Visara

Master Console Center

Scripting Guide

P/N 707131-001

visara international

Technical Support

Contacting the Visara Intellicenter

For US domestic customers, Visara provides technical support through its Intellicenter, 8:30 - 5:00 (ET) Monday through Friday at 888-542-7282.

Calls outside these hours are handled by automatic pager, so expect a delay. You can also call through our switchboard at 919-882-0200. For support outside the US, please contact the company that has sold the equipment to you.

Notices

Contents

List of Tables

About This Guide

Purpose of This Guide

This guide illustrates the Master Console Center Global Control Language (MCC GCL). It assumes a sound foundation in programming concepts such as arrays, variable manipulation, looping, parameter passing, function calls, and return values.

You should read the *Operations Guide* before writing scripts to become familiar with basic MCC operations.

Users of this Guide

This guide is intended for Master Console Center users who want to learn the MCC GCL, as well as providing a reference of the MCC GCL commands for experienced users.

Organization

This guide is organized into the following chapters:

Chapter	Topic
Chapter 1 Script Overview	Describes an overview of the structure and usage of scripts.
Chapter 2 Advanced Topics	Describes the product, Object Manager, and the basics of SNMP.
Chapter 3 Script Syntax	Describes the syntax of scripts including date/time, string expressions, numeric expressions, manifest constants, operators, comment statements, label statements, and an example of a script.
Chapter 4 Regular Expressions	Contains information on using and building regular expressions.
Chapter 5 Script Commands	Describes the scripts available to the MCC including the syntax, description, action, parameters, and any notes associated with the script.
Chapter 6 Obsolete Material	Describes script commands and manifest constants that are no longer used.
Appendix A ASCII Character Values	Lists ASCII character values.
Appendix B Command Syntax	Lists commands and their syntax.

The MCC Documentation Set

In addition to this manual, you may need to refer to other manuals in the MCC documentation suite. These are:

- **Software Installation Guide**. Provides instructions for the initial installation and configuration of the MCC software.
- **Getting Started**. Contains an initial overview of the MCC, and its applications.
- **Operations Guide**. Contains procedures for day-to-day operation of the MCC, including selecting consoles, and managing alerts and messages.
- **Administration Guide**. Provides information on administering the MCC software and hardware.
- **Installation Preparation Guide**. Contains information on how to install MCC hardware, and prepare mainframes and servers to communicate with the MCC.
- **Troubleshooting Guide**. Provides initial troubleshooting steps to take before contacting Technical Support.

Chapter 1 Script Overview

This chapter describes:

- Script concepts
- Script structures
- Master scripts
- Reserved scripts
- Script execution
- How to monitor and control equipment
- Script organization

Overview

The Master Console Center Global Control Language (MCC GCL) is a powerful high-level programming language that can be used to automate and monitor events on systems attached to the MCC. This guide provides reference information on writing scripts in GCL.

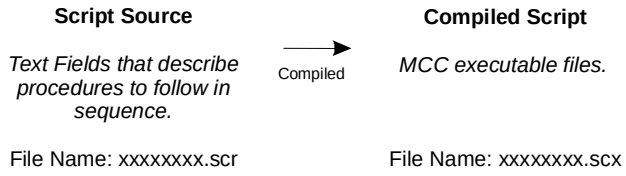
Script Concepts

The Master Console Center (MCC) can automate the monitoring, operating, and alerting of processes performed in the data center. Additionally, the MCC can monitor multiple consoles simultaneously with the same accuracy as monitoring only one console.

An operator or administrator can manually control the system IPL/boot, task/software startup, operations, and shutdown by interacting with the system console and/or OS console. All of the operations (such as entering commands, monitoring for special messages, and entering responses) are generally based on rules. These rules can be explicitly written as a detailed set of procedures. Each procedure can then be written as one or more MCC scripts in the Global Command Language (GCL). GCL is a powerful, flexible, easy-to-use programming language similar to other structured, high-level, common programming languages such as BASIC. Automation is accomplished using steps stored in scripts.

Procedures that were traditionally performed manually can be automated on the MCC. Before designing scripts for automation, create a detailed list of repetitive operations. MCC scripts can then be written to automatically perform repetitive operations.

Scripts are coded in the MCC Script Editor. After saving a script source from within the script editor, the script is automatically compiled and the user is informed of the nature and location of any errors.



A dialog box indicates if there are errors. Check the MCC Execution Log Window after saving a script to see the specific errors and the line numbers in which they occur.

> *Note*: Normally, scripts are automatically compiled when saved. It is also possible to manually compile a script using the /usr/ics/bin/gclcomp <filename>.scr command, where <filename> is the name of the script file.

A script source file format is a standard ASCII file—it can be written anywhere and copied to the MCC server when ready, for instance using FTP. The script source and executable files are stored as Unix files in the MCC system.

Script Source Structure

Source scripts can consist of several different steps. An example of the structure of a script source:

```
Script source begins.
   *Step 1:
         Turn on air conditioners, chiller, and CPU power.
   *Step 2:
         CPU IML reset.
   *Step 3:
         Load operating system.
   *Step 4:
         Boot subsystem.
Script source ends.
```

Script lines such as "*Step 1:" above are called "labels" or "label statements". Labels help make the script source easier to read, and, most importantly, are used as reference points for script logic control. Labels generally define a cohesive set of commands that are always executed together. As a reference point, a label is used for branch execution or a "go to"—an immediate transfer of script execution to the reference point. A go to can be made to any step within the current script.

Steps may include:

- Command statements, which affect the MCC processing, and
- Comment statements, which document the script and make it easier to read. Comment statements can be on separate line or on a line with the script, as shown below.

```
*Label Statement 1:
   //Comment statement
   COMMAND STATEMENT
   COMMAND STATEMENT
   COMMAND STATEMENT    // comment
   COMMAND STATEMENT
*Label Statement 2:
   //Comment statement
   COMMAND STATEMENT
   COMMAND STATEMENT
```

Naming Scripts

Each script must be given a unique name before it is saved and compiled.

> **Note**: The script name is case sensitive, so that "MYSCRIPT" and "Myscript" do not call the same script.

We recommend that you use all lower case characters for naming scripts (for example, "myscript"). The script name then has the same format as the script file stored on disk; a call to "myscript" executes a script in the file called "myscript.scx".

Master Scripts

Use "master scripts" whenever possible. A master script is a looping procedure that:

1. Reads the next message from a console.
2. Checks if the message needs processing and initiates any needed procedures or actions. (By calling another script, for instance.)
3. Begins again by reading the next new message.

For example, a master script might contain a **QREAD()** statement to retrieve the next message. Next, it includes a **SWITCH** statement to perform an initial check if the message requires processing. If it requires additional processing (that is, a **CASE** statement evaluates TRUE), the commands in the **CASE** statement of the **SWITCH** command do any additional checking of the message needed. They then call another script to process the message.

Reserved Scripts

Every MCC system includes a set of reserved scripts. Reserved scripts automatically execute when the MCC detects a status change. Customize these scripts to respond to status changes in the MCC software, and to environmental units such as digital input, digital output, sensor, and power units.

The MCC attempts to execute the appropriate reserved script when necessary. If the script does not exist, no action is taken and no error occurs.

The reserved scripts supplied with the MCC are simple, and only display informational messages—they do not initiate any corrective action. You can customize the reserved scripts to generate additional messages and/or initiate corrective action. For example, the "#snserrs.scr" script is executed when the temperature or humidity sensor limit is exceeded. This script may be modified (for example) to include instructions to start up a backup air conditioner, and to send a message to the pager of the person responsible for maintaining the air conditioners. You decide the actions and depth of automated responses.

WARNING:	Do not write a #shutdn script which takes a long time to finish. An excessively long script could cause a conflict when the MCC attempts to shut itself down. As a result, the script may not finish executing.

Status Change	Reserved Script Name Executed in Response
MCC is started (this event is not a login).	#startup.scr
MCC is shutdown (this event is not a logout).	#shutdn.scr
A log file is swapped from current to backup when full.	#logswap.scr
Temperature or humidity sensor limit is exceeded.	#snserrs.scr
Temperature or humidity sensor returns to within limits.	#snsredv.scr
Sensor unit is unreadable.	#snslost.scr
Sensor unit is recovered/readable.	#snsfind.scr
DI port #1 changes to ON.	#dier001.scr
DI port #2 changes to ON.	#dier002.scr
DI port #n changes to ON.	#diernnn.scr
DI port #n changes to OFF.	#dircnnn.scr
DI unit is unreadable.	#dilost.scr
DI unit is recovered/readable.	#difind.scr
DO unit is unreadable.	#dolost.scr
DO unit is recovered/readable.	#dofind.scr
Power unit is unreadable.	#pwrlost.scr
Power unit is recovered/readable.	#pwrfind.scr

Table 1. List of MCC Reserved Scripts executed in response to status changes

Some of these scripts are explained in the following sections.

#logswap.scr

When one of the MCC log files reaches its maximum record limit, the system:

- Deletes the current .BAK file if it exists
- Automatically renames the current log file to have a .BAK extension
- Opens a new log file

The MCC then executes the #logswap.scr script, sending it three parameters:

- **Parm 1**. A constant representing the type of backup log (*see Manifest Constants* on page 42). (A number.)
- **Parm 2**. The path of the backup log. (A string.)

- **Parm 3**. The name of the backup log. (A string.)

The default #LOGSWAP.SCR script shipped with the MCC does nothing more with the swapped log files. However, the script can be customized to respond to the log swap event with any number of actions, such as moving, renaming, or deleting the logs, or even issuing an Alert.

#startup.scr

The #startup.scr script is executed when the MCC software starts up for the first time after being shutdown. It is *not* run every time a user logs in. Actions to initiate automatically when the system first comes up (for example, fire off scripts, or log an alert) may be placed in this script.

Any Unix shell script started from this script must be stopped by the #SHUTDN.SCR script. If a shell script is stopped any other way, another copy of the shell script is started on restart of the MCC. Multiple copies of the same shell script will cause unexpected results.

#shutdn.scr

The #shutdn.scr script is run when the MCC software is fully shutdown, *not* when a single user logs out. Actions to perform at shutdown can be placed in this script. However, try to keep the number of actions to a minimum so as not to interfere with the MCC's normal shutdown procedure.

Executing Scripts

There are four ways to execute a script:

- **Manually**. A user selects a script for immediate execution.
- **Event Manager**. Exec Script is one of the available actions for a rule.
- **Script**. An executing script can initiate the execution of another script. (See *Scripts Executing Scripts* following.)
- **Reserved**. Certain status changes automatically execute reserved scripts. For example, the #startup script automatically executes at MCC startup.

The file `/usr/ics/config/gclrund.txt` can be used to configure the maximum number of concurrently executing scripts. The format of the file is a single whole number in the range of 2 to 128 inclusive. If this file does not exist, or if it contains invalid data, the value is defaulted to 35. When the configured maximum number of concurrent scripts is reached, script execution requests are queued. This script queue is a FIFO (First In First Out) queue with a maximum of 4096 scripts.

> *Note*: Reserved scripts are not subject to this limit, but are executed immediately.

Scripts Executing Scripts

An executing script can initiate the execution of another script. There are three ways to run a script from within another script, as described below:

> *Note*: Remember that script names are case-sensitive; "MYSCRIPT" is not the same as "myscript".

Calling the Function Directly

The syntax (and the "rules") for calling or executing another script from within a script is the same as for calling a built-in command. For example, the built-in **HEXSTR()** command requires one parameter—a number—and has a return value of a string. The format in a script looks like this:

```
$HexNum := HEXSTR( 15)
```

If a script is named MYSCRIPT and accepts one parameter—a number—and has a return value of a string, the format for calling MYSCRIPT from within another script looks like this:

```
$RetVal := myscript( 42)
```

Using the EXEC Command

A script may be executed from within another script with the EXEC command. The name of the script is stored in a string expression. The format of the command in a script looks like this:

```
$RetVal := EXEC(“myscript”,42)
```

For further information and examples, refer to the description of the **EXEC()** command in *Chapter 5 Script Commands*.

Using the START Command

A script may be executed from within another script for concurrent processing with the START command. The format of the command in a script looks like this:

```
$RetVal := START(“myscript”,42)
```

For further information and examples, refer to the description of the **START()** command in *Chapter 5 Script Commands*.

Note:	If the END command is used or a runtime error occurs in an executing script, every script in the “lineage chain” that has called the script will also end. RETURN may be a more appropriate command than END.

Chapter 2 Advanced Topics

This chapter contains:

- Script Writing Guidelines
- Descriptions of key MCC features, such as ports, the Object Manager, and icons.
- An overview of SNMP (Simple Network Management Protocol), and its use with the MCC.

Script Writing Guidelines

The following coding guidelines and style suggestions increase code readability and ease maintenance:

Coding Guidelines

- End all scripts with RETURN, so they can be easily called from other scripts.
- Always check return values.
- Use SCANP whenever possible, not SCANB.
- Avoid using GOTOs, except with functions such as SCANP.
- If a section of code is repeated, make it a subroutine.
- Initialize variables before use.
- Plan the script to ensure there are paths for each possible condition, and avoid "fall-through" situations. For example, if a RETURN is not specified, control of the script may fall through to the next subroutine by default, and not be processed correctly.
- Use Manifest Constants (see page 42), not values. For example, TRUE|FALSE is preferred to 0|1.
- SWITCH-ENDSWITCH statements should always have a DEFAULT: condition to handle unexpected values.

Style Suggestions

- Keep subroutines as brief and as simple as possible. If a subroutine exceeds 100 lines, consider breaking it into several subroutines.
- Precede all scripts or subroutines with a description, including pre- and post-conditions as well as exceptions and return codes that may be passed.
- Use variables in place of literals wherever possible.
- Comment at each gosub call, stating the reason for each call if it is not obvious.
- Include a commented usage statement for each subroutine if it is not obvious.
- Do not comment obvious statements, for example, "%Continue :=FALSE //don't continue".
- Try to use the same case for variable names throughout a script.
- Indent statements within control loops (IF-ENDIF, WHILE-ENDWHILE...).
- Put one space after each open parenthesis, and also after each comma. This allows use of the Ctrl-Left Arrow and Ctrl-Right Arrow keys to jump to the next word.

- To improve readability, put one blank line before and one after each control structure, such as IF-ENDIF and WHILE-ENDWHILE.
- If several assignment statements occur together, align them on the ":=" characters.
- Control structures and commands that do not return values should be all capitals, for example, IF, END, RETURN, ENDSWITCH, WHILE.
- Functions that return values should be mixed upper and lower case characters, for example, AssocKeys().
- Avoid using variables with generic names like $X or %Counter.
- Manifest constants should be all upper case characters.

MCC Concepts

Ports

Each MCC console interface has a unique number that is assigned when the system is configured. Port numbers are logical numbers assigned to each "MCC to customer equipment" interface. The **KEY()** command, which is used to type characters on a console, uses the port number to decide where to send the characters.

Each interface-type (console, DI, DO, power, sensor) has its own sequence of unique port numbers. Each interface-type command (**KEY()** command) matches only one type of interface.

Refer to the **PORT()** command description for more information about determining the logical port numbers.

Object Manager

The MCC Object Manager provides an open architecture framework that allows the user to define customized objects. Alternatively, the default objects that are inherent to the MCC may be utilized. Defining objects to specific needs allows you to manage operations more effectively.

Each object type has attributes such as type, name, and ID. The attributes of custom objects are user-defined, and are initialized in the MCC configuration files; the values are set and read with script commands. Refer to the *Administration Guide* for more information about the MCC Configuration files.

Object manager script commands all begin with "OBJ".

Object Type

Object type refers to the "template" or the definition of an object. Each object type has an object name, and is referred to by its name. The object type definition contains the information used by the object manager.

The "type" parameter for the object manager script commands is a string and is the type name.

Object Name

The object name is, obviously enough, the name of an object. In object-oriented terms, the object itself is an instance of the object type. Each object represents an instance of the object type and therefore has a unique name.

An object's name:

- Cannot contain a colon. (The colon is a delimiter.)
- Must be unique per MCC in the CPU class.
- Must be unique per CPU in the OS class.
- Must be unique per OS in the SW class.
- Must be unique per CPU in the UNIT class.

Object Key

Each object has a "hierarchical" key, referred to as the object key. The object key is a string expression specifying a precise object by referring to the chain of object names. The syntax rules for object keys state that colons should separate objects.

Only the script command **OBJID()** uses the object key. All other script commands utilize the object ID generated by that command.

An example of the full syntax of an object key is 'CPU:OS:SOFTWARE' or 'CPU:UNIT'.

Object ID

The Object ID is an integer value generated by the GCL scripting language to refer to an object. An object ID is unique to each object within individual scripts.

Advantages to implementing object IDs:

- Speeds execution time.
- Reduces script maintenance issues.
- Eases implementation.

> *Note*: Do not pass an ObjectID to different scripts as an argument. Rather, pass the object name, which is guaranteed to be unique across scripts. Generally, the MCC system assigns each script a different Object ID.

Object Field

Object fields exist for each object. Refer to the configuration management chapter of the *Administration Guide* for configuration information. There is only one intrinsic field, called 'Taskname'. It is always the left-most field displayed in the Task List (Taskman). Users can define their own additional fields and/or use some of the default MCC fields.

Refer to the OBJ family of GCL commands in *Chapter 5 Script Commands* for information on utilizing object fields. (Those commands beginning with "OBJ".)

Object Action

The status of objects can change with time. You can use object actions to wait for those changes.

> *Note*: These actions are used with the OBJEXEC command.

The following table describes object actions, descriptions of those object actions, parameters (described after the table), and the return value of each object action.

Object Action Type	Description	Parameters	Return Value
WAIT	The system waits until a field value changes on the specified object. Timeout parameter is optional (if not set, it defaults to zero) and specifies the number of seconds to wait before coming out of wait state. Zero means wait forever.	%Timeout	0 if successful. Negative on error.
WAITCHILD	The system waits until a field value changes on the child of the specified object. For example, a script on an OS can pause until a change is detected in the TaskMan fields (SW class—the child class of the OS class) for that OS. The return value is the Object ID of the object that changed (the task that had a field change).	%Timeout	Object ID of object that changed.
WAITQUEUE	The same as WAIT, except queues all change messages so as not to miss any (for sequentially processing all field change notices). For example, with WAITQUEUE, if the script receives a wakeup notice and the written script routine	%Timeout	0 if successful. Negative on error.

Object Action Type	Description	Parameters	Return Value
	takes two seconds to perform some processing, all wakeup notices that would have been missed are queued. The next entry into the WAITQUEUE then grabs the waiting change/wakeup notice. With only WAIT, those notices received during the two seconds of processing would be lost.		
WAITQUEUECHILD	The same as WAITCHILD, except with the queue functionality added as with WAITQUEUE.	%Timeout	Object ID of object that changed.

Table 2. List of Object Action Types

The only parameter available for object actions is %Timeout. This is an integer of the number of seconds to wait for a change before timing out. The value defaults to zero if not specified, which means wait forever.

Icon Class/Icon Name

A script executes on an object in a class. Objects are represented by an icon displayed on a window, such as the System Summary or CPU Configuration windows.

Class	Description
ROOM	Represents a "room". A room can hold many CPUs. The ROOM icon is on the "System Summary" window.
CPU	Represents a CPU. CPU icons are on the "System Summary" window.
OS	Represents an operating system. OS icons are on the "System Summary" window.
UNIT	Represents an I/O unit such as DASD or Tape. A UNIT icon is on the "I/O Unit" window (accessed by double-clicking a CPU icon).

Table 3. Icon Classes and Descriptions

Each icon in a class has a unique name referred to as the "Icon Name". This name is displayed on the icon, and is used in some script commands. The icon name uniquely identifies an icon in a class.

Icon classes (for example, "CPU" or "OS") are pre-defined, but icon names are user defined. Some examples of icon names are 3090, 9021, CPU #1, MVS #3, and Sys5. Icon names are case-sensitive. If an OS is defined in the system.cfg file as Sys5A, all references must be typed "Sys5A". Any other variation ("sys5a," or "SYS5A," for example) generate a run-time error.

Icon name and icon class are both optional arguments in script commands. However, if the icon name is specified, the icon class is required.

If class and name are both omitted on a relevant script command, the icon affected defaults to the icon on which the script is executing.

The default class and name for a script are determined when the script begins executing:

- A script started from another script defaults to the class and name of the calling script and, where applicable, uses parameters to override the defaults.
- The class and name of a scheduled script are set by its definition in the Event Manager.
- The class and name of an immediately executed script are set by the user in the script execution window.
- A reserved script's class and name are special, and are set by the system.

In a script executing on a lower-level class, when a higher-level icon class is specified without the icon name, the icon affected is the icon in the specified class that is an ascendant of the script's class.

> *Note*: Any class below ROOM is considered as a lower-level class.

In a script executing on a higher-level class, when a lower-level icon class is specified without the icon name, all icons in the specified class are affected.

> *Note*: Software programs or tasks are not represented by icons. A program or task is configured from the "SoftwareTask List" or "TaskMan", which can be displayed by double-clicking on the associated OS icon.

SNMP

A full discussion of SNMP (Simple Network Management Protocol) is beyond the scope of this guide. However, the following sections describe how the MCC uses SNMP.

NMS Alias

An NMS (Network Management Station) is an agent, the SNMP-enabled device that the MCC communicates with using SNMP — in other words, the device to be managed.

In the MCC, an alias must be defined to represent the agent if any of the SNMP_GET, SNMP_SET, and SNMP_GETTABLE commands are used (see pages 184 through 188). If only the SNMP_TRAPSEND command is used, an alias is optional.

The alias is defined using the "SNMP Setup" editor accessed from the Administration ⇒ Configuration ⇒ SNMP Setup menu option on the master window.

An alias in the MCC can also represent a group of agents, referred to as a group alias. This is particularly useful for the **SNMP_TRAPSEND()** command where, if the specified alias contains more than one agent (a group of agents), the same trap is sent to each agent.

MIB OID

MIB OID (Management Information Base Object ID) is an object in the MIB. MIB objects are accessed by their name.

In the MCC, the MIB OID is the name of the object to access. Both full syntax and shorthand syntax are supported. In the following example, the sysName object is referenced with both the full and shorthand syntax.

Full syntax:	1.3.6.1.2.1.1.5
Shorthand syntax:	system.5

Note	Values are often stored in the ".0" extension of an MIB OID. Thus, while sysName can be referred to as "sysName", "system.5", or "1.3.6.1.2.1.1.5", the actual value of sysName will be held in "sysName.0","system.5.0" or "1.3.6.1.2.1.1.5.0".

Chapter 3 Script Syntax

This chapter contains:

- General Scripting Syntax
- Basic Overviews of:
  - Variables
  - Date/Time
  - Expressions
  - Operators
  - Comments
  - Labels

General Script Syntax Information

General Syntax of a Script

The general structure of a script file is as follows:

```
// comments
// comments
PARMS var1, var2, ..., varN

statement
      ...
statement         // comment

RETURN value
```

Structuring a Script

When writing scripts:

- Any pre-defined MCC script item (for example, function, command, or manifest constant) is a reserved keyword.
- The script compiler is case insensitive except for string literals:
  - $MyVariable is the same as $myvariable
  - "IEF ERROR MSG" is different from "IEF Error msg"
- Each part of a multi-part statement must be on a separate line. For example, the following line is incorrect:

  ```
  IF (%A < %B) %A := %B ENDIF
  ```

  It must be split into three lines:

  ```
  IF (%A < %B)
        %A := %B
  ENDIF
  ```

- It is not required to assign return values from MCC functions or user scripts to variables. If a return value is not assigned or is not used in an expression, it simply disappears. However, ignoring return values is not good programming.
- Only one statement is allowed on each line (comments are not considered statements).
- The maximum number of characters per line is 2048.
- The number of lines per script is unlimited.
- The column starting position of each statement is optional. However, we strongly recommend following a good program indenting standard (for example, indenting steps, or IF logic).

- A script file name comprises up to eight characters followed by a period "." and a fixed three character extension:
  - "scr" for script source.
  - "scx" for script object (compiled script file).
- The script file name, without the extension, is the name used to call the script from another script.

Variables

Using and Naming Variables

- Variables may be character strings, numerics, or arrays; arrays may be strings or integers. The first character of a variable name must be a letter.
- A variable name can contain letters, numbers, and underscores.
- A variable name may not contain a dash.
- All variable names are prefixed with a special character denoting its type:

  $ string
  % integer

- The variable name's maximum length is 255 characters.
- The scope of all variables is local to the script in which they are used.
- Declaring variables is not necessary—they are allocated dynamically.
- "Deallocation" of variables is not necessary—they are automatically and properly deallocated when a script ends execution.

