal The time (in timestamp format) last authentication feature is accessed	get(no array)

sys%session:idletimeout
 	
sys%session:expiry
 	
sys%session:expire
sys%session:realmchar
	If set to non-zero, it indicates the number of seconds in which the session will be automatically logged out unless an authentication feature (e.g. going into a member only page) is accessed.
	The time (in timestamp format) by when the session will be automatically logged out.
	Same as "sys%session:expiry"
	The "Realm Character" (usually '@').
		get(no array)
		set(no array)

sys%site:name sys%site:url	The site name (scheme%SiteName) The idURL (can be overwritten by scheme%SiteURL)	get(no array)
sys%stdin	The input stream from begin to end	get(no array)
sys%textpagelist	The list of all pages with TEA (Text Edit Area). This always returns an array of which each item is in path/page format (with no leading '/'). If :*element* is specified, the result will always be blank.	get- always array
sys%time: sys%time:sec sys%time:second sys%time:seconds sys%time:usec sys%time:microsecond sys%time:microseconds sys%time:secusec sys%time:zoneoffset	The current time in number of second since 1 Jan, 1970 (GMT). (This is the default for any unrecognised element.) Same as sys%time: The "microsecond" (millionth second) part of the system clock. The "microseconds" since 1 Jan, 1970 (GST). Time Zone offset in second (e.g. Sydney standard time is GMT+10, the offset is 36000 seconds)	get(no array)
 sys%url:protocol sys%url:server sys%url:path sys%url:page sys%url:full	The current URL and its parts: http:// (or https://, same for below) http://mysite.com http://mysite.com/sub/dir http://mysite.com/sub/dir/mypage.htm http://mysite.com/sub/dir/mypage.htm?	get(no array)

	abc=123&xyz=How+are+you	
sys%urlssl:server	https://mysite.com	
sys%urlssl:path	https://mysite.com/sub/dir	
sys%urlssl:page	https://mysite.com/sub/dir/mypage.htm	
sys%urlssl:full	https://mysite.com/sub/dir/mypage.htm?abc=123&xyz=How+are+you	
sys%urlpart:server	mysite.com	
sys%urlpart:path	/sub/dir	
sys%urlpart:page	mypage.htm	
sys%urlpart:pathpage	/sub/dir/mypage.htm	
sys%urlpart:pathpageargs	/sub/dir/mypage.htm?abc=123&xyz=How+are+you	
sys%urlpart:args	abc=123&xyz=How+are+you	
sys%urlpart:full	http://mysite.com/sub/dir/mypage.htm?abc=123&xyz=How+are+you	
sys%urlredirect:pathpage sys%urlredirect:pathpageargs sys%urlredirect:args	The corresponding values of sys%urlpart for a redirected page (403 or 404 http server redirect)	get(no array)
sys%urlshown:pathpage	sys%urlredirect:pathpage if the page is redirected, or sys%urlpart:pathpage if otherwise.	get(no array)
sys%washform	The current "WashForm" settings. i.e. the scheme%WashForm:*accesslevel* value being applied to the current session.	get(no array)

text is an editable web page content. It is saved to the website permanently until changed. The value in a "text" class variable is used by the "text" command to display the web page content. If the visitor got Administrator access, an Edit Text icon will appear at the end of the text, which will click into the Text Edit Screen for online update. The new content will be saved into the "text" class variable.

All these are done by the "text" command without accessing the "text" variable. You usually access a "text" variable only if you want to process the content for some reason. In the "text" command, only the *name*:*item* part of the variable needs to be specified.

Each web page got a "text" class of its own, to nicely separate text contents from each other. If you want to refer to "text" variables for another page, you can use the *file*& part of the variable (e.g. text%/products/index.htm&headline).

There is a special scheme operation: "text%" without the *name* part. This itself returns a list of *names* in the "text" class. In addition, it has a "side-effect" of loading all "text" values in the scheme file into "*page*%*name*:*element*". That way, you can load all "text" class variables into a class (e.g. /product/index.htm%) under the Context in one go. e.g. "var text%*page*&;".

upload contains information about file or files uploaded via the client browser.

Variable	Value/meaning	Access
upload%[*index*&]*name*:name upload%[*index*&]*name*:type upload%[*index*&]*name*:size upload%[*index*&]*name*:uploaded	The name of the uploaded file The type of the uploaded file The size of the uploaded file non-zero if the file has been uploaded	get
upload%[*index*&]*name*:saveas	The name of the file to be saved on the server side. Uploaded files are held only temporarily until the end of the page run. To "actualise" the uploaded file, the "saveas" element must be set to a valid server file name (which is in FileDir – {sys%croft:filedir}, or {sys%croft:wwwdir} or {sys%croft:textdir} provided the full path is specified).	get set

Note: *index*& is used only for array input (e.g. <input type=file name="clientfile[]">). The first file is represented as "upload%0&clientfile" and the second "upload%1&clientfile" and so on.

A typical upload form is like:

```
<form method=post enctype="multipart/form-data" action="save.htm">
<input type=file name="clientfile">
<input type=submit value=Upload>
</form>
```

In the above example, "save.htm" will do <u>var upload%clientfile:saveas = "serverfile";</u> to save the uploaded file into "serverfile" on the server storage. The status of the "saveas" operation is in variables status%upload and message%upload.

3.2 BEE Variable Value

There are only one data structure in BEE Value – array of strings. This is the general form of a BEE Value:

`{[#|@][`*`class`*`%][`*`file`*`&]`*`name`*`[:[#]`*`element`*`][=`*`value`*`][|`*`conv`*`[:`*`arg`*`[,...]]]}`

Note 1: You can add a leading '$' to make a "naked" Value, which can be used outside of the Context (i.e. in other scripts or the HTML scope or attributes). For example, in the scope of the HTML Text: `Click here: <a href="${myLink}">${myCaption}</a>`

Note 2: With the optional sizeof operator (the '#' sign right after the open curly bracket), the BEE Value will evaluate to the length of the evaluated string or the number of elements in an array. (Please see Example 3.)

Note 3: With the optional keyof indicator (the '@' sign right after the open curly bracket), the BEE Value will evaluate to the key of the elements in an array. This looks trivial as the element name is itself the key. However, it is useful to get the key from a positional element. e.g. {@pet:#2} will evaluate to the key of the third element (first one is #0).

Note 4: If the optional *value* is specified (prefixed by '='), it will be assigned to the variable before the variable is being used (even before sizeof and BEE Conversion). If the variable name is omitted, a "null variable" will be created to temporarily hold the *value*. (A null variable value may sound trivial but can be useful as input to the Conversion without having to create a variable.) A null variable is treated as an array. If you want to force it to become a string (for the conversion for example), you can use ":=" instead of "=".

Note 5: The "conv" part is called a BEE Conversion, and it will be applied to the BEE Value before it is returned. Please see the "var" tag for details. (Operator sizeof is applied after the BEE Conversion.)

The value of a BEE Variable is specified by a pair of curly brackets surrounding the BEE Variable name. Whether the variable is in a "scaler" format (with the *element* part) or an array format (without the *element* part), it will be converted to string once it was surrounded by a pair of curly brackets. The resulting string will replace the curly bracket and the variable name in it. This operation is done repetitively until all BEE Variables in the string are resolved.

Please note that the variable evaluation is done literally. If variable "a" is "5", "{a}0" will evaluate to "50", and "{a} + 8" to "5 + 8" (not 13). To make calculation, you need to use "(expr){a} + 8".

Example 1:

```
var myinfo = "age";
var age = 18;
display "What is your {myinfo}?\n";  // What is your age?
display "I'm {{myinfo}}.\n";  // I'm 18.
```

Example 2:

```
// "customer" is a database select result currently
// pointing to a man called Peter Smith.
var person = "(database)customer";
var sexName = "(array)M=>Male,F=>Female";
var title = "(array)Male=>Mr,Female=>Ms";
display "Hello, {title:{sexName:{person:Sex}}} {person:Name}";
// Output: Hello, Mr Peter Smith
```

Example 3:

```
var abc = "IAmFine";
// {abc} -> IAmFine
// {abc|words} -> {abc:|words} I Am Fine
// {#abc} -> 1 (as the array has only one element)
// {#abc:} -> 7 (and that element evaluates to 7-char string)
// {#abc:|words} -> 9 (and it is "I Am Fine", 9-char long)
```

Please note that ${#myname} returns the number of elements in the *myname* array, and ${#myname:} returns the length of its default element (the one indexed by blank in the *myname* array). That's why from the above example, {#abc} evaluates to 1 and {#abc:} evaluates to 7.

3.3 BEE Conversions

As discussed before, a variable value can be appended a '|' and the name of a BEE Conversion, which convert the value into another value to be used in the expression instead of the original value.

The Conversion name can be optionally followed by a ':' and an argument string. If multiple arguments are required, it will be in a comma-delimited string format. All occurrences of "@value" in the argument string will be replaced by the input string.

There are two types of BEE Conversion: Intrinsic Conversions (built-in to the BEE system) and User-defined Conversions (defined in the code as a function). The latter take precedence. That is if a User-defined Conversion has the same name as one of the Intrinsic Conversions, the User-defined Conversion will be used instead.

3.3.1 Intrinsic Conversions

Intrinsic conversions are a set of built-in conversions that handles common needs in programming a commercial website.

In the following descriptions, the input string is passed by value instead of by reference (unless explicitly specified). It does not matter when converting a value, but when converting a variable, the changes will NOT be reflected on the variable.

Example:

```
var x = "apple";
var x conv="uppercase" display=1;     // display "APPLE"
display "{x}";                        // x is still "apple"
var y = "{x}" conv="uppercase";
display "{y}";                        // y is "APPLE"
```

Note: Convert-by-reference has a "side-effect" that an undefined variable will be defined after being converted by reference.

In the following descriptions of BEE Conversions, when a *true* or *false* is said to be returned, in fact, a literal 1 or 0 will be return respectively. You can use the result quoted or unquoted. They means the same. However, quoting even a boolean is safer in case there's a program bug that set a boolean-to-be as a null string.

Here is a list of Intrinsic BEE Conversions:

Conversion	**Input Type**	**Output Type**	**Description**

`++`	String	String	Increase the input string by one numerically.
			This input string is passed by reference. It must be either a variable as in:
			`var i conv="++";`
			or a value casted as a variable as in:
			`var j = "(var)i" conv="++";`

`+=`	String	String	Increase the input string by the number in the argument string numerically.
			This input string is passed by reference. It must be either a variable as in:
			`var i conv="+=:3";`
			or a value casted as a variable as in:
			`var j = "(var)i" conv="+=:3";`

`--`	String	String	Decrease the input string by one numerically.
			This input string is passed by reference. It must be either a variable as in:
			`var i conv="--";`
			or a value casted as a variable as in:
			`var j = "(var)i" conv="--";`

`-=`	String	String	Decrease the input string by the number in the argument string numerically.
			This input string is passed by reference. It must be either a variable as in:
			`var i conv="-=:3";`
			or a value casted as a variable as in:
			`var j = "(var)i" conv="-=:3";`

`?`	String	String	Take the argument string as a comma-delimited list of three arguments. The first argument is a logical expression representing a condition. If the condition evaluates to true, return the second argument. Otherwise, return the third one. Yes, the input argument is ignored. So you can use the shorthand {|?*condition*,*trueVal*,*falseVal*}.
`ana`	String	String	Return "an" if the input string starts with a vowel letter (A, E, I, O, U or their lowercase letter) or "a" otherwise.
`base64decode`	String	String	Take the input string as a MIME base64 encoded string and return the decoded string.
`base64encode`	String	String	Take the input string and return the MIME base64 encoded string.
`basename`	String	String	Take the input string as a file name with path (directory specification with "/") and return the file name with the path stripped off.
`bracketed`	String	String	Return the input string preceded with the first argument string and succeeded with the second argument string, if the input string is not blank. Otherwise, return blank.
`brackets`	String	String	Same as "bracketed"

`bracket`	String	String	Same as "bracketed"
`caseless`	String	String	Take the input string and produce a regular expression that can match the string case insensitively. For example, "abc" will become "[Aa][Bb][Cc]".
`classname`	String	String	Take the input string as a class name, but if it happens to be in a variable form, extract only the class name. Return the class name as such.

`concat`	String	String	Append the argument string to the input string.
			This input string is passed by reference. It must be either a variable as in:
			`var s conv="concat:abc";`
			or a value casted as a variable as in:
			`var x = "(var)s" conv="concat:abc";`

`concatvar`	String	String	Same as "concat", except that you can use variable costing like: `var s conv="concatvar:(var)y";` Like in "concat", "concatvar" is passed by reference as well. "concatvar" is usually used to join values together without macro substitution. Note: If the argument variable is specified as an array, only the default element will be appended to the input variable.
`count`	Array	String	Return the number of elements in the input array.
`countstr`	String	String	Return the number of times the argument string occurs in the input string.
`countstric`	String	String	Return the number of times the argument string occurs in the input string, ignoring case.
`countstring`	String	String	Same as "countstr"
`countstringic`	String	String	Same as "countstric"
`countstringre`	String	String	Same as "countstrre"
`countstrre`	String	String	Return the number of times the substring that matches the regular expression in the argument string occurs in the input string.
`crypt`	String	String	Encrypt the input string and return the result. The argument string specifies the type of encryption: StdDES (default), ExtDES, MD5 and Blowfish. If the argument string is 13 characters or more, it will be taken as an encrypted string to verify

			against the input string, and *true* is return if it is a match or *false* otherwise. In this case, the type of encryption can be optionally specified as the second argument.
`dbregexp`	String	String	Return a regular expression that the underlying database platform can understand, from the "common" regular expression contained in the input string. e.g. In MySQL, Common regular expression "a*b.c" will be converted to "^a.*b\.c$".
`dec`	String	String	Same as "-="
`decrease`	String	String	Same as "-="
`default`	String	String	If the input string is *false*, return the argument string. Otherwise, return the input string unchanged.
`dirname`	String	String	Take the input string as a file name with path (directory specification with "/") and return the path with the file name stripped off.
`empty`	String	String	Return *true* if the argument is '0' (literal zero or numeric zero), blank (”) or not defined; return *false* if otherwise.
`escape`	String	String	escape:quote or escape:quotes Add a backslash ('\') in front of any single or double quotation marks found in the input string. escape:single, escape:singlequote or escape:singlequotes Add a backslash ('\') in front of any single quotation marks found in the input string. escape:double, escape:doublequote or escape:doublequotes Add a backslash ('\') in front of any double quotation marks found in the input string. escape:comma or escape:commas Add a backslash ('\') in front of any commas found in the input string. escape:nonalnum Add a backslash ('\') in front of any non-alphanumeric characters found in the input string. escape:crlf Add a backslash ('\') in front of any carriage-return or line-feed characters. escape:lfcr, escape:newline, escape:linefeed Same as escape:crlf escape:escape

			Add a backslash ('\') in front of any backslashes.
`formatnum`	String	String	Take the argument string as a comma-delimited argument list. The first argument is the length of the resulting string, the second is the number of decimal places, and the third is a stuff character. The conversion will round the input number to the required decimal place (default 0), comma the integer part to separate groups of three digits from the left of the decimal point. Then if the resulting string is shorter than the specified length, the stuff character will be pre-pended (on the right to make up the required length.
`formatnumber`	String	String	Same as "formatnum"
`found`	String	String	Return *true* if the argument string occurs in the input string, or *false* if otherwise.
`foundic`	String	String	Return *true* if the argument string occurs in the input string ignoring case, or *false* if otherwise.
`foundre`	String	String	Return *true* if the substring that matches the regular expression in the argument string is found in the input string, or *false* if otherwise.
`highlight`	String	String	Take the argument string as a comma-delimited argument list. The first argument is the "begin-string", the second is the "end-string", and the rest is a list of highlighted strings. The conversion will find all occurrences of any highlighted strings from the input string and precede each with the begin-string and succeed each with the end-string. (If the begin-string is blank, it will be taken as "<b><i><u>"; if the end-string is blank, it will be taken as "</u></i></b>".)
`hmstosec`	String	String	Convert a duration from the H:M:S format to the number of seconds. e.g. 2:34:56 to 9296
`htmldisp`	String	String	Replace HTML sensitive characters with their proper representation. e.g. & to &amp;, < to &lt; and > to &gt;
`htmlstrip`	String	String	Strip any HTML tags found in the input string and convert hard spaces (&nbsp;) to soft ones and multiple spaces to single ones.
`if`	String or Array	String or Array	Take the argument string as a comma-delimited list of two arguments. The first argument is a logical expression representing a condition. If the condition evaluates to true, return the original input string unchanged. Otherwise, return the second argument.

			The second argument is sometimes called the alternative value, when the condition is not met. If the input is an array, the second argument and onwards will form the "alternative array", which will be returned when the condition evaluates to false.
`inarray`	Array	String	Search the input array for an element having the same value as the argument string and return *true* if it is found, or *false* if otherwise.
`inc`	String	String	Same as "+="
`increase`	String	String	Same as "+="
`inverse`	Array	Array	Convert the input array into a new array by swapping the key and the value. e.g. if the input is k1=>v1,k2=>v2, the output will be v1=>k1,v2=>k2. If more than one element have the same values, only the key of the last element having that value will be used.
`isset`	String or Array	String	For input string, return *true* if it is defined, or *false* if otherwise. For input array, return *true* if it is defined (even without any elements,) or *false* if otherwise.
`key`	Array	Array	Return an array which contains all the keys of the input array. The index of the output array starts from 0 (for the first element that contains the first key of the input array).
`keyof`	Array	String	Take the argument string as a numeric index and return the key of the element in the input array. The argument string is default to 0, which means the key of the first element.
`keys`	Array	Array	Same as "key"
`list`	Array	String	Generate a string representation of the input array according to the format specified in the argument string. The default format is: `('@key'=>'@value'),()`

			The format contains two small brackets, defining 5 format parts:
			OPEN **(** REPEAT **)** SEPARATOR **(** BEFORELAST **)** CLOSE
			OPEN The string before the list
			REPEAT The format of the element with the key represented by @key and value by @value.
			SEPARATOR The string that comes between any two adjacent elements.
			BEFORELAST The last separator. If blank, the normal separator will be used.
			CLOSE The string after the list.
			Here is an example:
			`var arr = "(array)goalie=>John, defender=>Matt, striker=>Bill";`
			`display "{arr|list:We have (@value as the @key), ( and ); what a winning team!}";`
			The output would be:
			`We have John as the goalie, Matt as the defender and Bill as the striker; what a winning team!`

`listall`	Array	String	Same as "list"
`listval`	Array	String	Same as "list" except that blank values will be excluded
`listvalue`	Array	String	Same as "listval"
`listvalues`	Array	String	Same as "listval"
`lowercase`	String	String	Convert a all letters in a string to lowercase.
`meansfalse`	String	String	Returns *true* if the input string is "false", "no", "off" or a numeric "0" (case insensitive). Returns *false* if otherwise. This conversion is usually used to handle user input.
`meanstrue`	String	String	Returns *true* if the input string is "true", "yes", "on" or a numeric non-zero (case insensitive). Returns *false* if otherwise. This conversion is usually used to handle user input.
`match`	String	Array	Match the regular expression in the argument string to the input string, and return a 10-element array (regardless the number of matches). The first element of the array (indexed by 0) is the string matching the regular expression. If small bracket pairs are found in the regular expression, the second element (indexed by 1) will contain the substring enclosed by the first left open bracket and its corresponding closing bracket exclusively, and the third element (indexed by 2) will contain the one by the second left bracket and so on.

			An empty array will return if it is not a match.
`merge`	Array	Array	Take the argument string as an array specification and merge it to the input string. That is to append the argument array to the input array and return the result array. For any common elements (with the same element name in both arrays), the one in the argument array element will replace that of the input array element. (Numeric key will be ignored and the element will take on the positional index – 0 for the first element and so on.)
`newkey`	Array	Array	Return an array containing strings (usually numeric) which can be used as the keys for new elements of the input array (i.e. guarantee uniqueness.) No element is created. The argument string is the number of keys to be returned. Default is to return one key only.
`newkeys`	Array	Array	Same as "newkey"
`newval`	Array	Array	Return an array containing strings (usually numeric) which are not equal to any elements of the input array (i.e. guarantee uniqueness.) No element is created. The argument string is the number of values to be returned. Default is to return one value only.
`newvalue`	Array	Array	Same as "newval"
`newvalues`	Array	Array	Same as "newval"
`nosysargs`	Array	Array	Remove all possible system arguments from the input array and return the result array.
`parseform`	String	Array	Convert a query string passed via the URL from a form GET submission, into an array of which each element correspond to a submitted value (indexed by the entry name.) The conversion "parseform" is usually used when you have acquired the query string in ways other than the URL. For those from the URL, it would be easier to access them via sys%form or form%.
`parseurl`	String	Array	Take the input string as a URL and returns the parts of it including scheme", "host", "port", "user", "pass", "path", "query", and "fragment".
`parsevar`	String	Array	Take the input string as a BEE variable name and return an array of at most five elements indexed by "class", "file", "name", "element" and "conv", representing the five parts of the BEE variable name in the input string. At least "class" and "name" will be returned ("class" is default to "value" and "name" can be

			blank) unless the input string is empty, in which case an empty array (one without elements) will be returned.
`plural`	String	String	Convert the number in the input string into a quantity with the unit specifies in the argument string. If the absolute value of the number is greater than one, the unit will be in plural form (which is generated by observing only simple rules like -s, -ies, etc.) If the number is zero, "No" followed by the plural unit will be returned.
`pluralunit`	String	String	Like "plural" except that it only returns the unit part without the quantity.

`Pop`	Array	String	Take the last element of the input array away, and return that element.
			The input array is passed by reference. It must be either a variable as in:
			`var a conv=pop;`
			or a value casted as a variable as in:
			`var a = "(var)arr" conv=pop;`

`precede`	String	String	Return the input string preceded with the argument string if the input string is not blank. Otherwise, return blank.
`present`	Array	String	Generate a string representation of the array, one item per line, or if the item is multiple lines, show each line under the key heading.
`printf`	String	String	Take the format from the input string and substitute the variables from the comma-delimited argument string, and return the result.
`propercase`	String	String	Convert the first letter of every word in the input string into uppercase. Optionally, you can use the argument string to specify a word boundary character. The default is space.
`push`	Array	Array	Append the argument string to the input array and return the result array. The input array is passed by reference. It will get an extra element at the end after this operation.
`regexp`	String	String	Return a regular expression from the "common" regular expression contained in the input string. e.g. Common regular expression "a*b.c" will be converted to "^a.*b\.c$".
`repeat`	String	String	Concatenate the input string to itself a number of times as specified in the argument string and return the result.
`replace`	String	String	Take the argument string as a comma-delimited

			list of two arguments, then find all occurrences of the first argument in the input string and replace them with the second argument. (See also "xreplace")
`replacere`	String	String	Take the argument string as a comma-delimited list of two arguments, then find from the input string all matches of the regular expression contained in the first argument and replace them with the second argument. If the regular expression contains small brackets, $*num* in the second argument will be replaced by the matching content of the "*num*"th brackets in the regular expression. (See also "xreplacere")
`reverse`	Array	Array	Returns an array in reverse order of the input array (both the value and the key.) That is the last element would become the first and the first would become the last.
`round`	String	String	Round the input string as a floating point number to the precision (number of digits after the decimal point) specified in the argument string.
`search`	Array	String	Search the input array for an element having the same value as the argument string and return the key of that element. If not found, return the second argument, or blank if none is specified.
`sectohms`	String	String	Convert a duration from the number of seconds to the H:M:S format. e.g. 9296 to 2:34:56
`setkeys`	Array	Array	Take the argument string as an array specification and use them to replace the keys in the input array, the return the result array. If there're duplicated keys in the argument array, all elements assigned with that key will be overwritten by the last one, but it will take the position of the first one. If there're more keys than are required, the excessive keys in the argument array will be ignored. If there're less keys than are required, the excessive elements in the input array will all have the blank key, and the last blank-keyed element will prevail taking the position of the first blank-keyed element.
`setvalue`	Array	Array	Set all the elements in the input array to the argument string and return the result array.
`shift`	Array	String	Take the first element of the input array away,

			and return that element.
			
			This input array is passed by reference. It must be either a variable as in:
			
			`var a conv=shift;`
			
			or a value casted as a variable as in:
			
			`var a = "(var)arr" conv=shift;`

`shuffle`	Array	Array	Randomise the position of the input array, and return the result array. If the argument string is "key" or "keys", only the keys are randomised. If "value" or "values", only the values are randomised. If "all" or "both", the keys and the values are both randomised.
`slice`	Array	Array	The argument string is taken as a comma-delimited list of two arguments: offset and length. The input array is then "stripped" starting from the offset position (first element is 0) until the number of elements specified by "length" is collected, or the end of the input array if "length" is not specified, or "length" elements before the end if "length" is negative.
`sort`	Array	Array	Sort the input array then return the sorted array as the result. The argument string is taken as a comma-delimited list of arguments, which has the following effects: value (default): keep key-value association valueonly: sort values only ignoring all keys key: sort the keys (only if "valueonly" is not set) reverse (or rev): sort in reverse order numeric: sort in numeric order string: sort in string order (only if "numeric" is not set)
`split`	String	Array	Take the input string as a space-delimited list and split the list into an array of its elements in the same order. If a argument string is specified, it will be used as the delimiter instead of space.
`splitre`	String	Array	Same as "split" except that the delimiter argument can be a regular expression.
`strtotime`	String	String	Convert a date-time string in the input string into timestamp format (number of seconds since 1 Jan, 1970 GMT). Common language like yesterday, next friday, last week, 3 months, -2 years, etc.