Character String Variables

- Character string variable names are prefixed with a dollar sign "$".
- The maximum number of characters per character string is limited only by available memory.
- Strings are automatically initialized to "" (empty string).
- String concatenation is allowed with the + operator.
- String literals can be delimited by single or double quotes. For example:

```
$VarName
$ScanText := "IEF"
$ScanText := 'IEF'
$All := $First + $Second
```

Numeric Variables

- Numeric variable names are prefixed with a percent sign "%".
- The maximum value of a numeric variable is 2^{31} -1.
- The minimum value of a numeric variable is -2^{31}.
- Numerics are automatically initialized to zero. For example:

```
%VarName
%Loop1 := 8
%Total := %Sub1 + %Sub2 + %Sub3
```

Arrays

Using and Naming Arrays

- There are two types of arrays, integer arrays and string arrays. Integer arrays are prefixed with a %. String arrays are prefixed by a $.
- The maximum number of elements per array is limited only by available memory. Array elements are dynamically allocated—no size declaration, no limits, no reallocation needed.
- The syntax for arrays is:

  ```
  ArrayName[ ArrayIndex].
  ```

- Array elements are accessed by surrounding the array index with square brackets "[ ]".
- Arrays are indexed by a numeric expression (normal array) or a character expression (associative array). The only limitation to index expressions is that the index cannot be the direct return value of a script. For example, the following statement:

  ```
  $Var := $Array[ Script()]
  ```

  must be written as

  ```
  %Index := Script()
  $Var := $Array[ %Index]
  ```

  to make clear what is the index type (array type). This assumes Script() returns a numeric value and $Array is a normal array.
- An array's indexing type (normal or associative) is determined by the first use of the array encountered by the compiler (processed from script beginning to end).
- Arrays can be copied to new arrays just like any other variable, if the destination array is new or of the same structure as the source:

  ```
  $A[1] := "Hello"
  $A[2] := "World"
  $B := $A
  // $B[1] will contain "Hello" and
  // $B[2] will contain "World"
  ```

Normal Arrays

Normal arrays are indexed by any expression that evaluates to a positive integer value.

The lowest normal array index value is 1.

Examples:

```
$Arr[ 4]
%Arr[ %ArrIndex]
$Arr[ (%Var1 + %Var2)]
```

Associative Arrays

Associative arrays are indexed by any expression that evaluates to a character string.

Examples:

```
$Arr[ "SYS5"]
%Arr[ $ArrIndex]
$Arr[ ($Var1 + $Var2)]
```

WARNING:	This is a string expression. As such, be careful of the case of the associative index—"SYS5" is different from "Sys5"! Use the "**UPPER**()" and "**LOWER**()" functions if necessary.

Date/Time

The date and time type functions allow great flexibility in manipulating and formatting of time values. All date and time values are based on the number of seconds past a certain point in time. The Date/Time function only uses two points in time:

- Seconds past epoch.
- Seconds past midnight.

Epoch Seconds

Epoch seconds is the number of seconds past the Epoch time. Epoch time is January 1, 1970 at the time of 00:00:00—the beginning of January 1, 1970. There are two ways of obtaining the epoch seconds for "now":

```
%EpochSecs := SECONDS()
%EpochSecs := DATE() + TIME()
```

Both obtain the same result.

Midnight Seconds

Midnight seconds is the number of seconds past midnight, in other words, the number of seconds that has elapsed for the day.

```
%MidSecs := TIME( "04:08:24") // %MidSecs == 14,904
```

Date and Time Literals

Just as character string literals are delimited with quotes, date/time literals are delimited by pound signs "#". All time literals are based on the 24-hour clock.

Using a date literal has the same results as using the "**DATE**()" command. Using a time literal has the same results as the "**TIME**()" command.

```
#04:00#        //means 4am
#16:00#        //means 4pm
#4/11/92#      //April 11, 1992
```

Expressions

There are two types of expressions, String and Numeric.

String Expressions

String expressions are one or more character strings used in a line of code in a script.

For example, in the following line, "$Old1 + $Old2" is a string expression of concatenating two strings together.

```
$NewString := $Old1 + $Old2
```

Numeric Expressions

Numeric expressions are plain mathematical expressions that calculate another numeric value for use in a script. Numeric expressions can only be performed on integers. For more information on numeric expressions, refer to *Variables* on page 36.

Manifest Constants

Intrinsic manifest constants have been defined to make scripts easier to read, write, and maintain. Constants are the equivalent to the value they represent and can be used anywhere in a script. For example, instead of using:

```
ALARM( 1)
```

Use the following:

```
ALARM( ON)
```

The second example is clearer because turning an alarm on is intuitive, while turning an alarm "1" has no meaning. Also, the value representing a constant may change, but the constant name always remains the same.

Although the following example is a valid numeric expression, it is not the true intention for the use of manifest constants:

```
%Num := TRUE + 1           //%Num would contain 2
```

> *Note*: Because values may change, Visara strongly recommends using the constants, NOT the values they represent.

List of Intrinsic Manifest Constants

Manifest Error Constants

Manifest Error Constants	Value	Associated String
Err_None	0	No Error
Err_BadArgs	1	Bad arguments to function
Err_ExecFailed	7	
Err_Exec_NotFound	7	Script execution failed. Script not found
Err_BadQueueId	8	
Err_Obj_InvalidId	1000	Invalid Object ID
Err_Alert_New	2001	
Err_Alert_Create	2001	ALERTCREATE failed
Err_Alert_Del	2003	ALERTDEL failed
Err_Alert_Mod	2002	ALERTMOD failed
Err_Snmp_TrapSend	3003	TrapSend failed
Err_Snmp_Get	3004	SNMP Get failed
Err_Snmp_GetNext	3005	SNMP Get Next failed
Err_Snmp_Set	3006	SNMP Set failed.
Err_Snmp_GetTable	3007	SNMP GetTable failed.
Err_Mvs_DaemonNoComm	4000	Unable to connect to gwMvsD, the MCC daemon that communicates with GW-MVS. Contact Visara Technical Support.
Err_Mvs_NoComm	4001	No GW-MVS configured for specified OS.
Err_Mvs_LostComm	4002	Lost communications to GW-MVS. Contact Visara Technical Support.
Err_Mvs_NoCmd	4010	No commands in MVSCommand() command array parameter.
Err_Mvs_CmdNotRun	4011	One or more commands

Manifest Error Constants	Value	Associated String
		not run.
Err_Mvs[1]	4012	General MVS error.
Err_Mvs_Failed[1]	4013	One or more commands failed.
Err_Mvs_NullCmd[1]	4014	Null command string found in command array parameter.
Err_Mvs_CmdTooLong[1]	4015	One or more commands too long.
Err_Mvs_BadCmd[1]	4016	Bad command string found.
Err_Mvs_FltMax[1]	4017	Too many filter elements found in filter array parameter.
Err_Mvs_NullFlt[1]	4018	Null filter string found in filter array parameter.
Err_Mvs_BadFlt[1]	4019	Bad filter string found.
Err_Mvs_InvMsgCnt[1]	4020	Invalid message count parameter.
Err_Mvs_InvWait[1]	4021	Invalid wait time parameter.
Err_Mvs_PortFailed[1]	4022	Commands sent via port failed!
Err_Mvs_PortSuccess[1]	4023	Commands sent via port successful!
Err_Mvs_InvPort[1]	4024	Invalid port found.
Err_Mvs_Timeout[1]	4025	Time-out error.
Err_QPreview_CommError	5200	
Err_QPreview_NoResponse	5201	
Err_Key_Timelock	5101	
Err_Key_Syslock	5102	
Err_Key_CommError	5103	
Err_Key_NotAccepted	5104	
Err_Key_BadFunction	5105	
Err_Key_BadLocation	5106	
Err_Key_TooMuchData	5107	
Err_Key_NumericOnly	5108	

[1] Used with TSOEREXX and MVSCOMMAND

Manifest Error Constants	Value	Associated String
Err_Key_CantReadStatus	5109	
Err_Key_ConsoleNotLocked	5110	
Err_Key_NoDaemonConnection	5111	
Err_Key_NoResponseReceived	5112	
Err_Key_BadParameter	5113	
Err_Key_LockQueueFull	5116	
ICLErr_MvsRsp_0000	10000	General MVS response message error.
ICLErr_MvsRsp_0001	10001	Storage shortage.
ICLErr_MvsRsp_0002	10002	Temporary failure.
ICLErr_MvsRsp_0003	10003	Invalid userid.
ICLErr_MvsRsp_0004	10004	Invalid password.
ICLErr_MvsRsp_0005	10005	Logon required prior to request.
ICLErr_MvsRsp_0006	10006	Unknown request.
ICLErr_MvsRsp_0007	10007	Not awaiting diagnosis.
ICLErr_MvsRsp_0008	10008	Not done with diagnosis.
ICLErr_MvsRsp_0009	10009	Missing required parameter.
ICLErr_MvsRsp_0010	10010	No session via any known Bxx node.
ICLErr_MvsRsp_0011	10011	No response data available.
ICLErr_MvsRsp_0012	10012	Userid already exists.
ICLErr_MvsRsp_0013	10013	Userid unknown.
ICLErr_MvsRsp_0014	10014	Userid database failed to open.
ICLErr_MvsRsp_0015	10015	Logic error during add.
ICLErr_MvsRsp_0016	10016	Userid has no administration authority.
ICLErr_MvsRsp_0017	10017	Userid has no administration authority.
ICLErr_MvsRsp_0018	10018	Suspended userid.
ICLErr_MvsRsp_0019	10019	Add operator rejected; administrator required.
ICLErr_MvsRsp_0020	10020	Change from administrator to operator not valid.
ICLErr_MvsRsp_0021	10021	Administrator cannot be suspended.

Manifest Error Constants	Value	Associated String
ICLErr_MvsRsp_0022	10022	Cannot delete last administrator.
ICLErr_MvsRsp_0023	10023	Authorization insufficient.
ICLErr_MvsRsp_0024	10024	Userid already logged on.
ICLErr_MvsRsp_0025	10025	Rejected virtual userid.
ICLErr_MvsRsp_0026	10026	Maximum virtual users logged on.
ICLErr_MvsRsp_0027	10027	Surrogate not logged on.
ICLErr_MvsRsp_0028	10028	Log on denied.
ICLErr_MvsRsp_0029	10029	Authorization insufficient.
ICLErr_MvsRsp_0030	10030	MVS command table exhausted.
ICLErr_MvsRsp_0031	10031	MVS command not outstanding.
ICLErr_MvsRsp_0032	10032	NMVT recording is off.
ICLErr_MvsRsp_0033	10033	Wrong release (incompatible).
ICLErr_MvsRsp_0034	10034	Site not authorized for online access.
ICLErr_MvsRsp_0099	10099	Missing or invalid parameters.
ICLErr_MvsRsp_0997	10997	File not found.

Table 4. List of Manifest Error Constants

Other Manifest Constants

Other Manifest Constants	Value	Assoc. String
On	1	
True	1	
Asc	1	
Left	1	
Off	0	
False	0	
Overwrite	0	
Both	0	
Desc	-1	
Error	-1	
Right	-1	
Append	1	
ReadOnly	2	
Room	1	
CPU	2	
Chan	3	
Unit	4	
OS	6	
SW	7	
Group	16	
All	255	
Log_CPU	2	
Log_Chan	3	
Log_Unit	4	
Log_Exec	1	
Log_Flt	8	
Log_Sys	9	
Log_Alert	10	
Log_Event	12	
Log_NewMsg	11	
Reset[2]	2	
Prn	32	

[2] Used only with QUEUE(), otherwise obsolete.

Other Manifest Constants	Value	Assoc. String
SkipEnd	2	
SkipNext	1	
Status_Error	1	
Status_Warning	4	
Status_InProcess	6	
Status_Normal	9	
Status_Down	14	
AlertState_New	0	
AlertState_Open	1	
AlertState_Closed	2	
AlertField_Status	2	
AlertField_State	4	
AlertField_Source	64	
AlertField_Msg	128	
AlertField_UserNote	256	
Alert_SendFailed	-20	
Alert_ReceiveFailed	-21	
Alert_NotFound	-22	
Alert_SyntaxError	-23	
Field_Separator	^|^	
Hmc_Activate	HMC_ACTIVATE	
Hmc_Deactivate	HMC_DEACTIVATE	
Hmc_Start	HMC_START	
Hmc_Stop	HMC_STOP	
Hmc_ResetNormal	HMC_RESETNORMAL	
Hmc_PswRestart	HMC_PSWRESTART	
Hmc_Load	HMC_LOAD	
Hmc_IpAddr	HMC_IpAddr	
Hmc_Status	HMC_Status	
Hmc_StatusAccept	HMC_StatusAccept	
Hmc_ImlMode	HMC_ImlMode	
Hmc_Id	HMC_Id	
Hmc_CpcName	HMC_CpcName	
Hmc_ImageName	HMC_ImageName	

Other Manifest Constants	Value	Assoc. String
Hmc_SnaAddr	HMC_SnaAddr	
Hmc_MachModel	HMC_MachModel	
Hmc_MachType	HMC_MachType	
Hmc_MachSerial	HMC_MachSerial	
Hmc_CSerial	HMC_CSerial	
Hmc_CId	HMC_CId	
Hmc_ActProf	HMC_ActProf	
Hmc_LastActProf	HMC_LastActProf	
Hmc_StatusError	HMC_StatusError	
Hmc_Busy	HMC_Busy	
Hmc_HasMsg	HMC_HasMsg	
Hmc_OSName	HMC_OsName	
Hmc_OSType	HMC_OsType	
Hmc_OSLevel	HMC_OsLevel	
Hmc_SysPlexName	HMC_SysplexName	
Event_Read	0	
Ptk_Type_Information	1	
Ptk_Status_Open	1	
Patrol_Events	1	
Ptk_Type_Change_Status	2	
Ptk_Status_Acknowledged	2	
Timeout	2	
Ptk_Type_Error	3	
Ptk_Status_Closed	3	
Ptk_Type_Warning	4	
Ptk_Status_Escalated	4	
Ptk_Type_Alarm	5	
Ptk_Status_Deleted	5	
Ptk_Type_Response	6	
Wait	WAIT	
WaitChild	WAITCHILD	
WaitQueue	WAITQUEUE	
WaitQueueChild	WAITQUEUECHILD	

Table 5. List of Other Manifest Constants

Operators

Mathematical

The following mathematical operators may be used in numeric expressions:

Character	Operator
+ -	Addition and Subtraction
* / %	Multiplication, Integer division, and Modulo
-	Unary Minus
:=	Assignment

Table 6. List of Mathematical Operators

Boolean

In Boolean expressions:

- Zero is interpreted as FALSE and all other values are interpreted as TRUE.
- Boolean expressions that evaluate to FALSE have a value of 0.
- Boolean expressions that evaluate to TRUE have a value of 1.

Boolean expressions consist of string and numeric expressions that evaluate to a TRUE or FALSE value. Often, a Boolean comparison operator is used, for example:

```
IF %A + %B == 4
```

This is not always the case. For example:

```
IF %A    //same as IF %A == TRUE
```

To stretch the realm of Boolean expression, consider the following example:

```
%Sum := (%A == %B) + 4

//%Sum will always be either 4 or 5
// (%A == %B) will always evaluate to true (1)
// or false (0).
```

The following Boolean syntax may be used:

Syntax	Means
==	Equals
<	Less than
>	Greater than
<= or =<	Less than or equal to
>= or =>	Greater than or equal to
!= or <>	Not equal
!	Unary not
AND	Logical “and”
OR	Logical “or”

Table 7. List of Boolean operators

Comment Statements

Comment statements are not part of a script, but describe parts of the script. Their purpose is to help others understand the logic of the script.

- Comment statements begin with two forward slashes “//” and finish at the end of the line.
- A comment may contain any character. It may be on a line by itself, or after a label or command statement.
- Blank lines are accepted and desired as comment statements for creating white space.

Examples:

```
// The entire line is a comment.
IF $MsgText == “ERROR”     //comment after a command
//statement
```

Label Statements

Label statements are used as a reference point within a script to which processing can be transferred.

- A label begins with an asterisk "*" and ends with a colon ":".
- A label cannot contain spaces.
- The maximum number of characters for the label text is 255, excluding the asterisk and colon.

For example:

```
*IPLBEGIN:
```

Chapter 4 Regular Expressions

This chapter discusses regular expressions including:

- Bracket expressions
- Special characters
- Ordinary characters
- Collating elements
- Rules for building regular expressions

Regular Expressions

Regular expressions (REs) or patterns are textual statements including specialized characters to control the search. REs provide a powerful way to search strings by specifying patterns. Matched search patterns also called substrings or a sequence (of characters).

Simple REs match a single character. More complex REs are built by concatenating simpler REs. Complex REs are classified as those that match a single character and those that match multiple characters. REs are defined recursively. For example, if you concatenate two REs, the resultant string is one RE.

A variety of utilities and languages use REs. This results in different syntax rules for REs. The MCC follows the Extended Regular Expression rules for Unix.

Important notes on the definition of regular expressions in this guide:

- The term “regular expression” in this guide includes extended regular expression rules.
- The terms “pattern” and “regular expression” can be used interchangeably.
- The term “match” describes a substring in a string that is successfully specified by a pattern or RE.

Simple Regular Expressions

```
Some Expression
```

This is a simple regular expression. It will match any text containing "Some Expression". There are no special characters contained in this example.

Bracket Expressions

```
Some[0-9]Expression
```

This expression searches for "Some" and "Expression" separated by a numeric character. Thus, "Some0Expression" and "Some6Expression" both match.

The brackets ‘[ ]’ searches "match any character contained within" and is referred to as a bracket expression. Lists of characters may be specified within brackets, as well as character ranges. For instance:

[A-Za-z] Matches any alphabetic character.

[ABC] Matches A, B, or C.

If the first character after the open bracket is a circumflex (^), the expression is reversed — the expression matches any character NOT in the brackets. Note that most of the other special characters are not special inside brackets, as in:

[^.] Match anything EXCEPT a dot.

[^A-Z] Match anything EXCEPT an upper case letter.

Subexpressions

```
Some(Other)*Expression
```

This expression example demonstrates a subexpression. The parenthesis '( )' contain a subexpression. Subexpressions are useful for two purposes:

- Specifying special characters after the regular expression.
- Some actions can be configured to allow specification of subexpression contents within the arguments to the action.

The asterisk '*' in this example searches for "zero or more occurrences" of the preceding expression. Thus, this example matches "SomeExpression", "SomeOtherExpression", "SomeOtherOtherExpression", and so on.

Regular Expressions Using Special Characters

```
Some.Expression
```

This regular expression includes a special character—the period. In regular expressions, a period matches any one character. Thus, this expression will match "Some Expression", "Some-Expression", "SomeXExpression", and so on.

```
Some.*Expression
```

This is a simple use of an asterisk. An asterisk means match zero or more occurrences of the previous item. Item examples are a character, a bracket expression, or a subexpression. This expression matches "Some" and "Expression" separated by zero or more characters—any characters. Thus, "SomeExpression", "SomexyzzyExpression" and "Some-Expression" all match.

Rules for Building Regular Expressions

When using the rules to build regular expressions (REs), first determine which substrings are matched by a pattern. Consider the following RE "c.*n". This pattern matches a substring beginning with c, ending with n, and with any number of characters between, including none. This pattern matches strings containing cn, cxn, and collision.

The following three strings are used to illustrate the discussion of REs:

String 1: "ab"
String 2: "acbcbc"
String 3: "12356"

As an introductory example, if the RE "b" is processed on each string, the results are:

String 1: match
String 2: match
String 3: no match

The RE "b", the pattern, matches the letter b in the first and second strings, and there is no b in the third string.

The RE "c" would match just the second string.

The RE "bc", built by concatenating the prior two REs, would match just the second string — there are two instances of bc in the second string.

A null pattern will match any character, so the RE "" matches all three strings.

The search for a match starts at the beginning of a string, scans from left to right, and stops when it finds the first sequence matching the pattern. If there is more than one possible leftmost match, the longest match is used. For example, in the following expression, the pattern c.*n matches the second through third characters and also the second through the fifth characters of the second string:

```
acncnc
```

The second match, being the longer in length, may be the desired match. However, a longer substring that is not the leftmost match is not the actual match.

A multi-character collating element is considered a single character that describes how to form a bracket expression. Bracket expressions are used to match a single character. However, when considering what is the longest sequence in a match involving a multi-character collating element, the element counts not as one character but as the number of characters it matches.

Pattern matching can search for case-insensitive strings. Case-insensitive processing permits matching of multi-character collating elements as well as characters. For example, the RE "[[.Ch.]]" would match ch, Ch, cH, or CH.

An RE ordinary character or an RE special character preceded by a backslash or a period matches a single character.

A bracket expression matches a single character or a single collating element.

An RE matching a single character enclosed in parentheses (a group) matches the same strings as the RE without parentheses.

Bracket Expressions

A bracket expression is a non-null string enclosed in brackets "[ ]" that matches any single character (or collating element) in the enclosed string. For example, the RE "[a3][c5]" means "look for the character 'a' OR the character '3' immediately followed by the character 'c' OR the character '5'. When processed on each of the following strings:

String 1: "ab"
String 2: "acbcbc"
String 3: "123456"
String 4: "1b2q3c4i"
String 5: "13579"

The results are:

String 1: no match
String 2: match
String 3: no match
String 4: match
String 5: match

The two contiguous bracket expressions in the pattern match the substrings 'ac', '3c', and '35' in those strings that "match".

A bracket expression is a matching list expression (searching for matching expressions) or a non-matching list expression (searching for everything except the expressions). The bracket expression consists of one or more

- Collating elements
- Collating symbols
- Equivalence classes
- Character classes
- Range expressions

Use bracket expressions to search for a matching expression or to search for everything *except* the expression.

A right bracket occurring first in the expression (after an initial circumflex "^", if any) loses its special meaning, and represents itself in the bracket expression. Otherwise, the right bracket terminates the bracket expression, unless it appears in a collating symbol (such as [.].]) or is the ending right bracket for a collating symbol, equivalence class, or character class.

Inside bracket expressions, the following are true:

- The special characters '.' , '*', '[', and '\' (period, asterisk, left bracket, and backslash) lose their special meanings within a bracket expression and become ordinary characters.
- The character sequences '[.', '[=', and '[:' (left bracket followed by a period, equal sign, or colon) are special inside a bracket expression and are used according to *Table 8. Beginning Character Sequences in Bracket Expressions*. A valid expression and the matching terminating sequence '.]', '=]', or ':]' must follow these symbols.

Beginning Character Sequence:	Used to:	Terminating Sequence:
[.	Delimit collating symbols.	.]
[=	Delimit equivalence class expressions.	=]
[:	Delimit character class expressions.	:]

Table 8. Beginning Character Sequences in Bracket Expressions

The rules in *Table 9. Bracket Expression Rules.* apply to:

- Creating and using matching and non-matching list expressions.
- Collating symbol.
- Equivalence class expression.
- Character class expression.
- Range expression in bracket expressions.

Bracket Expression Rule	Description
Matching List Expression	Specifies a list that matches any character or collating element in the list. The first character in the list cannot be a circumflex. [a3] matches either the character a or the character 3.
Non-matching List Expression	Specifies a list that matches any character or collating element except for the expressions in the list after the leading circumflex. A non-matching list expression begins with a circumflex "^". For example, [^abc] is an RE that matches any character or collating element except the characters a, b, or c. If the circumflex does not appear immediately following the left bracket, it loses its special meaning.
Collating Symbol	A collating element enclosed within bracket-period "[. .]" delimiters. Multi-character collating elements are represented as collating symbols to distinguish them from the individual characters in the collating symbol. For example, when using Spanish collation rules, [[.ch.]] is treated as an RE matching the sequence ch, while [ch] is treated as an RE matching c or h. In addition, [a-[.ch.]] matches a, b, c, and ch (see *Range Expression* later in this table). Collating symbols are valid only inside bracket expressions.
Character Class Expression	Delimiters enclosed in bracket-colon "[: :]" match any of the set of characters in the named class. Members of each of the sets are determined by the current setting of the LC_CTYPE environment variable. The supported classes are: alpha, upper, lower, digit, alnum, xdigit, space, print, punct, graph, and cntrl. Here is an example of how to specify one of these classes: "[[:lower:]]". This matches any single lowercase character for the current locale within the string.

Bracket Expression Rule	Description
Equivalence Class Expression	Specifies a set of collating elements that all sort to the same primary location. An equivalence class is enclosed in bracket-equal "[= =]" delimiters. An equivalence class generally is designed to deal with primary-secondary sorting, that is, for languages like French that define groups of characters as sorting to the same primary location and then having a tie-breaking secondary sort. For example, if x, y, and z are collating elements that belong to the same equivalence class, the bracket expressions "[[=x=]a]", "[[=y=]a]", and "[[=z=]a]" are equivalent to [xyza]. Here we use x, y, and z as variables representing characters in the same equivalence class. In a typical example, x might be the collating element **e** while y and z are the characters **è** and **é** respectively. If the collating element within [= =] delimiters does not belong to an equivalence class, the equivalence class expression is treated as a collating symbol; that is, the delimiters are ignored.
Range Expression	A set of collating elements that falls between two elements in the current collation sequence, inclusively. It is expressed as starting and ending points separated by a hyphen "-". For example, the RE "1[a-d]2", which includes the bracket expression "[a-d]" containing the range expression a through d, represents a pattern that will match any of these strings: 1a2, 1b2, 1c2, and 1d2. Range expressions depend on collating sequences. A construction such as [a-d-g] is invalid. A hyphen or right bracket may be represented as collating symbols, '[.-.]' or '[.].]', anywhere in a bracket expression. Otherwise, if both '-' and ']' are required in a bracket expression, the bracket must be first (after an optional initial ^) and the hyphen last. The hyphen character loses its special meaning in a bracket expression if it occurs first (after an initial '^', if any), last, or as an ending range point in a range expression. Examples: **[-df]** and **[df-]** are equivalent and match any of the characters d, f, or - **[^-df]** and **[^df-]** are equivalent and match any characters except d, f and - **[&-]** matches any character between & and - inclusive **[-;]** matches any characters between - and ; inclusive **[A--]** is invalid, because A follows - in the collation sequence

Table 9. Bracket Expression Rules.

Matching Multiple Characters in Bracket Expressions

The following rules are used to build multiple character-matching REs from bracket expressions (single character-matching REs).

- A concatenation of REs matches the concatenation of the strings matched by each component of the RE.
- A subexpression can be defined within an RE by enclosing it between parentheses "( )". Such subexpressions match whatever would have been matched without the parentheses.
- A concatenation of REs enclosed in parentheses matches whatever the concatenation without the parentheses matches. For example, both REs "ab" and "(ab)" match the second and third characters of the string "cabcdabc".
- An RE matching a single character or an RE enclosed in parentheses followed by the special character plus sign "+" matches what one or more consecutive occurrences of the RE would match. For example, the RE "(ab)a+" matches the second to sixth characters in the string "cabaaabc" and "c(ab)+" matches the first to seventh characters in the string "cababab".
- An RE matching a single character or an RE enclosed in parentheses followed by the special character asterisk "*" matches what zero or more consecutive occurrences of the RE would match. For example, the RE "b*c" matches the first character in the string "cabbbcde", and the RE "c*de" matches the second to sixth characters in the string "dcccdec". The REs "[cd]+" and "[cd][cd]*" are equivalent and "[cd]*" and "[cd][cd]" are equivalent when matching the string "cd".
- An RE matching a single character or an RE enclosed in parentheses followed by the special character question mark "?" matches what zero or one consecutive occurrence of the RE would match. For example, the RE "c?d" matches the third character in the string "abdbcccde".
- An RE matching a single character or an RE enclosed in parentheses, followed by an interval expression of the format "{i}", "{i,}", or "{i,j}", matches what repeated consecutive occurrences of the RE would match. Such an RE followed by:

- {i} matches exactly i occurrences of the character matched by the RE
- {i,} matches at least i occurrences of the character matched by the RE
- {i,j} matches any number of occurrences of the character matched by the RE from i to j inclusive.

  The values of i and j must be integers in the range 0 <= i <= j <= 255. Whenever a choice exists, the pattern matches as many occurrences as possible. Note that if i is zero, the interval expression is equivalent to the null RE.

For example, the RE "d{3}" matches characters eight through ten in the string "abcbcbcdddddde" and the RE "(bc){2,}" matches characters two to seven in the same string.

Alternation

If 'x' and 'y' are REs, then 'x|y' is an RE matching any string that is matched by either 'x' or 'y'. For example, the RE "((cd)|e)b" matches the string "cdb" and the string "eb". Single characters or expressions matching single characters, separated by the vertical bar and enclosed in parentheses, match a single character.

Expression Anchors—Restricting What Patterns Match

The search pattern (an entire RE) can be anchored to restrict a match:

- From the beginning of a line.
- Up to the end of the line.
- The entire line.

The following anchors restrict a search pattern:

Anchor	Restriction
"**^**"	The circumflex placed at the beginning of an expression or subexpression causes the pattern to match only a string that begins in the first character position in a string . For example, the pattern "^bc" matches bc in the string "bcdef" but does not match bc in "abcdef". The subexpression "(^bc)" also matches bcdef.
"**$**"	The dollar sign placed at the end of a pattern causes the pattern to match only if the last matched character is the last character (not including the new line character) in a string.
^pattern$	The construction ^pattern$ restricts the pattern to matching only an entire string. For example, the RE "^abcd$" matches strings containing the string "abcd", where 'a' is the first character in the string and 'd' the last.

Table 10. Regular Expression Anchors

Precedence of Special Characters

The order of precedence, from high to low, is shown below:

Order	Type	Symbols
1	Collation-related bracket symbols	[= =] [: :] [. .]
2	Escaped characters	\<special character>
3	Bracket expression	[ ]
4	Grouping	( )
5	Single-character duplication	* + ? {i,j}
6	Concatenation	
7	Anchoring	^ $
8	Alternation	|

Table 11. Regular Expression Matching, Order of Precedence

For example, the pattern 'ab|cd' is the same as '(ab)|(cd)' but is *not* equivalent to 'a(b|c)d'.

Special Characters in Regular Expressions

Table 12. Regular Expressions, Special Characters describes the various special characters that may be used in regular expressions. The characters are special — *except* in bracket expressions or following a backslash.

Character	Description
.	A period matches any single character.
\	When placed before a special character, a backslash nullifies that special character. This allows searches for a character that would otherwise be special (such as a period).
^	A circumflex matches the beginning of the line. The circumflex is special when used as an anchor or as the first character of a bracket expression.
$	A dollar sign matches the end of the line.
[ ]	Brackets match any single instance of the characters inside the brackets. Ranges can be specified, for instance, A-F. If the first character after the open bracket is a circumflex, the search is negated—matches any character not in the list. For example, G-Z if you specified A-F. If a dash is to be matched as a regular character, make it the first or last character in the list. It is not valid to use a left brace that is not part of an interval expression (of course, quoting with a backslash removes such invalidity).
( )	Parentheses form a subexpression. Up to nine subexpressions are allowed in a regular expression; they may be nested. Outside a bracket expression, do not use a left parenthesis unless it is preceded with a backslash and quoted "\(". To search for the string "()", use the quoted form by preceding with a backslash "\()".
*	[1] An asterisk matches zero or more occurrences of the previous character, bracket, or subexpression—depending on what precedes the asterisk.
+	[1] A plus sign matches one or more occurrences of the previous character, bracket, or subexpression—depending on what precedes the plus sign.
?	[1] A question mark matches exactly zero or one occurrences of the previous character, bracket, or subexpression—depending on what precedes the question mark.
{n}	[1] Brackets match exactly n occurrences of the previous character, bracket, or subexpression—depending on what precedes the {n}.
{n,m}	{n,} matches n or more. {,m} matches m or less. {n,m} matches n to m occurrences. Replace n and m with numbers specifying

Character	Description
	the quantity.
|	OR. "A|B" matches either "A" or "B". OR is very powerful when combined with subexpressions. The vertical line is a special character. It is not valid to use a vertical line: - First or last in an RE. - Immediately following another vertical line. - Immediately following a left parenthesis. - Immediately preceding a right parenthesis.

Table 12. Regular Expressions, Special Characters

[1] Outside of a bracket expression, it is *not valid* to use the '*', '+', '?', or '{':

- As the first character in an RE.
- Immediately following a vertical line.
- Immediately following a circumflex.
- Immediately following a left parenthesis.

Ordinary Characters

Any character, except for RE special characters (such as the period, question mark, or asterisk), is an ordinary character and is an RE that matches itself.

Collating Elements

The concept of a character is generalized to the concept of a collating element. For many purposes, especially in English-speaking locales, the term "collating element" may be considered synonymous with "character". In REs, collating elements are relevant to bracket expressions. For more information about bracket expressions, see *Bracket Expressions* on page 54.

A collating element is the smallest unit used to determine how to order characters. Collating elements are necessary for languages that treat some strings as individual collating elements. For example, in Spanish, the strings "ch" and "ll" each are collating symbols (that is, the Spanish primary sort order is a, b, c, ch, d,...,k, l, ll, m,...) and are considered one character.

Chapter 5 Script Commands

This chapter:

- Discusses the types of script commands
- Gives information about script syntax
- Lists the script commands available with their specific syntax and parameters.

Conventions in this Chapter

When writing scripts, text placed in brackets "[]" is an optional parameter or argument. Ellipsis "..." represent a multi-line command statement. Each ellipsis in the command syntax represents the line break, splitting the command onto multiple lines.

Script Command Types

Refer to *Appendix B Command Syntax* for grouping of script commands.

AICONNAMES

Syntax: AICONNAMES($AssocArray, %Class, $ParentIcon**)**

Description: Fill an array with all of the icon names in a class.

Action: An associative string array is created with its index keys being the names of the icons in the specified class. The set of icons obtained for the specified class is determined by the ParentIcon parameter. Not all icons in the specified class are obtained, only the icons whose parent icon name is ParentIcon.

Parameters: **$AssocArray**. Associative string array. The associative array to receive the icon names as its keys. Each icon name becomes an index key in the array. The data for each element remains at the initialization state—empty string “”.

%Class. Numeric expression. The icon class. Refer to *Icon Class/Icon Name* on page 29 for more information. Valid classes are CPU, OS, and UNIT.

$ParentIcon. String expression. The name of the parent icon to the class of icons. Refer to *Icon Class/Icon Name* on page 29 for more information.

Returns: N/A.

Notes:

1. Refer to *Manifest Constants* on page 42 for the constants reference list.
2. Refer to the **ASSOCKEYS()** command if it is necessary to convert the array keys to normal array data values.
3. For class of CPU, the ParentIcon parameter is ignored.
4. This command is not designed to get names from the SW level. To obtain the names at the SW level, refer to the **OBJIDARRAY()** command.

Example:

```
//place the names of all CPUs in the room called “Mpls”
//as keys in the $Icons associative string array.
AICONNAMES( $Icons, CPU, “Mpls”)
```

See Also: ASSOCKEYS, ICONNAME

ALARM

Syntax: ALARM(%Operation)

Description: Generates a repetitive alarm tone (beep) at the MCC terminal.

Action: The tone beeps once per second and continues until turned off with the ALARM(OFF) command.

Parameters: **%Operation**. Numeric expression. ON activates the tone and OFF deactivates the tone.

Returns: N/A.

Notes:
1. Refer to *Manifest Constants* on page 42 for information on the constants reference list.
2. Once activated, only the ALARM(OFF) command deactivates the beeping tone.

Example:
```
ALARM( ON)          //turn the alarm on
WAITFOR( 30)        //sound the alarm for 30 seconds
ALARM( OFF)         //turn the alarm off
```

See Also: DOUNIT

ALEN

Syntax: ALEN(Array) ==> %Elements

Description: Returns the number of elements in the array.

Action: The number of elements in the specified array is returned.

Parameters: **Array**. Array variable. The array to count the number of elements.

Returns: Numeric value. The number of elements in the array.

Notes:

Example:
```
%Num := ALEN( $Lpars)
```

See Also: LEN

ALERTCREATE

Syntax: ALERTCREATE(%Status, %State, $Source, $MsgText, $UserNote) ==> %AlertID

Description: Create a new Alert.

Action: A new Alert is created in the Alert Window, using the specified arguments.

Parameters: **%Status**. Numeric expression. The status number to set the alert to, which effectively sets its color. The range of status numbers is 1 to 16. For a list of the status constants, numbers, and colors, refer to the description of the *ICON()* command on page 130.

%State. Numeric expression. The current state of the alert. An alert is New, Open, or Closed. Use the following constants for clarity:

Manifest Constant	Value
ALERTSTATE_NEW	0
ALERTSTATE_OPEN	1
ALERTSTATE_CLOSED	2
ALERT_RECEIVEDFAILED	-21

$Source. String expression. Name of the source of the alert. In general, use the OS name of the system or device. Limited to 11 characters.

$MsgText. String expression. The alert text displayed in the Message field of the Alert.

$UserNote. String expression. The text displayed in the User notes field of the Alert.

Returns: The unique ID number of the Alert. The alert ID is used to update the alert.

Notes: Do not use $MsgText or $UserNotes strings longer than 450 characters. This can have unpredictable results.

Examples:

```
%AlertID := ALERTCREATE(STATUS_ERROR, ALERTSTATE_NEW,
$OSName, “JES2 ABEND”, “Script is performing auto-
recovery”)
%AlertID := ALERTCREATE(STATUS_WARNING, ALERTSTATE_NEW,
$OSName, “Login Failure”, “”)
```

See Also:

ALERTDEL

Syntax: ALERTDEL(%AlertID) ==> %ErrCode

Description: Deletes an existing Alert.

Action: The alert specified by %AlertID is deleted (removed) from the Alert Window.

Parameters: **%AlertID**. Numeric expression. The unique alert ID of the alert to be deleted.

Returns: Numeric value. Possible values are listed below:

Manifest Constant	Value	Associated String
Err_None	0	No Error
ERROR	-1	Could not communicate with Alert Manager
ALERT_NOTFOUND	-22	Specified Alert not found
ALERT_SENDFAILED	-20	The socket write failed
ALERT_RECEIVEDFAILED	-21	

Notes: None

Example:

```
// Create a bogus alert and delete it right away
%AlertID := ALERTCREATE(STATUS_ERROR, ALERTSTATE_NEW,
$OSName, “This is a test alert”, “Just testing”)
%ErrCode := ALERTDEL(%AlertID)
```

See Also: ALERTMOD, ALERTDEL, ALERTGETACTIVE

ALERTGETACTIVE

Syntax: ALERTGETACTIVE($AssocArray) ==>%ErrCode

Description: Retrieves information on all active alerts into an array.

Action: Information on all active alerts is put into the $AssocArray, with each alert being a key in the array.

Parameters: $AssocArray. Associative array. The array into which to put the alert information.

Returns: A manifest constant indicating the status of the operation, as follows:

Manifest Error Constant	Value	Associated Information
Err_None	0	No error, array has been filled with appropriate values.
Err_Alert_GetActive	2004	Error communicating with the alert manager daemon. No values set.

Notes: The ALERTGETACTIVE routine returns an array of values that come in pairs (name and value) as shown below:

Variable	Sample value	Description
$AlertInfo[1]	seq	
$AlertInfo[2]	000001	unique sequence number
$AlertInfo[3]	status	
$AlertInfo[4]	002	color (1-16)
$AlertInfo[5]	state	
$AlertInfo[6]	New	New, Open, Close
$AlertInfo[7]	createTime	
$AlertInfo[8]	10/09/2002_12:08:38	when created
$AlertInfo[9]	openTime	
$AlertInfo[10]	10/09/2002_12:08:38	when opened
$AlertInfo[11]	closeTime	
$AlertInfo[12]	10/09/2002_12:08:38	when closed
$AlertInfo[13]	devName	

$AlertInfo[14]	'Quilt'	source computer/OS
$AlertInfo[15]	message	
$AlertInfo[16]	'This\sis\sthe\smessage.'	message text Note: Spaces are represented by '\s'.
$AlertInfo[17]	userNote	
$AlertInfo[18]	'This\sis\sthe\user\snote.'	user note, if any
$AlertInfo[19]	whoLastModified	

Examples: Simple Script to log all of the alerts to the filtered message log.

```
%rc := ALERTGETACTIVE($AssocArray)
IF %rc == 0
%Num := ALEN($AssocArray) // Get the number of alerts
ASSOCKEYS( $AssocArray, $NormalArray) // Put into a normal
array
%Index := 1
WHILE %Index <= %Num // Log individual alert info
$Seq := $NormalArray[%Index]
$AlertInfo := $AssocArray[$Seq]
LOG(LOG_FLT, "Alert info is " + $AlertInfo)
INC %Index
ENDWHILE
ENDIF
```

Simple Script to break down all of the component parts of the returned alert array and log everything to the filtered message log.

```
%rc := ALERTGETACTIVE( $Array )
LOG( LOG_FLT, "There are currently " + ALEN($Array) + \ "
active alerts." , 5)
%ArrayIndex := 1
WHILE (%ArrayIndex <= ALEN($Array))
SPLIT($AlertInfo, $Array[STR(%ArrayIndex)], " ")
GOSUB *PROCESS_ALERT_INFO
%ArrayIndex := %ArrayIndex + 1
ENDWHILE
RETURN
*PROCESS_ALERT_INFO:
LOG(LOG_FLT, "Alert #: " + STR(%ArrayIndex), 1)
LOG(LOG_FLT, $AlertInfo[1] + " : " + $AlertInfo[2], 5)
LOG(LOG_FLT, $AlertInfo[3] + " : " + $AlertInfo[4], 5)
LOG(LOG_FLT, $AlertInfo[5] + " : " + $AlertInfo[6], 5)
LOG(LOG_FLT, $AlertInfo[7] + " : " + $AlertInfo[8], 5)
LOG(LOG_FLT, $AlertInfo[9] + " : " + $AlertInfo[10], 5)
LOG(LOG_FLT, $AlertInfo[11] + " : " + $AlertInfo[12], 5)
LOG(LOG_FLT, $AlertInfo[13] + " : " + $AlertInfo[14], 5)
LOG(LOG_FLT, $AlertInfo[15] + " : " + $AlertInfo[16], 5)
LOG(LOG_FLT, $AlertInfo[17] + " : " + $AlertInfo[18], 5)
LOG(LOG_FLT, $AlertInfo[19] + " : " + $AlertInfo[20], 5)
LOG(LOG_FLT, "----------------------------------------",
1)
RETURN
```

See Also: ALERTMOD, ALERTCREATE, ALERTDEL

ALERTMOD

Syntax: ALERTMOD(%AlertID, %AlertField, NewValue) ==> %ErrCode

Description: Modifies the value of a field in an existing Alert.

Action: In the alert specified by %AlertID, the value of the field specified by %AlertField is changed to the new value specified by NewValue.

Parameters: **%AlertID**. Numeric expression. The unique alert ID of the alert to be modified.

%AlertField. Numeric expression. The number representing the field to change the value of. Use the following manifest constants to reference the fields:

Constant	Value
ALERTFIELD_STATUS	2
ALERTFIELD_STATE	4
ALERTFIELD_SOURCE	64
ALERTFIELD_MSG	128
ALERTFIELD_USERNOTE	256

NewValue. String or numeric expression. The new value to set the field to. Be sure to use the correct data type for the field to be modified. Using the incorrect data type (for example, changing ALERTFIELD_STATE to a string) can have unpredictable results.

Returns: Numeric value, as follows:

Value	Status
0	SUCCESS
-22	%AlertID is invalid, or the ALERTMOD( ) failed

Notes: ALERTFIELD_MSG and ALERTFIELD_USERNOTE have a maximum length of 450 characters. Attempting to put longer strings into those fields will cause ALERTMOD( ) to fail.

Example:

```
%ErrCode := ALERTMOD(%AlertID, ALERTFIELD_STATE,
ALERTSTATE_CLOSED)
%ErrCode := ALERTMOD(%AlertID, ALERTFIELD_USERNOTE,
“Automatic correction procedures have failed.”)
```

See Also: ALERTCREATE, ALERTDEL, ALERTGETACTIVE

ARESET

Syntax: ARESET(Array)

Description: Reset the contents of an array to “empty”.

Action: All elements of Array are discarded. Array returns to its state before first use (its initialization state)—empty.

Parameters: **Array**. Array variable. The array to reset to “nothing”.

Returns: N/A.

Notes: ALEN(Array) equates to zero after **ARESET**() operates on Array

Example:

```
$Arr[ 4] := “Hello”
$Arr[ 5] := “There”
%Size := ALEN( $Arr)    // 2
$Var := $Arr[ 5]        // “There”
ARESET( $Arr)
%Size := ALEN( $Arr)    // 0
$Var := $Arr[ 5]        // “”
```

See Also: N/A

ASCII

Syntax: ASCII($String) ==> %Value

Description: Given a string $String, returns the integer ASCII value of its first character only, ranging from 0—255.

Action: Assign an integer value to the variable specified.

Parameters: **$String**. Normal string expression.

Returns: Integer value. The number corresponding to the ASCII code of the first character.

Notes:

1. A return value of zero denotes an empty (null) string.
2. Technically, ASCII values are only in the range of 0 to 128, but some systems (including the MCC itself) use the ISO 8859 "Latin 1" or ANSI character set, which contains values from 0 to 255. See *Appendix A ASCII Character Values* for a complete list of characters and their values.

Example:

```
$Name := "Alexandra"
%Code := ASCII($Name)
// Variable %Code will contain the number 65, the ASCII
// value for a capital A.
```

See Also: CHR

ASCRN

Syntax: ASCRN($Array, %Port)

Description: Fill an array with the full text of a console screen.

Action: Each row (or line) on the console screen becomes an element in $Array.

Parameters: **$Array**. Normal string array. The array to populate with the screen rows of text. Each array element will contain one row of text—element one will hold the text from screen row one, element two will hold the text from screen row two, and so on.

%Port. Numeric expression. The assigned OS port number for the console. Refer to *Ports* on page 24 for more information.

Returns: N/A.

Notes: **ASCRN()** is not for capturing the printer console text. Use the **QREAD()** command for reading the printer console instead.

Example:

```
// Sample ASCRN call
%PortID := PORT( OS, “System3”)
ASCRN( $ScreenContents, %PortID)
```

See Also: SCRIPTCANCEL

ASORT

Syntax: ASORT(NormArray, %Direction)

Description: Sort a normal array.

Action: The data elements in the specified array are sorted, in ascending or descending order, as specified by the Direction parameter.

Parameters: **NormArray**. Normal array variable. The normal array to sort. Can be a string or integer array.

Direction. Integer expression. Indicates the order in which to sort the data—ascending or descending. Valid constants are ASC for ascending and DESC for descending.

Returns: N/A.

Notes: The array is sorted "in-place". The order of the original array is lost.

Example:

```
ASSOCKEYS( $AssocArray, $Keys)
ASORT( $Keys, ASC)
// now we can traverse the associative array in a
// consistent order
```

See Also: ASSOCKEYS

ASSOCKEYS

Syntax: ASSOCKEYS(AssocArray, $NormArray) ==> %RetVal

Description: Populate a normal string array with the string index keys of an associative array.

Action: Each key of the associative array is placed in an element of the string normal array. Each element in the string normal array will contain one of the associative array's keys.

Parameters: **AssocArray**. Associative array variable. The associative array from which to get the keys. May be either a string or an integer array.

$NormArray. Normal array variable. The array to populate with the associative array's keys.

Returns: Integer. The number of items in the array. This value can be used for loops or other structures to ensure the entire array is populated.

Notes: The keys are not retrieved in any guaranteed order. However, if the associative array is not changed between two instances of ASSOCKEYS, ASSOCKEYS generates identical results. If a sorted order is required, use the **ASORT()** command.

Example:

```
// Retrieve list of all CPU names from a room
$RoomName := ICONNAME()
AICONNAMES( $ArrCPU, CPU, $RoomName)
%Number := ASSOCKEYS( $ArrCPU, $CPUList)
%Index := 1
WHILE %Index <= %Number
          $CPUName := $CPUList[ %Index]
          //additional processing
          %Index := %Index + 1
ENDWHILE
```

See Also: ASORT

ATSTR

Syntax: ATSTR($String, $Substring) ==> %StartPos

Description: Returns the starting position of a substring within a string.

Action: String is searched using the regular expression pattern Substring. The starting position of the matching text is returned.

Parameters: **$String**. String expression. The string to search for the pattern in.

$Substring. Regular expression. The regular expression pattern to search for in String. Refer to *Expressions* on page 41 for more information.

Returns: Numeric value, as follows:

Value	Meaning
0	No match found
Any other value	The starting position of the first instance of the matching text within String

Notes: N/A

Example:

```
$String := “The quick brown fox.”
$Substring := “brown”
%StringPosition := ATSTR ( $String, $Substring )
// %StringPosition == 11
```

See Also: FINDSTR, LEFTSTR, RIGHTSTR, REPSTR, SUBSTR, LEFTSTR

BASEDIRECTORY

Syntax: BASEDIRECTORY() ==> $DirectoryString

Description: Obtains the base directory for the product.

Action: Return a string giving the path of the base directory being used by the product.

Parameters: none

Returns: String value giving the base directory path (e.g. "/usr/ics")

Example:

```
$Dir := BASEDIRECTORY()
LOG( LOG_FLT, "Base directory is " + $Dir", 9)
```

See Also:

BLOCKSCAN

Syntax: BLOCKSCAN(%Wait, *Timeout [, $Array]) ...ENDBLOCK

Description: Enables up to 256 **SCANB()** commands to execute as a group.

Action: BLOCKSCAN delineates the beginning and ENDBLOCK delineates the ending of a group of **SCANB**() commands to execute as a group. The **SCANB**() commands are executed simultaneously and script execution continues with the branching logic of the first **SCANB**() command that fulfills its own scanning condition. If the Wait time expires, script execution branches to the label specified by the *Timeout parameter.

Parameters: **%Wait**. Numeric expression. The number of seconds to wait before timing out when none of the **SCANB**() commands are successful. This number specifies how long the command waits for the scans to be successful.

***Timeout**. Label literal. The label to jump to when the Wait time expires.

$Array. Normal string array. Optional. The array to populate with subexpression results, if any, from the Text parameter in a **SCANB**() command. Each array element will contain one subexpression result—element one will hold the result for subexpression one, element two will hold the result for subexpression two, and so on. The subexpressions are "numbered" from left to right in the Text parameter. $Array is only populated when the Text parameter contains subexpressions and the scan text is found.
Only the first nine subexpression results can be returned.

Returns: N/A.

Notes: Only **SCANB**() commands are allowed in the **BLOCKSCAN**() statement.