`stringtoarray`	String	Array	Convert the input string into an array of which each element contains a character of the corresponding position in the input string. (Element 0 is the first character.)
`strtoarr`	String	String	Same as "stringtoarray"
`strftime`	String	String	Convert a timestamp (number of seconds since 1 Jan, 1970) in the input string into the format specified in the argument string, according to the standard of the Unix "date" command. (Default argument is "%c".)
`strlen`	String	String	The length of the input string. (This is useful to avoid the ambiguity of the sizeof operator ('#') which may be applied to an array, as "strlen" work on string only.)
`strpos`	String	String	Search the input string for the argument string and return the position if found (0 is the first character). If not found, return -1. You can optionally specify the second argument, which is the position in the input string where the search starts.
`substr`	String	String	Return a sub-string of the input string starting from the position indicated by the number in the argument string. The first character is position 0. The argument string may contain the maximum sub-string length in the optional second argument separated from the first one by a comma. If the length is not specified, the sub-string will extend to the end of the input string.
`substring`	String	String	Same as "substr"
`succeed`	String	String	Return the input string appended with the argument string if the input string is not blank. Otherwise, return blank.
`surround`	String	String	Same as "bracketed"
`teafilter`	String	String	Used by the TEA Editor to adjust the content before saving. The input is NOT supposed to be macro-expanded. This conversion deals with "defects" in the Editor like putting "&" in a variable as "&amp;", and correct it.
`tolower`	String	String	Same as "lowercase"
`toproper`	String	String	Same as "propercase"
`toupper`	String	String	Same as "uppercase"
`trim`	String	String	Trim the leading and trailing white spaces from the input string.
`translate`	String	String	Take the array specification in the argument string (in the form of comma-delimited list of

			[key=>]value) and use the input string as an array index (the code) to find the corresponding value (the translated text), and return the value.
`truncate`	String	String	Take the argument string as a comma-delimited list of two arguments. The first argument is a number indicating the length which the truncation will occur if the input string is longer than. The optional second argument specifies a trailer to append to the end of the result if the truncation did occur. e.g. the Conversion of "truncate:10,..." will turn "This is an apple." to "This is an...".
`unescape`	String	String	Remove backslashes ('\') from the input string unless it is the last character.
`unique`	Array	Array	Remove all duplicated values from the input array and return the result array.
`unless`	String or Array	String or Array	Take the argument string as a comma-delimited list of two arguments. The first argument is a logical expression representing a condition. If the condition evaluates to false, return the original input string unchanged. Otherwise, return the second argument. The second argument is sometimes called the fallback value, when the condition is met. If the input is an array, the second argument and onwards will form the "fallback array", which will be returned when the condition evaluates to true.
`unshift`	Array	Array	Append the argument string to the input array and return the result array. The input array is passed by reference. It will get an extra element at the beginning after this operation.
`uppercase`	String	String	Convert a all letters in a string to uppercase.
`urlargstrip`	String	String	Strip from the input string the first occurrence of the question mark ('?') and everything afterwards.
`urldecode`	String	String	Convert all occurrences of a percentage sign ('%') followed by two hexadecimal digits into their corresponding literal characters.
`urlencode`	String	String	Convert all non-alphanumeric characters into a percentage sign ('%') followed by two hexadecimal digits, with the exception of hyphen ('-'), underscore ('_') and dot ('.').
`value`	Array	Array	Return an array which contains all the values of the input array. The index of the output array starts from 0 (for the first element that contains the first key of the input array).

`values`	Array	Array	Same as "value"
`varname`	String	Array	Take the input string as a variable name (optionally with initializer and conversion) and convert it into a 7-element array indexed by: var – The variable name (the input string without the initializer and conversion) class – The class part file – The file part name – The name part element – The element part value – The initializer conv – The conversion part
`verb`	String	String	Take the input string as an English verb and convert it into a form specified in the argument string, which can be do: return the input string (no conversion) doing: convert ~ or ~e into ~ing did: convert ~ or ~e into ~ed done: convert ~ or ~e into ~ed be: return "are" if the input string is 0 or a number greater than 1 or less than –1. Otherwise, return "is". where ~ represents the verb in the input string.
`washval`	String	String	Find all substrings that match the BEE Value Syntax, and replace the first character of such substrings by the first argument (default '[') and the last character by the second (default ']').
`washvalue`	String	String	Same as "washval"
`washvalues`	String	String	Same as "washval"
`word`	String	String	Separate joint words by insert a space before each capital letter or '@' sign. All caps string (e.g. acronyms) are recognised as one word.
`words`	String	String	Same as "word"
`xreplace`	String	String	Same as "replace" except that it handles special characters and escapes cleaner.
`xreplacere`	String	String	Same as "replacere" except that it handles special characters and escapes cleaner.

3.3.2 User-defined Conversions

User-defined conversions are functions written by the programmer that follows a certain input and output (argument and result) standard so that it can be used as a BEE Conversion. User-defined conversions take precedence over Intrinsic conversions, so that programmers can overwrite intrinsic conversions by providing his or her own.

<u>Input</u>

The conversion function will be given the following arguments:

arg%value

The value string regardless of the input structure. If the input happens to be in array form, this argument will contain the default element of the array.

This is used for functions that expects a string input.

This argument can be represented as "arg%function:value" (following the "arg" class convention) or "arg%value:".

arg%values

The value array regardless of the input structure. If the input happens to be a string, it will be put into the default element of arg%values.

This is used for functions that expects an array input. Avoid using this for string input. Here is an example of a pitfall:

```
var petowner:katty = "John Smith";
display "{petowner:katty|myconv}";
...
function myconv
{
      // arg%function:value is "John Smith"
      // arg%values is katty=>"John Smith"
      // arg%values:katty is "John Smith"
      // but the default element of arg%values
      //    is undefined
}
```

arg%argv

The argument array. The first element is arg%argv:0, which is the function name. The second element is arg%argv:1, which is the first argument, and so on.

arg%argc

The number of elements in the argument array. The value is the same as {#arg%argv}. Please note that the counting includes the function name in arg%argv:0, so if you have one argument, arg%argc is 2.

This argument can be represented as "arg%function:argc" (following the "arg" class convention) or "arg%argc:".

arg%arg	The argument string of the conversion. It is specified in the calling command after the conversion name and separate from it by a ':'. The system will present the argument string to the function as it is after variable evaluation on the command line. This allows the code to access the full argument string (instead of those in arg%argv). This argument can be represented as "arg%function:arg" (following the "arg" class convention) or "arg%arg:".
arg%var	The variable name (this is valid only if converting a "bare" variable as in var myVar conv="myConv"; (Please note that user-defined conversion is operating in a local Context, so your need to link it to the parent. e.g. parent "{arg%var}"; var argVar =& "{arg%var}"; and changes to argVar will be reflected to the converted variable.) This argument can be represented as "arg%function:var" (following the "arg" class convention) or "arg%var:".

Output

The conversion function is to return the value via result%function. If the returned value contains the default element and no other elements, the conversion will return a string. Otherwise, an array will be returned instead. (Even when the array contains one single element, as long as it is not the default element, the conversion will still return an array.)

Example 1 – Simple increment:

```
function inc
{
      var by = "{arg%argv:1}";
      if ({#by:} == 0) var by = 1;
      var result%function = "(expr){arg%value} + {by}";
}

display "2" conv=inc;                 // 3
display "2" conv=inc:8;               // 10

var a = 2 conv=inc;
display "a is {a}<br>\n";             // a is 3

var a conv=inc;
display "a is still {a}<br>\n";       // a is still 3
```

Example 2 – Increment by reference:

```
function inc
```

```
{
      var by = "{arg%argv:1}";
      if ({#by:} == 0) var by = 1;
      var result%function = "(expr){arg%value} + {by}";

      if ({#arg%var:} > 0)
      {
            // We need to modify the variable too
            parent "{arg%var}";
            var argVar =& "{arg%var}";
            var argVar = "{result%function}";
      }
}

var a = 2 conv=inc;
display "a is {a}<br>\n";            // a is 3

var a conv=inc;
display "a is now {a}<br>\n";        // a is now 4
```

Example 3 – Increment by multiple arguments:

```
function inc
{
      var by = "{arg%argv:1}";
      if ({#by:} == 0) var by = 1;
      var result%function = "(expr){arg%value} + {by}";

      if ({#arg%var:} > 0)
      {
            // by reference
            parent "{arg%var}";
            var argVar =& "{arg%var}";

            clear arg%argv:0;  // The function name is out of the loop
            foreach (arg%argv)
            {
                  var by = "{foreach}";
                  if ({#by:} == 0) var by = 1;
                  var result%function:{foreach:key}
                        = "(expr){arg%value} + {by}";
            }

            if ({#arg%var:} > 0) var argVar = "(var)result%function";
      }
}

var a = 2 conv=inc;
display "a is {a}<br>\n";            // a is 3

var a conv=inc;
display "a is now {a}<br>\n";        // a is now 4
```

```
var a conv=inc:10,20,30;
display "a:{a|list}<br>\n";     // a:’’=>’14’,’1’=>’14’,’2’=>’24’,’3’=>’34’
```

3.3.3 To String or Not to String

Some intrinsic conversions are used to convert string to string and some array to array. There are others that convert array to string as well. While the structure (whether it is a string or an array) of the output is controlled by the conversion, the input is not.

If the input is in element form (string) and the conversion takes a string as input, it would be a simple string conversion. If the input is in array form (element part missing) and the conversion takes an array as input, it would be a simple array conversion.

However, it is more complicated when the input is array form but the intrinsic conversion expects a string input, or when the input is a string but the intrinsic conversion expects an array. For a user-defined conversion, the system cannot tell whether it expects a string or an array, which adds to the complications.

To take the complication further, the array form is commonly used as a simple variable and means its default element (the one indexed by blank), and they’re indistinguishable in syntax.

Here is a table to summarise the situations and how they were handled (in sequence of precedence):

Input value	Conversion	How to handle
is an array or simple variable Examples: *class%name* "(array)..." "(db)..."	expects an array	Convert the input as an array.
	expects a string	The system will apply the conversion to each element in the input array one by one. This automatically takes care of the case of a simple variable, in which the default element will be converted.
	is user-defined	The system calls the user-defined function passing it the standard arguments.
is a string Examples: *class%name*:*elem* *class%name*: "(expr)..."	expects a string	Convert the input as a string
	expects an array	The system will take the conversion as a user-defined function and passes the standard arguments to it accordingly.
	is user-defined	The system calls the user-defined function passing it the standard arguments.

Please note that if a conversion can process both array and string, it is suggested to make the Input Value explicit array or string to avoid ambiguity. For example:

```
var x = -1;
display "{x:|if:@value > 0,unknown}";
```

The BEE Variable "x" is specified as a string (x: instead of x). Otherwise, the conversion "if" would be on the array of x, in which @value would not make sense, and the conversion would not work. This magic string-making colon is useful even in variable-less value initialization such as "{:=(expr){a} + {b}|if:@value > 10,not even ten}".

Please also note that curly bracketed variables always evaluate to a string, even the conversion output an array (in which case, the default element will be used to expand the curly bracket). If you really want to convert a string to an array, you need to use the "conv" parameter.

Example:

```
var myString = "abc";
var myArray = "{myString|strtoarr}";        // Got a blank
var myArray = "{myString}" conv=strtoarr;   // Got an array
```

4 BEE Syntax

BEE comes with two interchangeable syntax: BEE Script and BEE Tag.

BEE Script is program code like syntax with common constructs like variable assignment, if-else, looping, function and other controls. It starts with the command name, then followed by Attribute Name-Value Pairs in the form of name=value, and terminated with a semi-colon ';'. It is useful in sections where processing algorithm is the dominant process (occasional outputs can be done with the display statement.)

BEE Tag is HTML like tag system that starts with "<bee" or "<bee:" followed by a tag name then a series of Attribute Name-Value Pairs (name=value), and ends with ">" or "/>". (The tag name can be defined as an attribute as "tag=*tagname*".) It is useful in sections where displaying web page content is the dominant process (occasional controls can be inserted in BEE Tag forms, inter-mixing with the content and HTML Tags without causing syntax problems with non-BEE compliant authoring tools or web server.)

Internally, all BEE Script statements and structures are translated into a series of BEE Tags before being sent to the BEE Tag parser. So by learning BEE Script, you would be able to read and write BEE Tag with no problem.

In this document, the following presentation convention is followed:

Constructs within a pair of square brackets are optional.

'...' means repeats are allowed.

'|' is used to separate two or more alternatives of which one and only one will be employed.

4.1 General Syntax

There are syntax rules that apply to both BEE Script and BEE Tags.

1) Command Names must be a string of alphanumeric characters or underscore that starts with a non-numeric character. Leading underscore string will be ignored.

2) There must be at least a white space before the Attribute Name. If there are white spaces inside the value, the whole value should be quoted by a pair of double or single quotation marks.

3) Spaces around the equal sign in an Attribute Name-Value Pair are optional.

4) Unquoted Attribute Values are terminated by a white space or the terminator (';' for BEE Script and '>' for BEE Tag – if the BEE Tag is terminated by '/>', the last attribute value must be quoted).

5) Quoted Attribute Values open and close with the same quotation mark (both double or both single). If the value contains the quotation mark itself, it can be escaped by a back-slash. (Quoted Attribute Values can contain white spaces including line break. So you can specify multi-line values.)

6) An "Absolute" Attribute Value is one that is NOT evaluated automatically for BEE Variables before passing to the tag function (please see BEE Variables for details about the evaluation process.) An Attribute is "absolute" if the Attribute Name is preceded with an exclamation mark ('!').

7) Attribute Names must be a string of alphanumeric characters or underscore that starts with a non-numeric character. You can be optionally preceded by an underscore '_' or an exclamation mark '!'. Attribute Names are case insensitive and all spaces and underscores in it will be ignored.

8) If the Attribute Name and the equal sign is omitted, the Attribute Name is assumed to be the Tag Name. For example, "clear abc;" means "clear clear=abc;".

9) The Attribute Name "tag" is reserved for parser use. (In fact, the Tag Name is the Attribute Value of the "tag" parameter. "<beeclear clear=abc>" is the same as "<bee tag=clear clear=abc>".)

10) If the BEE Command Name is prefixed by an object name and a '%' sign, the object name and the '%' sign will be extracted and a new Attribute Name "this" will be inserted with the object name as the Attribute Value. For example, "myobj%func ..." will be translated to "func this=myobj ...".

One point that may not be obvious from the quotation rules (4 and 5): Because BEE Variables are non-typed, quotation marks do not change the type of the value they enclosed; they only serve to allow spaces inside the value without causing parsing problem. For example, `var abc = "flower";` is the syntactically and semantically the same as `var abc = flower;` or even `var "abc" = flower;`.

However, if there are white spaces in the value string, the quotation marks are required. For example, `var abc = "red flower";` is OK, but `var abc = red flower;` will assign only "red" into variable abc. The string "flower" will be taken as a separate parameter and ignored.

4.2 BEE Script Syntax

BEE Script: `tagname [name=value ...];`

Here are the rules specifically for BEE Script syntax:

1) A BEE Script section starts with <script language="bee"> and ends with </script>. You can optionally put src="*filename*" in the <script> tag. e.g. <script language="bee" src="myfunc.bs">.

2) White spaces between ';' and the start of the next statement are ignored.

3) Texts between '//' and the end of line, and between '/*' and '*/' are ignored.

4) A BEE Script Statement starts with a Command Name and ends with a semi-colon ';'.

5) Between the Command Name and the end of the command (the semi-colon) is a series of name=value pairs called attributes (except for "access", "if", "elseif", "foreach", "while" and "function" in which small brackets and curly brackets are usually involved.)

6) White spaces between two BEE Script statements are ignored.

4.3 BEE Tag Syntax

BEE Tag: `<bee`*`tagname`* `[`*`name1=value1`* `...]>`

Here are the rules specifically for BEE Tag syntax:

1) A BEE Tag is put in the HTML scope and there is no need to be surrounded by <script> tags.

2) White spaces between two BEE Tags are part of the HTML document and will therefore show up in the document source (not necessarily the display though). White spaces inside a BEE Tag are ignored unless within a Quoted Attribute Value.

3) Comments are surrounded by '<!--' and '-->' as in HTML tag. ('//', '/*' and '*/' are not recognised in BEE Tag and will be taken literally without hiding any display.)

4) A BEE Tag starts with a '<bee' or '<bee:' immediately followed by the Command Name, and ends with '>' or '/>'.

5) Between the Command Name and the end of the command ('>' or '/>') is a series of name=value pairs called attributes, even for "access", "if", "elseif", "foreach", "while" and "function", in which a "block-end" tag in the form of "</bee*CommandName*>" is required to close the block. For example, <beeif …> should be closed by </beeif>.

BEE Tag name is preceded by "<bee" or "<bee:" with no spaces in between, and the tag ends with ">" or "/>", just like an HTML tag or XML tag does. Optionally, the symbol "<" and ">" can be replaced by "[" and "]" respectively, which is useful when your authoring tool complains about a BEE Tag being misplaced (e.g. inside an HTML Tag).

4.4 From Script to Tag

BEE Script is abstracted from BEE Tag. All BEE Script statements are to be pre-processed into BEE Tags before being passed to the compiler.

The standard Script-to-tag translation is as following:

BEE Script: *`tagname`* `[`*`name=value`* `...];`

is translated to

BEE Tag: `<beetagname [name=value ...]>`

Some BEE Script statements have special syntax to make it more "natural" to programmers of other common scripting languages.

For example, Conditional statements ("if" and "elseif") and loop statements ("foreach" and "while") must have a small brackets containing the condition at the end of the statement. The block of statements that follow (for conditional execution or loop body) must be surrounded by a pair of curly brackets unless it is a single statement. Please see individual command syntax for details.

For function declaration, the "function" statement is NOT followed by a semi-colon (';'). Instead, it must be followed by an open curly bracket that starts the function body which will be ended by a close curly bracket. The pair of curly brackets cannot be omitted even when there is only one statement in the function body.

5 BEE Commands

BEE is designed to be simple to use. There are only 35 BEE Commands and 6 of them have no arguments and 11 others have just one. Among those with more than one argument, majority of the arguments have a default value that is rarely specified explicitly. That's why the syntax is plain and simple (partly helped by the versatility of the macro-style BEE Variables.)

There are nine groups of BEE Intrinsic Constructs:

1) **Variable Operations**
   var
   clear
   group
   link

2) **Conditional**
   if
   else
   elseif
   switch
   case
   hide
   show

3) **Loop**
   foreach
   while
   continue
   break

4) **Module Calling**
   function
   return
   global
   parent
   include
   exec
   exit

5) **Remote Calling**
   connector
   call

6) **Authentication**
   access
   login
   logout

7) **Data Access**
   database
   dbtree

8) **Socket**
socketcreate
socketbind
socketlisten
socketaccept
socketconnect
socketread
socketwrite
socketclose
socketcontrol

9) **Specific Commands**
mailto
text
auth
scheme
output

Even BEE Values are non-typed, BEE Commands do expect a certain pattern in some particular cases. Those patterns are notated in this section in the following way:

string	A string of characters. If white spaces are among them, the string needs to be enclosed by a pair of single or double quotation marks. If the enclosing quotation mark appears in the string, it needs to be "escaped" by inserting a backslash ('\') in front of the quotation mark.
char	A single character.
num	A string of numeric numbers (0 to 9, excluding dot)
bool	A non-zero number or a non-null string (except '0') to indicate a *true* value; and 0, '0', or a null string to indicate a *false* value. Here are some examples of boolean conditions:

1	true
'1'	true
'...'	true
0	false
'0'	false
''	false

Note 1: '...' denotes any non-null string except '0' (so '00' is true)

Note 2: '!' in front of a boolean value will negate it (e.g. !0 is true)

name	A string of alphanumeric characters or underscore that starts with a non-numeric character. Leading underscore string will be ignored.
var	A BEE Variable Name (Please see "BEE Variable Name").
value	A pattern in the form of (*type*)*string*, where type is interpreted by the specific BEE Command that the *value* is in.

class	A valid name for the "class" part of a BEE Variable. (Please see "BEE Variable Name".)
file	A valid file path and file name. Directory names are separated by forward slashes ('/'). For relative path, the system include path will be searched for the named file.
condition	A string that evaluates to a logical expression that can determine a TRUE or FALSE value. The logical expression follows the same format of other programming languages like PHP. (e.g. == means equal; != means not equal; >, >=, <, <= and other logical operators like &&, || etc can be used in the usual sense.)
statement	A valid BEE Script statement.
tag	A valid BEE Tag.
convSpec	A BEE Conversion name optionally followed by a colon (':') and an argument string.
souce	A special parameter in the (*type*)*string* format. It is for the "foreach" tag only. Please see "foreach" under "BEE Command Reference" for more details.
accessControlSpec	A special parameter for the "access" tag only. Please see "access" under "BEE Command Reference" for more details.
URL	A valid URL.
emailAddress	A valid email address.
action	A string of that indicate an action specific for the command.

const	A constant value in the "const" class. Some commands such as "socket" can interpret a string as a constant value.
	e.g. the followings are equivalent:
	socketcreate domain=const%socket:AF_INET;
	socketcreate domain="AF_INET";

The above patterns are interpreted after the variable evaluation. For example, if xyz is a variable containing "34", the value of `12{xyz}56` will be evaluated to "123456" and therefore is a *num*, instead of *string* as it may appear to be.

5.1 Variable Operations

5.1.1 var – assign a value to a variable, create an object, or display a value

The "var" command is the mother of all BEE Commands. It has some variance in syntax to performance different operations. The reason that these operations are not separated into different commands is that it reads more naturally to have an equal sign in it. It would be easier to learn as variance of one "assignment" command instead of many others with different parameters one needs to remember.

To ease the burden of the documentation of the "mother", we have the "link" command separated into a different section following this one.

Variable Mode:

BEE Script: `[var] `*`var`*` [=[!] `*`value`*`] [conv=`*`convSpec`*`];`

BEE Tag: `<bee[var] var=`*`var`*` [[!]value=`*`value`*`] [conv=`*`convSpec`*`]>`

Constructor Mode:

BEE Script: `[var] `*`var`*`[%] = new `*`constructor`*` [`*`name`*`=`*`value`*` ...];`

BEE Tag: `<bee[var] var=`*`var`*`%function:new value=`*`constructor`*`>`
`<bee`*`var`*`%new [`*`name`*`=`*`value`*` ...];`

Display Mode:

BEE Script: `display `*`value`*` [conv=`*`convSpec`*`];`

BEE Tag: `<bee[var] value=`*`value`*` [conv=`*`convSpec`*`]>`

Syntax Notes:

The command name "var" can be omitted in both BEE Script and BEE Tag syntax. This is because "var" is the centre piece of the BEE Tag repertoire, and has been made the default tag.

<u>Variable Mode</u>

In the Variable Mode, the command accepts a value and assigns it to the BEE Variable. The operation is "silent" (no display). If the "= value" part is omitted, the variable will still be accessed, but there will be no effect nor any display. One exception though of the "value-less var" operation is scheme file loading. e.g. `var scheme%`*`myschmfile`*`&;`. In this case, the command name "var" cannot be omitted. (Otherwise, it will be taken as a function call syntax-wise.) For details, please see "scheme" under the "Variable Name" section.)

<u>Constructor Mode</u>

Constructor Mode is for creating an object (or class). The *constructor* is the name of the function that defines the characteristics of the object and is generally taken as the object name. The *name*=*value* pairs are the parameters for the constructor. For details, please see "Objectes and Classes". (The '%' after the variable name is ignored. It is there only to make the syntax look better – more natural to see an object name followed by '%'.)

In BEE Tag, there is no one single command to create an object. It is done by assigning the "new" function of the object by the *constructor* name, then call the function with *var*%new followed by the constructor's parameters.

Display Mode

The Display Mode is characterised by the omission of the variable name. In fact, the "display" in BEE Script syntax is not a BEE Command. It is only an alias (shorthand writings) for the "var" command without the variable name *var*. That is why there is no "display" BEE Tag. The omission of *var* from the "var" BEE Tag indicates a display operation.

Parameters

var is the variable name. Please see the "Variable Name" section.

value is the value in the form of "(*type*)*string*". The value type is one of the followings: "literal" (or "lit", the default), "expression" (or "expr"), "var", "database" (or "db"), and "array". The whole value (type and string) is evaluated for BEE Variables before being used for assignment (Variable Mode) or display (Display Mode). For details of variable evaluation, please see the "Variable Value" section.

Type	**String**

`literal`
`lit`
`(default)`
	A string to be literally assigned or displayed. It is the default type, and therefore the type specification (literal or lit) is usually omitted.

`expression` `expr`	An expression to be arithmetically evaluated before being assigned or displayed.
`database` `db`	A database result name (see "database" tag). The database record of the current position will be retrieved and the fields assigned to the BEE Variable as individual elements. The position will move forward to the next database record after the retrieval. ("database" type is for assignment only. Display Mode will display nothing, but the position in the record set will still move forward.)
`var`	A variable name. The "var" type is different from the "literal" type. While "literal" is always a string, "var" can be a string or an array (if there is no element part in the variable name.) This is useful in array assignment or for BEE Conversions (conv=...) that require array input.

`array`	A comma separated list with each item being in the form of either "[string]" or "[key]=>[string]". In the Variable Mode, the resulting array will be assigned to the BEE Variable. (If the variable to be assigned to has an element part, the element part will be ignored.) In Display Mode, it will display nothing unless the array is converted into a string through a BEE Conversion (conv=...). Example: "Dog, Cat, Guinea Pig" would produce an array of three values: "Dog", "Cat" and "Guinea Pig". "John=>Dog, Mary=>Cat, Sam=>Guinea Pig" would produce the same array with their owner's name as keys (the element indexes). In the array value (the "[string]" part), leading and trailing spaces will be trimmed off unless quoted with single quotation mark. Comma inside an item can be escaped with a backslash to avoid being taken as a separator.

Please note that there are different forms to represent a variable in the "value" parameter. There is one simple rule: {...} always evaluate to a string.

(var)*class*%*name*	The array in variable *class*%*name*, whether it contains single or multiple elements.
(var)*class*%*name*:*elemet*	The string in the variable element *class*%*name*:*elemet*.
(var)*class*%*name*:	The default element of the *class*%*name* array. (The trailing ':' is compulsory to distinguish it from the array form.)
{*class*%*name*}	Same as (var)*class*%*name*:, NOT (var)*class*%*name*
{*class*%*name*:*elemet*}	Same as (var)*class*%*name*:*elemet*.
{*class*%*name*:}	Same as (var)*class*%*name*: (but the trailing ':' is optional, and usually omitted.)

(If the "*class*%" part is "value%", it can be omitted.)

Notes:

If you do not want BEE to evaluate the value for BEE Variables, you can precede the parameter name "value" with an exclamation mark ('!') (as mentioned in "General Syntax"). However, for BEE Script form, there is no parameter name. In that case, you can add the '!' after the equal sign but leaving at least one white space before the value.

Examples:

```
var var1 = "abc";
var var2 =! "{var1}";     // var2 will get "{var1}" literally, not "abc".
                          // "{var1}" will be evaluated with "var2" later.
var var1 = "def";
display "{var2}";         // will display {var2} at the time
                          // i.e. display "def", not "abc";

// Beware of the syntax
var a = ! "xyz";   // Good
var a =! "xyz";    // Good
var a=! "xyz";     // Good
var a=!"xyz";      // Bad! Taken as: var a = "!xyz";
```

conv is the BEE Conversion, which is a function applied after the value has been evaluated for BEE Variables but before it is assigned or displayed. For details, please see the "BEE Conversion" section.

name is not really a fixed parameter. The name *name* can be anything (except for the parameter names above) and can be more than one. They are used to pass parameters to the remote function (as indicated by the Attribute Name-Value Pairs in the syntax description). If a value is preceded by an "@" sign, the rest of the content indicates a JavaScript expression (instead of a BEE expression.)