Example:

```
//*********************************************
// Example 1
//*********************************************
*START:
      BLOCKSCAN(1800, *START, $Msg)
            SCANB( 1, "JOBA END", *JOBA)
            SCANB( 2, "JOBB END", *JOBB)
ENDBLOCK
//*********************************************
// Example 2
//*********************************************
*START:
      LOG( LOG_FLT, "JOB A–C REPLY SCAN")
      WAITFOR( 10)
      BLOCKSCAN(20, *START, $Msg)
      //put 2 digit job # in first cell of $Msg array
            SCANB( 2, "\*[0-9][0-9] JOB-A", *JOBA)
            SCANB( 2, "\*[0-9][0-9] JOB-B", *JOBB)
            SCANB( 2, "\*[0-9][0-9] JOB-C", *JOBC)
      ENDBLOCK
*JOBA:
      //send job # in pp with reply
      KEY( 2, "R " + $Msg[1] + ",CANCEL[ENT]")
```

See Also: SCANB, SCANP

CHR

Syntax: CHR(%Number) ==> $String

Description: Given an integer in the range 0 to 255, returns a string with the single character corresponding to that ASCII value.

Action: Assign a string value to the variable specified.

Parameters: **%Number**. A numeric value from 0 to 255.

Returns: String value, as follows:

String Value	Meaning
Empty	The ASCII value specified was 0, or outside the range 0 to 255
Single character specified	The character represented by the ASCII value

Notes: Technically, ASCII values are only in the range of 0 to 128, but some systems (including the MCC) use the ISO 8859 "Latin 1" or ANSI character set, which contains values from 0 to 255. See *Appendix A ASCII Character Values* for a complete list of characters and their values.

Example:

```
%Num := 66
$Character := CHR(%Num)
// The variable $Character will contain the letter B.
```

See Also: ASCII

CLASSNAME

Syntax: CLASSNAME([%Class]) ==> $ClassName

Description: Returns the class name for a class number.

Action: The class name for a class number is returned as a string.

Parameters: **%Class**. Numeric expression. Optional. The icon class. Refer to *Icon Class/Icon Name* on page 29 for more information.

If the class number is not specified, the current script class is used.

Returns: String value. The icon class manifest constant name as a character string.

Notes:

1. Refer to *Manifest Constants* on page *42* for the constants reference list.
2. This function is useful for display/log purposes.
3. CLASSNAME() is the inverse of CLASSNUM().
4. This function is not commonly used.

Example:

```
$Class := CLASSNAME()
```

See Also: CLASSNUM, ICONNAME

CLASSNUM

Syntax: CLASSNUM([$ClassName]) ==> %Class

Description: Returns the class number for a class name string.

Action: The class number for a class name string is returned.

Parameters: **$ClassName**. String expression. Optional. A valid class name in a string format (the icon class manifest constant name as a character string). Refer to *Icon Class/Icon Name* on page 29 for more information.

If the class name string is not specified, the current script class is used.

Returns: Numeric value. The icon class number.

Notes:
1. CLASSNUM() is the inverse of CLASSNAME().
2. This function is not commonly used.

Example:
```
%Class := CLASSNUM()
```

See Also: CLASSNAME, ICONNAME

CPUPOWER

Syntax: CPUPOWER(%Port, %Operation)

Description: Switches a CPU's power ON or OFF.

Action: The power to the CPU that is connected to the specified MCC power unit port is turned ON or OFF.

Parameters: **%Port**. Numeric expression. The assigned power port number to which the CPU is connected. Refer to *Ports* on page 24 for more information.

%Operation. Numeric expression. ON switches on the power and OFF switches off the power.

Returns: N/A.

Notes:
1. Refer to *Manifest Constants* on page 42 for the constants reference list.
2. When powering off, first ensure that the operating systems are shut down.
3. This function is not commonly used.

Example:
```
%Operation := OFF
CPUPOWER( 1, ON)
CPUPOWER( 1, %Operation)
```

See Also: DIUNIT, DOUNIT, HUMID, TEMP

DATE

Syntax: DATE([$DateString]) ==> %EpochSeconds

Description: Converts a date string to a date value.

Action: The date string is converted to epoch seconds. Epoch seconds can be easily used in time calculations.

Parameters: **$DateString**. String expression. Optional. The date string to convert. Must be in the "MM/DD/YY" format. "MM" represents the month number, "DD" represents the day number, and "YY"" represents the year. "/" is used as the separator. If not specified, the current system date is used. The year must be in the range 1/1/70 (1970) to 12/31/37 (2037).

Returns: Numeric value. The date expressed as the number of seconds past the epoch at the beginning of the specified date (the time of 00:00:00).

Notes: Refer also to *Date/Time* on page 40.

Example:
```
%Date := DATE( "08/22/98")
```

See Also: SECONDS, TIME, TIMESTR, VERSION, WAITFOR, WAITUNTIL

DEC

Syntax: DEC %Variable

Description: Subtracts one from a numeric variable's value.

Action: The value in the specified variable is reduced by one.

Parameters: **%Variable**. Numeric variable. Name of the integer variable to decrement.

Returns: N/A.

Notes: The **DEC()** command is a shortcut for subtracting one from itself. The following two statements are equivalent, but the first one executes many times faster:

```
DEC %Var
%Var := %Var–1
```

Example:

```
*START:
      %NumEx := 9                    //init to 9
*COUNTING:
      REPEAT
            //display value to filtered msgs window
            LOG( LOG_FLT, "COUNTER=" + STR( %NumEx))
            DEC %NumEx                     //subtract 1
      UNTIL %NumEx < 1
```

See Also: INC, SET

DECODE

Syntax: DECODE ($String[, $Key]) ==> $Result

Description: Decodes a GCL string, for example, a password.

Action: The specified string is decoded using the specified key, if any. If no key is specified, a default key is used. The decoded output string has the same length as the input string.

Parameters: **$String**. String expression. The input string to decode.

$Key. String expression. Optional. The decoding key to use.

Returns: **$Result**. String expression. The decoded output string.

Notes: The input string must contain only printable ASCII characters, from space to ~. If it contains unprintable characters (for example, control characters), **$Result** contains an empty string and an error message is logged in icsexec.log.

Example:

```
$a := "kdfjdkfj dkf@#%"
$c := decode($b)
```

See Also: ENCODE

DIUNIT

Syntax: DIUNIT(%Port) ==> %Status

Description: Check the status of a device connected to a DI unit.

Action: Set the value of the specified variable to the status of the DI unit connected to the specified port.

Parameters: **%Port.** Numeric expression. The assigned DI port number to which the device is connected. Refer to *Ports* on page 24 for more information.

Returns: Numeric value, the value of the DI status.

Value	Meaning
0	OFF
1	ON

> *Note*: Depending on the type of connected device, the values can have other meanings.

Notes:

1. Refer to *Manifest Constants* on page 42 for the constants reference list.
2. This function is not commonly used.

Example:

```
%Status := DIUNIT( 3)
IF %Status == ON
      LOG( LOG_FLT, “DI #3 is on”)
ENDIF
```

See Also: DOUNIT

DOUNIT

Syntax: DOUNIT(%Port[, %Operation]) ==> %DOSwitch

Description: Controls the device connected to a DO unit.

Action: The switch on the specified DO unit port is opened or closed, thereby changing the status of the connected equipment. This effectively switches the equipment ON and OFF.

Parameters: **%Port**. Numeric expression. The assigned power port number to which the CPU is connected. Refer to *Ports* on page 24 for more information.

%Operation. Numeric expression. Optional. ON closes the switch and OFF opens the switch. If not specified, the DO switch setting remains the same.

Returns: Numeric value, the setting of the DO switch.

Value	Meaning
0	OFF
1	ON

If the Operation parameter is specified, the return value is the old DO switch setting value.

If the Operation parameter is not specified, the return value is the current DO switch setting value.

> *Note*: This is not the current status of the device connected to the DO unit—use DIUNIT to retrieve that value.

Notes:

1. Refer to *Manifest Constants* on page 42 for the constants reference list.
2. This function is not commonly used.

Example:

```
DOUNIT( 4, ON)
%DOSwitch := DOUNIT( 4, ON)
%DOSwitch := DOUNIT( 4)
```

See Also: DIUNIT

ENCODE

Syntax: ENCODE ($String[,$Key]) ==> $Result

Description: Encodes a GCL string, for example, a password.

Action: The specified string is encoded using the specified key, if any. If no key is specified, a default key is used. The encoded output string has the same length as the input string.

Parameters: **$String**. String expression. The input string to encode.

$Key. String expression. Optional. The encoding key to use.

Returns: **$Result**. String expression. The encoded output string.

Notes: The input string must contain only printable ASCII characters, from space to ~. If it contains unprintable characters (for example, control characters), **$Result** contains an empty string and an error message is logged in icsexec.log.

Example:

```
$a := "kdfjdkfj dkf@#%"
$b := encode($a)
```

See Also: DECODE

END

Syntax: END

Description: Ends the execution of the script thread.

Action: The execution of the current script thread stops. A script thread begins immediately after the first "execution" point—from Event Manager, manually, or the START command. The script containing the START command is not part of the thread.

Parameters: N/A.

Returns: N/A.

Notes: N/A.

Example:

```
//Example showing a script end point
...
LOG(LOG_FLT, "This subscript has finished its task.")
END
```

See Also: RETURN

ERRORMSG

Syntax: ERRORMSG(%ErrNum)==> $ErrMsg

Description: Returns the error message associated with the error number.

Action: The error message for the error number specified by ErrNum is returned.

Parameters: %**ErrNum**. Numeric expression. The error number retrieved from the **ERRORNUM()** command.

Returns: String value. The error message associated with the error number.

Notes: See **ERRORNUM()** for more details.

Example:
```
%ErrCode := ERRORNUM()
$ErrMsg := ERRORMSG( %ErrCode)
LOG( LOG_FLT, “Error: ” + $ErrMsg)
```

See Also: ERRORNUM

ERRORNUM

Syntax: **ERRORNUM() ==> %ErrNum**

Description: Returns the error number for the most recent command.

Action: The error number from the most recently executed command is retrieved.

Parameters: N/A

Returns: Numeric value. The error code from the most recent command, as listed in *Possible Error Codes* below.

Notes:
1. The next successful command clears all error codes.
2. See ERRORMSG for information on receiving error message text.
3. The scope of the error number is per script thread. This means the last error number set in a child script (a called script) is accessible when returning to the parent script (the calling script).

Example:
```
%ErrNum := ERRORNUM()
IF %ErrNum == Err_Alert_Mod
      LOG( LOG_FLT, "Modify alert failed", STATUS_ERROR)
ENDIF
```

See Also: ERRORMSG

Possible Error Codes

Manifest Constant	Value	Associated String/Reason
Err_None	0	No Error
Err_BadArgs	1	Bad arguments to function
Err_Exec_NotFound	7	Script execution failed: Script not found
Err_Obj_InvalidId	1000	Invalid Object ID
Err_Alert_Create	2001	ALERTCREATE failed
Err_Alert_Del	2003	ALERTDEL failed
Err_Alert_Mod	2002	ALERTMOD failed
Err_Snmp_TrapSend	3003	TrapSend failed
Err_Snmp_Get	3004	SNMP Get failed
Err_Snmp_GetNext	3005	SNMP Get Next failed
Err_Snmp_Set	3006	SNMP Set failed.
Err_Snmp_GetTable	3007	SNMP GetTable failed.
Err_Mvs_Daemon_NoComm	4000	Unable to connect to gwMvsD, the MCC daemon that communicates with GW-MVS. Contact Visara Technical

Manifest Constant	Value	Associated String/Reason
		Support.
Err_Mvs_NoComm	4001	Unable to communicate with GW-MVS. This may be because no GW-MVS is configured for the specified OS, or it is not running, or because of an LU62/SNA program.
Err_Mvs_LostComm	4002	Lost communications to GW-MVS or the local gwMvsD. Some commands may have completed successfully. Contact Visara Technical Support.
Err_Mvs_NoCmd	4010	No commands found in MVSCommand() command array parameter.
Err_Mvs_CmdNotRun	4011	One or more commands not run. This may occur if TSOERREXX is passed more commands than it can process. Some commands may be discarded.
ICLErr_Mvs	4012	General MVS error.
ICLErr_Mvs_Failed	4013	One or more commands failed on the target OS. There may be one or more ICLErr_MvsRsp messages in the response array.
ICLErr_Mvs_NullCmd	4014	Empty command string found in command array parameter.
ICLErr_Mvs_CmdTooLong	4015	One or more commands too long. The maximum command length is 79 characters.
ICLErr_Mvs_BadCmd	4016	Bad command string found. One of the command strings in the command array may contain an invalid character.
ICLErr_Mvs_FltMax	4017	Too many filter elements found in filter array parameter. A maximum of three filter elements are permitted.
ICLErr_Mvs_NullFlt	4018	Null filter string found in filter array parameter.
ICLErr_Mvs_BadFlt	4019	Bad filter string found. One of the filter strings may contain an invalid character.

Manifest Constant	Value	Associated String/Reason
ICLErr_Mvs_InvMsgCnt	4020	Invalid message count parameter. The value must be between 0 and 999.
ICLErr_Mvs_InvWait	4021	Invalid wait time parameter. The wait time must be between 0 and 999.
ICLErr_Mvs_PortFailed	4022	Commands sent via port failed! One or more commands sent to the designated backup port failed. The error may also be reported in the response array(s) for the failed command(s).
ICLErr_Mvs_PortSuccess	4023	Commands sent via port successful!
ICLErr_Mvs_InvPort	4024	Invalid port found. The specified fallback port handle was invalid.
ICLErr_Mvs_Timeout	4025	Time-out error. The commands were sent to gwMvsD, but no response was received within the configured timeout period.
ERR_OED_QUEUE_NOT_OPEN	11002	The queue is not open.
ERR_OED_REQUEST_FAILED	11003	The script could not communicate with the outside event daemon. Make sure gwOed is running.
ERR_OED_RESPONSE_FAILED	11004	The script did not receive the correct response from the outside event daemon. Make sure gwOed is running.
ERR_OED_INVALID_EVENT_SOURCE	11005	The source of events is not supported.
ERR_OED_OS_DOES_NOT_EXIST	11006	Specified OS does not exist. Check config/system.cfg file.
ERR_OED_QUEUE_ID_MUST_BE_SPECIFIED	11008	At least one queue ID must be specified.

EXEC

Syntax: EXEC($ScriptName[, Parm1, ...]) ==> ReturnValue

Description: Executes a script whose name is stored in a string expression.

Action: The script name stored in the ScriptName parameter is executed, passing any of the values in the optionally specified parameters to the called script. The calling script waits for the completion of the called script and receives its return value.

Parameters: **$ScriptName**. String expression. The name of the script to execute.

Parm1. Numeric or string expression. Optional. Zero or more parameters to pass to the called script. Refer to the **PARMS()** command for more information.

Returns: Numeric or string value. The return value from the executed script.

Notes:

1. Execution of the parent script pauses until the child script has finished execution.
2. Script names are case sensitive, so that calling "MYSCRIPT" is not the same as a script called "Myscript". We recommend using all lower case characters for script names, so that the name of the script is the same as the name of the physical script file on disk. In that case, a call to "myscript" actually calls a script file on disk named myscript.scx).
3. Functionally, EXEC is the same as directly calling the script as demonstrated in the following examples.

```
//1–directly calling a script called "script1"
%ReturnValue := script1( $Parm1, %Parm2)

//2–using EXEC with the script name in a string expression
%ReturnValue := EXEC( "script1", $Parm1, %Parm2)

//3–using EXEC with the script name in a variable
$ScriptName := "script1"
%ReturnValue := EXEC( $ScriptName, $Parm1, %Parm2)
```

All of the examples above have the same result:

1. script1 is run
2. $Parm1 and %Parm2 are passed as parameters to script1
3. %ReturnValue is set to the return value from script1

Examples 1 and 2 above are the same and EXEC provides no added benefit.

Example 3 demonstrates when to use the EXEC command versus directly calling the script. EXEC has the added benefit of changing the script called at runtime, because the script to call is specified in a variable (and variables can be changed at runtime). The following simple example illustrates this further:

```
IF %Status == 1
```

```
      $ScriptName := “script1”
ELSE
      $ScriptName := “script2”
ENDIF
%ReturnValue := EXEC( $ScriptName, $Parm1, %Parm2)
```

Depending on the value of %Status, either script1 or script2 is executed.

Example:

```
$ScriptName := “myscript”     //script name to execute
//in this example, ‘myscript’ requires two numerical
//arguments
%ReturnValue := EXEC( $ScriptName, %Parm1, %Parm2)
```

See Also: START

FCLOSE

Syntax: FCLOSE(%FileNum)

Description: Closes an open file.

Action: Closes a file opened with **FOPEN**().

Parameters: **%FileNum**. Numeric expression. The file handle obtained from **FOPEN**(). The file to close.

Returns: Numeric value. 0 (zero) when the file is successfully closed. This maps to manifest ERR_NONE.

Notes: If FCLOSE is passed a bad file handle, an execution-time error is generated.

Example:

```
%Handle := FOPEN( “SYS5”, OVERWRITE)
// ... file processing commands
FCLOSE( %Handle)
```

See Also: FDELETE, FEXISTS, FOPEN, FREAD, FRENAME, FREWIND, FWRITE

FDELETE

Syntax: FDELETE($FileName) ==> %Success

Description: Permanently deletes a file.

Action: $FileName is permanently deleted from the system.

Parameters: **$FileName**. String expression. The case sensitive name of the file to delete. Include any necessary file path.

Returns: Numeric value, as follows.

Value	Meaning
0	False, file deletion was not successful
1	True, file deletion was successful

Notes:

1. Refer to *Manifest Constants* on page 42 for information on the constants reference list.
2. If FileName does not exist, FALSE is returned.
3. If FileName exists but could not be deleted, FALSE is returned.
4. FDELETE can be used to delete any file on the MCC unit.
5. Using NFS, it is possible to delete virtually any file on any server or mainframe from the MCC unit.

Example:

```
%Success := FDELETE( “SYS5”)
%Success := FDELETE( “user-data/SYS5”)
```

See Also: FCLOSE, FEXISTS, FOPEN, FREAD, FRENAME, FREWIND, FWRITE

FEXISTS

Syntax: FEXISTS($FileName) ==> %Success

Description: Determines if a file exists.

Action: The file is checked to ensure it exists in the current or specified directory.

Parameters: **$FileName**. String expression. The case sensitive name of the file to verify if it exists. Include any necessary file path.

Returns: Numeric value, as follows.

Value	Meaning
0	False, file was not found
1	True, file exists

Notes:

1. Refer to *Manifest Constants* on page 42 for information on the constants reference list.
2. If FileName does not exist, FALSE is returned.
3. FEXISTS can be used to verify any file on the MCC unit.
4. Using NFS, it is possible to verify virtually any file on any server or mainframe from the MCC unit.

Example:

```
%Success := FEXISTS( “SYS5”)
%Success := FEXISTS( “user-data/config/” + $OSName)
```

See Also: FCLOSE, FDELETE, FOPEN, FREAD, FRENAME, FREWIND, FWRITE

FILENO

Syntax FILENO(%FileHandle) ==> %FileDescriptor

Description: Obtains the system integer file descriptor from a file handle.

Action: Obtains the system integer descriptor from the given file handle. Some external system calls require the descriptor instead of a handle so this is most useful in interfacing with shell scripts.

Parameters: %FileHandle: Numeric expression. The file handle obtained from FOPEN.

Returns: Numeric value, as follows:

-1 = Error occurred (not a valid file handle)

Any other value = The system file descriptor.

Example:

```
%fd := FILENO(%handle)
IF ( %tmpfile == -1 )
  LOG( LOG_FLT, STR(%handle) + " is not a valid file
handle!", 6)
ELSE
  LOG( LOG_FLT, "Descriptor is " + STR(%fd), 6)
ENDIF
```

See Also: FOPEN, FREAD, FWRITE, FCLOSE, MKSTEMP

FINDSTR

Syntax: FINDSTR($String, $Substring) ==> $FoundText

Description: Searches a string for a regular expression pattern.

Action: $String is searched using the regular expression pattern $Substring. The resultant substring found by the pattern $Substring is returned.

Parameters: **$String**. String expression. The string to search for the pattern in.

$Substring. Regular expression. The regular expression pattern to search for in $String. Refer to *Regular Expressions* on page 54 for more information.

Returns: String value, as follows:

Value	Meaning
“” (null string)	Matching substring not found
Any other value	The substring that matches the pattern $Substring.

Notes: N/A

Example:

```
//Example 1
$Message := “The quick Brown Fox”
$Searchstr := “Brown”
$Result := FINDSTR( $Message, $Searchstr )
// The contents of $Result is “Brown”

//Example 2
$Message := “See Jane Run. Run Jane Run.”
$Searchstr := “Spot”
$Result := FINDSTR( $Message, $Searchstr )
// The contents of $Result is null (“”)

//Example 3
// Match a word beginning with “Test” and ending with 1
//or 3
$Message := “Test1 Test2 Test3”
$Searchstr := “Test[13]”
$Result := FINDSTR ($Message, $Searchstr )
// The contents of $Result is “Test1”
```

See Also: ATSTR, LEFTSTR, RIGHTSTR, REPSTR, SUBSTR, STR

FMODTIME

Syntax: FMODTIME(File) ==> %EpochSeconds

Description: Returns the time value of a last modified date/time stamp for a file.

Action: The epoch seconds for the specified file's last modified date/time stamp is returned.

Parameters: **File.** Numeric or string expression. The name of the file for which to return the date/time stamp. The file name or the file handle may be specified. For a numeric expression, specify the file handle obtained from **FOPEN()**. For a string expression, specify the case-sensitive file name, including any path.

Returns: Numeric value, as follows:

Value	Meaning
0 (Zero)	False. The file does not exist.
Any other value	The epoch seconds of the file's last modified date/time stamp.

For additional information, refer to *Epoch Seconds* on page 40.

Notes:

1. Refer also to *Date/Time* on page 40.
2. FMODTIME can be used to check any file on the MCC unit.
3. Using NFS, it is possible to check virtually any file on any server or mainframe from the MCC unit.

Example:

```
%Handle := FOPEN( $FileName)
%TimeStamp := FMODTIME( %Handle)
%TimeStamp := FMODTIME( $FileName)
```

See Also: FCLOSE, FDELETE, FEXISTS, FOPEN, FREAD, FRENAME, FREWIND, FWRITE

FOPEN

Syntax: FOPEN($FileName[, %Mode]) ==> %FileHandle

Description: Opens a file for I/O access.

Action: FileName is opened for read/write access in the mode defined by the Mode parameter.

Parameters: **$FileName**. String expression. The case sensitive name of the file to open as a data file. Include any necessary file path.

%Mode. Numeric expression. Optional. The mode in which to open the file. Valid constants are OVERWRITE, APPEND, and READONLY. If not specified, the default mode is OVERWRITE.

Mode Constant	Description
OVERWRITE	**FWRITE**() always places new entries at the current record pointer. If the current record pointer is at the end of the file, no data is overwritten. If the current record pointer is not at the end of the file, the data at the current position is overwritten and the data after the written position is of a questionable state.
APPEND	**FWRITE()** always places new entries at the end of the file, no matter where the current record pointer is. No data is overwritten.
READONLY	**FOPEN()** opens the file in read only mode. No write operations are allowed to the file.

Returns: Numeric value, as follows:

Value	Meaning
-1	Error occurred while attempting to open the file.
Any other value	A unique number used for subsequent access to the file in the current script (commonly referred to as the "file handle" or the "file number").

Notes:

1. Refer to *Manifest Constants* on page 42 for the constants reference list.
2. The file handle obtained from an **FOPEN**() call is only valid in the same script that contains the **FOPEN**().
3. If the file specified to open does not exist, a new one is created.
4. If the file to be opened already exists, the Mode parameter determines whether the current data in the file may be overwritten.
5. The maximum number of files that can be opened per script thread is 60.
6. **FOPEN**() can be used to access any file on the MCC unit.
7. Using NFS, it is possible to access virtually any file on any server or mainframe from the MCC unit.
8. Using **FOPEN**() on the same file twice in succession returns different numeric values, but both handles refer to the same file.

Example:

```
%Handle := FOPEN( “SYS5”, OVERWRITE)
$FileName := “SYS5”
%Mode := APPEND
%Handle := FOPEN( $FileName, %Mode)
```

See Also: FCLOSE, FDELETE, FEXISTS, FMODTIME, FREAD, FRENAME, FREWIND, FWRITE

FORMATSTR

Syntax: FORMATSTR($String [, expr1, [expr2, ..., [exprn]..]]) ==> $Formatted

Description: Formats a string by combining literal characters with conversion specifications.

Action: The characters and conversions specified in the String parameter are processed with any expression parameters to produce a formatted string.

Parameters: **$String**. String expression. The string to format that contains the literal characters and conversion specifications. Each conversion specification will use zero or more expr parameters.

Expr. Numeric or string expressions. Optional. The values to substitute into the conversion specifications in the String parameter.

The order of the expression parameters is important—they will be processed left to right, matching the conversion specifications left to right in String.

Returns: String value. The formatted string, with all conversions and formatting completed.

Notes:

1. The number of expression parameters must match the number of expressions the String parameter expects in the conversions or unexpected results will occur.
2. Each expression's parameter type must match its expected type in the String parameter.

Example:

```
//This code segment demonstrates some examples
//of the FORMATSTR function.
//
$MyString := "UNIX Console 22"
%Value := 10
%Value2 := 3
%Value3 := -3

$Return := FORMATSTR( "The value is: %d.", %Value2 )
LOG( LOG_EXEC, $Return, 12 )
OUTPUT=The value is: 3.

$Return := FORMATSTR( "The value is: %+d.", %Value2 )
LOG( LOG_EXEC, $Return, 12 )
OUTPUT=The value is: +3.

$Return := FORMATSTR( "The value is: %+d.", %Value3 )
LOG( LOG_EXEC, $Return, 12 )
OUTPUT=The value is: -3.

$Return := FORMATSTR( "The value is: % d.", %Value2 )
LOG( LOG_EXEC, $Return, 12 )
OUTPUT=The value is:  3.

$Return := FORMATSTR( "The value is: %*d.", %Value,
%Value2 )
LOG( LOG_EXEC, $Return, 12 )
OUTPUT=The value is:          3.
```

```
$Return := FORMATSTR( "The value is: %10d.", %Value2 )
LOG( LOG_EXEC, $Return, 12 )
OUTPUT=The value is:          3.

$Return := FORMATSTR( "The value is: %.10d.", %Value2 )
LOG( LOG_EXEC, $Return, 12 )
OUTPUT=The value is: 0000000003.

$Return := FORMATSTR( "The value is: %-*d.", %Value,
%Value2 )
LOG( LOG_EXEC, $Return, 12 )
OUTPUT=The value is: 3         .

$Return := FORMATSTR( "Alert Received from %s.",
$MyString )
LOG( LOG_EXEC, $Return, 12 )
OUTPUT=Alert Received from UNIX Console 22.

$Return := FORMATSTR( "Value is %d, string is %s.",
%Value, $MyString )
LOG( LOG_EXEC, $Return, 12 )
OUTPUT=Value is 10, string is UNIX Console 22.

$Return := FORMATSTR( "Integer %d is ASCII %c, integer %d
is ASCII %c.", 65, 65, 97, 97 )
LOG( LOG_EXEC, $Return, 12 )
OUTPUT=Integer 65 is ASCII A, integer 97 is ASCII a.

$Return := FORMATSTR( "Integer %d is hex %#X.", %Value,
%Value )
LOG( LOG_EXEC, $Return, 12 )
OUTPUT=Integer 10 is hex 0XA.

$Return := FORMATSTR( "Integer %d is hex %#x.", %Value,
%Value )
LOG( LOG_EXEC, $Return, 12 )
OUTPUT=Integer 10 is hex 0xa.

$Return := FORMATSTR( "Integer %d is hex %#.5X.", %Value,
%Value )
LOG( LOG_EXEC, $Return, 12 )
OUTPUT=Integer 10 is hex 0x0000A.

$Return := FORMATSTR( "Integer %d is hex %#.*X.", %Value,
%Value, %Value )
LOG( LOG_EXEC, $Return, 12 )
OUTPUT=Integer 10 is hex 0x000000000A.

$Return := FORMATSTR( "Integer %d is octal %#o.", %Value,
%Value )
LOG( LOG_EXEC, $Return, 12 )
OUTPUT=Integer 10 is octal 012.
```

See Also: N/A

Conversion Specifications Syntax

Each conversion specification in the String parameter has the following syntax:

- % (percent sign). Processes the elements of the parameter list in variable order. In such a case, the normal conversion character % (percent sign) is replaced by %digit$, where digit is a decimal number in the range from 1 to the total number of expr parameters. Conversion is then applied to the specified parameter, rather than to the next unused parameter.

  When variable ordering is used:

- It must be specified for all conversions.
- The * (asterisk) specification for field width in precision is replaced by *digit$.

- Zero or more flags that modify the meaning of the conversion specification. The flag characters and their meanings are as follows:

  - Left align within the field the result of the conversion.

  + Begin the result of a signed conversion with a sign (+ or -). Overrides the (space) conversion specification.

  (space) Prefix a space character to the result if the first character of a signed conversion is not a sign. If both the (space) and + flags appear, the (space) flag is ignored.

  # Convert the value to an alternative form.

- For o conversion, the function increases the precision to force the first digit of the result to be a zero.
- For x and X conversions, a non-zero result has 0x or 0X prefixed to it.
- For c, d, and s conversions, the flag has no effect.

  0 Pad to field width using leading zeros (following any indication of sign or base) for o, x, and X, conversions (no space padding is performed). If the 0 (zero) and—(dash) flags both appear, the 0 flag is ignored. For d, o, x, and X conversions, if a precision is specified, the 0 flag is also ignored. For other conversions, the behavior is undefined.

  An optional decimal digit string that specifies the minimum field width. If the converted value has fewer characters than the field width, the field is padded on the left to the length specified by the field width. If the left-adjustment flag is specified, the field is padded on the right.

A field width can be indicated by an * (asterisk) instead of a digit string. In this case, an integer value parameter supplies the field width. The value parameter converted for output is not fetched until the conversion letter is reached, so the parameters specifying field width or precision must appear before the value (if any) to be converted. If the corresponding parameter has a negative value, it is treated as a left alignment option followed by a positive field width. When variable ordering with the %digit$ format is used, the * (asterisk) specification for field width in precision is replaced by *digit$.

- An optional precision. The precision is a . (dot) followed by a decimal digit string. If no precision is given, the decimal digit string is treated as 0 (zero). The precision specifies the minimum number of digits to appear for the d, o, x, or X conversions. A field precision can be indicated by an * (asterisk) instead of a digit string. In this case, an integer value parameter supplies the field precision. The value parameter converted for output is not fetched until the conversion letter is reached, so the parameters specifying field width or precision must appear before the value (if any) to be converted. If the value of the corresponding parameter is negative, the value is treated as if the precision had not been specified. When variable ordering with the %digit$ format is used, the * (asterisk) specification for field width in precision is replaced by *digit$.
- A character that indicates the type of conversion to be applied:

  % Performs no conversion. A literal percent sign.

  c Accepts an integer value and converts it to an unsigned ASCII character.

  d Accepts an integer and converts it to a signed decimal number.

  o Accepts an integer value and converts it to unsigned octal notation. The precision specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a 0 (zero) value with a precision of 0 (zero) is a null string. Specifying a field width with a 0 (zero) as a leading character causes the field width value to be padded with leading zeros. An octal value for field width is not implied.

  s Accepts a string and copies (or pads) it until the specified precision is reached.

x, X Accepts an integer value and converts it to unsigned hexadecimal notation. The letters abcdef are used for the x conversion and the letters ABCDEF are used for the X conversion. The precision specifies the minimum number digits to appear. If the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a 0 (zero) value with a precision of 0 (zero) is a null string. Specifying a field width with a 0 (zero) as a leading character causes the field width value to be padded with leading zeros.

If the result of a conversion is wider than the field width, the field is expanded to contain the converted result. No truncation occurs.

The representation of the + (plus sign) depends on whether the + or (space) formatting flag is specified.

FPOS

Syntax: FPOS(%FileNum) ==> %Position

Description: Returns an open file's current record pointer position.

Action: The current record pointer offset position for the file represented by the file handle FileNum is returned.

Parameters: %**FileNum**. Numeric expression. The file handle obtained from **FOPEN()**. The file for which to return the current record pointer offset value.

Returns: Numeric value. The number of bytes from the beginning of the file the current record pointer is positioned. The beginning of the file is byte 0 on opening.

Notes: NEWLINE is considered a character when determining the pointer position.

Example:

```
%Handle := FOPEN( $FileName)
%Position := FPOS( %Handle)
```

See Also: FCLOSE, FDELETE, FEXISTS, FREAD, FRENAME, FREWIND, FWRITE

FREAD

Syntax: FREAD(%FileNum, var1[, var2, ..., [varn]...]) ==> %QtyRead

Description: Reads values from an open file into variables.

Action: Each value from the file is read into its respective positional variable in the parameter list.

Parameters: **%FileNum**. Numeric expression. The file handle obtained from **FOPEN()**. The file to read data from.

Var. Numeric or string variables. The variables to place the values from the file into. The number of variables is unlimited.

Returns: Numeric value, as follows:

Value	Meaning
0	All values have been read from the file (at EOF) or a read error occurred.
Any other value	The number of values read from the file.

Notes:

1. The file handle obtained from an **FOPEN()** call is only valid in the same script that contains the **FOPEN()**.
2. **FREAD()** has the same file format expectations as **FWRITE()** creates—the contents of one variable parameter will be all characters in a file from a starting position up to the next newline character (each newline character is discarded after reading). For example, if a file contains 1000 characters without a newline character, the first **FREAD()** variable parameter receives all of the 1000 characters from the file, and any other variable parameters receives no characters.
3. If a value in the file is numeric and is to be placed in a string variable in an **FREAD()** parameter, the value is automatically converted to a string.
4. If a value in the file is a character and is to be placed in a numeric variable in an **FREAD()** parameter, the value is automatically converted following the rules for the **VAL()** command.
5. The return value will never be larger than the quantity of variable parameters to read into provided.

Example:

```
%Handle := FOPEN( $FileName)
FREAD( %Handle, %ShutdownClean, $IPLInfo)
```

See Also: FCLOSE, FEXISTS, FOPEN, FRENAME, FREWIND, FWRITE

FRENAME

Syntax: FRENAME($CurrentName, $NewName) ==> %Success

Description: Renames a file.

Action: The file CurrentName is renamed to file NewName.

Parameters: **$CurrentName**. String expression. The name of the file to rename, which is case-sensitive. Include any necessary file path.

$NewName. String expression. The new name of the file, which is case-sensitive. Include any necessary file path.

Returns: Numeric value, as follows.

Value	Meaning
0 (FALSE)	File was not successfully renamed.
1 (TRUE)	File was successfully renamed.

Notes:

1. Refer to *Manifest Constants* on page 42 for the constants reference list.
2. If FileName does not exist, FALSE is returned.
3. If FileName exists but could not be renamed, FALSE is returned.
4. FRENAME can be used to rename any file on the MCC unit.
5. Using NFS, it is possible to rename virtually any file on any server or mainframe from the MCC unit.
6. **FRENAME()** can take several moments to complete.

Example:

```
%Success := FRENAME( “SYS5”, “SYS5OLD”)
%Success := FRENAME( “\DATA\LPARS\SYS5”, “\DATA\SAVE\SYS5”)
```

See Also: FCLOSE, FEXISTS, FOPEN, FREAD, FREWIND, FWRITE

FREWIND

Syntax: FREWIND(%FileNum)

Description: Moves an open file's current record pointer to the beginning of the file.

Action: The current record pointer for the file represented by the file handle FileNum is repositioned to the beginning of the file. The next **FREAD()** command reads the first entry in the file.

Parameters: **%FileNum**. Numeric expression. The file handle obtained from **FOPEN()**. The file for which to move the current record pointer.

Returns: N/A

Notes: Same as **FSEEK**(%Handle, 0).

Example:

```
%Handle := FOPEN( $FileName)
//FREAD() commands to advance the current record pointer
//through the data
FREWIND( %Handle) // to read from the beginning again
```

See Also: FCLOSE, FEXISTS, FOPEN, FREAD, FWRITE

FSEEK

Syntax: FSEEK(%FileNum, %Position) ==> %Success

Description: Moves an open file's current record pointer to a byte offset.

Action: The current record pointer for the file represented by the file handle FileNum is repositioned to the byte offset specified by the Position parameter.

Parameters: **%FileNum**. Numeric expression. The file handle obtained from **FOPEN()**. The file for which the current record pointer will be moved.

%Position. Numeric expression. The number of bytes from the beginning of the file to position the current record pointer. The beginning of the file is byte 0.

Returns: Numeric value, as follows:

Value	Meaning
0 (FALSE)	The current record pointer was not successfully moved to the requested Position.
1 (TRUE)	The current record pointer was successfully moved to the requested Position.

Notes: N/A

Example:

```
%Handle := FOPEN( $FileName)
%Success := FSEEK( %Handle, %Position)
```

See Also: FCLOSE, FEXISTS, FOPEN, FREAD, FREWIND, FWRITE

FWRITE

Syntax: FWRITE(%FileNum, expr [, %NEWLINE]) ==> %Success

Description: Writes the expression to an open file.

Action: The expr parameter is written to the open file (represented by the FileNum).

Parameters: **%FileNum**. Numeric expression. The file handle obtained from **FOPEN()**. The file to write data to.

Expr. Numeric or string expression. The evaluated expression to write into the file. Only one expression can be specified per **FWRITE()**.

%NEWLINE. Boolean expression. Optional. Indicates if a newline (ASCII 10) character should be appended to the end of the data. TRUE means append the character, FALSE means omit it. Default value is TRUE.

Returns: Numeric value, as follows:

Value	Meaning
0 (FALSE)	Write to open file failed.
1 (TRUE)	Write to open file was successful.

Notes: The format of the file created by **FWRITE()** is one parameter value per line. Each line is separated by a newline character (ASCII 10), unless %NEWLINE is FALSE.

Example:

```
%Handle := FOPEN( $FileName)
FWRITE( %Handle, “This is a sample string.”, TRUE)
```

See Also: FCLOSE, FEXISTS, FOPEN, FREAD, FREWIND

GETENV

Syntax: GETENV($Variable) ==> $Value

Description: Obtains the current value of the given environment variable.

Action: Searches the host system environment list for a string that matches the given variable. Returns the value if found or an empty string if there is no match.

Parameters: $Variable: String. The environment variable name.

Returns: String value for the environment variable (may be the empty string)

Example:

```
$Variable := GETENV( "PPID" )
LOG( LOG_FLT, "PPID has the value '" + $Variable + "'",
6)
```

See Also: SYSEXEC

GETPID

Syntax: GETPID() ==> %ProcessId

Description: Obtains the system process identifier for this script.

Action: Returns the host system process identifier for the process that is executing this script. This is useful for watching the action of a script using outside performance tools such as top.

Parameters: none

Returns: Numeric value, process identifier number

Example:

```
%pid := GETPID()
LOG( LOG_FLT, "This script is running as process
" + STR($pid), 2)
```

See Also:

GOSUB

Syntax: GOSUB *Label

Description: Immediately transfers script execution to the specified label, and waits until the called routine finishes execution.

Action: Script execution is transferred to the specified label and continues normal script execution until the RETURN statement is encountered. At that point, script execution continues with the statement immediately following the GOSUB statement.

Parameters: ***Label**. Label literal. Name of the label to transfer execution to.

Returns: N/A.

Notes:

1. The label must be in the current script.
2. Nesting of GOSUBs is allowed and encouraged for good script structure and organization. The maximum number of nested GOSUBs is 255.
3. The major difference between GOSUB and GOTO is:
   - GOSUB is used to execute a subroutine, wait until it is done, and then continue processing (perhaps by calling another subroutine).
   - GOTO is used to immediately transfer the flow of script execution to another step. There is no returning as with the GOSUB command, and therefore is no nesting of GOTOs.

Example:

```
*START:
      GOSUB *LABEL01    //calls a subroutine in the
                        //same script file
      RETURN            //done with this script,
                        //so let's return to whatever
                        //called us
*LABEL01:
                        // do some good things here...
      RETURN            //continues with the next
                        //statement after the GOSUB
```

See Also: GOTO, START

GOTO

Syntax: GOTO *Label

Description: Immediately transfers script execution to the specified label.

Action: Script execution is transferred to the specified label.

Parameters: ***Label**. Label literal. Name of the label to which execution is transferred.

Returns: N/A.

Notes:

1. The label must be in the current script.
2. The major difference between GOSUB and GOTO is:
   - **GOSUB** executes a subroutine, waits until it is done, and then continues processing (perhaps by calling another subroutine).
   - **GOTO** immediately transfers the flow of script execution to another step. There is no returning as with the GOSUB command, and therefore is no nesting of GOTOs.

   Generally, the use of GOTOs is considered bad programming style. GOSUBs are preferred, to have a guaranteed program flow. However, using GOTOs will not negatively impact the performance or execution of the script.

Example:

```
GOTO *MVS01
```

See Also: GOSUB, SCANP, START

HEXSTR

Syntax: HEXSTR(%Number) ==> $Hex

Description: Converts an integer to a hex string.

Action: The Number parameter is converted to a hex number. The hex number is returned as a character string.

Parameters: %**Number**. Numeric expression. The number to convert to hex.

Returns: String value. The hex number formatted as a string.

Notes: N/A

Example:

```
%Num := 62
$HexNum := HEXSTR( %Num)
// $HexNum will contain the string ‘3E’
```

See Also: STR, VAL

HMCEXEC

Syntax: HMCEXEC(%ObjID, $Action [, parm1, ...]) => %RetVal

Description: Send a command to an HMC-controlled CPC or Image on a CMOS generation mainframe.

Action: Any command that can be performed using the MCC's HMC Window can be automated using this function.

Parameters: **%ObjID**. Numeric expression. The unique Object ID number of the CPC or Image (CPU or OS) to which the command is to be sent. Refer to *OBJID()* for more information.

$Action. String expression. The HMC action to perform:

parm1, parm2, .. String or numeric expression. Optional. Only two Actions take optional parameters: HMC_ACTIVATE and HMC_LOAD. Details are given in *Table 13. Possible HMC actions and parameters* following.

Returns: Numeric value, as follows:

Value	Meaning
0 (FALSE)	Command failed.
1 (TRUE)	Command was successful.

Notes: Performing these commands on a non-HMC CPU or OS will have no effect other than returning a 0.

Example:

```
%ObjID := OBJID(OS, $ImageName)
%RetVal := HMCEXEC(%ObjID, HMC_START)
%ObjID := OBJID(CPU, $CPCName)
%RetVal := HMCEXEC(%ObjID, HMC_DEACTIVATE)
```

See Also: N/A

Possible HMC actions and parameters

CPC and Image Commands

Action	Description	Optional Parameter(s)
HMC_ACTIVATE	Activate a CPC or Image	**$Profile**: The name of the activation profile to use for the activate process.
HMC_DEACTIVATE	Deactivate a CPC or Image	

Image-only Commands

Action	Description	Optional Parameter(s)
HMC_RESETNORMAL	Perform a normal reset on an Image	
HMC_START	Start the selected Image	
HMC_STOP	Stop the selected Image	
HMC_PSWRESTART	Perform a PSW restart on an Image	
HMC_LOAD	Perform a Load on an Image	**$Addr**: The Hex Load Address (Default: Last used value) **$Parm**: The Parameter String for the Load (Default: Last used) **%Clear**: Whether or not memory should be cleared before performing the Load (Default: TRUE) **%Timeout**: Amount of time to wait for the Load to complete (Default: 60 [sec]) **%Store**: Whether or not status should be stored before performing the Load (Default: FALSE)

Table 13. Possible HMC actions and parameters

HUMID

Syntax: HUMID(%Port) ==> %Humidity

Description: Reads the current humidity from a sensor unit.

Action: The humidity value is read from the sensor unit connected to the specified port.

Parameters: **%Port**. Numeric expression. The assigned sensor port number to which the sensor is connected. Refer to *Ports* on page 24 for further information.

Returns: Numeric value. The current humidity reading.

Notes: This command is not commonly used.

Example:
```
%Hum := HUMID( 2)
LOG( LOG_EXEC, "HUMIDITY 2 = " + STR( %Hum))
```

See Also: TEMP

ICON

Syntax: ICON(%Status[, $Message [, %Class [, $Name]]])

Description: Changes icon characteristics.

Action: The status/color, message, class, and name are changed for the specified icon.

Parameters: **%Status.** Numeric expression. The status number to set the icon to, and consequently changes its color. The range of status numbers is from 1 to 16, as listed below.

Constant, if Available	Value	Default Message	Color
STATUS_ERROR	1	Error	Red
	2		VioletRed
	3		HotPink
STATUS_WARNING	4	Warning	Orange
	5		Yellow
STATUS_INPROCESS	6	In Process	Gold
	7		White
	8		Gray
STATUS_NORMAL	9	Normal	PaleGreen
	10		Aquamarine
	11		VioletRed
	12		Cyan
	13		LightSkyBlue
STATUS_DOWN	14	Down	LightSteelBlue
	15		Blue
	16		Brown

$Message. String expression. Optional. The text to appear on the bottom line of the icon. The maximum number of characters is 10. A message with length of greater than 10 is truncated to the first 10 characters.

%Class. Numeric expression. Optional, but required with Name. The icon class. Refer to *Icon Class/Icon Name* on page 29 for more information.

$Name. String expression. Optional. The icon name. Refer to *Icon Class/Icon Name* on page 29 for more information.

Returns: N/A.

Notes:

1. Refer to *Manifest Constants* on page 42 for the constants reference list.
2. The default message for a status is overridden with the "Message" parameter.
3. The default colors can be changed only by editing the colors configuration file. Refer to the *Administration Guide* for more information.
4. This command is case-sensitive. Type the CPU or OS name exactly as it appears in the system.cfg file.

Example:

```
//change current (the default) icon status to normal—9
ICON( STATUS_NORMAL)
//change current icon status to 5 and its msg to "ERROR"
ICON( 5, "ERROR")
//change 3090 CPU icon status to warning—4
ICON( STATUS_WARNING, , CPU, "3090")
//if this script were executing on an OS,
//the higher-level CPU icon for the OS would be changed
//because the CPU name is omitted
ICON( 2, , CPU)
```

See Also: ICONNAME, ICONSTATUS, ICONMSG

ICONMSG

Syntax: ICONMSG([%Class [, $Name]]) ==> $Message

Description: Returns an icon's current message.

Action: The message text for the specified icon is returned.

Parameters: **%Class**. Numeric expression. Optional, but required with Name. The icon class. Refer to *Icon Class/Icon Name* on page 29 for more information.

$Name. String expression. Optional. The icon name. Refer to *Icon Class/Icon Name* on page 29 for more information.

Returns: String value. The icon's message—the text that appears on the bottom line of the icon.

Notes: N/A

Example:

```
//Example 1
$MsgStr := ICONMSG()
LOG(LOG_EXEC, $MsgStr)
//Example 2
$IconName := "MCC OS"
%IconClass := OS
$MsgStr := ICONMSG(%IconClass, $IconName)
LOG(LOG_EXEC, $MsgStr)
```

See Also: ICON, ICONNAME, ICONSTATUS

ICONNAME

Syntax: ICONNAME([%Class [, %Port]]) ==> $Name

Description: Returns an icon's name.

Action: The name of the icon is returned for the specified class and port.

Parameters: **%Class**. Numeric expression. Optional, but required if Port is specified. The icon class. Refer to *Icon Class/Icon Name* on page 29 for more information. Valid constants are CPU, OS, SW, and PRN.

%Port. Numeric expression. Optional. The port number that the object the icon represents is connected to. Refer to *Ports* on page 24 for more information.

Returns: String value. The name of the icon.

If Class and Port are not specified, the name of the current icon (the icon that the script is executing on) is returned.

If Port is not specified but Class is, the name of the icon in the current icon's "lineage" in the specified class is returned.

Notes: Refer to *Manifest Constants* on page 42 for the constants reference list.

Example:

```
$Name := ICONNAME()
$Name := ICONNAME( CPU)
$Name := ICONNAME( CPU, 2)
```

See Also: CLASSNUM, ICON, ICONMSG, ICONSTATUS, PORT

ICONSTATUS

Syntax: ICONSTATUS([%Class [, $Name]]) ==> %Status

Description: Returns an icon's current status.

Action: The message text for the specified icon is returned.

Parameters: **%Class**. Numeric expression. Optional, but required if Name is specified. The icon class. Refer to *Icon Class/Icon Name* on page 29.

$Name. String expression. Optional. The icon name. Refer to *Icon Class/Icon Name* on page 29.

Returns: Numeric value. The icon's status.

Notes: N/A

Example: `N/A`

See Also: CLASSNUM, ICON, ICONMSG, ICONNAME

IF

Syntax: IF...[ELSE...]ENDIF

IF Expression

Group 1 command

[More group 1 commands]

[ELSE]

Group 2 command

[More group 2 commands]

ENDIF

Description: Evaluates an expression for TRUE or FALSE—if...then statement.

Action: If Expression evaluates to TRUE, then the Group 1 command(s) are executed. If Expression evaluates to FALSE, then, if specified, the Group 2 command(s) are executed.

Parameters: **Expression**. Boolean expression. The expression to evaluate. Refer to *Boolean* on page 50 for more information.

Returns: N/A.

Notes:

1. The number of nested IFs is unlimited.
2. The number of commands allowed in each section (the TRUE or FALSE sections) is unlimited.
3. The following command is invalid but will not generate a compiler error:

```
IF %A=1 or 2 or 3
```

Example:

```
IF %Number == 100
      LOG( LOG_FLT, “Numeric variable is 100”)
ELSE
      LOG( LOG_FLT, “Numeric variable is not 100”)
ENDIF

IF $MsgText[ 1] == “ERROR” AND $MsgText[ 2] == $SrchText
      //additional commands here
ENDIF

IF %Number <= 250
      LOG( LOG_FLT, “Numeric variable is less than 250.”)
ENDIF

IF %Number != 0
      LOG( LOG_EXEC, “Numeric variable does not equal 0.”)
ENDIF
```

See Also: SWITCH

INC

Syntax: INC %Variable

Description: Adds one to a numeric variable's value.