5.1.2 link – create a Reference to a variable

BEE Script: `var var =& var [context=string];`

BEE Tag: `<beelink link=var var=var [context=string]>`

A variable name is in fact an alias to a content storage (or more technically, an entry in the symbol table). You may have two different variable names referring to the same variable. This relationship is called a Reference. Please note that the two variables (or even more) are of equal footing. You cannot say that one is real and another is a shadow. They're both handles to the same content storage. They are both real and they are both shadows.

The "link" command establishes a new Reference (on the left-hand-side) to an existing variable (on the right-hand-side). References can only be created across compatible structure in the BEE Variable hierarchy. i.e. You can only link class to class, name to name, or element to element. Linking incompatible structure does not cause an error but has no effect at all.

Example:

```
// Link all variables in oldClass to newClass
var newClass% =& oldClass%;

// Link all elements in oldClass%oldName to newClass%newName
var newClass%newName =& oldClass%oldName;
```

```
// Link the two elements only
var newClass%newName:newElm =& oldClass%oldName:oldElm;
```

If you link two classes together, changing a variable in one class will cause the corresponding variable (that of the same name) in another class to have the new value as well. Moreover, creating new variables in one class will cause the other class to have the same new variable containing the same value. In fact, there is only one class.

This general principal applies to two linked variables too. Changing the value of an element in one variable will change the same element of the other, and creating an element in one will cause the other to have the same added to it, because they are the same variable.

Please note that the "clear" command is in fact clearing the reference to the variable's content storage, not the storage itself. If you have created multiple references on the same variable, clearing one will NOT destroy others. (Clearing the last reference to the variable effectively remove any means to access the variable from the Context. After that, the question whether the content storage still exists is meaningless.)

Parameters

link is the name of the new variable (shadow) created to link to an existing variable (real).

var is the name of the existing variable (real) that the new variable (shadow) is linked to.

context is the context identifier of the existing variable to be linked to. The {sys%context} variable always contains the context identifier of the current environment. With this facility, you can pass the whole context (and all the classes in it) around.

For example, you can pass the parent's context to a function to allow it to access the parent's variables (can be done with the "parent" command as well). On the other hand, a function can pass its context to the caller via the result%function variable so that the caller can access the function's local context using the "link" command. (Yes, the local context of the function survive its exist but not accessible unless through the context parameter.)

Please be very careful about linking context. It should be generally avoided unless there is an absolute need to do so and you know exactly what you are doing.

5.1.3 clear – undefine a variable

BEE Script: `clear var;`

BEE Tag: `<beeclear "var">`

Clearing a variable means to remove the variable from the Context so that it will not be defined in the Context anymore. It is different from assigning a blank to it. An element containing a blank will still appear in, say, a "foreach" loop and be counted as an element in the sizeof ('#') operation, but a cleared element will not because it does not exist anymore.

You can clear the entire array by omitting the element part. You can even clear the entire class (except for system classes) by omitting both the name and element parts.

Example:

```
clear abc:xyz;      // Clear value%abc:xyz
clear abc:;         // Clear value%abc: (the default elemnent)
clear abc;          // Clear value%abc (the entire array)
clear myclass%;     // Clear myclass% (the entire class)
```

Not all variables can be cleared. Please see "System Classes" for details.

Please note that if the variable has more than one Reference to it, clearing one Reference does not remove the others. Please see "link" for more details.

5.1.4 group – promote elements into variables (a matrix)

BEE Script: `group` *`var`* `[delim=`*`char`*`] [matrix=`*`class`*`] [result=`*`class`*`] [inverse=`*`bool`*`];`

BEE Tag: `<beegroup` *`var`* `[delim=`*`char`*`] [matrix=`*`class`*`] [result=`*`class`*`] [inverse=`*`bool`*`]>`

The "group" command creates out of a one-dimensional array a two-dimensional array, which is called a "matrix". The command generates under the "matrix" class a new variable for each element in the subject variable (*var*), then group the new variables by a common suffix which is separated from the variable name by a delimiter (the "delim" parameter, default to be '_'.)

After the operation, result%keys will hold all distinct variable names created, and result%fields will hold all distinct element names created. (The class name "result" can be overwritten by the result parameter.)

Examples:

```
// If we got the followings:
// cart:product_15 = "Orange"
// cart:qty_15 = "4"
// cart:unit_15 = "kg"
// cart:product_23 = "Spinach"
// cart:qty_23 = "1"
// cart:unit_23 = "bunch"

group cart;

// Now we have:
// matrix%15:product = "Orange"
// matrix%15:qty = "4"
// matrix%15:unit = "kg"
// matrix%23:product = "Spinach"
// matrix%23:qty = "1"
// matrix%23:unit = "bunch"
// result%fields = product, qty, unit
// result%keys = 15, 23
```

Here is a schematic illustration of the example:

Before

cart:product_15	cart:qty_15	cart:unit_15
cart:product_23	cart:qty_23	cart:unit_23

```
group cart;
```

After
(Variable "cart"
is still intact.)

matrix%15:product	matrix%15:qty	matrix%15:unit
matrix%23:product	matrix%23:qty	matrix%23:unit

result%fields

result%fields:0 is "product"	result%fields:1 is "qty"	result%fields:2 is "unit"
result%keycount:product = 2	result%keycount:qty = 2	result%keycount:unit = 2

result%keys

result%keys:0 is "15"	result%fieldcount:15 = 3
result%keys:1 is "23"	result%fieldcount:23 = 3

In fact, the "group" command creates a matrix out of an array variable. This is useful in a spreadsheet-like input where the columns are the fields and the row numbers are the keys. In such case, each form input tag should be named as *columnName_rowNumber*. If the spreadsheet is to be used in a database operation, the *rowNumber* should be replaced by the value of the read-only key. Please see "*dbobj*%keyfield" in the "database" command.

Parameters

delim is the delimiter character that splits the original element name into the element name part (before the delimiter) and the variable name part (after the delimiter) of the new variable. The default of "delim" is underscore: '_'.

If there no the delimiter characters are not found in the original element name, one is assumed in front of the name. That is the element name part of the new variable is blank. On the other hand, if multiple delimiter characters are found in the original element name, only the last one will be used as the delimiter.

matrix specifies the class under which the variables are created. Existing variables in the matrix class will not be cleared but may be overwritten by newly created variable of the same name if any. To avoid "polluting the environment", please make sure the matrix class is clean before using it for the "group" command. The default of "matrix" is "matrix".

result specifies the class for the %fields and %keys variables. (Please see parameter "matrix".) The default value of "result" is "result".

inverse indicates the inverse-format of the original element names. If inverse is true, the original element name is taken as *n_e*, where *n* is the variable name part and *e* is the element name part of the new variable. The default is false, which means the original element name is in the format of *e_n*, which is the usual case described above.

There are some points to make about the "matrix" and "result" parameters. If the matrix created is later on used in a database operation, please remember to specify the same matrix class in the "database" command as in the "group" command that created the matrix. Also, please use the database object name as the "result" parameter so that the "database" command will pick up the %fields and %keys variables properly.

Also, if the "group" command and the "database" command are in two different Contexts, please make sure you link the matrix and result classes properly using either the "global" or the "parent" command where necessary.

After the "group" command is executed, the following BEE Variables are made available (in additional to the matrix variables created):

result%fields	Distinct element names created. e.g. Name, Code, etc.
result%keys	Distinct variable names created. e.g. 15, 23, 56, etc.
Remarks:	If "*f*" is in *result*%fields and "*k*" is in *result*%keys, variable *matrix*%*k*:*f* should contain the value of the corresponding element in the matrix.
result%fieldcount:*key*	Number of elements in variable *key*. They usually contain the same number, which is the number of "columns" in the matrix.
result%keycount:*field*	Number of variables that got element *field* in it. They usually contain the same number, which is the number of "rows" in the matrix.

5.2 Conditional

5.2.1 if – conditional execution of a block

```
BEE Script:   if (condition) statement;
              [else statement;]
or            if (condition) { statement; ... }
              [else { statement; ... }]

BEE Tag:      <beeif "condition">
                    tag
                    ...
              [<beeelse>
                    tag
                    ...]
              </beeif>
```

Note: The *condition* is surrounded by a small bracket and therefore is taken literally. While you can (and sometimes need to) quote individual values in the *condition*, please do not quota the whole condition with double or single quote. Otherwise, the whole *condition* will be taken as a string (and will most likely to be evaluated to true unless it is blank or 0).

The "if" command starts a conditional block structure and therefore an "else" or a sequence of "elseif" tags can follow the conditional block.

The *condition* will be evaluated for any BEE Variables before the truth value is determined. Please note that variables are evaluated as macros and therefore one should bear in mind the logical expression syntax. In particular, you need to quote strings as required:

Example:

```
var num = 2;
var item = 'water melon';

if ({num} > 10) display 'We got plenty';  // Valid
// The condition will evaluate to (2 > 10)

if ('{num}' > 10) display 'We got plenty';  // Valid
// The condition will evaluate to ('2' > 10), which is OK.

if ('{item}' == 'lemon') display "Can't eat them";  // Valid
// The condition will evaluate to ('water melon' == 'lemon').

if ({item} == 'lemon') display "Can't eat them";  // Invalid
// The condition will evaluate to (water melon == 'lemon').
```

If the condition is blank, it will evaluate to false.

After the "if" command is executed, the following BEE Variables are made available:

`result%if:istrue`	Set to 1 if the condition evaluates to true, or 0 otherwise

`result%if:condition`	Set to the condition after variable evaluation

5.2.2 else – alternative execution block

BEE Script: `(see "if", "elseif", "access", "hide" and "show")`

BEE Tag: `(see "if", "elseif", "access", "hide" and "show")`

"else" starts an alternative block which will be executed if the previous conditional block is not (e.g. the condition in an "if" command evaluates to false, or the access-control-specification in an "access" command does not match with the current session.)

Only "if", "elseif", "access", "hide" and "show" commands can be followed by "else" after the conditional block. If an "else" command is "out-of-place", it will be ignored (and the block that follows will be joined with the previous one.)

Example:

```
if ({form:Qty} > 0) display "Thank you!";
else display "Please enter a positive quantity.";
```

5.2.3 elseif – conditional alternative execution block

```
BEE Script:   if ...
              elseif (condition) statement;
              [else statement;]
or            if ...
              elseif (condition)] { statement; ... }
              [else { statement; ... }]

BEE Tag:      <beeif ...>
                     ...
              <beeelseif "condition">
                     tag
                     ...
              [<beeelse>
                     tag
                     ...]
              </beeif>
```

Note: The *condition* is surrounded by a small bracket and therefore is taken literally. While you can (and sometimes need to) quote individual values in the *condition*, please do not quota the whole condition with double or single quote. Otherwise, the whole *condition* will be taken as a string (and will most likely to be evaluated to true unless it is blank or 0).

The "elseif" command starts a conditional block structure and therefore an "else" or a sequence of another "elseif" tag can follow the conditional block.

The *condition* will be evaluated for any BEE Variables before the truth value is determined. Please note that variables are evaluated as macros and therefore one should bear in mind the logical expression syntax. In particular, you need to quote strings as required. Please see the "if" command for examples.

The "elseif" construct is equivalent to "else" followed by "if". The only difference is that you do not need to have multiple curly brackets and excessive indentation for clarity.

Example 1: Using "else" then "if"

```
if ({form:Qty} > 100) {
      var deliveryCharge = 0.0;
      display 'Free delivery.';
} else {
      if ({form:Qty} > 50) {
            var deliveryCharge = 8.0;
            display 'Delivery charge discounted.';
      } else {
            if ({form:Qty} >= 3) {
                  var deliveryCharge = 10.0;
                  display 'Thank you.';
            } else {
                  var abort = 1;
                  display 'Minimum quantity is 3.';
            }
      }
}
```

Example 2: Using "elseif"

```
if ({form:Qty} > 100) {
      var deliveryCharge = 0.0;
      display 'Free delivery.';
} elseif ({form:Qty} > 50) {
      var deliveryCharge = 8.0;
      display 'Delivery charge discounted.';
} elseif ({form:Qty} >= 3) {
      var deliveryCharge = 10.0;
      display 'Thank you.';
} else {
      var abort = 1;
      display 'Minimum quantity is 3.';
}
```

If the condition is blank, it will evaluate to false.

After the "elseif" command is executed, the following BEE Variables are made available:

`result%elseif:istrue`	Set to 1 if the condition evaluates to true, or 0 otherwise
`result%elseif:condition`	Set to the condition after variable evaluation

5.2.4 switch – conditional execution of blocks based value matching

BEE Script:

```
switch (value)
{
case value:
       statement;
       ...
       [break;]
[case value:
       statement;
       ...
       [break;]]
...
[default:
       statement;
       ...
       [break;]]
}
```

BEE Tag:

```
<beeswitch "value">
<beecase "value">
       tag
       ...
       [<beebreak>]
[<beecase "value">
       tag
       ...
       [<beebreak>]]
...
[<beecase>
       tag
       ...
       [<beebreak>]]
</beeswitch>
```

Note 1: In BEE Tag form, "case" without a value is the "default" case. (In BEE Script form, the alias "default" will be translated to a "case" command with no parameter.)

Note 2: All contents (scripts, HTML codes or text), if any, between "switch" and the first "case" will be ignored.

The "switch" command specify a value for matching with the one in the "case" command. If one is matched, the block specifies after the "case" command will be executed until a "break" command is encountered, then exist the "switch" block.

If there is no "break" statement in the block of matching "case" value, the execution will fall through to the next block until a "break" command is executed. So it is possible that multiple blocks are executed before the exist of the "switch" block.

If none of the "case" command got a value matching the one in the "switch" command, the "default" block if exists will be executed. If the "default" block is not specified, the whole "switch" block will be skipped with no execution at all.

Example:

```
display "Please wear ";
switch ("{patron}")
{
case "woman":
case "girl":
      display "skirt";
      break;
case "man":
      display "tie and ";
      // Fall through
case "boy":
      display "shirt";
      break;
default:
      display "properly";
}
```

5.2.5 case – execution block matching a value

BEE Script: `(see "switch")`

BEE Tag: `(see "switch")`

Note: In BEE Tag form, "case" without a value is the "default" case. (In BEE Script form, the alias "default" will be translated to a "case" command with no parameter.)

The "case" command specifies a value and starts a block which will be executed if the specified value matches the value in the previous "switch" command.

5.2.6 hide – unconditional non-execution of a block

```
BEE Script:   hide statement;
              [else statement;]
or            hide { statement; ... }
              [else { statement; ... }]

BEE Tag:      <beehide>
                     tag
                     ...
              [<beeelse>
                     tag
                     ...]
              </beehide>
```

The "hide" command specifies the following command or block of commands to be skipped in execution. It is equivalent to "if (false)". "hide" is useful in debugging or leaving inactive code in the script for future reactivation.

The script form of the "hide" command is rarely used because you can always inactivate a block of code by putting a pair of /* and */ around it. (Commented text are stripped before compilation but blocks hidden by "hide" are still compiled in under an "if (false)". In both cases, it will not show up to the client browser because BEE is a server side script. No execution means no display.)

In Tag form, "hide" is useful in hiding block of HTML code from the client browser. If you simply commented out the code with a pair of <!-- and -->, the inactive block will still show up in the client browser because HTML comments are stripped only at the client display, not even from the page source.

Note: "hide" starts an "if" block structure and therefore an "else" or an "elseif" tag can follow the conditional block.

5.2.7 show – unconditional execution of a block

```
BEE Script:   show statement;
              [else statement;]
or            show { statement; ... }
              [else { statement; ... }]

BEE Tag:      <beeshow>
                     tag
                     ...
              [<beeelse>
                     tag
                     ...]
              </beeshow>
```

The "show" command specifies the following command or block of commands to be executed. It is equivalent to "if (true)" and is trivial in nature. It is included in the design for the ease of reversing the effect of "hide".

For example, you can prepare two blocks of codes, one for debugging and one for live. Usually, you put the debug code into the "hide" block and the live code into the "else" block. When you want to use the debug code temporarily, you can change the "hide" command to a "show" command. When you finish, change "show" back to "hide" to resume live operation.

Note: "show" starts an "if" block structure and therefore an "else" or an "elseif" tag can follow the conditional block.

5.3 Loop

5.3.1 foreach – loop through a variable or a data access result

```
BEE Script:   foreach [maxiter=num] (source [as var]) statement;
or            foreach [maxiter=num] (source [as var])
                     { statement; ... }
```

```
BEE Tag:        <beeforeach source var=var [maxiter=num]>
                       tag
                       ...
                </beeforeach>
```

"Foreach" loop is commonly used to populate an HTML table. For example, you can put a template data row within a "foreach" loop, such that in each iteration a data record is extracted for a display.

It is easier to have it done than said.

Example (Script Form):

```
database "customer" query="select * from Company
      where Balance > 1000";
display '<table>\n';
foreach ((db)customer as custrec)
{
      display '<tr>\n';
      display '<td>{custrec:CompanyName}</td>\n';
      display '<td>{custrec:ContactPerson}</td>\n';
      display '<td>${custrec:Balance}</td>\n';
      display '</tr>\n';
}
display '</table>\n';
```

Example (Tag Form):

```
<beedatabase "customer" query="select * from Company
      where Balance > 1000">
<table>
<beeforeach (db)customer var=custrec>
      <tr>
            <td>${custrec:CompanyName}</td>
            <td>${custrec:ContactPerson}</td>
            <td>$${custrec:Balance}</td>
      </tr>\n';
</beeforeach>
</table>
```

Parameters

foreach (the *source*) is the structure to loop through. There are four types of source: var (default), csv, database, and dbtree.

(var)*source*

"var" type is the default if no source type is specified. The variable name *source* is taken as the name of the array to loop through. (If a single element is specified, only that element will be used and the loop will be executed only once.)

The elements of the array will be extracted, one in every iteration, and assigned to the "Loop Variable": "foreach:value" is the element value, "foreach:key" is the element key. As a short hand, "foreach" (the default element of the loop variable) contains the same value as "foreach:value".

Example:

```
foreach (sys%auth)
      display "User's {foreach:key} is {foreach}<br>\n";
// Sample display:
//    User's username is jack
//    User's Name is Jack Lee
//    User's Tel is 98765432
```

The following BEE Variables will be made available within a "var" loop:
("foreach" is to be substituted by the Loop Variable name if one is explicitly specified.)

`foreach:key`	The current key (index) of the Loop Variable
`foreach:value`	The current value of the Loop Variable
`foreach:`	Same value as foreach:value
`result%foreach:iteration`	The iteration count. It contains 1 in the first iteration and 2 for the second etc.

(csv)*source*

"csv" type is very similar to the "var" type. The only difference is that instead of the usual {foreach:key}, {foreach:value} (or {foreach}), the Loop Variable will contain an array derived from the comma-delimited value list contained in the source element. (The "key" is insignificant in this case. If you're interested in its value, it is stored in result%foreach:key.)

The "csv" type is useful in processing a CSV file.

Example:

```
foreach ((csv)file%myFile) {
      display "Month: {foreach:#0}<br>\n";
      display "Income: {foreach:#1}<br>\n ";
      display "Expense: {foreach:#2}<br>\n ";
}
```

(database)*dbobj* or (db)*dbobj*

"database" type can be shortened to just "db". The *dbobj* that follows is from a previous "database" command, as shown in the example. (Please see the "database" command for more details about the data access mechanism.)

Data records will be fetched from the last data access via the database object (*dbobj*), one record in every iteration, and assigned to the "Loop Variable". The fields are assigned to individual elements, indexed by the field name.

Example:

```
foreach ((db)phonebook) {
      display "Name: {foreach:Name}";
      display "Telephone: {foreach:Tel}";
}
```

The following BEE Variables will be made available within a "database" loop:
("foreach" is to be substituted by the Loop Variable name if one is explicitly specified.)

`foreach:`*`fieldname`*	Value of the field in the data record
`status%database` also in *`dbobj`*`%status`	Error code of the database retrieval, or 0 if successful
`message%database` also in *`dbobj`*`%message`	Error message of the database retrieval, or blank if successful
`result%foreach:iteration`	The iteration count. It contains 1 in the first iteration and 2 for the second etc.

(dbtree)*dbobj*

In this version, a "foreach" loop is the only mechanism to access a "dbtree". (Please see the "dbtree" command for more details about building and accessing a dbtree.)

The nodes of the dbtree will be fetched from the last data access via the database object (*dbobj*), one record (or node) in every iteration, and assigned to the "Loop Variable". The attributes of the dbtree are assigned to individual elements, indexed by the attribute name.

Three more values will be made available inside a dbtree foreach loop: result%foreach:activeness, result%foreach:isparent, and result%foreach:level.

The following BEE Variables will be made available within a "dbtree" loop:
("foreach" is to be substituted by the Loop Variable name if one is explicitly specified.)

`foreach:`*`fieldname`*	Value of the field in the dbtree node
`result%foreach:activeness`	The "activeness" value of the current node
`result%foreach:isparent`	0 if the current node is a leaf node, or 1 if it is a parent node.
`result%foreach:level`	The number of active parents inclusively between itself and the root. (Note: The root does not count because there is no root node.)

`result%foreach:iteration`	The iteration count. It contains 1 in the first iteration and 2 for the second etc.

var is the Loop Variable. The default Loop Variable is "foreach". Explicitly specifying the Loop Variable helps to make the code reads better, and also is necessary in a nested loop to avoid crashing of the Loop Variables if the inner loop use the same Loop Variable as the outer one.

The Loop Variable is NOT a reference to the source variable being looped through. You can modify the Loop Variable and use the new value within the loop, and the change has NO effect on the source variable. Upon the next iteration, the Loop Variable will be assigned the next value it should receive.

The Loop Variable is an array. If you specify an element as the Loop Variable, the element part will be ignored and the entire array will be used as the Loop Variable.

maxiter is the maximum number of iterations the loop will execute. It is default to 10000. Zero means infinity.

5.3.2 for – loop through a series of numeric values

```
BEE Script:  for [maxiter=num] ([var] [from num] [to num] [step num])
                    statement;
or           for [maxiter=num] ([var] [from num] [to num] [step num])
                    { statement; ... }

BEE Tag:     <beefor var [maxiter=num] [from=num] [to=num] [step=num]>
                    tag
                    ...
             </beefor>
```

Note: While the numeric values (*num*) are usually integer, BEE can accept floating point values as well.

"For" loop is used to loop through a series of uniformly increasing or decreasing numeric values (with an equal interval (step) in between).

Example 1:

```
for (i from 3 to 10)
{
        display "Children of year {i} please come here<br>\n";
}
```

Example 2:

```
display "Ready ... ";
for (i from 3 to 1) display "{i} - ";
display "Go!<br>\n";
```

Example 3:

```
display "All even numbers between 100 and 110:\n";
for (i from 100 to 110 step 2) display "{i}\n";
```

In most cases, a "for" loop can be replaced by a "foreach" loop, using no Loop Variable but specifying the "maxiter" parameter as the number of iterations required.

Example: The following two loops display the same result

```
for (i from 1 to 3) display "{i}\n";
foreach maxiter=3 { display "{result%foreach:iteration}\n"; }
```

Parameters

var is the Loop Variable. The default Loop Variable is "for". Explicitly specifying the Loop Variable helps to make the code reads better, and also is necessary in a nested loop to avoid crashing of the Loop Variables if the inner loop use the same Loop Variable as the outer one.

A "for" loop will be "inactive" if the ending condition is satisfied even before the first iteration. For example, the parameter "from" is larger than "to" and "step" is positive, in such case the loop is taken as ended before it even starts. The Loop Variable receives a value ONLY IF at least the first iteration of the loop is executed.

The Loop Variable is NOT a reference to the source variable being looped through. You can modify the Loop Variable and use the new value within the loop, and the change has NO effect on the source variable. Upon the next iteration, the Loop Variable will be assigned the next value it should receive.

The Loop Variable is an array. If you specify an element as the Loop Variable, the element part will be ignored and the entire array will be used as the Loop Variable.

from is the value assigned to the Loop Variable in the first iteration (unless the "for" loop is in active, in which case the Loop Variable will not be altered at all.) The "from" parameter is default to 0.

to is the value that specifies the end of the number series. Before the Loop Variable is assigned the next number in the series, the number will be checked and if it goes beyond the "to" value (larger than "to" if "step" is positive or smaller than "to" if "step" is negative), the loop will exit without assigning the number to the Loop Variable. The "to" parameter is default to 0.

step is the number used to generate the number series. The first number of the series is the value of the "from" parameter. Every number after the first one is obtained by adding the value of the "step" parameter to the previous number (the one contained in the Loop Variable for the last iteration).

If the "step" value is positive, the number is increasing in every iteration. If the "step" value is negative, the number is decreasing. The loop will be "inactive" (not executed) if the series is not expected to terminate (i.e. "from" is larger than "to" and "step" is positive, or "from" is smaller than "to" and "step" is negative.)

If the "step" value is zero or omitted, it will be taken as 1 if the "from" value is not larger than the "to" value, or -1 if otherwise. By omitting the "step" parameter, you guarantee that the number series will be stepping from the "from" value to the "to" value by 1 each term towards to right direction to terminate. (e.g. "from 1 to 3" will generate 1, 2, 3, and "from 3 to 1 will generate 3, 2, 1.)

maxiter is the maximum number of iterations the loop will execute. It is default to 10000. Zero means infinity.

5.3.3 while – loop while a condition remains true

BEE Script: `while [maxiter=`*`num`*`] (`*`condition`*`)` *`statement`*`;`

or
```
while [maxiter=num] (condition)
      { statement; ... }
```

BEE Tag:
```
<beewhile "condition" [maxiter=num]>
      tag
      ...
</beewhile>
```

Note: The *condition* is surrounded by a small bracket and therefore is taken literally. While you can (and sometimes need to) quote individual values in the *condition*, please do not quota the whole condition with double or single quote. Otherwise, the whole *condition* will be taken as a string (and will most likely to be evaluated to true unless it is blank or 0).

The "while" loop iterates for as long as the condition evaluates to true.

Please note that variables in *condition* are evaluated as macros and therefore one should bear in mind the logical expression syntax. In particular, you need to quote strings as required. Please see the "if" command for examples.

Example:

```
var i = 0;
while ({i} < 5)
{
      display 'i=<br>\n';
      var i = "(expr){i} + 1";
}
// Sample display:
//    i=0
//    i=1
//    i=2
//    i=3
//    i=4
```

Parameters

maxiter is the maximum number of iterations the loop will execute. It is default to 10000. Zero means infinity.

The following BEE Variables will be made available within a "while" loop:

`result%while:istrue`	Set to 1 if the condition evaluates to true, or 0 otherwise
`result%while:condition`	Set to the condition after variable evaluation
`result%while:iteration`	The iteration count. It contains 1 in the first iteration and 2 for the second etc.