Action: The value in the specified variable is increased by one.

Parameters: **%Variable**. Numeric variable. Name of the integer variable to increment.

Returns: N/A

Notes: The **INC()** command is a shortcut for adding one to itself. The following two statements are equivalent, but the INC() command executes much faster:

```
INC %Var
%Var := %Var + 1
```

Example:

```
*START:
      %Num := 0               //init to 0
*COUNTING:
      REPEAT
            //display value to filtered msgs window
            LOG( LOG_FLT, "COUNTER=" + STR( %Num))
            INC %Num          //add 1
      UNTIL %Num > 9
```

See Also: DEC, SET

JOIN

Syntax: JOIN($Array, $Delimiter) ==> $String

Description: Combines the elements of an array into a string.

Action: Each element of $Array is sequentially copied into $String. A delimiter is placed between each element in the string.

Parameters: **$Array**. String normal array variable. The array to combine into a string.

$Delimiter. String expression. The string characters to place between the element values in the returned string.

Returns: String value. The combined string of $Array elements and Delimiter.

Notes:

Example:
```
$Text := JOIN( $Arr, ":")
$Msg := JOIN( $MsgArr, " ")
```

See Also: SPLIT

KEY

Syntax: KEY(%Port, $Keys [,%Timeout]) ==> %RetCode

Description: Enters a character string to the specified console.

Action: Keyboard keys, represented by the characters in the Keys parameter, are sent to the console connected to the specified port just as if they were typed from the console keyboard. The keys are sent in exactly the order specified, left to right, in the $Keys string. If the console is unable to accept the keystrokes, the KEY command waits a specified amount of time until it allows them. See *Notes* for additional information.

Parameters: **%Port**. Numeric expression. The assigned console port number to which the console is connected. Refer to *Overview* on page 12 for more information.

$Keys. String expression. Characters and key representations to send as keyboard typing. Refer to the *KEY Command Specifics* tables following for details.

%Timeout. Numeric expression. Optional. The number of seconds to wait for an input inhibited condition (X clock or X SYSTEM message) to clear on a mainframe console before returning an error code signifying the inability to key the command. If not specified, the default value is zero, which means the KEY command does not wait for an input inhibited condition to clear before returning.

Returns: Numeric Value. Refer to *Table 14. KEY Command* following for more information.

Notes:

1. If a mainframe console is input-inhibited in a way that will resolve without user input (that is, without needing to press the RESET key), the KEY command waits the amount of time specified in the parameter %Timeout until the console is free. If this does not occur within the timeout period, the appropriate error code is returned.
2. If a mainframe console is input-inhibited in a way that requires the RESET key to be pressed, the KEY command issues a RESET before entering its string.
3. If sending a RESET key does not restore the mainframe console, an error is returned, allowing the script to engage in appropriate error trapping.
4. The KEY command processing checks the status line after sending keys to the console. If the status line indicates one of the following conditions, the appropriate status is returned to the script.

   - Last key not accepted (X ?+)
   - Bad function (X -f)
   - Too much data (X man >)
   - Numeric only (X man #),

   If the status line indicates X (clock) or X SYSTEM, it is assumed the command was submitted properly and no error is returned to the script. This is because a KEY command might set the X (clock) or X SYSTEM flag on the status line while the command is being processed. Consequently, it may be difficult to distinguish between

a command being processed, and a console that has failed or hung.

5. The KEY command processing also checks the status line before sending keys to the console. In this case, if the status line indicates X SYSTEM or X (clock), it waits for that indication to be removed from the status line if the user provided a timeout to the KEY command. For example, if the user entered KEY(%portOS, "$pjes2[ENTER]", 30), and the status line said X SYSTEM when this command started, the command waits up to 30 seconds for the X SYSTEM status to clear. If the command was just KEY(%portos, "v 000-999,offline[ENTER]"), and the status line contained X SYSTEM, the command would immediately return the error (number 5102).
6. If you use the KEY command with a [CLEAR] statement, do not put additional text in the same statement as KEY ([CLEAR]). Also, check the return code and take appropriate action if it fails.
7. If sending commands to an HP console, use [RETURN] or [RET] to send the RETURN key. Do not use [ENTER] or [ENT] unless you are in block mode, or specifically want to send the [ENTER] key.

Example:

```
// Logout of a Unix machine using a control-d
$ErrCode := KEY( %OSPort, “[CTRL-D][ENTER]”)
// Logout of a Unix machine using the exit command
%ReturnCode := KEY( %OSPort, “exit[ENTER]”)
// Display the date and time on a mainframe console
%ErrCode := KEY (%OSPort, “d t[ENTER]”, 10)
```

See Also: BLOCKSCAN, SCANB, SCANP

KEY Command Return Values

Manifest Constant	Value	Description
Err_None	0	No error
Err_Key_Timelock	5101	Time is required for the function to complete.
Err_Key_Syslock	5102	The system has locked the keyboard while processing data.
Err_Key_CommError	5103	Controller error, SNA protocol error, or communication error.
Err_Key_NotAccepted	5104	Last input was not accepted. Typically caused by an attempt to enter keys when the X clock or X printer busy message was displayed.
Err_Key_BadFunction	5105	The user is trying to perform a function that is not currently available.
Err_Key_BadLocation	5106	The user is trying to enter data in a location that cannot accept keys.
Err_Key_TooMuchData	5107	The user is trying to enter too much data into a field.
Err_Key_NumericOnly	5108	The user is trying to enter non-numeric characters into a numeric field.
Err_Key_CantReadStatus	5109	The MCC could not read or parse the status line.
Err_Key_ConsoleNotLocked	5110	The script did not lock the console before trying to send keys to it.
Err_Key_NoDaemonConnection	5111	The script could not connect to the daemon to send keys or read the status line.
Err_Key_NoResponseReceived	5112	The script was able to send keys to the console, but it did not get a return code back indicating whether the command was successful.
Err_Key_BadParameter	5113	One or more of the arguments to the KEY command is invalid.
Err_Key_LockQueueFull	5116	The maximum of 20 scripts are already running on this console.

Table 14. KEY Command Return Values

KEY Command Specifics

> *Note*: The KEY command is not case sensitive. Characters within brackets ([or]) are interpreted the same, regardless of whether they are upper or lower case. Uppercase is used as the standard for readability.

The following table lists keyboard keys and their KEY command equivalents:

Keyboard Key	Key Command
ASSIGN CONS	[CNS] or [ASGNCNS]
ATTN	[ATN] or [ATTN]
BACK TAB	[BKTAB]
BKWD	[BWD] or [BKWD]
BREAK	[BREAK]
¢	[CENT]
CHANGE DISPLAY	[CHGDPLY]
CLEAR (see Notes)	[CLR] or [CLEAR]
CNCL(PA2)	[CP2] or [CNCL]
CONTROL-A through CONTROL-Z	[CTRL-A] through [CTRL-Z]
CURSOR UP	[CUP]
CURSOR DOWN	[CDW] or [CDN]
CURSOR LEFT	[CLT]
CURSOR RIGHT	[CRT]
DEL or DELETE	[DEL] or [DELETE]
DEVICE CANCEL	[DVCNL]
DUP	[DUP]
END	[END]
ENTER	[ENT] or [ENTER]
ERASE EOF	[EOF] or [EREOF]
ERASE INPUT	[INP] or [ERINP]
ESCAPE	[ESC] or [ESCAPE]
FIELD MARK	[FMK] or [FLDMRK]
FIND	[FIND]
FWD	[FWD]
HOME	[HOME]
INDEX	[INDEX]

Keyboard Key	Key Command
INS (^a) or INSERT	[INS] or [INSERT]
IRPT	[IPT] or [IRPT]
ISTEP EOF	[ISTEP]
LAST CMD	[LASTCMD]
LEFT	[LEFT]
NEXT SCREEN	[NEXTSCREEN]
¬	[NOT]
PA1	[PA1]
PA2	[PA2]
PF key nn (01 to 24)	[Fnn]
PREV SCREEN	[PREV SCREEN]
REFRESH	[REF] or [REFRESH]
REMOVE	[REMOVE]
RESET	[RESET]
RESTART	[RST] or [RESTART]
RETURN	[RET] or [RETURN]
RIGHT	[RIGHT]
SELECT	[SELECT]
Shift+F6 through Shift+F20	[SHIFTF6] through [SHIFTF20]
START	[STR] or [START]
STOP	[STP] or [STOP]
SWAP CONS	[SWAPCNS]
SYS REQ	[SRQ] or [SYSREQ]
TAB	[TAB]
TOD	[TOD]
VIEW LOG	[VIEWLOG]

Table 15. Keys and Command Equivalents

Note: The date and time formats embedded in the KEY command have been deprecated and should not be used. Instead use the TIMESTR command, referring to *KEY Command (Date and Time Formats)* on page 214 for replacement commands.

LEFTSTR

Syntax: LEFTSTR($String, %Count) ==> $SubStr

Description: Returns the leftmost specified number of characters of a string expression.

Action: A substring is extracted from a string expression. The substring begins with the first character in the string expression.

Parameters: **$String**. String expression. The character string from which to extract characters.

%Count. Numeric expression. The number of characters to extract. If Count is negative or zero, LEFTSTR() returns an empty string “”. If Count is larger than the length of String, LEFTSTR() returns the entire String.

Returns: String value. The leftmost Count characters of String as a character string.

Notes:

Example:

```
$Msg := “This is a test.”
$Var := LEFTSTR( $Msg, 4)
// $Var will contain the string ‘This’

$Var := LEFTSTR( “****” + $Msg[ 1] + $Msg[ 2] + “****”, 20)
// $Var will contain a string starting with 4 asterisks, and
// containing the first 16 characters of the combination of
// strings $Msg[1] and $Msg[2].  If there are less than 16
// characters in those variables, $Var will also contain up to
// 4 asterisks at the end.
```

See Also: ATSTR, FINDSTR, RIGHTSTR, REPSTR, STR, SUBSTR

LEN

Syntax: LEN($StringExpr) ==> %Count

Description: Returns the number of characters in a string expression.

Action: Counts the number of characters in an evaluated character expression.

Parameters: **$StringExpr**. String expression. The character string to count.

Returns: Numeric value. The length of the character string.

Notes:
1. If the string is empty “”, zero is returned.
2. Each byte in the string counts as one.

Example:
```
$String := “Hello”
%Len := LEN( $String)
// %Len will contain the value 5.
%Len := LEN( $Text)
IF LEN( $Name) == 0
      //some commands
ENDIF
%Count := LEN( “****” + $Msg[ 1] + $Msg[ 2] + “****”)
```

See Also: ALEN

LOG

Syntax: LOG(%LogType, $Message[, %Status])

Description: Enters a message in a log.

Action: The message is entered into the specified log file and displayed on the appropriate log window.

Parameters: **%LogType**. Numeric expression. Determines the log into which the messages are entered, as follows.

Constant	Description
LOG_CHAN	Channel messages log
LOG_CPU	CPU messages log
LOG_EXEC	MCC execution messages log
LOG_FLT	Filtered messages log
LOG_UNIT	I/O Unit messages log
LOG_SW	Software messages log

$Message. String expression. The message to enter in the log file. The maximum number of characters per message is 80; extra characters are truncated.

%Status. Numeric expression. Optional. Determines the message color for display purposes. If not specified, the default status is STATUS_NORMAL.

Returns: N/A.

Notes:

1. Messages logged to the Filtered messages log are also displayed on the Filtered Messages window.
2. Messages logged to the execution messages log are also displayed on the Execution Log Display.

Example:

```
LOG( LOG_FLT, “Shutdown started”, STATUS_INPROCESS)
$MsgText := “This is a message”
LOG( LOG_EXEC, $MsgText + “ into the execution log”)
```

See Also: N/A

LOWER

Syntax: LOWER($String) ==> $Lowercase

Description: Converts uppercase characters to lowercase.

Action: The uppercase characters of an evaluated string expression are converted to lowercase characters.

Parameters: **$String**. String expression. The character string to convert to lowercase.

Returns: String value. A copy of the evaluated string expression with all alphabetic characters in lowercase. All of the other characters remain the same as in the original expression.

Notes:
1. The original string expression is not changed.
2. Only the alphabetic characters are converted; numerics and special characters are not changed.

Example:
```
$Original := “ThIS iS HaRd tO REaD”
$LowerCaseText := LOWER( $Original)
// $LowerCaseText will contain the string ‘this is hard to
read’
$Text := LOWER( $Text)
IF LOWER( $Name) == “fred”
      //Commands here will be processed if $Name is equal to
      //any of the following: ‘Fred’ ‘FRED’ ‘FrEd’ etc.
ENDIF
```

See Also: UPPER

MKDTEMP

Syntax: MKDTEMP($Pattern) ==> $DirectoryName

Description: Creates a unique temporary directory for I/O access.

Action: A directory that is guaranteed to be uniquely named and not accessible by other users is created. As this is a temporary directory, it and all of the contained files will be removed from the system at the time the calling script terminates.

Parameters: $Pattern: String expression. This is a case sensitive template to be used by the system to create a directory name. The last six characters of the template must be XXXXXX (all uppercase) and these are replaced with a string that makes the directory name unique. The directory is created with permissions insuring access only this user.

Returns: String value, as follows:

"" (empty string) = Error occurred while attempting to create the directory.

Any other value = A unique directory name to be used for subsequent access by the current script.

Example:

```
$tmpdir := MKDTEMP("/usr/ics/tmpXXXXXX")
IF ( $tmpdir == "" )
  LOG( LOG_FLT, "Error creating temporary directory!",
6)
ELSE
  LOG( LOG_FLT, "Directory is " + $tmpdir, 6)
ENDIF
```

See Also: MKSTEMP, FREAD, FWRITE, FCLOSE, FILENO

MKSTEMP

Syntax: MKSTEMP($Pattern) ==> %FileHandle

Description: Opens a unique temporary file for I/O access.

Action: A file that guaranteed to be uniquely named and not accessible by any other process is opened for read/write access. As this is a temporary file, it will be removed from the system at the time the calling script terminates.

Parameters: $Pattern: String expression. This is a case sensitive template to be used by the system to create a filename. The last six character of the template must be XXXXXX (all uppercase) and these are replaced with a string that makes the filename unique. The file is created with mode read/write and permissions 0666. The file is opened for exclusive access guaranteeing that when this routine returns successfully, this script will be the only user.

Returns: Numeric value, as follows:

-1 = Error occurred while attempting to create and open the file.

Any other value = A unique number used for subsequent access to the file in the current script (commonly referred to as the "file handle" or the "file number").

Example:

```
%tmpfile := MKSTEMP("/usr/ics/tmp/foobarXXXXXX")
IF ( %tmpfile == -1 )
  LOG( LOG_FLT, "Error creating temporary file!", 6)
ELSE
  LOG( LOG_FLT, "File handle is " + STR(%handle), 6)
  %rc := FWRITE( %tmpfile, "Here is some output",
TRUE )
  IF ( %rc == 0 )
    LOG( LOG_FLT, "Error writing to temporary file!",
6)
  ELSE
    %rc := FREAD( %tmpfile, $FirstWord )
    IF ( %rc == 1 )
      LOG( LOG_FLT, "Read '" + $FirstWord + "'", 6)
    ENDIF
  ENDIF
ENDIF
```

See Also: FREAD, FWRITE, FCLOSE, FILENO

MKTEMP

Syntax: MKTEMP($Pattern) ==> $FileName

Description: Returns a unique file name.

Action: Creates a file name based on the given pattern that is guaranteed to be unique (not used) at the time of creation. This is intended for use in situations where a temporary file name is needed. Please note that MKSTEMP is preferred to avoid race conditions. MKTEMP is provided for backward compatibility.

Parameters: $Pattern: String expression. This is a filename with some number of 'Xs' appended to it. The 'Xs' will be replaced with letters and numbers so that the filename is unique.

Returns: String value, unique file name (or am empty string on failure)

Example:

```
$TempFileName := MKTEMP("/tmp/fooXXXXXX")
$Command := "ls -la > " + $TempFileName
%rc := SYSEXEC( $Command )
IF ( %rc >= 0 )
  %Handle := FOPEN( $TempFileName )
  IF ( %Handle != ERROR )
    %rc := FREAD( %Handle, $Temp )
    IF ( %rc >= 0 )
      LOG( LOG_FLT, "First line is " + STR(
$Temp ), 8)
    ENDIF
  ENDIF
ENDIF
```

See Also: MKSTEMP, SYSEXEC, FOPEN

MONIKER

Syntax: MONIKER() ==> $Name

Description: Obtains the product name.

Action: Returns the product name string.

Parameters: none

Returns: String value giving the name of the product.

Example:

```
$CallMe := MONIKER()
LOG( LOG_FLT, "Running " + $CallMe, 7)
```

See Also:

OBJEXEC

Syntax: OBJEXEC(%ObjID, $Action[, Parms...]) ==> %ReturnValue

Description: Executes an action on an object.

Action: The action specified by the Action parameter is performed on the object specified by ObjID.

Parameters: **%ObjID**. Numeric expression. The ID of the object to perform an action on. Refer to *Object ID* on page 26 for more information.

$Action. String expression. The action to perform on the object. Valid actions are WAIT, WAITCHILD, WAITQUEUE, and WAITQUEUCHILD. Refer to *Object Action* on page 27 for further information.

Parms. Numeric or string expressions. The optional and/or required parameters depend on the specified Action. Refer to *Object Action* on page 27 for further information.

Returns: Numeric or string value. The return value depends on the specified Action. Refer to *Object Action* on page 27 for further information.

Notes: Use OBJEXEC() to monitoring tasks on an OS. Once the task list is set up in a script, OBJEXEC can be set on the OS to wait until a change occurs on one of its child objects. The script can then make an appropriate response.

Example:

```
// In this example script segment, '%OSObjID' contains the
// ID of the OS you want to monitor.  The return value
// 'TaskObjID' will be the object ID of the child which
// has changed.
%Wait := 3
WHILE TRUE
      %TaskObjID := OBJEXEC(%OSObjID, WAITQUEUECHILD,
%Wait)
      IF %TaskObjID > 0
            // Take appropriate change actions.
      ENDIF
ENDWHILE
```

See Also: OBJGET, OBJGETARRAY, OBJID, OBJIDARRAY, OBJSET, OBJSETARRAY

OBJGET

Syntax: OBJGET(%ObjID, $ObjFieldName) ==> $CurrentValue

Description: Returns the current value in an object's field.

Action: The current value in the field specified by ObjFieldName for the object specified by ObjID is returned.

Parameters: **%ObjID**. Numeric expression. The ID assigned to the object from which to get the value. Refer to *Object ID* on page 26 for more information.

$ObjFieldName. String expression. The field name in the object to get the value from. Refer to *Object Field* on page 26 for more information.

Returns: String value. The current value in the field. Integer values are converted and returned as strings.

If $CurrentValue is the empty string, "", the **ERRORNUM**() function will return a value according to the following table:

Manifest Constant	Description
Err_None	No error has occurred. The correct value for the object's field is the empty string.
Err_BadArgs	$ObjFieldName is not a member of the object.
Err_Obj_InvalidId	%ObjID does not contain a valid object id.

Notes:

1. Refer to the description of the VAL() command for information on string to numeric conversions.
2. When retrieving more than 25% of the fields of an object, **OBJGETARRAY()** is more efficient than **OBJGET()**.

Example:

```
//This example gets the current status of the Cron
// task, running on the Solaris OS, running on the
// Webserv CPU.
$CronExpr := "WEBSERV:SOLARIS:CRON"
%CronID := OBJID( SW, $CronExpr)
$Current_fieldname := "Current_status"
$Cron_Current_Status := OBJGET (%CronID, $Current_fieldname)
```

See Also: OBJEXEC, OBJGETARRAY, OBJID, OBJIDARRAY, OBJSET, OBJSETARRAY

OBJGETARRAY

Syntax: OBJGETARRAY(%ObjID, $AssocArray) ==> %ErrCode

Description: Populates an associative string array with the current field values from an object.

Action: The current values in the fields for the object specified by ObjID are populated into an associative array. The index values of the associative array are the field names for the object.

Parameters: **%ObjID**. Numeric expression. The object id. The ID assigned to the object from which to get the values. Refer to *Object ID* on page 26 for more information.

$AssocArray. Associative string array. The array to populate with the current values from the object. Integer values are converted to strings.

Returns: A numeric value representing the status of the operation. If %ErrCode is ERROR, the **ERRORNUM**() function returns a value as follows:

Manifest Constant	Description
Err_None	No error has occurred.
Err_Obj_InvalidId	%ObjID does not contain a valid object id.

Notes:

1. Refer to the description of the VAL() command for string to numeric conversions.
2. The associative string array is cleared before being populated with the field values. Refer to the description of **ARESET**() for more information.
3. The TaskName field is fixed and cannot be changed by a script. Consequently, OBJGETARRAY cannot return its value.
4. **OBJGETARRAY()** is more efficient than **OBJGET()** when retrieving more than 25% of the fields of an object.

Example:

```
$MyObject := "MyCPU:MyOS:cron"
%MyID := OBJID( SW, $MyObject )
      %ReturnCode := OBJGETARRAY( %MyID, $aAllStatus )
      $ReturnCurrent := $aAllStatus["Current"]
      $ReturnDesired := $aAllStatus["Desired"]
      $ReturnGroup := $aAllStatus["Group"]
      LOG(LOG_EXEC,"Current Status is: " + $ReturnCurrent,3)
      LOG(LOG_EXEC, "Desired Status is: "+ $ReturnDesired,3)
      LOG(LOG_EXEC, "Group Status is: " + $ReturnGroup, 3)
```

OUTPUT:

```
Current Status is: UNKNOWN
Desired Status is: UNKNOWN
Group Status is: NONE
```

See Also: OBJEXEC, OBJGET, OBJID, OBJIDARRAY, OBJSET, OBJSETARRAY

OBJID

Syntax: OBJID(%Class, $ObjKeyExpr) ==> %ObjectID

Description: Returns the ID (unique only for the particular script thread) for a MCC object.

Action: The ID (ObjID) for the object specified by the ObjKeyExpr parameter in the class specified by the Class parameter is returned.

Parameters: **%Class**. Numeric expression. The icon class of the lowest level (target) object specified in the ObjKeyExpr parameter. Valid classes are CPU, OS, MVSAGENT (see below), and SW. Refer to *Icon Class/Icon Name* on page 29.

$ObjKeyExpr. String expression. The object key expression. Refer to *Object Key* on page 25 for more information.

Returns: Numeric value. If **OBJID**() succeeds, %ObjectID will be the Object ID assigned to the object. If an error occurs, %ObjectID will be the value ERROR. If %ObjectID is ERROR, the **ERRORNUM**() function returns one of the following values:

Manifest Constant	Description
Err_BadArgs	$ObjKeyExpr does not resolve to an object that matches %Class.
1001	%Class is not CPU, OS, or SW.

Refer to *Object ID* on page 25 for more information.

Notes:

1. If OBJID() is called with an invalid object key expression, %ObjectID is set to -1 (ERROR).
2. The MVSAGENT class is required when using QREAD to read messages from the MVS agent. Essentially, this treats the MVS agent as a printer:

```
%oid := OBJID(MVSAGENT, "ECS OS")
%objids[1] := %oid
%qid := QOPEN(%objids)
// read from the printer Q
$text := QREAD(%qid, $msgarray, 0)
LOG(LOG_EXEC, "text = " + $text)
```

Example:

```
//=======================================
//EXAMPLE 1: SW (task) ObjID
/=======================================
//This gets an ObjID for the cron task, running
// on the Solaris OS, on the Webserv cpu
$CronExpr := “WEBSERV:SOLARIS:CRON”
%CronID := OBJID( SW, $CronExpr)
//=======================================
//EXAMPLE 2: OS ObjID
//=======================================
//This sample an ObjID for the BETA OS,
// running on the 9672-1 CPU
$BetaExpr := “9672-1:BETA”
%BetaID := OBJID( OS, $BetaExpr)
//=======================================
//EXAMPLE 3: OS ObjID array, for use with QOPEN
//=======================================
```

```
//This example builds an array of ObjIDs, for use
// with the QOPEN command.
$Lpar1 := “9672-1:BETA”
$Lpar2 := “9672-1:PROD”
$Lpar3 := “3090:DEV”
%OSIDarray[1] := OBJID(OS, $Lpar1)
%OSIDarray[2] := OBJID(OS, $Lpar2)
%OSIDarray[3] := OBJID(OS, $Lpar3)
```

See Also: OBJEXEC, OBJGET, OBJID, OBJIDARRAY, OBJSET, OBJSETARRAY

OBJIDARRAY

Syntax: OBJIDARRAY(%Class, %ObjIDParent, %AssocArray) ==> %Children

Description: Populates an associative numeric array with object ids from the children of a parent object.

Action: Populates the child object ids in the class specified by the Class parameter for the parent object specified by the ObjIDParent parameter into an associative array. The associative array index values are the object names (names only, not the fully qualified object key— refer to the descriptions of *Object Name* on page 25 and *Object Key* on page 25 for more information. The associative array data values are the object ids.

Parameters: **%Class**. Numeric expression. The icon class of the child objects for which to get the object ids. Valid classes are CPU, OS, and SW. Refer to *Icon Class/Icon Name* on page 29 for more information.

%ObjIDParent. Numeric expression. The object id of the parent object. This object is the parent of the objects for which to get the object ids. Refer to the description of Object ID for more information.

%AssocArray. Associative numeric array. The array to populate with the child object ids.

%Children. Numeric value. The number of children of the parent object.

Returns: Integer. The number of children of the parent object. If %Children is 0, the **ERRORNUM**() function returns one of the following values:

Manifest Constant	Description
Err_None	No error occurred. The correct value for the object's field is the empty string.
1001	%Class is not CPU, OS, or SW.
Err_BadArgs	%ObjIDParent does not contain any objects of the requested %Class.
Err_Obj_InvalidId	%ObjIDParent does not contain a valid object id.

Notes:

1. This command is intended to get names from the SW level. Conversely, the **OBJIDARRAY()** command does not get commands from the SW level.
2. In the following example, the '/' character means that the code line continues on the next line. Do not interpret it literally.

Example:

```
// The following example assumes a CPU named 'MyCPU', an
// OS named 'MyOS', and three tasks listed on the swlist
// for 'MyOS'.
$MyObject := "MyCPU:MyOS"
%MyID := OBJID( OS, $MyObject )
%ReturnCode := OBJIDARRAY( SW, %MyID, %aaKidIDs )
ASSOCKEYS( %aaKidIDs, $aKidIDs )
%Length := ALEN( $aKidIDs )
%Count := 1
WHILE %Count <= %Length
      LOG(LOG_EXEC, "Index [" + %Count + "] is: " + /
            $aKidIDs[%Count], 1)
```

```
      LOG(LOG_EXEC, $aKidIDs[%Count] + "Object ID is: " + /
            %aaKidIDs[$aKidIDs[%Count]], 1)
      INC %Count
ENDWHILE
```

OUTPUT:

```
Index [1] is: inetd
inetd Object ID is: 3
Index [2] is: cron
cron Object ID is: 2
Index [3] is: lpd
lpd Object ID is: 4
```

See Also: AICONNAMES, ASSOCKEYS, OBJEXEC, OBJGET, OBJGETARRAY, OBJID, OBJSET, OBJSETARRAY

OBJSET

Syntax: OBJSET(%ObjID, $ObjFieldName, $NewValue) ==> %ErrCode

Description: Sets the current value in an object's field.

Action: The field specified by ObjFieldName for the object specified by ObjID is set to the value specified by the NewValue parameter.

Parameters:

%ObjID. Numeric expression. The object ID assigned to the object for which to set the value. Refer to the description of *Object ID* on page 26 for more information.

%ObjFieldName. String expression. The field name in the object to get the value from. Refer to *Object Field* on page 26 for more information.

$NewValue. String expression. The value to set the specified field to. A target of an integer field will have its value automatically converted from the string value to a numeric value in the same manner as the **VAL**() command.

Returns: A numeric value indicating the status of the operation, as follows:

Manifest Constant	Description
ERR_NONE	No error
ERROR	Error

If an error occurred, the **ERRORNUM**() function returns one of the following values:

Manifest Constant	Description
Err_None	No error has occurred. The correct value for the object's field is the empty string.
Err_BadArgs	%Class is not a supported class type.
Err_Obj_InvalidId	%ObjIDParent does not contain a valid object id.

Notes:

1. Refer to the description of the **STR**() command for string to numeric conversions.
2. When setting more than 25% the fields of an object, **OBJSETARRAY()** is more efficient than OBJSET().

Example:

```
//This example sets the desired status of the Cron
// task ( running on the Solaris OS, running on the
// Webserv CPU ) to down.
$CronExpr := "WEBSERV:SOLARIS:CRON"
%CronID := OBJID( SW, $CronExpr)
$Desired_fieldname := "Desired_status"
$New_Desired_value := "down"
%ErrCode := OBJSET( %CronID, $Desired_fieldname,
$New_Desired_value)
```

See Also: OBJEXEC, OBJGET, OBJGETARRAY, OBJID, OBJIDARRAY, OBJSETARRAY

OBJSETARRAY

Syntax: OBJSETARRAY(%ObjID, $AssocArray) ==> %Success

Description: Sets the field values for an object from an associative string array.

Action: The current values in the fields for the object specified by ObjID are set to the values in the associative array. The index values of the associative array are the field names for the object (see *Notes* below).

Parameters: **%ObjID**. Numeric expression. The ID assigned to the object for which to set the values. Refer to *Object ID* on page 26.

$AssocArray. Associative string array. The array from which to set the object's current field values. Integer fields are converted from the string array.

Returns: A numeric value indicating the status of the operation. The following table describes the error codes:

Manifest Constant	Description
ERR_NONE	No error
ERROR	Error has occurred. No values set.

Notes:

1. Refer to the **STR**() command description for numeric to string conversions.
2. Not all of the object's field names must be specified. Only the fields specified in the associative array are set, and the unspecified fields remain unchanged.
3. OBJSETARRAY() is more efficient than **OBJSET()** when setting more than 25% of the fields of an object.

Example:

```
$SetCurrent := "UP"
$SetDesired := "UP"
$SetGroup := "UNIX"

$MyObject := “MyCPU:MyOS:cron”
%MyID := OBJID( SW, $MyObject )
GOSUB *DoOBJGETARRAY

$SetArray["Current"] := $SetCurrent
$SetArray["Desired"] := $SetDesired
$SetArray["Group"] := $SetGroup

%ReturnCode := OBJSETARRAY( %MyID, $SetArray )

GOSUB *DoOBJGETARRAY
END

*DoOBJGETARRAY:
      %ReturnCode := OBJGETARRAY( %MyID, $aAllStatus )

      $ReturnCurrent := $aAllStatus["Current"]
      $ReturnDesired := $aAllStatus["Desired"]
      $ReturnGroup := $aAllStatus["Group"]
      LOG(LOG_EXEC,"Current Status is: " + $ReturnCurrent,3)
     LOG(LOG_EXEC, "Desired Status is: "  $ReturnDesired,3)
     LOG(LOG_EXEC, "Group Status is: " + $ReturnGroup, 3)
```

```
RETURN

OUTPUT:
Current Status is: UNKNOWN
Desired Status is: UNKNOWN
Group Status is: NONE

Current Status is: UP
Desired Status is: UP
Group Status is: UNIX
```

See Also: OBJEXEC, OBJGET, OBJGETARRAY, OBJID, OBJIDARRAY, OBJSET

PARMS

Syntax: PARMS var1[, var2 [, var3, ..., [varn]...]]]

Description: Receives parameters into the script.

Action: Values passed from the calling entity are received sequentially, one-to-one, into the variables in the PARMS command.

Parameters: **Var**. Numeric or string variable. The passing of arrays is allowed.

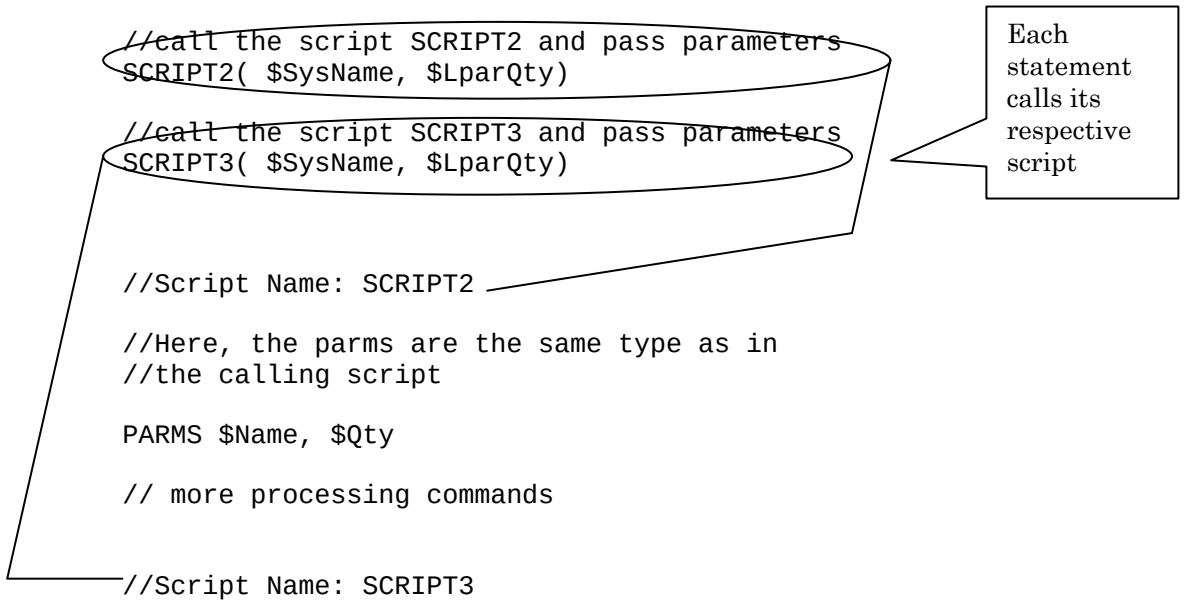
Returns: N/A.

Notes:

1. If used, the PARMS command must be the first executed command in the script.
2. Up to 255 parameters can be passed or received.
3. Automatic variable type conversion will happen when necessary—strings are converted to numerics (with the same result as **VAL**()), and numerics are converted to strings (with the same result as **STR**()). Refer to the examples below.

Example:

```
//Script Name: SCRIPT1

$SysName := "SYS5"
%LparQty := 4

//call the script SCRIPT2 and pass parameters
SCRIPT2( $SysName, $LparQty)

//call the script SCRIPT3 and pass parameters
SCRIPT3( $SysName, $LparQty)


//Script Name: SCRIPT2

//Here, the parms are the same type as in
//the calling script

PARMS $Name, $Qty

// more processing commands


//Script Name: SCRIPT3

//here the "$Qty" parm is automatically converted
//to a string via its syntax in the PARMS
//statement

PARMS $Name, $Qty

// more processing commands
```



See Also: N/A

PORT

Syntax: PORT(%Class[, $IconName]) ==> %Port

Description: Returns the port number for a console definition.

Action: A console's port number is returned for a class and an optional icon name.

Parameters: **%Class**. Numeric expression. The icon class. Refer to *Icon Class/Icon Name* on page 29 for more information. The unique port sequence type. Only CPU, OS, PRN, and SW are valid for the PORT function.

$IconName. String expression. Optional. The name of the icon whose port number should be returned. If not specified, the icon used is in the class specified by the port type that is in the same lineage as the icon on which the script is executing.

Returns: Numeric value, as follows:

Value	Meaning
-1	The requested console definition or port does not exist.
Any other value	The port number for the specified console.

Notes:

1. Refer to *Manifest Constants* on page 42 for the constants reference list.
2. This command allows the QREAD command to read messages from the printer queue.
3. The PORT command can be used to get the port number for an LU6.2 printer port. The MCC handles an LU6.2 printer port in the same way as any other printer port. Note that there may be more than one printer port per OS, for example, an LU6.2 printer port and a coax printer port.

Example:

```
%Port := PORT( OS)
%Port := PORT( CPU, "RS6000")
%Port := PORT( CPU)
%Port := PORT( OS, "Prod1")
```

See Also: ICONNAME, QREAD

QCLOSE

Syntax: QCLOSE(%QueueID)

Description: Closes a message queue.

Action: Closes the specified queue in a script by removing the current message pointer for the specified queue.

Parameters: **%QueueID**. Numeric expression. The id of the queue to close.

Returns: N/A.

Notes: Use **QCLOSE()** only on queues that were opened with **QOPEN()**.

Example:
```
QCLOSE( %QID)
```

See Also: QOPEN, QREAD, QSKIP

QOPEN

Syntax: QOPEN([%ObjIdArray]) ==> %QueueID

Description: Opens a new queue of OS printer console messages.

Action: Creates a new message queue in a script by creating a new current message pointer. The current message pointer is set to the end of the currently available messages—the next new message received will be the first message in the script's queue and therefore the next message retrieved for processing. Messages from the printer console for each OS ObjID specified in the array are placed in the queue represented by the QueueID return value.

Parameters: **%ObjIdArray**. Normal numeric array. Optional. An array of OS ObjIDs for which to create a message queue. Refer to *Object ID* on page 26 for more information. If not specified, a queue is created only for the OS that the script is running on.

Returns: Numeric value, as follows:

Value	Meaning
-1	An error occurred – see below.
Any other value	The numeric ID of the newly created queue pointer—a unique number used for subsequent access to the queue in the current script thread.

Use the manifest constant ERROR for clarity in your code (refer to *Manifest Constants* on page 42 for more information).

Possible Errors

An invalid array of object IDs. All object IDs must be created by the **OBJID**() command or the **OBJIDARRAY**() commands.

Notes:

1. The other queue commands utilize the queue id for their operations.
2. QOPEN() only creates a queue for OS printer console messages. If object IDs are specified, they must be object IDs for OSs. If object IDs are not specified, the script must be executing on an OS icon.
3. The MCC has one large message queue that holds all of the incoming printer console messages. The queue holds the most recent 10,000 messages. The oldest message is discarded when a new message is added (a circular message queue).
4. Each script can have multiple current message pointers into the message queue. The message pointers are created with **QOPEN**(), and advanced with **QREAD**() and **QSKIP**().
5. A queue ID is only valid within the script thread in which it was created. It becomes invalid in another script when a new thread is begun, such as with the **START**() command.
6. As with any array, the %ObjIdArray parameter can contain one or more elements. Having multiple object IDs enables a **QREAD**() command to automatically read the next message from any of the OSs specified in the **QOPEN**() command. Refer to the **QREAD**() command for more information.
7. The OS must have a valid printer defined. The Q series of

commands does not function with an OS that uses an RS232 or Telnet connection with the no_printer flag defined.

8. The maximum number of QOPEN commands that can be included in one script is 256.

Example:

```
//=================================================
//EXAMPLE 1: One input
//=================================================
//Read msgs from the OS the script is running on
%QID := QOPEN( )
IF %QID == ERROR        //if error
      LOG( LOG_EXEC, “ERROR: can’t open Queue”,
STATUS_ERROR)
ELSE
      //continue code here
ENDIF
RETURN
//=================================================
//EXAMPLE 2: Multiple inputs
//=================================================
//Read msgs from the Beta1 and AIX OSs
%OSArray[ 1] := OBJID( OS, ”3090:Beta1”)
%OSArray[ 2] := OBJID( OS, ”RS6000:AIX”)
//This queue will have msgs from Beta1 & AIX OS
%QueueID:= QOPEN( %OSArray)
IF %QID == ERROR        //if error
      LOG( LOG_EXEC, “ERROR: can’t open Queue”,
STATUS_ERROR)
ELSE
      //continue code here
ENDIF
RETURN
//=================================================
//EXAMPLE 3: Multiple inputs
//=================================================
// Using the OSIDArray, from the OBJID examples:
%QueueID := QOPEN( %OSIDArray)
IF %QID == ERROR        //if error
      LOG( LOG_EXEC, “ERROR: can’t open Queue”,
STATUS_ERROR)
ELSE
      //continue code here
ENDIF
RETURN
```

See Also: QCLOSE, QREAD, QSKIP

QPREVIEW

Syntax: QPREVIEW(%QueueID, $ResultArray) ==> %RetCode

Description: For non-mainframe printer queues, this command read or "previews" the text on the current line before it is added to the message queue (that is, before a carriage return is sent). The text returned is essentially a "snapshot" of the line on the screen as it is being built. It may be a complete message; alternatively, it may be an incomplete message, with more text being added after the snapshot is taken.

Action: The current line of the screen is read and the text is returned as a string. The message source (the OS name the message came from) becomes the key for the $ResultArray parameter, and the current previewed message becomes its value.

Parameters: **%QueueID**. Numeric expression. The message queue from which to read the message. This is the unique queue ID created by **QOPEN**().

$ResultArray. Associative string array. The array is associatively indexed with the message source (the OS name that the message came from), and assigned the current message pending on the console. Each OS in the %QueueID has an entry in this array.

Returns: String value in $ResultArray, and error code in %RetCode, as follows:

Value	Meaning
-1	An error occurred. The preview text was not read.
0	Success. The preview text was not read.

If %RetCode is -1, the ERRORNUM function returns one of the following values:

Error Code	Manifest Constant	Description
8	ERR_BADQUEUEID	The queue ID passed is invalid.
5200	ERR_QPREVIEW_COMMERROR	Iclrun could not communicate with Serial Manager.
5201	ERR_QPREVIEW_NORESPONSE	Iclrun did not receive responses from Serial Manager for all of the ports it was trying to preview.

Notes:

1. **QPREVIEW**() only reads messages from serial printer console queues created with the **QOPEN**() command.
2. If used on a mainframe printer console queue, **QPREVIEW**() returns an empty string in $ResultArray.
3. There is no guarantee that the expected prompt will be in the queue at the time **QPREVIEW**() is executed, due to timing and other host-related issues. Additional keys may need to be sent to the host to produce the desired prompt. (For instance, if you are waiting for a "Login:" prompt and a user is already logged in on the

console, you may wish to key “exit” before issuing your **QPREVIEW**().)

Example:

```
// Get the Object IDs for two systems
%OSArray[ 1] := OBJID( OS, ”DECUnix:Alpha”)
%OSArray[ 2] := OBJID( OS, ”RS6000:Beta”)
// Open the message queue
%QueueID := QOPEN( %OSArray)
%Preview := QPREVIEW( %QueueID, $MsgArray)
// If the DECUnix system is at the “Login:” prompt and
the
// RS6000 is at a root (“#”) prompt, the values of
// $MsgArray will be:
// $MsgArray[ “Alpha” ] ==> Login:
// $MsgArray[ “Beta” ] ==> #
```

See Also: QCLOSE, QOPEN, QREAD, QSKIP, SCANB, SCANP

QREAD

Syntax: QREAD(%QueueID, $MsgArray, %Wait[, $Filter]) ==> $MsgLine

Description: Reads the next message from a message queue.

Action: The next message in the specified queue is read. The message is defined by the current message pointer for the message queue created by the **QOPEN**() command.

The entire "raw" message is returned as a string. The message source (the name of the OS that the message came from) and the entire message are placed in the normal string array specified by the $MsgArray parameter.

When the queue is empty, **QREAD**() acts as specified by the Wait parameter.

Parameters: **%QueueID**. Numeric expression. The message queue from which to read the next message. This is the unique queue ID created by **QOPEN**().

$MsgArray. Normal string array. The array to populate with the message source (the OS name from which the message came) and each word from the message. The first array element contains the message source. Each subsequent element contains a word from the message, sequentially by position in the message. For example:

$MsgArray[1] contains the message source

$MsgArray[2] contains the first word of the message

$MsgArray[3] contains the second word of the message

$MsgArray[4] contains the third word of the message etc.

The length of the array (the number of elements) is determined by the number of words in the message. The words are split by "white space"—one or more space characters; **SPLIT**() works in the way, with the split characters parameter not specified.

%Wait. Numeric expression. The number of seconds to wait before timing out. When the queue is empty, this number specifies how long the command waits for a message before returning a null string ("").

$Filter. String expression. Optional. The text in this string acts as filter criteria—the criteria text must be in the message in order for **QREAD**() to receive it. Messages will only "appear" to **QREAD**() if they contain the criteria text. A Filter value of null string "" is the same as omitting the Filter parameter.

Returns: String value. The complete original message.

Notes:

1. **QREAD**() only reads messages from printer console queues, queues created with the **QOPEN**() command.
2. Although platforms such as Unix and AS/400 do not have mainframe-style printer consoles, the MCC internally creates a "printer console-like" facility for non-mainframe platforms. This facility is a sequential message queue of the messages received from the OS consoles. To the **QREAD**() command, this is the same queue as created from the mainframe's printer consoles.

   For the message from the OS console to appear in the printer message queue, the message must be completed with a newline or carriage return first. For example, when logging into a Unix system, the user is prompted with "login:". This prompt is on the screen but not yet in

the printer message queue. It is placed in the printer message queue after the user presses <ENTER>.

3. Incomplete messages are not available to **QREAD**().
4. Each message pointer can be moved with **QSKIP**().
5. If the OS Console is an RS232 or Telnet connection, and was defined with the no_printer flag, this command will **not** work with that OS. See **QOPEN**() for more details.

Example:

```
//=============================================
// Example 1
//=============================================
%Wait := 180
%QID := QOPEN()         //open queue on current OS
WHILE TRUE              //repeat process continually
      $Msg := QREAD( %QID, $MsgArray, %Wait)
      IF $Msg == ""     //if null, then no msg was
      received
      //the building of the $Temp string to display
      //on the log is done this way here simply due to
      //width restrictions of the printed page
      //Normally, build it on one line or place the
      //entire string in the LOG command
            $Temp := "No messages in " + STR( %Wait)
            $Temp := $Temp + " seconds on OS: "
            $Temp := $Temp + $MsgArray[ 1])
            LOG( LOG_FLT, $Temp, STATUS_WARNING)
      ENDIF
                        //check first word of msg for IO
                        // err $MsgArray[ 1] will always
                              //have OS name
      IF $MsgArray[ 2] == "IOS000I"
      //call IO err handling script passing OS name
            IOERROR( $MsgArray[ 1])
      ENDIF
ENDWHILE
//=================================================
// Example 2
//=================================================
%Wait := 30
%ObjIDSys5[ 1] := OBJID( OS, "SYS5")
%QIDSys5 := QOPEN( %ObjIDSys5)
$Msg := QREAD( %QIDSys5, $MsgArray, %Wait, "error")
IF $Msg == ""         //if null, then no msg was received
      $Temp := "No msgs for " + STR( %Wait)
      $Temp := $Temp + " seconds: " + $MsgArray[ 1]
      LOG( LOG_FLT, $Temp, STATUS_WARNING)
ELSE
      LOG( LOG_FLT, $Msg, STATUS_ERROR)
ENDIF
```

See Also: QCLOSE, QOPEN, QPREVIEW, QSKIP, PORT, SCANB, SCANP

QSKIP

Syntax: QSKIP(%QueueID, %Skip)

Description: Moves a current message pointer for a queue.

Action: The current message pointer for the specified message queue moves according to the %Skip parameter.

Parameters: **%QueueID**. Numeric expression. The message queue that the message to skip is in. It is the unique queue ID created by **QOPEN**().

%Skip. Numeric expression. The amount of messages to skip.

Skip Constant. Description.

SKIPNEXT. Moves the message pointer to the next full message. Used only to skip through the rest of a multi-line message.

SKIPEND. Moves the current message pointer to the end of the messages, thereby quickly skipping any remaining messages (unread with **QREAD**()) in the script's queue. This is the same as executing:

```
QCLOSE( %QueueID)
%QueueID := QOPEN(.)
```

Returns: N/A.

Notes: N/A

Example:

```
QSKIP(%QID, SKIPNEXT)   //skips remainder of multiline
                        //message
QSKIP(%QID, SKIPEND)    //skips all remaining messages
                        //in the message queue
```

See Also: QCLOSE, QOPEN, QPREVIEW, QREAD

REPEAT

Syntax:

REPEAT...UNTIL
    REPEAT
        Commands
    UNTIL Expression

Description: Repeats a sequence of commands until an expression evaluates to TRUE.

Action: The commands in the REPEAT block are repeatedly executed in sequence until the expression evaluates to TRUE.

Parameters: **Expression**. Boolean expression. The expression to evaluate that determines whether or not to continue looping. Refer to *Boolean* on page 50 for more information.

Returns: N/A.

Notes:

1. The commands in the REPEAT block always execute at least once because the expression is evaluated at the end of the loop.
2. The number of nested REPEATs is unlimited.
3. The number of commands allowed within the REPEAT block is unlimited.

Example:

```
%Num := 1
REPEAT
      INC %Num
UNTIL %Num > 10
```

See Also: WHILE

REPSTR

Syntax: REPSTR($String, %Count) ==> $RepeatedString

Description: Returns a string repeated a specified number of times.

Action: The String expression is repeated the number of times specified by "Count", and is returned as a one-character string.

Parameters: **$String**. String expression. The character string to repeat.

%Count. Numeric expression. The number of times to repeat String.

Returns: String value. The String repeated the number of times specified by "Count".

Notes: N/A

Example:

```
$Str := REPSTR( "*", 5)       //RESULT: "*****"
$Str := REPSTR( "Hi", 3)      //RESULT: "HiHiHi"
$Str := REPSTR( "Hi ", 3)     //RESULT: "Hi Hi Hi "
```

See Also: ATSTR, FINDSTR, LEFTSTR, RIGHTSTR, STR, SUBSTR

RETURN

Syntax: RETURN [Expression]

Description: Returns execution to the calling routine, passing an optional return value.

Action: Execution of the current script stops, and resumes with the next statement in the calling routine— either from a GOSUB command or from a script call. All variables in a script are released when execution is stopped. The optional Expression parameter is evaluated and returned only to a calling script. The calling script can assign the return value to a variable for further usage.

Parameters: **Expression**. Numeric or string expression. Optional. Evaluates to the value returned to the calling script. If not specified, the value is null string "".