An interesting remark on the "while" command: The *condition* is declared once but evaluated multiple times, each in every iteration. Internally the *condition* is passed in through an implicit "absolute" attribute, which means no pre-evaluation will be done until the execution starts.

5.3.4 continue – jump to the beginning of the loop

BEE Script: `continue;`

BEE Tag: `<beecontinue>`

Command "continue" brings the execution back to the beginning of the loop ("foreach", "for" or "while"). Please note that the beginning of the loop means the loop statement, not the first command of the loop. For example, "continue" will cause the next element in a "foreach" loop to be fetched, the Loop Variable in a "for" loop to increase, or the condition to be checked for a "while" loop.

You can only "continue" in a loop. If a "continue" command is "out-of-place", it will be ignored (and the execution will fall through to the next command after "continue".)

5.3.5 break – exit the loop and jump to after the end of it

BEE Script: `break;`

BEE Tag: `<beebreak>`

Command "break" brings the execution to after the end of the loop ("foreach", "for" or "while").

You can only "break" in a loop. If a "break" command is "out-of-place", it will be ignored (and the execution will fall through.)

5.4 Module Calling

5.4.1 function – define a function and its arguments

BEE Script: `function` *`name`* `[`*`name`*`=`*`value`* `...] {` *`statement`*`; ... }`

BEE Tag:
```
<beefunction name [name=value ...]>
        tag
        ...
</beefunction>
```

BEE Function is a block of code between the "function" command and the closing curly bracket. In BEE Tag, it is the block between <beefunction [function_name]> and </beefunction>. It is not executed at the time of declaration. Instead, the code is executed at the time of "calling". The "calling" syntax is exactly the same as calling a BEE Script statement or a BEE Tag, except that the command name is substituted with the function name.

Unlike in other programming languages, a BEE Function does not return a value syntactically. Instead, it passes out values through the "result", "status" and "message" classes. This enables the caller to receive from the function an array, a status code and a message text respectively in one go.

The argument list is in name-attribute pair format in both the function declaration and the function calling (unlike in most other languages which put the argument list in a comma-delimited position sensitive list inside a pair of small brackets.)

The function declaration line does not have to include all the arguments. Those that are included in the declaration line will be in name-attribute pair format, where each attribute represents the default value (the value the function takes if the argument is missing from the calling line).

<u>If there is no default value for an argument, DO NOT include it in the declaration line.</u> In such case, if the argument is missing from the calling line, it will be undefined. You can use the "isset" BEE Conversion on the arg%function:*name* or arg%*name*, to find out whether a particular argument is defined or not.

Variables if any in the default value is interpreted from the parent Context at calling time, instead of at the declaration time. For example:

```
var a = 123;
func1;

var a = 456;
function func1 x="{a}"
{
      // {arg%x} evaluates to "123"
}
```

Within the function declaration, arguments are referred to by the variable arg%function:*argname* or directly via arg%*argname* (useful for array passing.) For example, if function "myfunc" has two arguments: a=1 and b=2, then arg% contains:

`arg%function:a`	1
`arg%function:b`	2
`arg%function:function`	myfunc
`arg%a`	1
`arg%b`	2

The arg%function variable contains the argument in a non-structural form. For example, if argument b is an array like b="(array)x=>11,y=>12", arg%function:b will look exactly like this, without the evaluation of the array:

`arg%function:a`	1
`arg%function:b`	(array)x=>11,y=>12
`arg%function:function`	myfunc
`arg%a`	1
`arg%b:x`	11
`arg%b:y`	12

All arguments are optional syntactically. That means missing arguments of passing more of them do not cause syntax problem. Missing arguments will take on the default value. If no default value is specified for a missing argument, it will be undefined within the function.

Excessive arguments simply stay in the arg% variable list without causing trouble. That enables the function to work on a variable argument list by scanning the arg%function variable for argument values.

Arguments are passed by value, not by reference. To pass variables by reference, you can pass the variable name and declare the variable "parent". (Please see the "parent" tag.) For example:

```
function plusOne
{
       parent "{arg%v}";
       var "{arg%v}" = "(expr){{arg%v}} + 1";
}

var a = 2;
plusOne v="a";
// Now {a} is 3
```

There is an argument named "ignoreerror", which if defined as true, will suppress the error message if the function to call does not exist. This argument will be passed to the function as usual if the function exists. The function may use the "ignoreerror" argument to suppress error message of its own if so required.

Variables within a function are "local", which means that they do not inherit the values they got before the function call, and any changes to them inside the function would not be effective outside (except for System Classes.)

Example:

```
function calculate num1=0 num2=0
{
      var result%function:sum="(expr){arg%num1}+{arg%num2}";
      var result%function:diff="(expr){arg%num1}-{arg%num2}";
}

calculate num1=5 num2=2;
display "Five plus two is {result%calculate:sum}<br>\n";
display "Five minus two is {result%calculate:diff}<br>\n";
```

If the argument names is preceded by an "!", it means the BEE Variable inside the argument will NOT be evaluated ("absolute" passing).

Example:

```
var something = "out there";
showme what="{something}";  // Show me out there
showme !what="{something}";  // Show me in here

function showme
{
      var something = "in here";
      display "Show me {arg%what}<br>\n";
}
```

If the calling command does not pass in the argument, and the argument name in the function declaration line for default value is preceded by an "!", the argument value will be "absolute", which means it will NOT be evaluated in the parent Context. For example:

```
var something = "out there";
showme1;  // Show me out there
showme2;  // Show me in here

function showme1 what="{something}"
{
      var something = "in here";
      display "Show me {arg%what}<br>\n";
}

function showme2 !what="{something}"
{
      var something = "in here";
      display "Show me {arg%what}<br>\n";
}
```

<u>Parameters</u>

function specifies the name of the function. It must be an alphanumeric string starting with a letter. Function name is case insensitive. It will be converted to lowercase internally.

Within the function declaration, the following BEE Variables are made available:

`arg%function:`*`argname`*	The argument value referred to within the function. Variable arg%function can be scanned to discover arguments if the function accepts flexible argument list.
`arg%`*`argname`*	The argument values referred to within the function. It is the same as arg%function:*argname* except that it can hold an array. Also, with arg%*argname*, argument scanning is via class%list:arg.
`result%function`	The result array referenced from within the function.

After calling the function, the following BEE Variables are made available:

`status%`*`functionName`*	The status code returned by "return status=*code*;".
`message%`*`functionName`*	The message text returned by "return message=*text*;".
`result%`*`functionName`*	The result array referenced after calling the function.

5.4.2 return – stop executing a function and return to the caller

BEE Script: `return [status=num] [message=string];`

BEE Tag: `<beereturn [status=num] [message=string]>`

The command "return" stops the function execution and returns to the caller, optionally with a status code and a message text, which can be accessed by the caller via status%*functionName* and message%*functionName*. (Please see "function".)

Parameters

status specifies a positive number to return to the caller via status%*functionName*. Default is 0.

message specifies a message text to return to the caller via message%*functionName*. Default is an empty string.

5.4.3 global – declare a variable to be from the global Context

BEE Script: `global [var];`

BEE Tag: `<beeglobal [var]>`

The "global" command is valid only within a function. It declares that a variable is in the "global" Context, which means that the variable inherit the value set in the main script, and any changes to them inside the function will reflect onto the main script.

In another word, "global" variables are references to top level variables.

However, if the function calls another function, the "callee" function would not see the variables declared global in the "caller" function, unless the "callee" declares these variables "global" as well.

Example:

```
function welcome
{
      global sitename;
      var sitename = "Good Buy Shopping Mall";
      var who = "{sys%auth:username}";
      display "Welcome to {sitename}, {who}.<br>\n";
}

var sitename = "a website";
var who = "a user";
welcome;  // Welcome to Good Buy Shopping Mall, johnc.
display "{sitename} got {who}.<br>\n";
// Good Buy Shopping Mail got a user.
```

Please note that while variable "sitename" has been set after the function call, changes to "who" inside the function did not show up in the display.

If the variable name (*var*) is missing, the whole Context (all variables in the global Context) will be declared global. (Please use this carefully to avoid polluting the global environment.)

5.4.4 parent – declare a variable to be from the parent Context

BEE Script: `parent var;`

BEE Tag: `<beeparent "var">`

The "parent" command is valid only within a function. It declares that a variable is equivalent to the one in the "parent" (the calling function's Context), which means that the variable inherit the value set before the function call, and any changes to them inside the function survive the exit of it.

In another word, "parent" variables are references to the caller's variables.

However, if the function calls another function, the "callee" function would not see the variables declared "parent" in the "caller" function, unless the "callee" declares these variables "parent" as well.

If the variable name *var* is missing, the whole Context (all variables in the parent Context) will be linked with the one of the parent. (Please use this carefully to avoid polluting the parent's environment.)

5.4.5 include – include a block of code from a file

BEE Script: `include` *`file`* `[ignoreerror=`*`bool`*`];`

BEE Tag: `<beeinclude "`*`file`*`" [ignoreerror=`*`bool`*`]>`

The "include" command includes a file (not URL) into the current position. It must be in the current directory or a subdirectory under it. Your BEE Hosting Provider may have set up an "include path" which contains various directories in the order of searching for the include file. Please check with your BEE Hosting Provider for available include files.

Parameters

ignoreerror indicates whether or not to suppress the error message if the file to be included cannot be found. If "ignoreerror" is true, no error message will be displayed. The default is false.

5.4.6 exec – execute a system program

BEE Script: `exec prog [args=`*`var`*`] [istream=`*`var`*`] [ostream=`*`var`*`];`

BEE Tag: `<beeexec prog [args=`*`var`*`] [istream=`*`var`*`] [ostream=`*`var`*`]>`

The "exec" command executes a program in the operating system. In fact, the name *prog* is merely a "stub" to invoke the necessary system commands to do the job. This "stub" needs to be installed by the BEE Administrator and is assigned with proper permission, ownership and CROFT mapping.

The "exec" command will set status%exec and message%exec accordingly.

Parameters

args is a variable that contains an array of command line arguments to be included when executing the specified file.

istream is a variable that contains an array of input passed as the input stream of the program to be executed. Each element in the istream array represents a line in the input stream.

ostream is a variable that contains an array of output passed out from the output stream of the program after being executed. Each element in the ostream array represents a line in the output stream.

After calling the function, the following BEE Variables are made available:

`status%exec`	The status code returned by the program after being executed.
`message%exec`	The last line in the ostream. This is useful for programs returning only one line. ("ostream" is usually used when the program returns multiple lines.)

5.4.7 exit – stop execution and end the web page display

BEE Script: `exit` *`num`* `[to=string];`

BEE Tag: `<beeexit` *`num`*`>`

The command "exit" stops the execution and returns an exit code to the web server process (like a CGI script termination with an exit status.) unless the "to" parameter is specified.

to is used to specify where to exit to. If to="parent", the execution will return to the calling file (the one that "include" the current file). The default is "system", which means to exit to the web server process.

5.5 Remote Calling

5.5.1 connector – define a connection point for a remote function call

BEE Script: `connector` *`name`* `[mode=`*`string`*`] [style=`*`string`*`] [method=`*`string`*`]`
`[domain=`*`string`*`]`
`[auth=`*`string`*`];`

BEE Tag: `<beeconnector` *`name`* `[mode=`*`string`*`] [style=`*`string`*`] [method=`*`string`*`]`
`[domain=`*`string`*`]`
`[auth=`*`string`*`]>`

The "connector" command encapsulates the remote function call mechanism. Remote function call enables a client web page to call a function on the server without refreshing the page.

Two processes are involved: the client calls the server, and the server "calls back" the client. On the client side, the calling process is indicated below:

The "call" command -> Client Connector -> Server Connector -> Remote function

On the server side, the callback process is indicated below:

Remote function "returns" -> Server Connector -> Client Connector -> Callback function

The "call" command and is invoked in a client script, e.g. JavaScript (and should be in the BEE Tag syntax because nested <script> tag is not a good idea.) The tag specifies the URL to the Server Connector page, and the Client Connector used to deliver the call.

The Client Connector then packs the call parameters and sends them to the Server Connector (specified by the URL). The Server Connector will unpack the parameters and call the Remote function (specified in the Server Connector itself.)

When the function returns the result, status and message, the Server Connector will pack them and call the Callback function on the client page via the Client Connector.

The Callback function is in a client script (e.g. JavaScript) which will use the results for its operation (e.g. display in the proper field or pop up a message box etc.) The Callback function accepts three parameters: "result" (which is an array that the server function set up in rsult%function), status and message. They're also available globally via JavaScript variable connector_*name*.result, connector_*name*.status and connector_*name*.message.

For example:

Client-side

```
<script language="JavaScript">

function searchOnClick(fld)
{
      // In JavaScript, we need to use BEE Tag instead of BEE Statement.
      <beecall "http://myserver.com/ws/search.htm" connector="srch"
            key="@fld.value">
      // Arbitrary parameters (like "key") for the remote function allowed
      // "@" indicates that "fld.value" is a JavaScript expression.
}

// The callback function (named after the connector)
function srch(result, status, message)
{
      // The arguments (result, status and message) are returned
      // from the server function.
      // They are also available globally via connector_srch.result,
      // connector_srch.status and connector_srch.message.
      if (status > 0) alert(message);
      else myForm.myField.value = result['Name'];
}

...
</script>
...

<beeconnector "srch">
```

Server-side

```
<script langauge="bee">

connector "findname" mode="server";

function findname
{
       ...
       var result%function = ...;
       return status=... message=...;
}
</script>
```

The "connector" command will set status%connector and message%connector accordingly.

Parameters

connector specifies the name of the connector. The connector's name is not only a handle to the connector but also the default name of the remote function (mode="server") or the callback function (mode="client").

The Client Connector is an HTML construct and therefore its name is in the name space of the client document object hierarchy. It can be accessed by the "connector_name" name in JavaScript. (e.g. if connector="abc", the object name is connector_abc.)

In particular, three JavaScript variables are made available after the remote call: connector_name.result (an array containing the result%function of the remote function), connector_name.status and connector_name.message (which are status%function and message%function respectively.)

Unlike in the Client Connector, the Server Connector name is not used for any other purpose (not even in the call command, which calls the URL containing the Server Connector, not the Server Connector's name).

mode indicates what the connector does. A Client Connector is located in the client page for call tags to pass parameters to the remote function. The Client Connector is an HTML construct. It is hidden unless style="..." is specified that makes it displayable. Displaying the Client Connector (e.g. by style="") is useful for debugging as you can see what parameters are being passed and what results are passed back.

Since a Client Connector is a displayable construct (even when it is hidden), it must be put in the <body> section and not contained in a JavaScript section. (A Client Connector can exist within a BEE Script section as long as the section is in the <body> section.)

A Server Connector is used in the remote function page to be called. It is translated to an execution procedure of the remote function and therefore should be the only executable construct in the function page, but you can add other debugging messages which is to be showing up on the Client Constructor (if it's not hidden).

The remote function can be on the Server Connector page or another page included in by the "include" command. Basically, the remote function can be on any page as long as it is accessible by the Server Connector.

style specifies the style="..." attribute for the Client Connector HTML construct. Normally, the Client Connector is a hidden HTML construct, but you can use style="..." to show it. The simplest way is to use a null style (e.g. style=""), which will show the connector as a display box of default size at where the connector is located on the client page. Other style attributes can be used to, say, relocate the connector box or show different background or font etc.

domain is a security feature for connecting across web hosts. e.g. if you call a remote function on server.mydomain.com from a web page on client.mydomain.com, you need to specify domain="mydomain.com" in the Server Connector (at server.mydomain.com). On the client side, the "call" command needs to have the same domain="mydomain.com" parameter as well. The "domain" parameter is optional if both the server and the client is on the same web host.

auth if set to true indicate that the server connector is for BEE Remote Authentication. This parameter contains a *secret* string that the caller needs to provide in order to authenticate remotely. The "auth" parameter is effective only for a server connector (mode=server). Please refer to the "call" command for details.

Note: In the current implementation of BEE, the client connector is in fact an <IFRAME ...> tag. Parameters other than the above will be used as additional attributes of the IFRAME. A typical use is to include a src="/blank.htm" attribute to avoid browser complaining about an SSL page containing non-secure parts (the unspecified IFRAME content).

5.5.2 call – call a remote function via a URL

BEE Script:

```
call [location=URL] [operation=string] [connector=string]
[callback=string] [method=string] [window=bool]
[domain=string] [auth=string]
[name=value ...];
```

BEE Tag:

```
<beecall [location=URL] [operation=string] [connector=string]
[callback=string] [method=string] [window=bool]
[domain=string] [auth=string]
[name=value ...]>
```

The "call" command initiate a "remote call" to a function (the "server function") in another page (the "server page") indicated by the URL parameter. This command is useful in performing an operation at or retrieving data from the server without refreshing the current page.

The "call" command must run from a client script (e.g. JavaScript) environment. For example:

```
<script language="JavaScript">
<beecall "http://www.someserver.com/serv.htm" connector="testcon">
</script>
```

The server function is indicated by the "operation" parameter and needs to be accessible from the server page indicated by the "location" parameter. The server function is operating under the Scope of the server page, not the client page.

The result (the result%*function* array) of the server function will be passed back to a client script (e.g. JavaScript) on the client page, so that the script and handle the result without refreshing the client page.

Two Connectors, one for the client page and one for the server page, are required to facilitate the communication. Please see the "connector" command for more details.

<u>Calling a function on another site</u>

If the server page happens to be on a different website (defined as a collection of URL prefix of the same "owner" as mentioned in the "CROFT" section), it will use the Scope (authentication and data access) of the "server site", not the "client site". This mechanism not only provides security for the server function but also bridges the data access gap of the two sites.

For example, let's suppose Site A needs to access a piece of information on Site B. The client page on Site A, a.mydomain.com/caller.htm, calls the server page on Site B, b.mydomain.com/callee.htm. The connector at callee.htm will invoke a server function (which is just an ordinary function on Site B) and pass the result back to caller.htm on Site A via JavaScript.

<u>Access control</u>

To the server page, the request comes from the visitor's browser, not from the client page. However, the sys%client:referrer variable still show the client page address, and therefore can be used by the server function to restrict access from only the client page it allows.

Since the server function works under the scope of the server function, it can restrict access based on visitor's login and access level. That means the visitor needs to have logged into the server site already before gaining access by means of the client site. This way, the server site security will not be compromised by granting access via the client site.

If SSL is available on the callee site, the caller can include "https://" in the callee URL to indicate a secure submission.

<u>Calling another web host</u>

At the moment, due to a security restrictions on Internet Explorer, BEE needs to restrict the calling to between two pages of the same host (e.g. www.mydomain.com/client.htm and www.mydomain.com/server.htm), and between two pages of two different hosts in the same domain (e.g. client.mydomain.com/a.htm and server.mydomain.com/b.htm, both in the domain of mydomain.com). In the later case, the "domain" parameter is required on both the client "call" command and the server "connector" command.

<u>BEE Remote Authentication</u>

The Remote Calling machenism of BEE can be used to obtain (or transfer) authentication information from the server site. For example, if Site A and Site B are two closely related websites (e.g. two departments of the same company), how can we allow users who have logged into Site B to automatically have access to Site A? The answer is to set the "auth" parameter in the "call" command to invoke a server function via a "server" connector with "auth" set as well.

To begin, Site A initiates an authentication call to Site B. Since there is no session information in the call, the authentication server connector on Site B refreshes the caller page on Site A with the session information (which come from the scope of Site B in which the server function is running in). Site A after being refreshed initiates a second authentication call to Site B, this time with the session information.

Once the *secret* string is confirmed valid, the server function will be called. It is to return the sys%auth array via result%*function*. These are usually the attributes of the authentication information, and are optional. In fact, you can have a null function without returning anything, but the function still needs to be there. If you want to fail the authentication, you can return a non-zero status, and the connector will not pass any authentication information to the client (e.g. return status=1 message="Not allowed";).

Here is an example:

```
connector myauth mode="server" auth="mysecret";

function myauth
{
      var result%function = "(var)sys%auth";
}
```

Programmers do not need to concern the above details. The BEE Remote Authentication machenism is encapsulated in the "auth" parameter of the "call" command and the server "connector" command on the server page. Once these parameters are specified, everything else will happen automatically.

The "call" command will set status%call and message%call accordingly.

Parameters

location specifies the URL of the server page. If the location URL is not specified, the Client Connector will still prepare the parameters but stop just before calling. For example, <beecall operation=myfunc connector=myconn a=1 b=xyz> (with a style="" in the connector to make it visible) will show the calling sequence in a small window at where the connector is without actually calling the server function. This is useful to debug the parameters without they being overwritten by the server results.

operation specifies the name of the server function (at the server page). The default is the name of the connector.

connector specifies the name of the connector used in the remote call.

callback specifies the name of the callback function (client script such as JavaScript). The default value is the name of the connector (in which case, you need to name your callback function the same as your client connector.) If "auth" is set, "callback" specifies a URL to callback with the session information. In this case, the default is the URL of the calling page itself (i.e. {sys%url:page} of the calling client).

method specifies the mechanism of "calling" (submitting the request from the client to the server). It is in fact the "method" attribute of the <form> tag used internally by the "call" command. The value is either "get" or "post". The default is "post". (Sometimes the "post" method may cause problem if the access to the server page is somehow redirected. It is recommended to use the default first and use method="post" only if it does not work without.)

formattr specifies extra attributes, if any, of the <form> tag used internally by the "call" command.

window indicates whether the connector will open a separate window on the browser to deliver the call. By default, the Client Connector will deliver the call through a hidden <IFRAME ...>. You may set this parameter to true if you want the call delivered via a separate visible window. This is useful if you want to observe the server connector in operation, for example, when the server does show some display. This is also used if the call is delivered when the client page is unloaded (for example to save something to the server). In this case, the hidden <IFRAME ...> will close as soon as the client is closed, killing the server connector process. Setting "window" to true will allow the server connector keep running even after the client is closed. The Default of the "window" parameter is false.

domain is a security feature for connecting across web hosts. e.g. if you call a remote function on server.mydomain.com from a web page on client.mydomain.com, you need to specify domain="mydomain.com" in the "call" command. On the server side, the "connector" command needs to have the same domain="mydomain.com" parameter as well. The "domain" parameter is optional if both the server and the client is on the same web host.

auth if set to a *secret* string indicates a BEE Remote Authentication call. In this case, the "callback" parameter contains the URL to call back. The *secret* string needs to be acceptable to the connector on the server side.

name is not really a fixed parameter. The name *name* can be anything (except for the parameter names above) and can be more than one. They are used to pass parameters to the remote function (as indicated by the Attribute Name-Value Pairs in the syntax description). If a value is preceded by an "@" sign, the rest of the content indicates a JavaScript expression (instead of a BEE expression.)

5.5.3 xmlparse – parse an XML document

BEE Script: `xmlparse` *`string`* `[xml=`*`class`*`] [index=`*`class`*`];`

BEE Tag: `<beexmlparse` *`string`* `[xml=`*`class`*`] [index=`*`class`*`]>`

The "xmlparse" command parse the XML document contained in the *string* and create an XML class variable and an optional index class variable. These two variables should not be accessed directly. They should be passed to an XML object which encapsulate the XML operation.

In fact, the "xmlparse" command itself should not be directly accessed as well. The constructor of the XML object (contained in "common/xml.bs") will parse the XML document for you.

The "xmlparse" command will set status%xmlparse and message%xmlparse accordingly.

Parameters

xml specifies the name of the "xml" class, which contains in each variable an XML construct, such as XML tag, element and CDATA. The elements of such variable is the meta data and attribute of the XML construct. The default of the "xml" parameter is "xml".

index specifies the name of the "index" class, of which each variable correspond to an XML tag. The name is the XML tag name, and the elements are keys that can be used to address the "xml" class as variable name for all occurrence of the XML tag. (The "index" class is optional.)

5.6 Authentication

5.6.1 access – define an access control block

```
BEE Script:   access (accessControlSpec) statement;
              [else statement;]
or            access [(accessControlSpec)] { statement; ... }
              [else { statement; ... }]

BEE Tag:      <beeaccess [accessControlSpec]>
                     tag
                     ...
              [<beeelse>
                     tag
                     ...]
              </beeaccess>
```

Note: The *accessControlSpec* is surrounded by a small bracket and therefore is taken literally. So please do not quota it with double or single quote or the quotation marks will be taken literally as part of the value.

Example 1:

```
access display 'Hello {sys%auth:username}';
else display 'Please login first';
```

Example 2:

```
access (admin) {
        // ... display the logout button here
        display 'Go to <a href=/admin>Admin Site</a>';
} else {  // Not admin user
        access (at most public) {
                display 'Please login first';
                // ... display the login form here
        } else {  // higher than public
                access (@friendly.site.com) display 'Hi Pal';
                else {
// Stop if it is user "john" of realm "myown.site.com"
                        access (john@myown.site.com) exit;
                        else display 'Welcome my dear member';
                }
        }
}
```

The "access" command defines a block that is executed only when the current session matches the *accessControlSpecification* (ACS) enclosed in the small bracket.

There are various authentication settings that you can select when your site is established, such as the password encryption scheme, the location and access method of the user table and how to map the information on that table.

The concept of "realm" (a group of users who share the same name space) is built into the BEE platform and implemented in CROFT. You can use a field in the user table to indicate the realm (e.g. the Department field of the Staff table), or you can have physically separated tables for different realms.

For example, you can allow the sales staff of a branch office in Melbourne to login via a realm called "melsales". When someone logs in using, say, "matthew@melsales", CROFT will check the Auth table at the Melbourne branch server to verify the password for "matthew". You can use the "access" command to grant/restrict access of such "affiliate" member.

The "Auth" table can locate at any server as long as the BEE server where your web page is running from have access to the "Auth" table via the Internet. This allows the end user to run their own authentication system in house (or in any facilities they choose).

The "Auth" table can in fact be a scheme File. For example, if you establish the website authentication with @itguys realm pointing to the "staff" scheme file and have

```
john:Password=abc
john:AccessLevel=2
john:Name=John Lee
```

then "john@itguys" can login with password "abc" and gain member access. In fact, you may have more than one realms in the same authentication scheme file. For example:

```
mary@marketing:Password=def
mary@marketing:AccessLevel=2
mary@marketing:Name=Mary Young
```

Of course, you will need to establish the realm @marketing to point to the "staff" scheme file as well. However, extra care is needed in this case. If no realm is specified in the login, only entries without realm will match. If a realm is specified, entries with the specified realm will be searched first, and if there is no match, entries without realm will be searched. In the above example, "mary" (without realm in the login) will get no match, while "john@marketing" and "john@marketing" will both match the entry "john".

It is recommended to use separate authentication scheme files for different realms to avoid confusion. In such case, there is no need to put realm specification in the entries.

For flexibility in integration, the Auth table can have any attributes of any arbitrarily names. All attributes is accessible via the sys%auth:*attribute* variable. CROFT recognises six attributes in order to operate the authentication process. Three CROFT attributes are for user identification: Username, Password and Realm. There are another three for session control: Access Level, Active and Expiry. CROFT requires at least the Username and Password. All the other four can be omitted in the Auth table without affecting CROFT.

- If Realm is omitted from the Auth table, CROFT will take the whole table to be from a single pre-defined realm (usually blank). If any of the three session control attributes are omitted, a default value needs to be entered into CROFT. For example, #2 for Access Level if all users in the Auth table has Member access; #1 for Active if all are active; and #0 for Expiry if none will expire.

User Identification Attributes:

Username: login user ID

Password: password (encrypted with a scheme chosen at set up time)

Realm: an optional suffix following the user ID (ie user@realm) for user grouping and access control

Session Control Attributes:

AccessLevel: access privilege (a common name mapping is public=0, affiliate=1, member=2, vip=3, editor=6, manager=8, admin=10; Please see note.)

Active: whether the user record is effective (non-zero means active)

Expiry: timestamp indicating when the user record is deactivated (0 means never)

Note on AccessLevel: Besides the "public" and "affiliate", all other Access Levels are assigned largely arbitarily, but it is not recommended to change as many library applications or functions assume such mapping. The "admin" access level should always been highest, and its value "10" is entered into the Auth table in CROFT. The "manager" access level is used by the User Management application in the Administration site; the "editor" access level is used by the Portal Object to allow menu editing and setting changing. Only the "admin" user can edit the site contents (TEA). If you want to allow the "editor" user to do TEA editing, you must have the scheme setting "scheme%AllowEdit=6".

Parameters

access specifies the ACS (Access Control Specification). Default is "higher than public".

The ACS comes in two different forms:

Form 1: [*relation*][*accessLevel*]

Specifies a range of access level at which the user can access the controlled block that follows.

relation is default to ">=". *accessLevel* can be the access level number as well. The default is "member".

Here're some examples:

higher than affiliate
at least vip
over 2

Common expression can be used to a certain extend. The followings are valid relational words:

less than
below
under
greater than
above
over
equal to
same as
at least
at most
not ...
lt
le
eq
ne
ge
gt

All these will be translated to the proper mathematical symbol. e.g. less than is < and at least is >= etc.

Form 2: [*user*]@[*realm*]

Specifies a user or a group of users who can access the controlled block that follows. The "@" sign is compulsory (even if the website do not use the "realm" feature). If *user* is missing, it means all users in the *realm*. If *realm* is missing, it means the "blank" realm.

Note: The "Realm Character" is defined in scheme%RealmChar. The default is "@". If you use email address as login name and do not want to confuse the system into taking the domain as realm, you may define an impossible character in scheme%RealmChar.

Here're some examples:

@marketing	All users in realm "marketing"
john@	User "john" in the blank realm
mary@billing	User "mary" in the realm "billing"

In particular, "not" means to exclude the user or group of users from accessing. e.g. "not @staff" means NOT in the realm "staff".

After the "if" command is executed, the following BEE Variables are made available:

`result%access:isallowed`	Set to 1 if allowed access, or 0 otherwise

5.6.2 login – authenticate a session with username and password

BEE Script:
```
login [username=string]
      [password=string]
      [realm=string]
      [signup=num]
```

BEE Tag:
```
<beelogin    [username=string]
             [password=string]
             [realm=string]
             [signup=num]>
```

The "login" command authenticates the session so that subsequent "access" command and "sys%auth" variables will operate on the new session.

All parameters got proper default value from the login form (via the sys%form variables.) So "login" mostly appears to be parameter-less command unless your login form got field names different from "username", "password" and "realm". ("realm" is optional as login will extract the part of the username after the "@" sign if any as the "realm" specification. The "@" is called the Realm Character, which can be redefined in scheme%RealmChar.)

If both "username" and "password" are both evaluated to blank (including their default values from the form entry), no login function will be performed, status:login will be set to 0, and message:login will be set to blank. This design is to avoid a login error when the user first open the page.

Here is a typical way to handle a "member-only" page:

Example (forms submit back to the same page):

```
if ('{sys%form:Submit}' == 'Login') login;
elseif ('{sys%form:Submit}' == 'Logout') logout;
display '{message%login}';

access {
      display '<form method="post" action="{sys%url:page}">';
      display '<input type=submit name=Submit value=Logout>';
      display '</form>';
      // Member-only info here
} else {
        display '<form method="post" action="{sys%url:page}">';
        display 'Username: <input type=text name=username><br>\n';
        display 'Password: <input type=password name=password><br>\n';
        display '<input type=submit name=Submit value=Login>
        display '</form>';
}
```

The "login" command will set status%login and message%login accordingly.

Parameters

username specifies the username used to access the Auth table. It is default to the value of the form entry field named "username" (i.e. {sys%form:username}). If "username" evaluates to blank but "password" is non-blank, "login" will give an error. If both "username" and "password" evaluate to blank, "login" will do nothing and return no error.

If "username" evaluates to a value that contains an "@" sign, the part after the "@" will be used as "realm". Realm specified this way (after the "@" sign) takes precedence over the "realm" parameter.

password specifies the plain text version of the password that is used in the authentication process to match up with the one (encrypted or not) in the Auth table. It is default to the value of the form entry field named "password" (i.e. {sys%form:password}). If "password" evaluates to blank but "username" is non-blank, "login" will give an error. i.e. blank password is not allowed. If both "username" and "password" evaluate to blank, "login" will do nothing and return no error.

realm specifies the realm if the "username" value contains no "@" sign.

After the "login" command is executed, the following BEE Variables are made available:

`status%login`	Error code or 0 if successful 1: user already logged in 2: no username is entered 3: no password is entered 4: username is incomplete 10 or above: Error code from the BEE system
`message%login`	Error message or blank if successful

5.6.3 logout – unauthenticate a session

BEE Script:	`logout;`
BEE Tag:	`<beelogout>`

The "logout" command "unauthenticates" the session so that subsequent "access" command and "sys%auth" variables will operate on a public session.

After the "logout" command is executed, the following BEE Variables are made available:

`result%logout:user`	The full username that was logged out (with @realm if applicable)

5.7 Data Access

5.7.1 database – access the database and prepare the result

BEE Script:

```
database dbobj [name=string] [query=string]
[action=string] [matrix=class]
[seek=[num[,num[,num]]]]
[dbid=name]
```

BEE Tag:

```
<beedatabase dbobj [name=string] [query=string]
[action=string] [matrix=class]
[seek=[num[,num[,num]]]]
[dbid=name]>
```

In BEE, there is no initialisation or connection to the database before data access operation and no closing or disconnection from the database afterwards. The database can locate at any server as long as the BEE web server has access to the database via the Internet. Also, there are no platform-dependent database operations. All the dependent operations are handled by the system transparently and are completely concealed from the program code.

When writing database access code, you go straight into the data access operation, which is usually a one-liner, then you retrieve the results for display or further processing. This design helps to simplify your web page and reduce risk of error by eliminating the database connection process from your code. It is also more secure way of data access as BEE websites can access only the database predefined for it at set up time.

At the moment of writing, BEE supports:

BEE VirtualBase (flat file)
MySQL
mSQL
Microsoft SQL Server
Oracle 8
ODBC
InterBase
PostgreSQL
Sybase

In the BEE architecture, database operation is a big topic and needs to be covered in a separate section. Please see "Database Operation" for details.

The "database" command will set status%database and message%database accordingly.

Parameters

name is a global name uniquely identifying the data access result. This is used internally by the system to avoid resource allocation conflict. It is usually omitted so that the system will generate a unique name for you. If you need to access the "name" value, it is in {*dbobj*%name}.

Explicitly specifying the name parameter is useful when you want to access the query information stored in the "session" class. You need to use a constant name to guarantee that you get back the same "session" variable. However, BEE store a company of the query information in "session%database" anyway. So you need to define a constant name across sessions only if you have more than one database object to carry across.

Please note that this "name" parameter has nothing to do with the name of the underlying database that the operation is binding to. (The script does not contain any knowledge about the underlying database.)

query is an SQL query statement that operates on the underlying database. For data retrieval, the record set can be accessed via the "database" data type in the "get" command or in a "foreach" loop. If "query" is specified, parameters "action" and "matrix" will be ignored.

action is an advanced query method that draws input arguments from the matrix and the database object (named by *dbobj* in the "database" command). Besides data access, "action" can also be used to manipulate the "BEE VirtualBase" table ("create", "load", "unload", "showcreatetable" and "showinsertinto") and roll-back record retrieval ("push").

If the value of "action" is preceded by an "!", the operation will stop after deriving the SQL statement but before executing it. The resulting SQL statement will be stored in *dbobj*%query as usual. This is useful in debugging or testing to let the programmer to check that the SQL statement is right before applying it physically to the database.

Parameter "action" is a big topic and deserves a separate section. Please see "Database Action".

(Parameter "action" is ignored if parameter "query" is specified)

matrix specifies a class of variables each contains a data row, indexed by a key value. e.g. "matrix%12:Tel" represent the "Tel" column of row "12" (the row that is indexed by "12", not the 12th row.) The list of keys (the variable names) are held in *dbobj*%keys, and the list of fields (the element names) in *dbobj*%fields. The default value of "matrix" is "matrix".

Please note that like any object reference, the matrix class is locally referenced unless explicitly declared as "global" or "parent".

(Parameter "matrix" is ignored if parameter "query" is specified)

seek specifies the position in the retrieved record set where the record fetching will start. The default is "seek=1", which means to start fetching from the first record. "seek=last", "seek=end" and "seek=bottom" all mean that the fetching will get the last record. "seek=random" will start the fetching on a record randomly selected between the first one and the last inclusively. Subsequent fetches after "seek=random" will remain sequential following the randomly selected record.

The position in parameter "seek" can be optionally followed by two more values: "records per page" and "pages per block". This is useful for paging the display of the record set, and is described in "Paging the Data Display". (If either of these two values are specified, the "seek" position will be automatically adjusted to the first record in the page.)

dbid is an identifier to the database if there are more than one database associating with the URL of the web page where the request is originated from. This parameter is rarely used even when you have different database to access, because the URL is sufficient to identify the database to use. It is only when you want to access multiple databases from the SAME page that you need to use "dbid" to indicate which one to use. (The value of "dbid" is an identifier defined at set up time of the website, NOT the name of the underlying database.)

After calling the function, many BEE Variables are made available. Please see "Database Variables".

5.7.2 dbtree – convert a database result into a non-circular tree

BEE Script: `dbtree` *`dbobj`* `[root=`*`string`*`] [self=`*`string`*`] [parent=`*`string`*`] [maxlevels=`*`num`*`] [maxnodes=`*`num`*`] [withroot=`*`bool`*`];`

BEE Tag: `<beedbtree` *`dbobj`* `[root=`*`string`*`] [self=`*`string`*`] [parent=`*`string`*`] [maxlevels=`*`num`*`] [maxnodes=`*`num`*`] [withroot=`*`bool`*`]>`

The "dbtree" command does not generate any output. It only generates an internal structure for the specified database object. This structure can only be accessed via a foreach loop of type "dbtree".

A dbtree is a non-circular tree structure converted from a database query result. The typical use of a dbtree is in a site-map style hierarchical menu. The minimum implementation is a VirtualBase table with two fields: Self and Parent. "Self" is a unique identifier of the menu item and "Parent" is the upper level menu the item belongs to, which must exist as a "Self" node in another item. It is like a file system directory structure.

In this context, we follow the common convention in describing "node", "parent", "child", "sibling", "leaf" and "root". (See Terminology for more details).

Each record in the record set corresponds to a node in the dbtree. The node record has at least one unique key field (of which the name is specified in the "self" parameter) and a parent pointer field (of which the name is specified in the "parent" parameter.) Circular relations will be detected and removed.

The root node does not necessarily exist as a record (virtual root), but the sibling relation for all children nodes having the same parent pointer still holds.

A record set may contains several roots (or virtual roots), each holds a disjoint dbtree in the same record set. In this case, you may specify root="..." to indicate a sub-dbtree formed by all siblings of that parent.

The root will not be included in the dbtree result unless withroot="1". If the root does not exist in the record set but withroot="1", a faked root node will be generated in the dbtree result (not in the record set). The faked root (if there's one) has only the "self" field containing the virtual root pointer and the "parent" field being null.

Example:

In the following example, "indent" is a variable containing spaces and the "lead" strings are the symbols you display before the node to indicate its type. e.g. ">" (or &gt;) implies a non-opened parent etc.

Please note that the "lead" variable is indexed by a two digit number, the first one being the "activeness" and the second one "isparent". There're 4 possible values for "activeness" (0, 1, 2 and 3) and 2 possible values for "isparent". Zero "activeness" should not be shown. So you can specify 6 different symbols.

(For the sake of an example, the "self" node is the "Code" field in the table, and the "parent" node is the "Parent" field in the table, which is the default.)

```
var indent = ". ";
var "lead" = "(array)10=>o, 11=>&gt;, 21=>v,
      30=><font color=#ff0080>o</font>,
      31=><font color=#ff0080>v</font>";
database "db" query="select * from Menu";
dbtree "db" self="Code" active="form%pagecode";
foreach ((dbtree)db as node)
{
      if ({result%node:activeness} == 0) continue;

      display "{indent|repeat:{result%node:level}}";
      display
"{leadchar:{result%node:activeness}{result%node:isparent}}";
      var href = "{sys%url:page}?pagecode={node:Code}";
      display '<a href="{href}">{node:Name}';
      display ' ({node:Description})</a><br>';
}
```

Terminology:

Every "node" in the tree can have zero or more "children" nodes, which in turn may have their own children nodes. Children nodes with the same parent node are called "sibling" nodes of each other. Nodes with no children nodes are called "leaf" nodes; nodes with children nodes are called "parent" nodes; the node with no "parent" node are called the "root" node (usually unique in a tree).

Parameters

root specifies the root node that the dbree is built under. The default is blank.

active indicates the active node in the dbtree. By specifying the "active" parameter, you assign an activeness value to each node "related" to the active one in the following scheme:

`3`	CURRENT: The active node itself
`2`	OPEN: An active parent, i.e. a parent of the active node or a parent of another active parent. All the active parents form a path from the root to the active node.
`1`	CLOSE: An active sibling, i.e. a sibling of the active node or an active parent. All the active siblings form an active tree that resumbles a function menu hierarchy.
`0`	None of the above. (Nodes with zero activeness should not be shown.) If "active" is blank, the activeness of all nodes will be set to 0.

self specifies the name of the field which contains the unique key of the node. The default is "Self".

parent specifies the name of the field which contains the parent pointer of the node. The default is "Parent".

maxlevels specifies the maximum number of levels the process will go down to. This restriction is introduced to avoid endless looping in ever deeper search for more levels due to a fault in the record set. The default is 10.

maxnodes specifies the maximum number of nodes the process will take from the database. This restriction is introduced to avoid resource draining problem with a huge record set. The default is 1000.

withroot indicates whether to include the root into the dbtree. To include the root, please enter "yes", "on", "true", or a positive number. If the root does not exist in the record set (virtual root), a faked dbtree node will be generated. To exclude the root (the default case), please enter "no", "off", "false", or 0.

After calling the function, the following BEE Variables are made available:

`status%database`	Error code of the dbtree operation, or 0 if successful
`message%database`	Error message of the dbtree operation, or blank if successful

The followings are available only in the "foreach ((dbtree)... as *var*) { ... }" loop:

`result%`*`var`*`:activeness`	The "activeness" value of the current node. (Please see the "active" parameter.)
`result%`*`var`*`:isparent`	0 if the current node is a leaf node. 1 if the current node is a parent node.

`result%`*`var`*`:level`	The number of active parents inclusively between itself and the root. If "withroot" is 1, the root is at level 0 and all its children nodes are at level 1; if "withroot" is 0, there is no root, and all top level nodes are at level 0. In another word, you always have level 0 in a dbtree.

5.8 Socket

Sockets are endpoints for communication between processes or hosts.

The most common model of communication is the client-server one, in which the "server" process or host is running all the time, waiting for requests to serve, and the "client" process or host sends requests to the server and get the reply from it for further processing.

The typical live cycle of a server socket is *create-bind-listen-accept-read-write-close*, while that of the client is *create-connect-write-read-close*. The status%socket and message%socket variables will be set accordingly after each operation. However, in the following examples, the status and message variables are ignored for simplicity.

A Typical Server Process:

```
socketcreate serverSock;
socketbind serverSock address="the.server.com" port=100;
while (true) {
      socketlisten serverSock;
      socketaccept serverSock newsocket=msgSock;
      socketread msgSock;
      display "Request: {msgSock%read:content}<br>\n";
      if ('{msgSock%read:content}' == 'quit') break;
      socketwrite msgSock content="I heard: {msgSock%read:content}";
      socketclose msgSock;
}
socketclose serverSock;
```

A Typical Client Process:

```
socketcreate clientSock;
socketconnect clientSock address="the.server.com" port=100;
socketwrite clientSock content="hello world";
socketread clientSock;
display "Reply: {clientSock%read:content}<br>\n";
socketclose clientSock;
```

5.8.1 socketcreate – create an endpoint for communication

BEE Script:

```
socketcreate socketObj
[domain=string] [type=string]
[protocol=string] [listen=bool];
```

BEE Tag:

```
<beesocketcreate socketObj
[domain=string] [type=string]
[protocol=string] [listen=bool]>
```

The "socketcreate" command creates an endpoint for communicating with another process or host. It does not do the actual communication but to specify the property of the socket for subsequent socket commands. These properties are stored in the *socketObj* object.

The "socketcreate" command will set status%socket and message%socket accordingly.

Parameters

socket (the *socketObj*) is the name of the object holding the socket's properties.

domain is one of the followings:
const%socket:AF_INET (default): IPv4 Internet based protocols
const%socket:AF_UNIX: Local communication good for Interprocess Communication

type is one of the followings:
const%socket:SOCK_STREAM (default): Sequenced, full-duplex byte stream (for TCP)
const%socket:SOCK_DGRAM: Datagrams (connectionless for UDP)
const%socket:SOCK_RAW: Raw network protocol access (used to construct any protocols like ICMP)
const%socket:SOCK_SEQPACKET: Sequenced, two-way for datagrams (entire packet in each read)
const%socket:SOCK_RDM: Reliable Datagram layer (order not guaranteed, likely not implemented)

protocol is one of the followings:
const%socket:SOL_TCP (default): Transmission Control Protocol
const%socket:SOL_UDP: User Datagram Protocol
const%socket:SOL_ICMP: Internet Control Message Protocol

listen is the port number on which the socket is opened to accept connection from other processes or hosts. If listen is specified, the "domain", "type" and "protocol" parameters will be ignored. "domain" will be taken as const%socket:AF_INET and "type" will be taken as const%socket:SOCK_STREAM. ("listen" is usually used is a "server" process.)

5.8.2 socketbind – bind an address and a port to a socket

BEE Script:

```
socketbind socketObj address=string port=integer;
```

BEE Tag:

```
<beesocketbind socketObj address=string port=integer>
```

The "socketbind" command binds an address and a port number to an existing socket. It does not do communication but to specify the address and port for subsequent socket commands. These properties are stored in the *socketObj* object.

The "socketbind" command will set status%socket and message%socket accordingly.

Parameters

socket (the *socketObj*) is the name of the socket to be bound.

address is the address that the socket will communicate on. For AF_INET domain, it is an IP address, or a host name resolvable to an IP via the DNS (Domain Name Service).

port is a service identifier. e.g. SMTP (port 25) for e-mail and HTTP (port 80) for the web. It may be specified as a service name recognisable by the system.

5.8.3 socketlisten – listen (wait) for a connection on a socket

BEE Script: `socketlisten` *`socketObj`* `[backlog=`*`integer`*`];`

BEE Tag: `<beesocketlisten` *`socketObj`* `[backlog=`*`integer`*`]>`

The "socketlisten" command is usually used by a server process. This command "registered" its interest to the system on connections made to the address and port specified in a previous "socketbind" command. It is usually followed by a "socketaccept" command to wait for and accept the new connection given to the process by the system.

The "socketlisten" command will set status%socket and message%socket accordingly.

Parameters

socket (the *socketObj*) is the name of the socket to be suspended for "listening".

backlog is the maximum number of connections allowed to be queued before the process accept the first connection (see "socketaccept"). The default is 5.

5.8.4 socketaccept – accept a connection on a socket

BEE Script: `socketaccept` *`socketObj`* `[newsocket=`*`string`*`];`

BEE Tag: `<beesocketaccept` *`socketObj`* `[newsocket=`*`string`*`]>`

The "socketlaccept" command is usually used by a server process. This command accept a connection from the queue (see "socketlisten") and create a new socket for that connection. After the connection is processed, the new socket should be closed and the original socket should be "listened" to again.

If there are multiple connections in the queue, the first one will be "accepted". If there are no connections in the queue, the process will be suspended waiting for the arrival of a new connection.

If the socket is set to non-blocking (see "socketcontrol") and there is no connection in the queue, the process will return an error.

The "socketaccept" command will set status%socket and message%socket accordingly.

Parameters

socket (the *socketObj*) is the name of the socket where the new connection is to be accepted.

newsocket is the name of the new socket object to be created for the process to communicate with the process or host making the connection. The address and port of the connection (of the server process) can be obtained from *newSock*%address and *newSock*%port respectively, where *newSock* is the value of the "newsocket" parameter. The address and port of the (client) process and host making the connection can be obtained from *newSock*%remote:address and *newSock*%remote:port respectively.

5.8.5 socketconnect – make a connection on a socket

BEE Script: `socketconnect socketObj address=string port=integer;`

BEE Tag: `<beesocketconnect socketObj address=string port=integer>`

The "socketconnect" command initiates a connection to a remote (server) address and port via the socket. This command is usually used by a client process.

The "socketconnect" command will set status%socket and message%socket accordingly.

Parameters

socket (the *socketObj*) is the name of the socket to connect.

address is the address that the socket is making connection to. For AF_INET domain, it is an IP address, or a host name resolvable to an IP via the DNS (Domain Name Service).

port is a service identifier. e.g. SMTP (port 25) for e-mail and HTTP (port 80) for the web. It may be specified as a service name recognisable by the remote system.

5.8.6 socketread – read a message from a socket

BEE Script: `socketread socketObj [maxlength=integer];`

BEE Tag: `<beesocketread socketObj [maxlength=integer]>`

The "socketread" command reads a message from the socket. The content and length of the read message are stored in *socketObj*%read:content and *socketObj*%read:length.

In order to read all bytes in the message, you may need to implement the "socketread" command as a while loop:

```
socketread mySock;
var msgRead = "";
```

```
while ({mySock%read:length}) {
        var msgRead = "{msgRead}{mySock%read:content}";
        socketread mySock;
}
// Now msgRead contains the message read from the socket.
```

The "socketread" command will set status%socket and message%socket accordingly.

Note: "socketread" is binary safe.

Parameters

socket (the *socketObj*) is the name of the socket to read.

maxlength is the maximum number of bytes that the command will read. If there are more to read beyond this limit, the excessive data will be left unread until the next "socketread" command.

5.8.7 socketwrite – write a message to a socket

BEE Script: `socketwrite socketObj [content=string] [maxlength=integer];`

BEE Tag: `<beesocketwrite socketObj [content=string] [maxlength=integer]>`

The "socketwrite" command write a message to the socket.

The "socketwrite" command will set status%socket and message%socket accordingly.

Note: "socketwrite" is binary safe.

Parameters

socket (the *socketObj*) is the name of the socket to write.

content is the content to be written, which can be subsequently accessed via the *socketObj*%write:content variable.

maxlength is the maximum number of bytes to be written. The number of bytes have actually been written to the socket can be accessed via the *socketObj*%write:length. If there are more bytes in the "content" value, only the first "maxlength" bytes will be written. Otherwise, the entire content will be written. The latter case is the default if "maxlength" is not specified.

5.8.8 socketclose – close a socket

BEE Script: `socketclose socketObj;`

BEE Tag: `<beesocketclose socketObj>`

The "socketclose" command closes a socket.

The "socketclose" command will set status%socket and message%socket accordingly.

Parameters

socket (the *socketObj*) is the name of the socket to close.

5.8.9 socketcontrol – control some behaviours of the socket

BEE Script: `socketcontrol socketObj [blocking=bool] [timeout=integer];`

BEE Tag: `<beesocketcontrol socketObj [blocking=bool] [timeout=integer]>`

The "socketcontrol" command changes the behaviour of the socket.

The "socketcontrol" command will set status%socket and message%socket accordingly.

Parameters

socket (the *socketObj*) is the name of the socket to have the behaviour changed.

blocking is used to change the socket to non-blocking by specifying "blocking" as false (e.g. 0). In the current version of BEE, you cannot set a non-blocking socket back to blocking.

timeout is the number of seconds the blocking will persist until the process resume its operation even without an input.

5.9 Special Functions

5.9.1 mailto – send an e-mail

BEE Script:
```
mailto emailAddress [subject=string]
[from=emailAddress] [fromname=string]
[cc=emailAddress] [bcc=emailAddress]
[errorsto=emailAddress]
[ignore=string] [header=string] [body=string];
```

BEE Tag:
```
<beemailto emailAddress [subject=string]
[from=emailAddress] [fromname=string]
[cc=emailAddress] [bcc=emailAddress]
[errorsto=emailAddress]
[ignore=string] [header=string] [body=string]>
```

Note: If the message is sent to multiple addresses, please separate them but commas.

The "mailto" command differs from other form-mail scripts in several ways:

Firstly, "mailto" command does not show up in the <form ...> tag. You simply set the action="..." to the confirmation (or thank-you) page, where the "mailto" command is included.

Secondly, "mailto" command can send an e-mail without a form submission. You can use this to get alert messages directly from the website.

Thirdly, "mailto" command does not disclose any parameters to the users including the recipient's e-mail address, not even from the page source view. All the users can see is the confirmation page, and by then the mail has already been sent.

This is to protect the privacy of the recipient and prevent the e-mail address from being scanned on the website by online marketers.

Parameters

subject of the e-mail. Default is "Online Submission".

from is the e-mail address appearing at the "From" line of the e-mail.

fromname is the name of the sender appearing at the "From" line of the e-mail.

cc is the CC recipient of the e-mail. If there are multiple addresses, please separate them by commas.

bcc is the BCC recipient of the e-mail. If there are multiple addresses, please separate them but commas.

errorsto is the e-mail address errors are sent to. In another word, if the e-mail cannot be delivered, the "bounce back" will be sent to the "errorsto" address. (The default will be the "from" e-mail address if specified, or otherwise, the e-mail address of the user which the web server runs on.)

header is a string specifying any extra header lines you want to append to the original header. For example, header="{const%mail:HTML_HEADER}" will turn the message from plain text into HTML format.

Please be careful not to repeat the header lines. For example, if you specify the "From" line in the header string, do not specify the "from" parameters. Otherwise, there will be two "From" lines in the header, which may result in unpredictable result.

body is the message body to send. If "body" is blank, the elements in the sys%form variable (the form entries) will be listed to form the message body. You can use body="..." to send an arbitary message, whether a form is submitted or not. This is useful to keep track of access to pages of importance. You can send the administrator an e-mail notification whenever the page is accessed and/or a condition is met.

ignore is a list of names (separated by commas) specifying the sys%form elements to be ignored in the e-mail. Example: ignore="MyEmail, Submit" would cause "mailto" tag not to include MyEmail and Submit into the message body. The parameter "ignore" will be ignored if the parameter "body" is specified, in which case all sys%form elements will be ignored anyway.

After calling the function, the following BEE Variables are made available:

`result%mailto:header`	The message header built for the e-mail
`result%mailto:body`	The message header passed in (or built) for the e-mail.

5.9.2 text – specify an online editable and searchable text

BEE Script:

```
text name [default=string]
[editprompt=string] [init=string]
[webpath=string] [textpath=string]
[display=bool] [hidden=bool] [showvar=bool]
[edit=bool] [anchor=string]
[highlight=var] [hlbegin=string] [hlend=string]
[bgshield=string]
[allowview=accessControlSpec]
[allowedit=accessControlSpec];
```

BEE Tag:

```
<beetext name [default=string]
[editprompt=string] [init=string]
[webpath=string] [textpath=string]
[display=bool] [hidden=bool] [showvar=bool]
[edit=bool] [anchor=string]
[highlight=var] [hlbegin=string] [hlend=string]
[bgshield=string]
[allowview=accessControlSpec]
[allowedit=accessControlSpec]>
```

The Text Command is used to specify a unit of content that can be displayed, searched. If the "admin" user is logged in, the text can be edited. In such case, at the end of each text unit there is a link to a Text Edit page, which allows modification of the text content via a word processor-like interface.

The *textName* is a "Text Variable" in the format of "page&text:item". The "page" part is default to be the path of the caller page's URL ({sys%urlshown:pathpage}). For example, If the "text" command is called from "http://www.foo.com/blah/mypage.htm?abc=1", the default "page" part will be "/blah/mypage.htm".) The "text" part is the name of the text unit and should be unique within the page. The "item" part is optional. If used, it helps to group different text units under the same name for clarity.

The value of the text variable is stored with the website as part of the content. It can be accessed via the "text" class: "text%*page*&*text*:*item*" (internally implemented as a "scheme" class object). Since "page" is default to the current page, you can take it that every page has a "text" class of its own (i.e. text%*text*:*item* refers to text%*CurrentPage*&text:*item*). The "page&" part is used only if you want to refer to the "text" class of another page.

The "item" part is rarely used. However, one special item called "AllowView", which will let you specify an access level below which the text is not visible. For example, if you specify "text%member.htm&sensitive:AllowView" to "member", any sessions with a privilege under that of "member" will not see the text. (AllowView defaults to "public".) There are another special item called "AllowEdit" which control the editing of page text. More details can be found later in this section.

<u>Parameter</u>

<u>default</u> specifies the string to display if the text content is blank.

<u>editprompt</u> specifies the string to display in the text editor if the text content is blank.

<u>init</u> specifies the string to display and save automatically if the text content is blank. Upon encountering of a "text" command with the "init" parameter, the text content is created and contains the value of the "init" parameter (unlike "default" which does not create any content unless the administrator explicitly click the Edit icon, enter the new text and Save.)

The "init" parameter is usually used to convert an existing page into a template without affecting the appearance. You just surround the original text with <u><beetext *textName* init='*originalText*'></u> and after the first run of that page, the content is saved. (The "init" value has no effect in subsequent runs and therefore can be removed.)

<u>webpath</u> is for "non-standard" text storage structure. For example, if your website's home path is http://*server*/mysite, so for page http://*server*/mysite/mypage.htm you naturally want its content to be stored under text%/mypage.htm&. However, according to the "standard" text storage structure, the content should be in text%/mysite/mypage.htm. To breach the two structures, you can specify webpath=/mysite and the "text" command will work out that the text file is in fact in text%/mypage.htm& (instead of text%/mysite/mypage.htm&).

<u>textpath</u> is for "non-standard" text storage structure. For example, if for some reasons your want to "bury" your content under a subdirectory (usually for version control), you can specify textpath=/subdir and the "text" command will use text%/subdir/mypage.htm& (instead of text%/mypage.htm&) for the content of http://*server*/mypage.htm.

<u>display</u> indicates whether to display the text unit or not. The default is true. The usual case to turn "display" off is to include a server-side hidden text in the page so that the browser at the client-side cannot see, not even with "View Source". For example, if a product information page shows pricing that is derived from a cost figure that you set in the page, you would not want the customer to see the cost. You can use the Text Command with display=false so only the admin login will see it.

<u>hidden</u> indicates whether to hide the text regardless of the "admin" login. A true for "hidden" is not the same as a false for "display". If "display" is false, it will not run the "text" command at all unless it is the "admin" login. That means "display" has no effect when logged in as "admin". On the other hand, if "hidden" is true, it will run regardless, even for non "admin" user. It is only when it comes to showing the text, it will skip the text unless it is in the editor. "hidden" is usually used for text of non-display purposes, such as to initialise a variable or to file up an editor to capture a user entered text for application use.

<u>showvar</u> indicates whether to evaluate for BEE values before displaying the string. This does not affect the text in edit, which is always showing the text without evaluation so that the user can modify the expression. The default is false.

edit indicates whether to allow edit on this text unit or not. Even when "edit" is true, you still need to login as the "admin" user in order to be able to edit the text unit. If "edit" is false, the text is not editable in any circumstances.

This parameter can be used to turn off the edit feature for the text unit, or to specify a "duplicated" text unit of which the value will inherit from another editable text unit of the same name. The default is true.

anchor if specified indicates the name of the HTML anchor tag (e.g. <a name=...></a>) for direct reference/link to the text position. You can turn anchor off by specifying anchor="". If the "anchor" parameter is not specified and the "edit" parameter is true, then the system will use the "text" part of the *textName* as the anchor's name (or if the item part is specified, an underscore followed by the item name will be appended to the anchor's name to make it unique.)

highlight is the name of the variable that contains the string to highlight (usually as a result of a search). If the variable is an array, every strings in the array will be highlighted.

hlbegin specifies the string displayed before the highlighted text. The default is "<strong>".

hlend specifies the string displayed after the highlighted text. The default is "</strong>".

bgshield specifies the opacity of the background shield, which stops the editor from going beyond the Text Edit Area. The value can be from 0 (clear) to 100 (complete white). The default is 65.

allowview specifies the minimum access level required to view the text. This parameter overwrites all AllowView specification in the text% and scheme% class (see below).

allowedit specifies the minimum access level required to edit the text. This parameter overwrites all AllowEdit specification in the text% and scheme% class (see below).

There are some special "text" and "scheme" class variables to govern the user privileges of text viewing and editing:

`text%`*page*`&`*text*`:AllowView`	The minimum access level required to view the text. Default is "public".
`text%`*page*`&AllowView`	The minimum access level required to view any text on the page, except for those individually governed by text%*page*&*text*:AllowView. Default is "public".
`scheme%AllowView`	The minimum access level required to view any text on the website, except for those individually governed by text%*page*&*text*:AllowView or text%*page*&AllowView. Default is "public".
`text%`*page*`&`*text*`:AllowEdit`	The minimum access level required to edit the text. Default is "edit".

`text%`*page*`&AllowEdit`	The minimum access level required to edit any text on the page, except for those individually governed by text%*page*&*text*:AllowEdit. Default is "admin".
`scheme%AllowEdit`	The minimum access level required to edit any text on the website, except for those individually governed by text%*page*&*text*:AllowEdit or text%*page*&AllowEdit. Default is "admin".

There are some "scheme" class variables to control the text viewing and editing functions:

`scheme%TextEdit:ArgPage`	The argument name via which that the "page" part of "name" is passed to the Text Edit page. The default is "pagecode".
`scheme%TextEdit:ArgText`	The argument name via which that the "text" part of "name" is passed to the Text Edit page. The default is "textcode".
`scheme%TextEdit:ArgItem`	The argument name via which that the "item" part of "name" is passed to the Text Edit page. The default is "itemcode".
`scheme%TextEdit:Link`	The HTML code to specify the link to be displayed immediately after the text unit. You can use BEE Variable syntax within the code. Some more "local" variables are made available: param%text:pagecode - The page part of "name" param%text:textcode - The text part of "name" param%text:itemcode - The item part of "name" (If scheme%TextEdit:ArgPage, scheme%TextEdit:ArgText and scheme%TextEdit:ArgItem is specified, use their values in intead of "pagecode", "textcode" and "itemcode" respectively.) If you are writing your own Text Edit page, all it does is to accept the three arguments, retrieve the text via "text%[text]:[item]", modify it and save it into the same. If scheme%TextEdit:Link is not specified, the following scheme parameters will be used.
`scheme%TextEdit:URL`	The URL of the Text Edit page WITHOUT the pagecode, textcode and itemcode arguments.

	The default is "/textedit/edit.htm".
`scheme%TextEdit:Caption`	The caption appear in the Text Edit link. The default is "Edit".
`scheme%TextEdit:Target`	The target window that the Text Edit page will appear. The default is "_blank" (a new window).
`scheme%TextEdit:Title`	The "title" attribute for the anchor tag. (In IE 4 and above, the title string will appear when you move your mouse over the link.) The default is "Edit Text".

There are some special "session" and "scheme" class variables to govern the image and file insertion:

`session%ImageFile`	If not blank, it will be used as the image address after the "webpath". (In this case, the user will not be able to change the image address in the InsertImage dialogue box.) For example, if the webpath is http://*server*/mysite and the session%ImageFile is /img/mypict.gif, the image URL will be "http://*server*/mysite/img/mypict.gif".
`session%ImageDir`	If not blank, it will be used as the default image directory, and the local file name to upload will be appended to this image directory to form the image address. The user can overwrite this image directory in the InsertImage dialogue box if they choose to. The default is "{scheme%ImageDir}" (which in turn defaults to "/images").
`scheme%ImageDir`	Same as "{session%ImageDir}" but effective only if session%ImageDir is blank. The default is "/images".
`scheme%ImageRoot`	The "root" path preceding the image address. For example, if {scheme%ImageRoot} is "/cust", the image address will be under /cust. e.g. "/cust/images/mypict.gif". This setting usually requires the upload directory (sys%croft:filedir) to point to the physical directory corresponding to the "/cust" path.
`session%ImageSubroot`	This will be appended to {scheme%ImageRoot}. For example, if {scheme%ImageRoot} is "/cust" and {session%ImageSubroot} is "/brochure", the

	image address will be under /cust/brochure. e.g. "/cust/brochure/images/mypict.gif".
`session%LinkDir`	If not blank, it will be used as the default file upload directory, and the local file name to upload will be appended to this link directory to form the link address. The user can overwrite this link directory in the InsertFile dialogue box if they choose to. The default is "{scheme%LinkDir}".
`scheme%LinkDir`	Same as "{session%LinkDir}" but effective only if session%LinkDir is blank. The default is blank.
`scheme%FileRoot`	The "root" path preceding the link address. For example, if {scheme%FileRoot} is "/cust", the link address will be under /cust. e.g. "/cust/mydoc.pdf". This setting usually requires the upload directory (sys%croft:filedir) to point to the physical directory corresponding to the "/cust" path.
`session%FileSubroot`	This will be appended to {scheme%FileRoot}. For example, if {scheme%FileRoot} is "/cust" and {session%FileSubroot} is "/brochure", the link address will be under /cust/brochure. e.g. "/cust/brochure//mydoc.pdf".
`session%Phantom`	The "Phantom" mode. When set, no text will be saved and no images or files will be uploaded. Instead, the local link will be returned into the Text Edit page for display (so that the user sees the same as if Phantom mode is off). "Phantom" mode is provided by the Portal Object, not the BEE platform. But the object requires platform support (via this session variable) to stop text saving or file uploading.

After calling the function, the following BEE Variables are made available:

`result%text:length`	The length of text displayed. This is usually for checking whether any text has been displayed (e.g. so as to display a default).

5.9.3 auth – control authentication information

BEE Script:	`auth` *`action`*`;`
BEE Tag:	`<beeauth` *`action`*`>`

The command "auth" is to control authentication information.

There are only one "auth" action currently available in BEE:

`save`	Save all authentication attributes to the user record (Authentication attributes are data for the logged in user, and can be changed by assigning new values to variable sys%auth:*attributeName*.) This operation will set status%auth and message%auth accordingly.

Note: The "password" attribute cannot be changed in this way. You must assign an array of two values: *old* and *new* into the System Variable sys%password. e.g. <u>var pswd:old = "myoldpswd"; var pswd:new = "mynewpswd"; var sys%password = (var)pswd;</u>

5.9.4 scheme – access the scheme file

BEE Script:	`scheme` *`action`*`;`
BEE Tag:	`<beescheme` *`action`*`>`

The command "scheme" is used to access the scheme file.

The followings are the "scheme" actions currently available in BEE:

`reload`	Reload all scheme settings (ignoring all changes made to the non-file "scheme" class since the last save)
`unload` or `save`	Save all scheme settings (all changes made by made to the non-file "scheme" class. Note: Other users will not get the new settings until they restart their browser.

5.9.5 output – control the display

BEE Script:	`output` *`action`*`;`
BEE Tag:	`<beeoutput` *`action`*`>`

The command "output" is used to control the display.

The followings are the "output" actions currently available in BEE:

`bufferbegin`
`begin`
`(default)`
	Begin redirecting display to an internal output buffer. In another word, stop displaying and instead, build a display string internally.
	
	The output buffer can be accessed via the sys%output:content variable. Its length (number of characters in it) is in sys%output:length.

`bufferend` `end`	Display the output buffer onto the web page, destroy the content of the output buffer and stop further display redirection to the output buffer.
`buffercancel` `cancel`	Destroy the content of the output buffer without displaying it, and stop further display redirection to the output buffer.
`now`	Flush the output to the client. (Server platform and client browser dependent.)
`instantly`	Flush after each output (as if using "output now" after every "display"). This will implicitly assume an "output bufferend" so that all buffered content will be output at once and buffering will end immediately. Note: On some platform, using "output instantly" may have an impact on performance.
`noninstantly`	Turn off "output instantly", which means to display output according to the platform and browser default behaviour in bufferring.

5.9.6 sleep – pause the operation

BEE Script: `sleep [sleep=`*num*`];`

BEE Tag: `<beesleep [sleep=`*num*`]>`

The command "sleep" is used to pause the operation for a specific period of time. The parameter:

Parameters

sleep is the number of seconds that the operation will pause for. The default value of "sleep" is 1.

5.9.7 virtualpage – indicate a Virtual Page template page (optional)

BEE Script: `virtualpage;`

BEE Tag: `<beevirtualpage>`

The command "virtualpage" output an HTTP page header with Status Code 200 (OK) to indicate that the page is a valid page. This is useful to overwrite the default Status Code 404 (Not Found) when the web server redirect a virtual page (missing page) to the template page.

This command is <u>not</u> required in most cases. However, if it is ever used, it <u>must</u> be the first line of the template page before any output (even before the <html>).

The reason to have this command is that some browsers (notably Internet Explorer) and browser extension software trap the 404 Status Code and substitute the entire page with a "friendly" page or search-advertisement page. On such browers, the BEE Virtual Page mechanism (missing-page redirection to a template page that load real content based on the requested URL) will not work. This "virtualpage" command will "cheat" the browser or extension software into believing that the page is "real" and therefore not to alter it.

5.9.8 franchise – to acquire CROFT access of an idURL (CLI only)

BEE Script: `franchise franchise=idURL;`

BEE Tag: `<beefranchise>`

BEE allocates resources based on the idURL (the URL without the http:// part and arguments), but if you are running the script from the Shell through the CLI (The PHP Command Line Interpretor), there is no idURL associated with the script. If you require CROFT resources like database access, you will need to assign an idURL to the script through the "franchise" command.

For security reason, the script file needs to be owned by a "franchisee" of the idURL in order to franchise to it. The list of franchisees of any particular idURL is specified in the CROFT system. Such list is usually empty unless the idURL allows franchising.

The "franchise" command is for CLI only and will be ignored if you are running it on a webpage.

<u>Parameters</u>

<u>**franchise**</u> is the idURL of which the CROFT access the script would assume after the execution of the command. Please note that the idURL is in the format of webserver/webpath format. For example, www.mywebsite.com or www.mywebsite.com/mypath.

6 Database Operation

The BEE Database Operation is carried out by several commands, mainly the "database" command. For data retrieval, subsequent "get" commands (most commonly represented in "var" assignment) or "foreach" loop can be used to fetch the records in the retrieved record set. For details of the syntax and parameters of these commands, please see "BEE Command Reference".

There are two modes of operation with the "database" command: "query" and "action". In either mode, the system will eventually submit an SQL query to the database engine and obtain the result from it. The difference of the two modes is in how the SQL query is built.

In the "query" parameter is specified, the "query" mode is assumed and the "action" parameter will be ignored.

In "query" mode, the SQL query statement is specified by the user explicitly. The system simply take the query string, evaluate for embedded variables, and submit it to the database engine. For data retrieval, the system will retrieval the record set for later fetching. If "seek" is specified, the required record in the record set will be located for subsequent fetching.

If the "query" parameter is not specified but the "action" parameter is, the "action" mode is assumed.

In "action" mode (only if the "query" parameter is NOT specified), the SQL query statement is built from the "Argument Variables" in the database object (represented in this document as *dbobj*) and the matrix. (Please see the "matrix" parameter under the "database" command.)

If neither the "query" parameter nor the "action" parameter is specified, the "database" command does not do anything unless "seek" is specified, in which case, the "seek" operation is performed on the last retrieved record set.

6.1 Database Variables

The "database" command uses many BEE Variables implicitly, especially in "action" mode, which is almost entirely variable driven. The variables are under the database object (the *dbobj* as represented in this document), the matrix class and in some system classes (e.g. status%, message% and session%).

They can be categorised into three groups: Argument Variables (input to the system in "action" mode), System Variables (output from the system), and Paging Variables (also output from the system but for the purpose of display paging).

6.1.1 Argument Variables (Input to the system)

Argument Variables are variables specified by the user and not changed by the system. They are used as input arguments to the database operation.

Please note that all Argument Variables are ignored if the parameter "query" is specified. In such case, the SQL statement contained in the "query" parameter is all the database operation is needed to complete the job. (The variable *dbobj*%decode is still effective even in an SQL statement query retrieval, but it will not be used until the record set is later retrieved via a "get" command or a "foreach" loop.)

The following Argument Variables form part of the table definition. Even though these Argument Variables can be set up or changed at any time, it is recommended to define the necessary ones in the table definition section of the code (except for *dbobj*%search, which is usually done when the search keys are processed, for example, from the form input.)

Variable	**Description**	
`dbobj%table`	If specified, the table name used in an "action". Otherwise, the name of *dbobj* will be used as the table name.	
`dbobj%keyfield`	The name of the key field name.	
	The value of *dbobj*%keyfield is used by different actions in different ways:	
	select:	*dbobj*%keyfield is not used
	update:	*dbobj*%keyfield, if specified, overwrites the record filter (the generated where-clause) with *keyfield* = ’*keyValue*’ (the *keyValue* is from the matrix.) Also, the key field and its corresponding value will be removed from the set-list to avoid the key field being updated.
	insert:	*dbobj*%keyfield, if specified, its value will be disgarded and replaced with one generated by the underlying database platform. If the database does not generate keys, it will be an error.
	delete:	*dbobj*%keyfield, if specified, overwrites the record filter (the generated where-clause) with *keyfield* = ’*keyValue*’ (the *keyValue* is from the first row of the matrix). In this case, only one record can be deleted.
`dbobj%mustfields`	If specified, contains a list of compulsory fields. A compulsory field is one that must appear in *dbobj*%fields. The system will examine *dbobj*%fields and if there are compulsory fields missing, their names will be appended to *dbobj*%fields.	
	This feature is useful when some fields in the matrix	

	might be missing (e.g. if the matrix receives its values from a form, and so happens that none of the checkboxes or radio entry of a particular column are checked by the user). When such a "matrix" is processed by the "group" command, the missing fields will not be included into *dbobj*%fields at all, and the whole column will be ignored (instead of setting them all to the unchecked status.) Therefore, it is recommended to include every checkbox and radio entry into *dbobj*%mustfields.
dbobj%wild	To avoid accidental operations on the entire table (e.g. due to programming or operational error), "update" and "delete" will return an error if there is neither a key field nor a where-clause implicit (record filter) or explicit (*dbobj*%where). By specifying *dbobj*%wild to be true (non-null and non-zero), you overwrite this protection and allow operation to be carried out for the entire table in one go without a where-clause and without a key field being specified. For "select", selecting multiple tables without a where-clause will result in an error (to protect the platform from being overloaded with a huge record set) unless *dbobj*%wild is true. Selecting on a single table without a where-clause is valid, regardless of the value of *dbobj*%wild. *dbobj*%wild has no effect if the "query" parameter is specified in the "database" command. (You can do anything with a "query".)
dbobj%orderby	The "order by" clause without the word "order by". It is a comma-delimited list of field names, each is optionally followed by "desc" to indicate a reverse sorting order. This variable is used only for data retrieval operation. (If multiple tables are involved, please make sure the table name qualifiers and a dot is included in each field to be sorted.)
dbobj%match:*matchName*	If specified, contains a search criterion named *matchName*. The criterion is in the format of *matchType*:*matchField*, where *matchField* is the field in the table to match (default is *matchName*), and *matchType* is one of these: equal Field content is equal to the key

substr	Field content contains the key as a substring
regexp	Field content matches the regular expression represented by the key
clike	Field content matches the clike expression represented by the key
rlike	Field content matches the rlike expression represented by the key
min	Field content is equal to or higher than the key
max	Field content is equal to or lower than the key
minx	Field content is higher than the key
maxx	Field content is lower than the key

dbobj%search:*matchName*

If specified, contains the key for match criterion *matchName*. The value of the key is to be used to build the record filter (the generated where-clause). e.g. If "*dbobj*%match:CanAfford" contains "max:Price" and "*dbobj*%search:CanAfford" contains "1000", the where-clause will contain the condition "Price <= 1000". All these conditions are "and"ed together to form the where-clause.

If *dbobj*%match:*matchName* does not exist, it is assumed to be "equal". This simplify the process: setting *dbobj*%search:*field* to *value* generate a criterion of *field* = '*value*'.

Usually, the *dbobj*%search variable copies the values from the search form via sys%form as in:

```
var dbobj%search = "(var)sys%form";
```

dbobj%where

If specified, will be taken as the first condition in the record filter (the generated where-clause). i.e. the condition contained in *dbobj*%where will be "and"ed to the generated where-clause from the left.

dbobj%selectfrom

If specified, overwrites the field list (*dbobj*%fields) and table (*dbobj*%table).

The variable *dbobj*%selectfrom is useful to join multiple tables in a "select" operation. In that case, the *dbobj*%where variable needs to contain the conditions that link the tables together. That is why the variable *dbobj*%selectfrom is effective only if *dbobj*%where is specified (unless *dbobj*%wild is true, which is a dangerous and useless things to do as the

	joining will become an unrestricted crossing of all the tables involved, and is likely to end up timing out.)
dbobj%encode:*field*	If specified, contains a BEE Conversion to apply to the value to be written to Field *field*, or the value to be used in the record filter (the where-clause). e.g. `var friend%encode:Birthday = "strtotime";`
dbobj%decode:*field*	If specified, contains a BEE Conversion to apply to the value read from Field *field*. e.g. `var friend%decode:Birthday = "strftime:%Y-%m-%d";`
dbobj%quote:*field*	If specified, contains a quotation mark to use when generating the query for Field *field*. The default is single quotation mark ('). Set it to blank if no quotation mark is required. e.g. with mSQL `var friend%quote:ID = "";`
dbobj%fieldnames	If specified, the elements will be used as keys of the record fetched from the record set. This is useful if you want to overwrite the field names defined in the database. The list is position sensitive as the first element will be used as the key for the first fetched field, the second element as the key for the second field and so on. In multiple table joining, the same field names may exist in different tables. In such case, *dbobj*%fieldnames is the only way to distinguish them by giving each field a distinct name. *dbobj%fieldnames* are used at the time of reading the database result (via the (db)*dbobj* cast), not at the time of "query". That means you can modify *dbobj%fieldnames* even after reading has started. This will cause subsequent readings to bear different fieldnames.

6.1.2 System Variables (Output from the system)

There are variables that are updated automatically by the system as a result ("output") of the database operation.

The followings are general System Variables in the database object:

Variable	**Description**

dbobj%tablelist	Contains a list of table names available in the underlying database. Value initialised upon the first access of "database" command regardless of what operation and table it is on (if at all), and the value does not changed in the entire page run.
dbobj%fieldlist	Contains a list of field names available in the table specified in *dbobj*%table, or if it is not specified, the name of *dbobj*. If you modify the value of *dbobj*%table, the next access to the "database" command will see *dbobj*%fieldlist updated with a new field list of the newly specified table. Any successful data retrieval operation will update *dbobj*%fieldlist as well, even if it was via the SQL statement containing in the "query" parameter.
dbobj%query	Last query statement regardless of whether it was successful or not
dbobj%queryaction	Action of the last query statement (i.e. "select", "update", "insert" or "delete") regardless of whether it was successful or not

The followings are general System Variables in system classes:

Variable	**Description**
dbobj%status or status%database	Error code of the database operation, or 0 if successful
dbobj%message or message%database	Error message of the database operation, or blank if successful
session%database:query	Last successful query saved as persistent value that is kept through out the client session
session%database:select	Last successful select query saved as persistent value that is kept through out the client session
session%database:update	Last successful update query saved as persistent value that is kept through out the client session
session%database:insert	Last successful insert query saved as persistent value that is kept through out the client session
session%database:delete	Last successful delete query saved as persistent value that is kept through out the client session

`session%database:queryaction`	Action of last successful query saved as persistent value that is kept through out the client session

A brief note on session%database, upon a successful database operation, the system will save *dbobj*%query into session%database:query, and *dbobj*%queryaction into session%database:queryaction, then save *dbobj*%query again into session%database:{*dbobj*%queryaction}.

These "session" variables also have a copy in session%{*dbobj*%name}, where {*dbobj*%name} is the unique name the system generates or you explicitly specify in the database operation.

There are some System Variables that are related to data retrieval only:

Variable	**Description**
`dbobj%numrecs` or any of these: `dbobj%numrec` `dbobj%numrecords` `dbobj%numrecord` `dbobj%numrows` `dbobj%numrow`	Number of records in the record set retrieved in the last query, or 0 if it was unsuccessful or was not a data retrieval query.
`dbobj%thisrecord:field`	Contains the Field *field* in the record just retrieved. (If the record just retrieved is empty, for example, when retrieved beyond the last record in the record set, *dbobj*%thisrecord will not be updated (and will retain its last value).
`dbobj%lastrecord:field`	Contains the Field *field* in the record retrieved before the one just retrieved. If the record just retrieved is empty, for example, when retrieved beyond the last record in the record set, *dbobj*%lastrecord will not be updated (and will retain its last value).
`dbobj%datacount`	Contains the number of records read so far.
`dbobj%datacount:field`	Contains the number of last consecutive retrievals in which the data of Field *field* has not changed. For example, if vehicle%thisrecord:make is "BMW" and vehicle%datacount:make is 3, then you can tell that the last two retrievals on "make" were also "BMW". This is useful in detecting change of value (i.e. when *dbobj*%datacount:*field* is 1) to determine the end of a subtable. If the record just retrieved is empty, for example, when retrieved beyond the last record in the record set, *dbobj*%lastrecord will not be updated (and will

retain its last value).

There are some variables that are related to data insertion only:

Variable	Description
`dbobj%lastseq`	The last sequence number inserted into the table. It is an output from the "nextseq" action. If the value is negative (-1), that means the previous "nextseq" came up with an error. Please note that not all database platform supports a retrieval of the last generated sequence number. Please see Database Action "nextseq".

6.1.3 Paging Variables (for display paging)

It is a common design for a commercial website to display a large record set in separate pages. But it may not be efficient (if possible at all) to keep the database connection and record set handles across sessions, as you will never know when the client will finish and release the resources.

It would be much easier to implement record set display paging in separate queries. That is to make a new query every time a new page of records is required. For example, if 200 records are returned and the first page of 10 is on the screen, the visitor can click "next page" to get the 11th to the 20th records. The web page will submit the same query again (as the previous one has been completed and resources released) but start retrieving from the 11th record. If the visitor click, say, page 6 on the page navigation bar, the web page will again submit the query and go straight to record 51 and start retrieval from there.

The "database" command handles this mechanism automatically and neatly. The "Paging Variables" of the database object are for this purpose. But before we do into the details of those variables, we need to explain how the display paging work.

6.1.3.1 How paging works

As mentioned before, the "seek" parameter specify the position to start the next retrieval. There are two more optional numbers in the "seek" parameter after the record to seek: the number of records per page (RPP, default is 10) and the number of pages per block (PPB, default is 10). (A block is the collection of pages on the page navigation bar.)

With these three numbers, BEE will calculate the positioning of the pages on the page navigation bar and return an array. The target page will contain the sought record. For example, seek="71,10,5" means to seek record 71, with 10 records per page and 5 pages per block. The page navigation bar should contain page 6 (record 51-60), page 7 (record 61-70), page 8 (record 71-80), page 9 (record 81-90) and page 10 (record 91-100). The sought record (number 71) is in page 8.

If the sought record does not fall into the first record of the page, BEE will automatically change the "seek" position internally so that the next retrieval will always start from the first record of the page. In the above example, if seek="73,10,5", the next retrieval will still start from 71 because it is the first record in the sought page (page 8).

This automatic repositioning does not apply unless the paging numbers (the second and/or the third number) are specified. This will guarantee that you get what you want to seek if you are not interested in display paging.

Besides the positioning, we need to make sure the same record set is carried through to various pages of the same search. This is done by keeping track of the last successful "select" query and resending it to the database platform in each page run.

The last successful query has been stored as a session variable. All you need to do is to have query="{session%database:select}". (Alternatively, if you want to use "action" mode, you can save the search criteria in *dbobj*%search and use it to generate a query in subsequent paging display. This is a more complicated solution. The "session" variable method is preferable unless you have a good reason not to use it.)

The format of the hyperlink to the pages can be specified in the *dbobj*%pagebar variable (input), and the system will generate the "cells" on the page bar and put them into the *dbobj*%page variable (output).

Example:

```
    var href = "href=@self?record=@record";

    var mydb%pagebar:first = "<a {href}>|&lt;&lt;</a>";
    var mydb%pagebar:previous = "<a {href}>&lt;&lt;</a>";
    var mydb%pagebar:back = "<a {href}>&lt;-</a>";
    var mydb%pagebar:page = "<a {href}>@page</a>";
    var mydb%pagebar:sought = "<b>@page</b>";
    var mydb%pagebar:forward = "<a {href}>-&gt;</a>";
    var mydb%pagebar:next = "<a {href}>&gt;&gt;</a>";
    var mydb%pagebar:last = "<a {href}>&gt;&gt;|</a>";

    var rec = "{form%record}";
    if ({#rec:} == 0) var rec = 1;

    var recordsperpage = 5;
    var pagesperblock = 3;

    database "mydb" query="{session%database:select}"
          seek="{rec},{recordsperpage},{pagesperblock}";

    display "{pb%page|list:(@value)&nbsp;&nbsp;()}";
    // If rec is 71 and there are 200 records in the record set,
    // display          |<<  <<  <-  6  7  8  9  10  ->  >>  >>|
    // hyperlink record=  1  41  61 51 61    81  91  81 101  191
```

Note: The "pagebar" definition in the above example is the default. So the above initialisation is trivial unless you want to use different values (e.g. some button images.)

6.1.3.2 Variables for paging

The followings are the Paging Variables for the page navigation bar display:

Variable	Description
dbobj%pagebar	Contains a list of format strings used by the system to generate *dbobj*%page for displaying a page navigation bar. The elements are positionally sensitive as the page bar (in *dbobj*%page) will follow the same order as in *dbobj*%pagebar (except for *dbobj*%pagebar:sought.) Not all the format strings will be used. For example, if the sought record is already in the first page of the first block, then "first", "previous" and "back" will not be included in the *dbobj*%page output variable. Likewise, for the last record, "last", "next" and "forward" will not be included. In the format string, the following string will be replaced by the corresponding values shown below: @page the page number of the sought page @record the record number of the first record of the sought page @self the path-page of the URL (i.e. {sys%self}). @query the SQL of the last query URL-encoded.
dbobj%pagebar:first	The format string for the first page in the first block. (|<< in the above example.)
dbobj%pagebar:previous	The format string for the last page of the previous block. (<< in the above example.)
dbobj%pagebar:back	The format string for the page before the sought one. (<- in the above example.)
dbobj%pagebar:page	The format string for the pages of the sought block. (The numbers in the above example.)
dbobj%pagebar:sought	The format string for the sought page. It usually set to contain no hyperlink and is in a different font, so that it can be easily recognised as the sought page (the one currently on display.) If *dbobj*%pagebar:sought is missing, its value will be assumed to be the one in *dbobj*%pagebar:page.

	Since the sought page is among the other pages in the sought block, the system will determine its position and therefore the format string position in the *dbobj*%pagebar variable is not significant and does not affect the output.
dbobj`%pagebar:forward`	The format string for the page after the sought one. (-> in the above example.)
dbobj`%pagebar:next`	The format string for the first page of the next block. (>> in the above example.)
dbobj`%pagebar:last`	The format string for the last page in the last block. (>>| in the above example.)
dbobj`%page`	Contains in each of its elements the display of a page in the navigation bar, formated by the corresponding format string specified in *dbobj*%pagebar. *dbobj*%page is available only if the "seek" parameter is specified in the "database" command. For details, please refer to the example presented earlier in this section.

There are other information that helps to determine where the record, the page and the block are:

Variable	**Description**
dbobj`%sought:record`	The record number of the sought record Note: this value always equals to the first number in the "seek" parameter, regardless the possible repositioning of the "seek" position to align with the first record in the page. (Please see *dbobj*%first:rip.)
dbobj`%sought:page`	The page number of the page that the sought record is in (the sought page)
dbobj`%sought:block`	The block number of the block that the sought page is in (the sought block)
dbobj`%first:recordinpage` or *dbobj*`%first:rip`	The record number of the first record in the sought page Note: this value always reflects the actual "seek" position, so that the next retrieval always start from the first record in the page.
dbobj`%first:pageinblock`	The page number of the first page in the sought

or *dbobj*%first:pib	block
dbobj%first:recordinblock or *dbobj*%first:rib	The record number of the first record in the first page of the sought block
dbobj%last:record	The record number of the last record in the record set. The value is the same as *dbobj*%numrecs
dbobj%last:page	The page number of the last page in the record set.
dbobj%last:block	The block number of the last block in the record set.
dbobj%last:recordinpage or *dbobj*%last:rip	The record number of the last record in the sought page
dbobj%last:pageinblock or *dbobj*%last:pib	The page number of the last page in the sought block
dbobj%last:recordinblock or *dbobj*%last:rib	The record number of the last record in the last page of the sought block
dbobj%count:recordsinpage or *dbobj*%count:rip	The number of records in a page
dbobj%count:pagesinblock or *dbobj*%count:pib	The number of pages in a block
dbobj%count:recordsinblock or *dbobj*%count:rib	The number of records in a block

6.2 Database Actions

Most of the "magic" of database operation in BEE are provided in the "action" mode (when the "action" parameter is specified without the "query" parameter.)

There are eleven Database Actions. Among them are four data access (select, update, insert and delete), five VirtualBase operations (create, load, unload, showcreatetable and showinsertinto) and two supplementary (push and nextseq). You can specify any one of these in the "action" parameter of the "database" command.

6.2.1 Data Access Actions

The four data access actions come with some synonyms and are shown in a smaller font in the following:

Data Access Actions	**Description**
select search get	Read from the database. The system will first check whether *dbobj*%selectform is

`read`
`retrieve`

specified and the select range is restricted by *dbobj*%where (unless *dbobj*%wild is true). If so, they will be used to generate the SQL query statement.

Otherwise, the generated SQL statement will be a select on the table defined in *dbobj*%table on the fields in *dbobj*%fields. If *dbobj*%table is not specified, the name of *dbobj* will be used as the table name. If *dbobj*%fields is not specified, all fields ("*") will be retrieved.

Whether "selectfrom" is used or not, the key values in *dbobj*%search will be used to generate the where-clause (and "and"ed after *dbobj*%where if one is specified,) and *dbobj*%orderby will be used in the "order by" clause of the generated SQL statement.

The key values contained in *dbobj*%search will be properly encoded with the BEE Conversions in *dbobj*%encode before being used to generate the where-clause.

The SQL statement will be submitted to the database engine and the retrieved record set can be fetched via "get" commands or a "foreach" loop, after the BEE Conversions in *dbobj*%decode are applied to their corresponding fields.

`update`
`change`
`modify`

Update the database.

The update will be on the table defined in *dbobj*%table. If *dbobj*%table is not specified, the name of *dbobj* will be used as the table name.

The system will get the fields to be updated from *dbobj*%fields, appended with field names in *dbobj*%mustfields (if specified) for those not in *dbobj*%fields already. Then *dbobj*%keys will be looped through and an SQL update statement will be generated for each element in *dbobj*%keys, based on the values in the matrix. (Please see the "matrix" parameter under the "database" command.)

All the field values used in the SQL update statement will be properly encoded using the corresponding BEE Conversions in *dbobj*%encode.

If *dbobj*%keyfield is specified, the corresponding key field will NOT be updated, and the generated where-clause will be ignored. Instead, the value of the key field found in the matrix will be used to find the record to update.

If *dbobj*%keyfield is not specified, the search criteria in *dbobj*%search will be used to generate the where-clause used in the SQL update statement.

Please note that when updating multiple records from a matrix, it is important to always specify *dbobj*%keyfield. This way, the system will generate multiple SQL update statements, each update only one record with "where *keyfield* = ...", which make sure the corresponding row on the matrix is being updated.

To avoid updating the entire table by mistake, if the SQL update statement is found to have no where-clause, the system will return an error (unless *dbobj*%wild is true).

`insert`
`put`
`add`

Add a new record into the database.

The new record will be inserted into the table defined in *dbobj*%table. If *dbobj*%table is not specified, the name of *dbobj* will be used as the table name.

The system will get the fields to be included in the new record from *dbobj*%fields, appended with field names in *dbobj*%mustfields (if specified) for those not in *dbobj*%fields already. Then *dbobj*%keys will be looped through and an SQL insert statement will be generated for each element in *dbobj*%keys, based on the values in the matrix. (Please see the "matrix" parameter under the "database" command.)

All the field values used in the SQL insert statement will be properly encoded using the corresponding BEE Conversions in *dbobj*%encode.

The last sequence number used with the database object will be checked (assuming that the database platform will not change its value by mere checking.)

If *dbobj*%keyfield is specified and the last sequence number can be obtained from the database platform, the system will assume that the database platform is capable of generating the key value internally and therefore would not bother to include the key field in the inserted record.

However, if the last sequence number is not available, the system will try to obtain the next sequence number from the database platform for the key value. If that cannot be obtained, it will be an error. In another word, if the database platform is not capable of generating either last sequence or next sequence, *dbobj*%keyfield should not be specified at all.

Note: If *dbobj*%keyfield is specified in an "insert" operation, the key field value should be available in the matrix (even as a dummy value) in order to trigger the key generation process.

`delete`
`erase`
`remove`

Delete a record or records from the database.

The record will be deleted from the table defined in *dbobj*%table. If *dbobj*%table is not specified, the name of *dbobj* will be used as the table name.

If *dbobj*%keyfield is specified, the corresponding key field of the first row of the matrix will be used to identify the record to be deleted. (The second row and onwards in the matrix are ignored in the "delete" data access action.)

If *dbobj*%keyfield is not specified, the search criteria in *dbobj*%search will be used to generate the where-clause used in the SQL delete statement.

To avoid deleting the entire table by mistake, if the SQL delete statement is found to have no where-clause, the system will return an error (unless *dbobj*%wild is true).

Please note that only one "delete" SQL will be executed. For deletion via the matrix, only one record will be deleted. For deletion using the search criteria in *dbobj*%search (when *dbobj*%keyfield is not specified), multiple record can be deleted. Deleting without a where-clause (deleting the entire table) is possible only if *dbobj*%wild is true.

6.2.2 VirtualBase Actions

VirtualBase is a "table-on-the-fly". It is a data structure that can be accessed via the usual database interface. A VirtualBase table does not exist on any database platform as such. Instead, it was created, used, and disposed of all in the single page run. Alternatively, you can save the VirtualBase into a file (the "unload" action) so that you can re-create it in the future (the "load" action).

VirtualBase is useful in keeping temporary data that require database-like access interface like sorting, filtering, column and row selective processing etc. Sometimes VirtualBase is used to hold design attributes for run-time input. For example, Site-map can be easily done with VirtualBase and the "dbtree" command.

There are only three VirtualBase specific actions. Once a VirtualBase table is instantiated, you can access it in the "query" mode or the "action" mode as if it is an ordinary database table.

VirtualBase Actions	**Description**
`create`	Create a new VirtualBase. Table definition can be specified using "create" SQL statement.
`load[:file]`	Parse a file for VirtualBase query, which is usually starting with a "create" statement followed by many "insert" statements. (Action "load" does a "create" implicitly before parsing the file.) The *file* argument is default to the name of *dbobj*.
`unload[:file]`	Save the VirtualBase to a file in the form of VirtualBase query. The saved file is in the format that the "load" action can parse.

	The *file* argument is default to the name of *dbobj*.
`store[:sess]`	Store the VirtualBase to the session named *sess*. Note: The *sess* is NOT a session variable. It is just a name specifically for VirtualBase session storage.
`restore[:sess]`	Restore the VirtualBase from the session named *sess*. (See notes from "store".)
`showcreatetable`	Return the "create table" SQL statement via the *dbobj*%result variable (string).
`showinsertinto`	Return the "insert into" SQL statements of all data rows via the *dbobj*%result variable (array, one element for each row).

6.2.3 Supplementary Actions

The two supplementary actions are used sparingly and only when there are absolute needs to do so:

Supplementary Actions	**Description**
`push:`*`var`*	Put the record (in the named BEE Variable array) back to the database so that it can be retrieved as the next record. The "Pushed in" record will be cleared everytime a new query or a seek operation is performed.
`nextseq:[`*`seq`*`]`	Get the next sequence for insert-key generation. The result is stored in the variable *dbobj*%lastseq. Please note that not all database platforms support automatic key generation. For those that do, the implementation may be very different. Some database platforms generate a new key value implicitly whenever a record is inserted, and allow you to the generated key without affecting its value (like MySQL). Some others only allow you to get the key value from a sequence name (the *seq* argument) and a new value is generated every time it was accessed (like mSQL). The sequence name is usually the table name (but does not have to be). Therefore, the table name in *dbobj*%table is the default *seq*. (Please note that accessing to the variable *dbobj*%lastseq will not cause a key generation as the variable is only a buffer holding the sequence returned by the action "nextseq".) The behaviour of "nextseq" is platform dependent and it is

dangerous to use it in the insert statement as two processes may get the same value if the platform does not generate a new key every time it was accessed. It is recommended to insert via the "insert" action, instead of an explicit SQL insert statement via the "query" parameter, unless you know exactly what you’re doing and have the need to do so.

7 Content Management

To allow end users (website owners) to update the web content is one of the major features that make BEE a class of its own.

The "content" of a business website is a synthesis of effort contributed by many different people: art designers who put in beautiful graphics and background, operation staff who collect and enter business data, programmers who write programs to process and present raw data into useful information, and writers and individual business units who define the message they want to deliver to the visitors.

BEE Content Management focus on helping the last group of people, whom have been left out of the content cycle for too long. Every time they need to change a piece of content or have a new message for the visitors, they need to rely on the web designers or programmers to do it for them. BEE is going to turn this around, allowing the owner of the content to take control of their content directly on the website with a browser, instead of waiting (and in most cases paying) other people to do it for them.

7.1 What is Web Content

Web "content" is sometimes referred to as static information, as they remain unchanged for majority of web pages. Those that are dynamically changed are usually the result of an application operations (such as displaying the content of a shopping cart or a listing of all staff in the department).

However, some information do not change often, but they do change sometimes. For example, the "what's new", the contact information, special promotion, newsletter etc.

In BEE, we define web content as information presented in a predefined area of a specific web page. Web content comes in "units", called Web Content Unit. The system handles Web Content Units as text variables (a special type of scheme class variables). You can have a program script to handle Web Content Units, show them, hide them, even change them, like they do variables.

When a Web Content Unit is being edited online, it will be surrounded by a dotted line square so that the user what can be changed.

7.2 Virtual Page

The separation of design and content, and the independence of code and resources, have made possible the online creation and maintenance of new pages without having to go through the local authoring and uploading processes.

In another word, you can have only one page, and have it automatically load different content into the TEA (Text Editing Area) based on the path-page of the URL. That is done by setting up the web server to redirect a page-not-found error to a template page. Then you can have that template page load the text content based on the *path*/*page* information obtained from the web server.

Only one page is needed as the template and you can have as many pages as you want, all with different contents and can be individually updated via the browser:

```
<h1><beetext "headline" default="Page not found"></h1>

<beetext "content" default="Please check the web address and try again">
```

When a visitor requests a non-existing page, say, http://www.mysite.com/news/sports.htm, the web server will display the above page with the default text. However, if the visitor is logged in as the administrator, instead of the page-not-found text (the "default"), two Text Editing icons will display. The administrator can then click into the TES (Text Editing Screen) and enter the text for each of the TEAs.

Upon saving the first TEA, the administrator has created the page by making the text item available in the "text" class. No actual web page has been created on the web server; only a piece of new content has been entered. That is what we called a "virtual page".

Note: In the current implementation of BEE, you can only submit a form to a Virtual Page using the "GET" method.

8 Objects and Classes

8.1 Classes

Unlike in the Object-Oriented world that a "class" is a template of an "object" and an "object" is an instance of a "class", in BEE terminology, "class" is "object" and "object" is "class". Object templates are represented by their constructor functions and therefore are not called "class" as in the proper Object-Oriented language.

To differentiate the "classes" that represent objects and those defined by the system, you can call the former "object classes" and the latter "system classes" (or just "classes"). For details, please refer to the "BEE Variable Name" Section.

8.2 Objects

BEE Objects are implemented with Variable Class. They can be read, written and cleared. They are also locally defined. (i.e. remain in the local Context and become undefined outside of the function unless declared "global".)

8.2.1 The Constructor

Since BEE Objects are just Classes, you can create an object out of thin air by assigning values into the class variables.

Alternatively, you can create an object with a Constructor function. You define the Constructor function that initialise the class variables through the "this" class. Then you call this Constructor function with the class name prefix. For example:

```
function vehicle
{
        var this%make = "(var)arg%m";
        var this%capacity = "(var)arg%c";
}

myVehicle%vehicle m="Toyota" c="4";
```

Every class has a special variable called "function", which contains the references to all the functions (or methods) associated with that class.

For example, var myVehicle%function:new = "vehicle" will define a function "myVehicle%new", which will refer to the function "vehicle" when it is called. In that case, the constructor call can be rewritten in the following way:

```
var myVehicle%function:new = "vehicle";
myVehicle%new m="Toyota" c="4";
```

There is a special syntax to express the above in one statement:

```
var myVehicle% = new vehicle m="Toyota" c="4";
```

The "new" operator is commonly used in calling the constructor, instead of direct call in the first two examples.

However, in BEE Tag, there is not "new" operator and you need to use direct call as in the following:

```
<beemyVehicle%vehicle m="Toyota" c="4">
```

or

```
<bee var="myVehicle%function:new" value="vehicle">
<beemyVehicle%new m="Toyota" c="4">
```

8.2.2 Calling an Object Method

BEE Script: `ObjName%funcname [name=value ...];`

BEE Tag: `<beeObjName%funcname [name=value ...]>`

When calling a function in the above format, it will first find the function *funcname* from the object *ObjName*. The object's function needs to be set in the constructor using var this%function:*funcnam* = "*actual_function_name*";, where the *actual_function_name* is a name defined with the "function" command.

If the function name of the object is not set, the actual function of the same name will be called.

There are three extra arguments that can be accessed within the function: arg%function:this, which evaluates to *ObjName*, arg%function:thisfunction, which evaluates to *funcname*, and arg%function:function, which evaluates to *actual_function_name*.

For example, if the object "myobj" gets var this%function:mymethod = 'myfunc'; and the method is invoked with myobj%mymethod a=1 b=2; then:

`arg%function:a`	1
`arg%function:b`	2
`arg%function:this`	myobj
`arg%function:thisfunction`	mymethod
`arg%function:function`	myfunc
`arg%a`	1
`arg%b`	2
`arg%this`	myobj
`arg%thisfunction`	mymethod

Note: *ObjName* can contain BEE Variables. However, the current version of the parser handle BEE Variable as *ObjName* only if there is no space, ">" or "]" in the variable syntax. e.g. `{v}%func;` is OK, but `{v|replace:\ ,x}%func` is not because of the space in the conversion argument. Also, *funcname* cannot contain an "%" in its syntax.

8.2.3 Polymorphism

Polymorphism in the BEE is implemented with function reference. Every object class has a "function" variable, of which every element is a definition of a function name. For example:

```
var car1%model = "Camry";
var car2%model = "Diablo";

var car1%function:whatType = "familyCar";
var car2%function:whatType = "playCar";

function familyCar
{
      display "{this%model} is a car for the family.";
}

function playCar
{
      display "{this%model} is for fun.";
}

car1%whatType;  // Camry is a car for the family.
car2%whatType;  // Diablo is for fun.
```

8.2.4 Inheritance

Inheritance in BEE is done by the child's constructor calling the parent's, so the object have all the child's variables (object attributes) and functions (method), PLUS all the parent's variables and functions.

In the following example, Vehicle has two Child classes: car and plane. We use the same constructor vehicle in the previous example and create two more:

```
function vehicle
{
      var this%make = "{arg%m}";
      var this%capacity = "{arg%c}";
      var this%function:do = "dontknow";
}

function car
{
      this%vehicle m="{arg%m}" c="{arg%c}";
```

```
        var this%wheel = "{arg%w}";
        var this%function:do = "drive";
}

function plane
{
        this%vehicle m="{arg%m}" c="{arg%c}";
        var this%engine = "{arg%e}";
        var this%function:do = "fly";
}

function drive
{
        display "{arg%who} drive {arg%this}<br>\n";
}

function fly
{
        display "{arg%who} fly {arg%this}<br>\n";
}

function dontknow
{
        display "{arg%who} don't know what to do with {arg%this}<br>\n";
}

var myCar% = new car m="Toyota" c="4" w="alloy";
var myPlane% = new plane m="Boeing" c="400" e="jet";
var mySpaceship% = new vehicle m="Alien Nation" c="4000";

myCar%do who="I";
myPlane%do who="You";
mySpaceship%do who="We";
```

Note: When an object function needs to call another function of the same object, it is recommended to call into the absolute name instead of the object function name in case the callee function is overloaded. For example:

```
function vehicle
{
        var this%make = "{arg%m}";
        var this%capacity = "{arg%c}";
        var this%function:do = "dontknow";
        var this%function:start = "vehicle_start";
        var this%function:findkey = "vehicle_findkey";
}

function car
{
        this%vehicle m="{arg%m}" c="{arg%c}";
        var this%wheel = "{arg%w}";
        var this%function:do = "drive";
}

function vehicle_start
{
      this%findkey;  // For safety, replace this by this%vehicle_findkey;
      display "Start Vehicle.<br>\n";
}

function vehicle_findkey
{
      display "Find the key of the Vehicle.<br>\n";
}
```

The above worked fine, but if the constructor of "car" overload the function findkey (e.g. var this%function:findkey = "car_findkey";), then vehicle_start will call into "car_findkey". If this is not your intention, please replace "this%findkey;" with "this%vehicle_findkey;".

Calling on the actual function name allows the children object to call an overloaded parent function without ambiguity.

8.2.5 Object within Object

While the "this%" class holds variables of the object, there is no equivalent construct to hold another object within an object. However, this can be worked around by context linking into the constructor's local context.

The constructor is a function to create the object. It can also create other objects in its local context. These "member" object can be in possession of the "main" object.

However, once the constructor exists, the local context is lost and gone with the "member" object. One way to keep the context (and therefore any "member" objects in it) alive is to save the context into an object variable (e.g. this%context). Subsequent calls to the object's methods can reacquire access to the constructor's local context (and therefore the "member" objects in it).

```
function person
{
      var this%name = "{arg%name}";
}

function car
{
      var this%context = "{sys%context}";               // Save context
      var ownerobj = new person name="{arg%name}";      // An object name

      var this%make = "{arg%make}";         // A string
      var this%function:belongsto = "car_belongsto";
}

function car_belongsto
```

```
{
      // Link "ownerobj" from "car"'s local context to the current one.
      var ownerobj% =& ownerobj% context="{this%context}";
      display "This {this%make} belongs to {ownerobj%name}";
}

var myCar = new car make="Ford" name="John Lee";
myCar%belongsto;  // This Ford belongs to John Lee
```

Sometimes, you may not want to keep the object externally and pass to the object method when calling. In that case, the "parent" command is required to access the external object from the method.

```
function car
{
      var this%make = "{arg%make}";         // A string
      var this%function:belongsto = "car_belongsto";
}

function car_belongsto
{
      parent "{arg%owner}%";   // The owner object is from the parent
      display "This {this%make} belongs to {{arg%owner}%name}";
}

var jl = new person name="John Lee";
var myCar = new car make="Ford";

myCar%belongsto owner="jl";
```

The need for a "parent" command is not always obvious. For example, if the method does not access the external object but calls another function that does, "parent" is still required in the method function to "pass on" the linkage.

```
function car_belongsto
{
      parent "{arg%owner}%";   // This is required to pass on
      this%car_ownersname owner="{arg%owner}";
      display "This {this%make} belongs to {result%car_ownersname}";
}

function car_ownersname
{
      parent "{arg%owner}%";   // linked to the caller's context
      var result%function = "{{arg%owner}%name}";
}
```

In "car_ownersname", the {arg%owner}% is linked to the same variable in the caller's context. If "parent" is missing from "car_belongsto", "car_ownersname" will link to no object.

8.2.6 Database Object – an Example

Here is the complete working source code of the database object (dbobj) in the common library (common/dbobj.bs), as an example to show how to wrap a database in a nutshell, I mean, object:

```
<script language="bee">

/*********************************************************
 *      dbobj: Database Object
 *
 *      arg%table = the table name (default is the object name {arg%this})
 *
 *      Object variables:
 *      %matrix         the name of the matrix used.  Default "matrix"
 *      %dbid           the dbid.  Default ""
 *      %status         the status of last database operation
 *      %message        the message of last database operation
 *
 *      All other object variables are "inherited" from the database command
 *********************************************************/

function dbobj
{
        var this%table = "{arg%table}";
        if ({#this%table:} == 0) var this%table = "{arg%this}";
        var this%matrix = "{arg%this}_matrix";
        var this%dbid = "{arg%dbid}";

        var this%status = 0;
        var this%message = "";

        var this%function:query = "dbobj_query";
        var this%function:select = "dbobj_action";
        var this%function:update = "dbobj_action";
        var this%function:insert = "dbobj_action";
        var this%function:delete = "dbobj_action";
        var this%function:create = "dbobj_dbtact";
        var this%function:load = "dbobj_dbtact";
        var this%function:unload = "dbobj_dbtact";
        var this%function:group = "dbobj_group";
        var this%function:menu = "dbobj_menu";
        var this%function:matrixdump = "dbobj_matrix_dump";
        var this%function:dump = "dbobj_dump";
}


/*********************************************************
 *      dbobj_query: Send a query to the database
 *
 *      arg%query (default) = the query string
 *      arg%seek = the seek parameter for the database command
 *********************************************************/

function dbobj_query
{
        if ({#arg%function:seek} > 0) var param%database:seek = "{arg%function:seek}";
        database this query="{arg%query}" dbid="{this%dbid}";
}


/*********************************************************
 *      dbobj_action: Called via dbobj%select, dbobj%update, dbobj%insert, dbobj%delete
 *              to carry out the corresponding database command
 *
 *      arg%thisfunction = the action parameter for the database command
 *      arg%seek = the seek parameter for the database command
 *********************************************************/

function dbobj_action
{
        parent "{this%matrix}%";
        if ({#arg%function:seek} > 0) var param%database:seek = "{arg%function:seek}";
        database this matrix="{this%matrix}" action="{arg%thisfunction}" dbid="{this%dbid}";
}


/*********************************************************
 *      dbobj_dbtact: Called via dbobj%create, dbobj%load, dbobj%unload
 *              to carry out the corresponding dbtree command
 *
 *      arg%function:this = the name of the dbtree object in the calling statement
 *      arg%thisfunction = the action parameter for the database command
 *********************************************************/

function dbobj_dbtact
{
        parent {arg%function:this}%;
        database "{arg%function:this}" name="{this%name}" action="{arg%thisfunction}"
dbid="{this%dbid}";
}


/*********************************************************
 *      dbobj_group: Perform the "group" command on the specified
 *              variable into the class specified by this%matrix and
 *              set the group result into the object.
 *
 *      arg%group (default) = the variable to be grouped.
 *      If this%matrix is blank, set it to {arg%this}_matrix
 *********************************************************/

function dbobj_group
{
        var gvar = "{arg%{arg%thisfunction}}";
        if ({#gvar:} == 0) return;
        parent "{gvar}";

        if ({#this%matrix} == 0) var this%matrix = "{arg%this}_matrix";

        // The "group" command always operates in the local context.
        // Needs to link the variables it changes to the parent.
        parent "{arg%this}%";
        parent "{this%matrix}%";
        group "{gvar}" matrix="{this%matrix}" result="{arg%this}";
}


/**********************************************************
 *      dbobj_matrix_dump: Dump the matrix
 **********************************************************/

function dbobj_matrix_dump
{
        if ({#this%matrix} == 0) return;
        if ({#this%keys} == 0) return;
        if ({#this%fields} == 0) return;

        parent "{this%matrix}%";

        display "<table>\n<tr valign=top><td>Key</td>\n";
        foreach (this%fields) display "<td>{foreach}</td>\n";
        display "</tr>\n";

        foreach (this%keys as key)
        {
                display "<tr valign=top><td>{key}</td>\n";
                foreach (this%fields as field) display
"<td>{{this%matrix}%{key}:{field}}</td>\n";
        }

        display "</table>\n";
}


/**********************************************************
 *      dbobj_dump: Dump the database object
 *
 *      arg%fieldlist = the name of the fields to display in the format of
 *              key=>val where key is the caption and value is the field name.
 *              If the caption is preceeded by '!', val is a derived value with
 *                      @f for value, where f is the field name;
 *      arg%pagesize = the maximum number of records in a page
 *      arg%form = the input tag attributes
 *              @key for the value of the key field
 *              (form is displayed only if the object got the keyfield defined)
 *      arg%td = the td tag attributes
 *      arg%format = the field formats (read-only fields only)
 *      arg%hiddenvalue = the array of values that are hidden, indexed by fieldname
 *      arg%readonlyvalue = the array of values that are read-only, indexed by fieldname
 *********************************************************/

function dbobj_dump pagesize=10
{
        if ('{this%table}' == '' || '{this%status}' > 0) return;
        // Now we got a proper table with OK status and at least some records.

        var hl = "(array)#e0ffff,#ffffff";  // Alternative highlighting
        var hli = 0;  // Highlight index

        var fieldlist = "(var)arg%fieldlist";
        if ({#fieldlist} == 0) var fieldlist = "(var)this%fieldlist";

        display "<table>\n";

        // Display the header
        display "<tr bgcolor=#808080>\n";
        foreach (fieldlist)
        {
                if ({foreach:key|foundre:^[0-9]}) var caption = "{foreach}";
                else var caption = "{foreach:key|replacere:^!}";
                display "<td><font color=#ffffff>{caption|words}</font></td>\n";
        }
        display "</tr>\n";

        foreach maxiter="{arg%pagesize}" ((db)this as rec)
        {
                display '<tr valign=top bgcolor="{hl:{hli}}">\n';
                //var fldcnt = 0;
                var keyValue = "{rec:{this%keyfield}}";
                foreach (fieldlist as fld)
                {
                        display "<td {arg%td:{fld:key}}>";
                        //display "{fld:key}:" conv="if:'{fld:key}' !=
'{this%fieldlist:{fldcnt}}'";
                        //display "{fld|{{name}%decode:{fld:key}}}</td>\n";
                        if ({fld:key|foundre:^!})
                        {
                                var fldValue = "{fld}";
                                foreach (this%fieldlist) var fldValue =
"{fldValue|replace:@{foreach},{rec:{foreach}}}";
                                display "{fldValue}";
                        } else
                        {
                                var fldFormatted = "{rec:{fld}}";
                                if ({#arg%format:{fld}}) var fldFormatted =
"{arg%format:{fld}|replace:@fldval,{fldFormatted}}";
                                if ({#keyValue:} == 0 || ('{fld}' != '{this%keyfield}' &&
{#arg%form:{fld}} == 0))
                                {
                                        display '{fldFormatted}';
                                } else
                                {
                                        var hiddenFld = '<input type=hidden
name="{fld}_{keyValue}" value="{rec:{fld}|replace:\\\\",&quot;}">';
                                        var readonlyFld = '{hiddenFld}{fldFormatted}';
                                        var shownFld = '<input type=text
name="{fld}_{keyValue}" value="{rec:{fld}|replace:\\\\",&quot;}"
{arg%form:{fld}|replace:@key,{keyValue}}>';
                                        if ({arg%hiddenvalue:{fld}|isset} && '{rec:{fld}}' ==
'{arg%hiddenvalue:{fld}}')
                                                display '{hiddenFld}';
                                        else if ({arg%readonlyvalue:{fld}|isset} &&
'{rec:{fld}}' == '{arg%readonlyvalue:{fld}}')
                                                display '{readonlyFld}';
                                        else if ('{fld}' == '{this%keyfield}')
                                                display '{readonlyFld}';
                                        else
                                                display '{shownFld}';
                                }
                        }
                        display "</td>\n";

                        //var fldcnt = "(expr){fldcnt} + 1";
                }
                display "</tr>\n";

                if ({hli} > 0) var hli = 0;
                else var hli = "(expr){hli}+1";
        }
        display "</table>\n";
}


/************************************************************
 *      dbobj_menu: Display the record set as a menu
 *
 *      arg%self = the name of the field holding the "Self" value
 *      arg%root = the root node name
 *      arg%active = the active node name
 *      arg%show = the minimum "activeness" to show
 *      arg%indent = the indentation "tab"
 *      arg%format:node         the format of the node
 *      arg%format:current      the format of the current node (default arg%format:node)
 *      arg%format:open         the format of opened nodes (default arg%format:current)
 *      arg%format:close        the format of closed nodes (default arg%format:open)
 *      arg%format:off          the format of off nodes (default arg%format:close)
 *      arg%format:currentparent        the format of the current node if it's a parent
(default arg%format:current)
 *      arg%format:openparent           the format of opened parent nodes (default
arg%format:open)
 *      arg%format:closeparent          the format of closed parent nodes (default
arg%format:close)
 *      arg%format:offparent            the format of off parent nodes (default arg%format:off)
 *      arg%prev = the parent variable name to receive the record preceeding the current one
 *      arg%next = the parent variable name to receive the record following the current one
 ************************************************************/
```

```
function dbobj_menu show=1 indent="&nbsp;&nbsp;"
{
        var format = (var)arg%format;
        if (!{format:current|isset}) var format:current = "{format:node}";
        if (!{format:open|isset}) var format:open = "{format:current}";
        if (!{format:close|isset}) var format:close = "{format:open}";
        if (!{format:off|isset}) var format:off = "{format:close}";
        if (!{format:currentparent|isset}) var format:currentparent = "{format:current}";
        if (!{format:openparent|isset}) var format:openparent = "{format:open}";
        if (!{format:closeparent|isset}) var format:closeparent = "{format:close}";
        if (!{format:offparent|isset}) var format:offparent = "{format:off}";

        var lead = "(array)
                30=>{format:current},
                20=>{format:open},
                10=>{format:close},
                00=>{format:off},
                31=>{format:currentparent},
                21=>{format:openparent},
                11=>{format:closeparent},
                01=>{format:offparent}";

        dbtree this self="{arg%self}" root="{arg%root}" active="{arg%active}";
        var minact = "{arg%show}";
        if ({#minact:} == 0) var minact = 1;

        var gotCurrent = 0;
        foreach ((dbtree)this as rec)
        {
                if ({result%this:activeness} >= {minact})
                {
                        var dispstr = "{lead:{result%this:activeness}{result%this:isparent}}";
                        var dispstr = "{dispstr}"
conv="replace:@@INDENT,{arg%indent|repeat:{result%this:level}}";
                        var dispstr = "{dispstr}"
conv="replace:@@ACTIVENESS,{result%this:activeness}";
                        var dispstr = "{dispstr}"
conv="replace:@@ISPARENT,{result%this:isparent}";
                        var dispstr = "{dispstr}" conv="replace:@@LEVEL,{result%this:level}";
                        foreach (rec) var dispstr = "{dispstr}"
conv="replace:@{foreach:key},{foreach}";

                        display "{dispstr}";
                }

                switch ({gotCurrent})
                {
                case 0:
                        if ({result%this:activeness} == 3) var gotCurrent = 1;
                        else var prev = "(var)rec";
                        break;
                case 1:
                        var next = "(var)rec";
                        var gotCurrent = 2;  // The work to obtain prev and next is finished
                        break;
                }
        }

        if ({arg%prev|isset} && {prev|isset})
        {
                parent "{arg%prev}";
                var "{arg%prev}" = "(var)prev";
        }

        if ({arg%next|isset} && {next|isset})
        {
                parent "{arg%next}";
                var "{arg%next}" = "(var)next";
        }
}
</script>
```

9 Interface with other languages

BEE is a server side scripting language, but it was designed to be compatible with most client-side scripts and scripting languages. BEE Tag was introduced so that it can intermix with HTML. Building client-side JavaScript was a common way of passing server information to the interactive client interface. Also, BEE can interleave with its implementation language (PHP at the moment) within one web page.

9.1 BEE and HTML

Every BEE Command got a BEE Tag form of which the format is similar to HTML Tag. Its usage with HTML is limited only by the programmers imagination. For example, you can maintain the form context by inserting the BEE Form Variables:

```
<input type=text name=GivenName value="${sys%form:GivenName}"><br>
<input type=text name=Surname value="${sys%form:Surname}"><br>
<input type=text name=Age value="${sys%form:Age}"><br>
<input type=text name=Title value="${sys%form:Title}"><br>
```

To keep the <select …> tag value is straight forward in BEE:

```
<bee var=sel:{sys%form:Title} value="selected">
<select name=Title>
<option value="">--select Title--</option>
<beeforeach "(array)Mr,Ms,Mrs,Miss,Dr">
<option value="${foreach}" ${sel:{foreach}}>${foreach}</option>
</beeforeach>
</select>
```

9.2 BEE and JavaScript

JavaScript is a client-side language and is therefore part of the visible web page source. BEE can build JavaScript by outputting its code. For example, the following build a JavaScript array variable from a BEE (server side) array:

```
display ’<script language="JavaScript">\n’;
display ’var jsCars = new Array\n’;
foreach (cars as model)
      display ’jsCars["{model:key}"] = "{model}";\n’;
display ’</script>\n’;
```

BEE can conditionally turned on or off section of JavaScript:

```
if (’{scheme%discount}’)
{
      display ’<script language="JavaScript">\n’;
      display ’price = price * (1 - {scheme%discount} / 100);\n';
      display 'alert("Now you can get {scheme%discount}% discount!");\n';
      display '</script>\n';
}
```

9.3 BEE and PHP

The current version of BEE is implemented in PHP. Naturally, you can run sections of a BEE Web Page in native PHP code. All global PHP variables that BEE uses in the implementation start with "BEE_". If you are writing PHP codes in a BEE Web Page, please make sure you do not use a variable name that starts with "BEE_".

To access the BEE environment from a PHP section, there are several functions you can use.

9.3.1 BEE_get($*value*)

Return the BEE Value contained in the expression *value*. Anything you can put into the "value" parameter of the "var" command, can be used as the argument of BEE_get().

The value after BEE evaluation will be returned. Please note that this is not a reference. It is a copy of the value, even if it is specified as "(var)…". Changing the PHP Variable returned from BEE_get("(var)…") will not affect the value of the underlying BEE Variable. If you want to retrieve a reference to a variable so that your modification to that variable from PHP will be effective in the BEE Scope, please use BEE_var().

9.3.2 BEE_var($*varname*)

Return a PHP variable reference to the BEE Variable identified by *varname*. If the element part is omitted, the array form of the BEE variable will be returned.

Please note the followings:

Remember to use "=&" to receive the reference. e.g. $a =& BEE_var("apple");

Be aware that an array is returned if the element part is omitted. For example:

```
<script language="bee">
var myVar = "abc";
</script>
<?php
$arr =& BEE_var("myVar");  // an array in which $arr[""] is "abc"
$elm =& BEE_var("myVar:"); // a scaler "abc"
?>
```

Specifying a non-existing *element* will create that element in the variable array with null. (This is the same in BEE Script when referring an element with the (var) cast, like "(var)myVar:newElement".)

```
<script language="bee">
var myVar:x = "abc";  // a single-element array
</script>
```

```
<?php
$elm =& BEE_var("myVar:y");
// Now the BEE Variable myVar got a new element y (null value)
// and becomes a two-element array.
$elm = "def";  // myVar now x=>"abc",y=>"def"
?>
```

System Class BEE Variables have no corresponding PHP variable reference and therefore cannot be accessed via BEE_var(). You can only use BEE_get () and BEE_set() to access them. If you retrieve a System Class BEE Variables by BEE_var(), the returned reference has no effect on the BEE Variables at all. In fact, such PHP Variables is not in the BEE Scope.

The "file" part in *varname* will be ignored if specified.

9.3.3 BEE_isset($*varname*)

Return a PHP *true* value if the BEE Variable identified by *varname* is defined, or *false* if otherwise. This is the same as "{*varname*|isset}" except that it returns a PHP *true* or *false* value instead of 1 or 0.

Please note the followings:

BEE_isset will return *false* if a System Class BEE Variable name is specified.

The "file" part in *varname* will be ignored if specified.

9.3.4 BEE_set($*varname*, $*string*)

Set the string to the variable *varname*. Please note that *string* is not subject to any BEE evaluation. It is put straight into the variable named by *varname*. In fact, BEE_set is equivalent to "var *varname* =! *string*" in BEE Script.

If you want the *string* to be evaluated, please use BEE_get() first.
e.g. BEE_set($myPHP, BEE_get("{myBEE}"));

9.3.5 BEE_clear($*varname*)

Clear the variable *varname*.

9.3.6 BEE_convert($*value*, $*conversion*)

Return the result of the PHP expression or variable after the BEE Conversion specified in *conversion*. If the argument *value* is a PHP variable reference, changes by the BEE Conversion (if the BEE Conversion accept pass by reference) will be effective on the PHP variable. For example:

```
<?php
$a = 1;
$b = BEE_convert($a, "inc");
printf("a=%d b=%d\n", $a, $b);  // a=1 b=2
$x = 1;
$y = BEE_convert(&$x, "inc");
printf("x=%d y=%d\n", $x, $y);  // x=2 y=2
?>
```

9.3.7 BEE_do($*funcname*)

Call the BEE Function or Command named by *funcname*. The arguments can be set via *param%funcname:argname* variable before the function call. The result can be accessed by means of *result%funcname*.

Please note that the parameters are to be set exactly as they appear in the calling statement. If a variable is to be passed, please use "(var)…" instead of BEE_var(). If you want strings to be BEE evaluated, please use BEE_get(). Object function calls need to set the object name as the "this" parameter instead of including it in the function name. For example:

```
<script language="bee">
myobj%myfunc argone="abc" argtwo="(var)myvar" timenow="{sys%time}";
foreach (result%myfunc) display "{foreach:key}: {foreach}<br>\n";
</script>
```

is equivalent to

```
<?php
$param["argone"] = "abc";
$param["argtwo"] = "(var)myvar";
$param["timenow"] = BEE_get("{sys%time}");
$param["this"] = "myobj";
BEE_set("param%myfunc", $param);
BEE_do("myfunc");
$result = BEE_get("(var)result%myfunc");
foreach ($result as $k => $v) printf("$k: $v<br>\n");
?>
```

Please note that the first four groups of BEE Commands, Variable Operations, Conditional, Loop and Module Calling, CANNOT be called via BEE_do(), with the exeption of "clear", "group", "include" and "exec" (which you can use even they fall into one of the four groups).

10 Glossary

AMS – Please see "Authentication Mechanism Specification".

Attribute Name – The Name part of an Attribute Name-Value Pair.

Attribute Name-Value Pair – Parameter specification in a BEE Command in the form of *name=value*.

Attribute Value – The Value part of an Attribute Name-Value Pair, optionally quoted by a pair of single or double quotation marks if the value string contains white spaces.

Authentication Mechanism Specification – a CROFT record that tells the BEE system where to find the user/password table, its database type, the server, the database and table name, field mapping and how to decode the password.

BEE – Please see "Business Electronic Enterprise".

BEE Code Section – The section in the web page between two neighbouring <script language="bee"> tag and </script> tag inclusively.

BEE Command – One of the commands specified in the "BEE Command Reference" section of this document. A BEE Command is used in this document to refer to the operation (the BEE Command Name) and its parameters. A BEE Command may appear as a BEE Statement or a BEE Tag.

BEE Command Name – The operation of a BEE Command. It is the first word of the BEE Command, less the object reference if exists in BEE Script, or in the case of BEE Tag, less the "<bee" part. (Please see "BEE Command".)

BEE Conversion – A function that convert a value into another value, which is used in place of the original one.

BEE Hosting Administrator – The person who administer the BEE Hosting Server

BEE Hosting Provider – The organisation who provides the BEE Hosting Service

BEE Hosting Server – The computer which runs the BEE software along with other necessary programs and/or system to serve web pages from a BEE Website to the visitors.

BEE Hosting Services – The service of providing, running and administering a BEE Hosting Server.

BEE Script – Please see "BEE Script Syntax" and "BEE Statement".

BEE Script Syntax – A BEE program code syntax in which the command starts with the command name, is optionally followed by some name=value pairs, and is terminated with a semi-colon ';'.

BEE Section – The part of the web page containing BEE operation, which will be replaced by its output before the web page is sent to the client browser. BEE Script surrounded by <script language="bee"> and </script> inclusively is a BEE Section. A BEE Tag is a BEE Section by itself. Content between two neighbouring BEE Tags (white pages or else) is NOT part of a BEE Section.

BEE Statement – The smallest unit of independent operation in BEE Script syntax. A BEE Statement starts with a BEE Command and ends with a semi-colon (';'). A BEE Statement must be placed in a BEE Code Section.

BEE Web Path – Identified by an idURL of a BEE Website, and contains all the web pages of which the URL starts with the idURL after the protocol specification (http://).

BEE Tag – The smallest unit of independent operation in BEE Tag syntax. A BEE Tag starts with "<bee*command*" and ends with ">", where *command* is a valid BEE Command. A BEE Tag must NOT be placed in a BEE Code Section.

BEE Tag Syntax – A BEE program code syntax in which the command starts with "<bee" followed by the command name immediately after it (with no spaces), then a series of optional name=value pairs, and ends with ">".

BEE Variable – A run-time storage content of a BEE program, implemented as an array containing BEE Elements.

BEE Variable Name – Name of a BEE Variable in the form of `[class%][file&]name[:[#]element]`. A Variable Name without the element part (i.e. `[class%][file&]name`) mean the whole array or the string value of the Default Element (the element indexed by blank). Please see the "To String or Not to String" section.

BEE Website – A collection of BEE Web Paths, each identified by an idURL.

Business Electronic Enterprise – A web technology to enable easy development of commercial website through server-side scripting.

Code Section – Please see "BEE Code Section".

Command – Please see "BEE Command".

Command Name – Please see "BEE Command Name".

Conversion – Please see "BEE Conversion".

Context – A section of program code where a set of BEE Variables are defined separately from those outside of the section. When the execution leaves the section (e.g. a function), the BEE Variables defined in that section will be undefined.

CROFT – Please see "Customer Resource Online Facility Tables".

Customer Resource Online Facility Tables – An automatic system to bind resources to website according to their URL, so that the knowledge of the resource identification is concealed from the code.

DAS – Please see "Database Access Specification".

Database Access Specification – A CROFT record uniquely identified by an idURL and a DBID. A DAS describes a database access and contains information like Database Type, Database Host, Database Name, and the Username and password to gain access to the database on its data server. This information is stored within CROFT.

Default Element – The element in a variable which is indexed by blank. When a variable is specified without the element part (i.e. *class%name*), it will be taken as the default element unless in the Context where an array will be assumed.

DBID – An identifier that, together with an idURL, identify a DAS. This is useful for a web page to access multiple databases when necessary. For those pages that access none or a single database, DBID is not necessary (or is set to blank).

Element – An item in the array of a BEE Variable.

Hosting Administrator – Please see "BEE Hosting Administrator".

Hosting Provider –Please see "BEE Hosting Provider".

Hosting Server – Please see "BEE Hosting Server".

Hosting Services – Please see "BEE Hosting Services".

idURL – A partial URL that starts with the full web server name without the protocol specification (http://), and optionally followed by a path and/or a page. An idURL identifies a BEE Web Path. A collection of BEE Web Paths forms a BEE Website.

Loop Variable – The variable in a "foreach" or "for" loop that is updated by the system in every iteration by either assigning it the next item (in "foreach" loop) or increasing or decreasing its value (in "for" loop).

Matrix – A set of variables defined under a class (object) which serves as a two-dimensional array, with each variable representing a record (the variable name is the key) and its elements representing a field in the record.

OnMyWeb – The developer and intellectual property owner of BEE.

Owner-Service Duple – A dual-key uniquely identify a BEE Website. The Owner usually stands for the organisation that uses the BEE Website for its own purpose (or a web hosting customer in a BEE Hosting Provider environment). The Service usually stands for the name of the BEE Website name within the organisation, or just "main" if the organisation has only one BEE Website on the server.

Parameter – Arguments passed into a BEE Command or a function in the form of an Attribute Name-Value Pair.

Reference – An alias of a variable name. All BEE Variable Names are in fact references to the variable itself.

Scope – The set of resources that a web page or website have access to. The resources are allocated to the URL when the website is set up.

SME – Small to Medium Enterprise.

Statement – Please see "BEE Statement".

Tag – Please see "BEE Tag".

Variable – Please see "BEE Variable".

Variable Name – Please see "BEE Variable Name".

Variable Value – Please see "BEE Variable Value".

Virtual Page – A BEE feature that allows the website administrator to modify web content or create new web pages through the very website, without any authoring tools but the browser (Internet Explorer 6 or later).

Web Content – Information presented in a predefined area of a web page

Web Content Unit – A piece of Web Content that is always handled in a whole unit and allow online editing in a square shape area.