Returns: N/A.

Notes:
1. The RETURN command returns to calling routines in a "last in, first out" order.
2. An error occurs if the RETURN command is used when no calling routines exist.

Example:
```
N/A
```

See Also: GOSUB

RIGHTSTR

Syntax: RIGHTSTR($String, %Count) ==> $SubStr

Description: Returns the right-most specified number of characters of a string expression.

Action: A substring is extracted from a string expression beginning with the last character in the string expression.

Parameters: **$String**. String expression. The character string from which to extract characters.

%Count. Numeric expression. The number of characters to extract. If Count is negative or zero, RIGHTSTR() returns an empty string “”. If Count is larger than the length of String, RIGHTSTR() returns the entire String.

Returns: String value. The rightmost Count characters of String as a character string.

Notes: N/A

Example:

```
$Var := RIGHTSTR( $Msg, 4)
```

See Also: ATSTR, FINDSTR, LEFTSTR, REPSTR, STR, SUBSTR

SCANB

Syntax: SCANB(%Port, $Text, *Found)

Description: Searches an OS console for a specified character string. Used with the BLOCKSCAN() command.

Action: BLOCKSCAN delineates the beginning and ENDBLOCK delineates the end of a group of SCANB commands to execute as a group. The SCANB commands execute simultaneously, and script execution continues with the branching logic of the first SCANB command that fulfils its own scanning condition. If the Wait time expires, script execution branches to the label specified by the *Timeout parameter.

Parameters: **%Port**. Numeric expression. The assigned console port number to scan. Refer to the *Overview* chapter for more information.

$Text. Regular expression. The text expression to scan for in the console. Refer to *Regular Expressions* on page 54 for more information.

***Found**. Label literal. The label to jump to when the Text parameter is found within the time limit specified by the Wait parameter in the **BLOCKSCAN()** command.

Returns: N/A

Notes:

1. QREAD() is the preferred method of scanning a console for messages. Use SCANB() only if the desired messages are not coming out of a printer console.
2. The position of the scan text on the console and character attributes (for example, only highlighted characters) cannot be specified.

Example:

```
//*********************************************
// Example 1
//*********************************************
*START:
      BLOCKSCAN(1800, *START, $Msg)
            SCANB( 1, "JOBA END", *JOBA)
            SCANB( 2, "JOBB END", *JOBB)
ENDBLOCK
//*********************************************
// Example 2
//*********************************************
*START:
      LOG( LOG_FLT, "JOB A–C REPLY SCAN")
      WAITFOR( 10)
      BLOCKSCAN(20, *START, $Msg)
      //put 2 digit job # in first cell of $Msg array
            SCANB( 2, "\*[0-9][0-9] JOB-A", *JOBA)
            SCANB( 2, "\*[0-9][0-9] JOB-B", *JOBB)
            SCANB( 2, "\*[0-9][0-9] JOB-C", *JOBC)
      ENDBLOCK
*JOBA:
      //send job # in pp with reply
      KEY( 2, "R " + $Msg[1] + ",CANCEL[ENT]")
```

See Also: BLOCKSCAN, KEY, QREAD, SCANP

SCANP

Syntax: SCANP(%Port, $Text, %Wait, *Found[, $Array])

Description: Searches an OS console for a specified character string.

Action: A console is scanned for a regular expression match. Script execution can branch depending on whether the scan was successful within a specified time limit. If found within the number of seconds specified by the Wait parameter, script execution continues with the next statement. If not found within the number of seconds specified by the Wait parameter, script execution jumps to the label specified by the *Found parameter.

Parameters:

%Port. Numeric expression. The assigned console port number to scan. Refer to the *Overview* chapter for more information.

$Text. Regular expression. The text expression to scan for in the console. Refer to *Regular Expressions* on page 54 for more information.

%Wait. Numeric expression. The number of seconds to wait before timing out when the Text parameter is not found. This number specifies how long the command waits for the scan to be successful.

***Found**. Label literal. The label to jump to when the Wait time expires.

$Array. Normal string array. Optional. The array to populate with subexpression results, if any, from the Text parameter. Each array element will contain one subexpression result—element one will hold the result for subexpression one, element two will hold the result for subexpression two, and so on. The subexpressions are "numbered" from left to right in the Text parameter. $Array is only populated when the Text parameter contains subexpressions and the scan text is found. Only the first nine subexpression results can be returned.

Returns: N/A.

Notes:

1. QREAD() is the preferred method of scanning a console for messages. Use SCANP() only if the desired messages are not coming out of a printer console.
2. The position of the scan text on the console and character attributes (for example, only highlighted characters) cannot be specified.

Example: See BLOCKSCAN command.

See Also: BLOCKSCAN, KEY, QREAD, SCANB

SCRIPTCANCEL

Syntax: SCRIPTCANCEL($ScriptName,$Class,$Name)

Description: Obtains the system integer file descriptor from a file handle.

Action: Cancels all active scripts that match the given input.

Parameters: $ScriptName: String. The name of the script (file name) or "*" as a wildcard to match all names.

$Class: String. Class name passed to script or "ALL" as a wildcard to match all classes. Typical class names are: ROOM, CPU, OS, UNIT. Refer to the <Icon Class/Icon Name> section for more information.

$Name: String. The icon name passed to the script or "*" as a wildcard to match all names. Refer to the <Icon Class/Icon Name> section for more information.

Returns: Numeric value, as follows:

-1 = Error occurred (invalid input parameters)

0 = Success

Example:

```
$OSName := ICONNAME(OS)
SCRIPTCANCEL("timeralert", "CPU", $OSName)
```

See Also: EXEC, START

SCRIPTGETACTIVE

Syntax: SCRIPTGETACTIVE($AssocArray) ==> %ErrCode

Description: Retrieves information on all active scripts into an associative array.

Action: Information on all active scripts are put into the $AssocArray, with each script being a key in the array.

Parameters: $AssocArray. Associative array. The array into which to put the script information.

Returns: A manifest constant indicating the status of the operation, as follows:

Manifest Error Constant	Value	Associated Information
Err_None	0	No error, array has been filled with appropriate values.
	-1	Error communicating with the script manager daemon. No values set.

Notes: The SCRIPTGETACTIVE routine returns an array of values as shown below:

variable	sample value	description
$ScriptInfo[1]	4	Unique script number
$ScriptInfo[2]	9669	Host system process identifier (i.e. PID)
$ScriptInfo[3]	1	Script class value - manifest constant 1 – Room 2 – CPU 3 – Chan (i.e. Channel) 4 – Unit 6 – OS 7 – SW (i.e. Software/Application) 255 – All
$ScriptInfo[4]	236	Current script source line number
$ScriptInfo[5]	0	Nonzero => in scan wait Gives the number of seconds defined in the

		current WAIT command
$ScriptInfo[6]	0	Number of seconds which remain for the current WAIT command
$ScriptInfo[7]	0	Echo status
		value - manifest constant
		0 - Off
		1 - On
$ScriptInfo[8]	3	Status
		value - meaning
		0 - inactive
		1 - wait
		2 - hold
		3 - active
		4 - pause
		5 - end
		6 - canceled
$ScriptInfo[9]	ROOM	Class name passed to script
$ScriptInfo[10]	MCC	Object name passed to script
$ScriptInfo[11]	mccstars	Script name
$ScriptInfo[12]	SYSEXEC	Command being executed
$ScriptInfo[13]	26229	Number of instructions executed
$ScriptInfo[14]	3	Interpreter running script (i.e. initiator)
$ScriptInfo[15]	1183667241	System time when this script information was last updated (given in system standard units of seconds since midnight UTC January 1, 1970)

Examples: Simple Script to log all of the script information to the filtered message log:

```
%rc := SCRIPTGETACTIVE($AssocArray)
IF %rc == 0%
Num := ALEN($AssocArray) // Get the number of scripts
ASSOCKEYS( $AssocArray, $NormalArray) // Put into a normal
array
%Index := 1
WHILE %Index <= %Num // Log individual script info
$Seq := $NormalArray[%Index]
$AlertInfo := $AssocArray[$Seq]
LOG(LOG_FLT, "Script info is " + $AlertInfo)
INC %Index
ENDWHILE
ENDIF
```

Simple Script to break down all of the component parts of the returned script array and log everything to the filtered message log:

```
%rc := SCRIPTGETACTIVE( $Array )
LOG( LOG_FLT, "There are currently " + ALEN($Array) + \"
active scripts." , 5)
ASSOCKEYS( $Array, $NormalArray) // Put into a normal array
%ArrayIndex := 1
WHILE (%ArrayIndex <= ALEN($Array))
  SPLIT($ScriptInfo, $Array[$NormalArray(%ArrayIndex)], " ")
  LOG(LOG_FLT, "Script #: " + STR(%ArrayIndex), 1)
  LOG(LOG_FLT, $ScriptInfo[1] + " : " + $ScriptInfo[2], 5)
  LOG(LOG_FLT, $ScriptInfo[3] + " : " + $ScriptInfo[4], 5)
  LOG(LOG_FLT, $ScriptInfo[5] + " : " + $ScriptInfo[6], 5)
  LOG(LOG_FLT, $ScriptInfo[7] + " : " + $ScriptInfo[8], 5)
  LOG(LOG_FLT, $ScriptInfo[9] + " : " + $ScriptInfo[10], 5)
  LOG(LOG_FLT, $ScriptInfo[11] + " : " + $ScriptInfo[12], 5)
  LOG(LOG_FLT, $ScriptInfo[13] + " : " + $ScriptInfo[14], 5)
  LOG(LOG_FLT, $ScriptInfo[15], 5 )
  LOG(LOG_FLT, "----------------------------------------",
1)
  INC %ArrayIndex

ENDWHILE
RETURN
```

See Also: Executing Scripts

SCRNTEXT

Syntax: SCRNTEXT(%Port, %Start, %Length) ==> $Text

Description: A full or partial screen snapshot. Returns characters from a console screen.

Action: The characters beginning at absolute position specified by "Start", and continuing for the specified "Length" on the console represented by "Port" are returned as a string. In effect, a screen snapshot is taken of the specified area.

Parameters: **%Port**. Numeric expression. The assigned console port number. Refer to *Ports* on page 24 for more information.

%Start. Numeric expression. The absolute position to begin copying characters from the console. The minimum value is 1 and the maximum value is the console's row quantity * its column quantity.

%Length. Numeric expression. The quantity of characters to copy from the console. The minimum value is 1 and the maximum value is the console's row quantity * its column quantity less the Start parameter.

Returns: String value. The characters copied from the console.

Notes: **SCRNTEXT()** is not intended for capturing the printer console text. Use the **QREAD()** command for reading the printer console.

Example: N/A

See Also: ASCRN

SECONDS

Syntax: SECONDS() ==> %EpochSeconds

Description: Returns the time value for the current time.

Action: The epoch seconds for the current system time is returned.

Parameters: N/A.

Returns: Numeric value. The number of seconds past the epoch for the current system time.

Notes:
1. Refer also to the description of *Date/Time* on page 40.
2. To convert the returned value into a date or time, use the **TIMESTR()** command.

Example:
```
%Now := SECONDS()
```

See Also: DATE, TIME, TIMESTR, WAITFOR, WAITUNTIL

SET

Syntax: SET Variable := Expression

Description: Make the contents of a variable equal to the specified expression.

Action: The variable is made equal to the value of the evaluated expression.

Parameters: **Variable**. Numeric or string variable. The variable to receive the value. If the expression evaluates to a numeric value, a numeric variable must be specified; likewise for a string.

Expression. Numeric or string expression. The evaluated value to place in the variable.

Returns: N/A.

Notes: SET is an optional keyword. The following statements have the same function:

```
SET %Var := 22
%Var := 22
```

Example:

```
SET %Num := 200
SET $Str := "ABCDEFG"
%Num := VAL( $Str)            // 0
$Str := RIGHTSTR( $Str, 1)    // "G"
SET $Array[ 4] := "IEF"
```

See Also: DEC, INC

SNMP_GET

Syntax: SNMP_GET($Alias, $MIBOID) ==> $Value

Description: Retrieves the value of a specified MIB object.

Action: The value of the object specified by the MIBOID parameter in the MIB of the Alias is returned as a string.

Parameters: **$Alias**. String expression. The NMS alias from which to get the MIB object value. The specified alias must not be a group alias. Refer to *NMS Alias* on page 31 for more information.

$MIBOID. String expression. The MIB Object ID to get the value for. Refer to *MIB OID* on page 31 for more information.

Returns: String value, as follows:

Value	Meaning
"" (null string)	The MIB object does not exist.
Any other value	The value of the MIB object.

Notes:

1. Append ".0" to the MIB Name. Alternatively, use the **SNMP_GET()** command to enter a branch of the MIB, and the **SNMP_GETNEXT()** command to retrieve specific values from that branch.
2. For error checking of **SNMP_GET()**, refer to the description of the **ERRORNUM()** command.

Example:

```
//This sample code enters the sysName branch and grabs
// the value of sysName.
//Note that the alias must be defined in SNMP Setup.
$Alias := "RS6000"
$MIBOID := "sysName"
$RetVal := SNMP_GET( $Alias, $MIBOID)
$SystemName := SNMP_GETNEXT( $Alias, $MIBOID,
$NextMIBOID)
//$SystemName now contains the value of sysName.
//Example2
//This gets the value of sysName by appending a ".0"
// to the end of the branch name.
$Alias := "RS6000"
$MIBOID := "sysName"
$MIBOIDValue := $MIBOID + ".0"
$SystemName := SNMP_GET( $Alias, $MIBOIDValue)
//$SystemName now contains the value of sysName.
```

See Also: SNMP_GETNEXT, SNMP_GETTABLE, SNMP_SET, SNMP_TRAPSEND

SNMP_GETNEXT

Syntax: SNMP_GETNEXT($Alias, $MIBOID, $NextMIBOID) ==> $Value

Description: Retrieves the value and the Object ID of the next logical MIB object to a specified MIB object.

Action: The value of the next logical object to the object specified by the MIBOID parameter in the MIB of the Alias is returned as a string. Additionally, the MIB object id of the next object is returned in the NextMIBOID parameter. In other words, given the current object id in MIBOID, this command retrieves the next MIBOID from the specified alias. This allows the user to "walk" the MIB.

Parameters: **$Alias**. String expression. The NMS alias to get the MIB object value from. The specified alias must not be a group alias. Refer to *NMS Alias* on page 31 for more information.

$MIBOID. String expression. The MIB object id for which to get the value. Refer to *MIB OID* on page 31 for more information.

$NextMIBOID. String value. The MIB object id of the next logical MIB object from the specified MIB object (specified by the MIBOID parameter).

Returns: String value, as follows:

Value	Meaning
"" (null string)	The MIB object does not exist.
Any other value	The value of the MIB object.

Notes:

1. Use the **SNMP_GET()** command to enter a branch of the MIB, and the **SNMP_GETNEXT()** command to retrieve specific values from that branch.
2. For error checking of **SNMP_GETNEXT()**, refer to the description of the **ERRORNUM()** command.

Example: Refer to the example for SNMP_GET.

See Also: SNMP_GET, SNMP_GETTABLE, SNMP_SET, SNMP_TRAPSEND

SNMP_GETTABLE

Syntax: SNMP_GETTABLE($Alias, $MIBOID, $TableArray[, $Delimiter]) ==> %ReturnCode

Description: Retrieves a complete table of information from the MIB on the specified agent.

Action: Given an OID that points to a table-type piece of data, this function returns the SNMP MIB table in the TableArray parameter. Each row in the MIB table will be stored as one element in the array with the columns separated by the Delimiter parameter.

Parameters:

$Alias. String expression. The NMS alias to get the MIB object values from. The specified alias must not be a group alias. Refer to *NMS Alias* on page 31 for more information.

$MIBOID. String expression. The MIB object ID for which to get the values. Refer to *MIB OID* on page 31 for more information.

$TableArray. Associative string array. The array that will be populated with the table data.

$Delimiter. String expression. Optional. A string of separator characters to be used to separate columns in each array element. This is the character(s) used by the SPLIT() command to break the data apart into fields. If not specified, it defaults to "^ | ^" (a very unique sequence of characters).

Returns: Numeric value, as listed below:

Return Code	Description
0	Success
-1	Agent not defined in SNMP Setup
-2	Invalid Argument
-3	Agent not found
-5	Unable to open configuration file
-6	Internal memory allocation error. Contact Visara Technical Support immediately
-7	Bad MIB
-8	Configuration file error

Notes:

1. Do not use **SNMP_GETTABLE()** to retrieve an entire MIB, as this may overwhelm the MCC. Instead, use **SNMP_GETTABLE()** to retrieve specific branches and leaves from the MIB.
2. For information on error checking of **SNMP_GETTABLE ()**, refer to the **SNMP_GETTABLE()** command.
3. The associative array includes an element called "_header_" that can be used to specify the name of each column (field) in the table to retrieve. The header must exist in the MIB.

Example:

```
// gets the table at “interfaces” from the named host
//  alias
$Alias := “Galileo”
$MIBOID := “interfaces”
%result := SNMP_GETTABLE( $Alias, $MIBOID, $TableArray)
```

See Also: SNMP_GET, SNMP_GETNEXT, SNMP_SET, SNMP_TRAPSEND

SNMP_SET

Syntax: SNMP_SET($Alias, $MIBOID, $Value) ==> %ReturnCode

Description: Sets the value of a specified MIB object.

Action: The value of the object specified by the MIBOID parameter in the MIB of the Alias is set to the value specified by the Value parameter.

Parameters: **$Alias**. String expression. The NMS alias to set the MIB object value on. The specified alias must not be a group alias. Refer to *NMS Alias* on page 31 for more information.

$MIBOID. String expression. The MIB object ID for which to set the value. Refer to *MIB OID* on page 31 for more information.

$Value. String expression. The MIB object is set to this value.

Returns: Numeric value, as follows:

Return Code	Description
0	Success
-1	Agent not defined in SNMP Setup
-2	Invalid Argument
-3	Agent not found
-5	Unable to open configuration file
-6	Internal memory allocation error. Contact Visara Technical Support immediately
-7	Bad MIB
-8	Configuration file error

Notes:

1. For information on error checking of **SNMP_SET()**, refer to the description of the **ERRORNUM()** command.
2. Note that valid MIB branch names must have ".0" appended to them to reference the value to which they refer.

Example:

```
//sets the value of "sysLocation" in the MIB on the
RS6000
//to "down"
$Alias := "RS6000"
$MIBOID := "sysLocation.0"
%Result := SNMP_SET( $Alias, $MIBOID, "Ops Center")
```

See Also: SNMP_GET, SNMP_GETNEXT, SNMP_GETTABLE, SNMP_TRAPSEND

SNMP_TRAPSEND

Syntax: SNMP_TRAPSEND($Alias, %TrapNum[, %EntNum [, $MIBOID] [, $VARBINDS]]) ==> %ReturnCode

Description: Sends a trap to one or more hosts.

Action: An SNMP trap is sent to the agent(s) represented by the Alias parameter.

Parameters: **$Alias**. String expression. The NMS alias to send the trap to. Any alias may be specified, including a group alias. If a group alias is specified, the same trap is sent to each agent in the alias group. Refer to *NMS Alias* on page 31 for more information.

%TrapNum. Numeric expression. The number of the SNMP trap to send. Possible trap numbers are listed below:

Trap Number	Description
0	Cold Start
1	Warm Start
2	Link Down
3	Link Up
4	Authentication
5	EGP Neighbor Loss
6	Enterprise

%EntNum. Numeric expression. Optional. The enterprise number for enterprise traps. This parameter is ignored if the trap number is not an enterprise trap. It is not required for enterprise traps, but if not specified, the enterprise number will be 0.

$MIBOID. String expression. Optional. The MIB OID to be specified in the trap. If not specified, the default is 0.0.0.0. Refer to *MIB OID* on page 31 for more information.

$VARBINDS. String expression. Optional. Specifies variable bindings, which must exist in the MIB. See Example 2 following, which specifies variable bindings called "$trapBinds".

Returns: Numeric value, as listed below:

Return Code	Description
0	Success
-1	Error

Notes:

Example:

```
************************************
Example 1
************************************
$Alias := “RS6000”
%Return := SNMP_TRAPSEND( $Alias, 6, 215)
// sends an enterprise trap to an NMS aliased to “RS6000”
// using enterprise number 215
************************************
Example 2
************************************
// Try sending a trap
$trapBinds["sysContact"] := "Fred Flintstone"
$trapBinds["sysDescr"] := "Mail Server"
%retV := SNMP_TRAPSEND("mothra", 6, 1, "snmpDot3RptrMgt",
$trapBinds)
```

See Also: SNMP_GET, SNMP_GETNEXT, SNMP_GETTABLE, SNMP_SET

SPLIT

Syntax: SPLIT($Array, $String, $Delimiter)

Description: Populates an array with the fields of a string delimited by a string.

Action: String is divided into fields by the Delimiter expression. Each field, from left to right in String, is placed into its corresponding array element in $Array. The first field is placed in $Array[1], the second field is placed in $Array[2], and so on.

Parameters: **$Array**. String normal array variable. The array to populate with the field results.

$String. String expression. The string to split into fields.

$Delimiter. String expression. The string pattern (one or more characters) that divides the String into fields. The Delimiter expression result characters will not be a part of any field. A single space is a special delimiter that causes a contiguous sequence of spaces in String to be reduced to one space before splitting. This special handling eases the manipulation of host messages that often have multiple spaces between words.

Returns: N/A.

Notes: Determine the number of fields created with the **ALEN()** function.

Example:
```
SPLIT( $Array, $Text, ":")
```

See Also: JOIN

START

Syntax: START(ScriptName(Parms)[, %Class[, $Name]])

Description: Initiates execution of another script for concurrent processing.

Action: The specified script executes in parallel with other script(s) that are currently running (multitasking).

Parameters: **ScriptName**. Script name, literal, or a string expression (including string variables). Name of the script to execute. For a literal, explicitly state the name of the script to execute. For a string expression, the name of the script is derived from the evaluation of the string expression, e.g. "myscript" or $MyScript. The parentheses after the literal or variable are required; they may optionally contain parameters (numeric or string variables or arrays) to pass to the script.

%Class. Numeric expression. Optional. The icon class. Refer to *Icon Class/Icon Name* on page 29 for more information.

$Name. String expression. Optional. The icon name. Refer to *Icon Class/Icon Name* on page 29 for more information.

Returns: N/A

Notes:

1. Refer to *Manifest Constants* on page 42 for the constants reference list.
2. If Class and Name are not specified, the new script will default to the same Class and Name as the current script.
3. Script names are case sensitive, so that calling "MYSCRIPT" is not the same as calling "Myscript". We recommend using all lower case characters for script names, so that the name of the script is the same as the name of the physical script file on disk. In that case, a call to "myscript" actually calls a script on the disk named myscript.scx).

Example:

```
//Example showing the script name as a literal
START( OtherScr( %Num, $Str, $StrArray))
LOG( LOG_FLT, "Simultaneous operation started")
//Example showing the script name as a literal
START( IPLSYSA())
//Example showing the script name in a string variable
$ScriptName := "IPLSYSA"
START( $ScriptName())
```

See Also: EXEC, STOP

STOP

Syntax: STOP(ScriptName[, %Class [, $Name]])

Description: Halts execution of a script.

Action: Execution of the specified script is canceled.

Parameters: **$ScriptName**. String expression. Name of the script to cancel execution. Use an asterisk "*" to specify all scripts.

%Class. Numeric expression. Optional. The icon class. Refer to *Icon Class/Icon Name* on page 29 for more information. Use ALL to specify all classes.

$Name. String expression. Optional. The icon name. Refer to *Icon Class/Icon Name* on page 29 for more information. Use an asterisk "*" to specify all icon names.

Returns: N/A

Notes:

1. Refer to *Manifest Constants* on page 42 for the constants reference list.
2. If Class and Name are not specified, the script to cancel must be executing on the same Class and Name as the script that contains the **STOP()** command.
3. The script that contains the **STOP()** command will not be canceled.

Example:

```
//cancel the "MVSIPL" script
STOP( "MVSIPL")
//cancel all 3090 CPU scripts
STOP( "*", CPU, "3090")
//cancel all scripts everywhere
STOP( "*", ALL, "*")
```

See Also: START

STR

Syntax: STR(%Number) ==> $String

Description: Converts a numeric expression to a string.

Action: The numeric expression is returned as a character string.

Parameters: **%Number**. Numeric expression. The numeric expression to convert to a character string.

Returns: String value. The number formatted as a character string.

Notes: N/A

Example:
```
$Var := STR( 4)          // the same as $Var := “4”
$Var := STR( %Num)
```

See Also: ATSTR, FINDSTR, LEFTSTR, REPSTR, RIGHTSTR, SUBSTR, VAL

SUBSTR

Syntax: SUBSTR($String, %Start[, %Count]) ==> $SubStr

Description: Extract a substring from a character string.

Action: A substring is extracted from a string expression, beginning with the specified character position in the string expression and taking the specified number of characters.

Parameters: **$String**. String expression. The character string from which to extract characters.

%Start. Numeric expression. The starting character position in String. If Start is positive, the starting position is relative to the leftmost character in String. If Start is negative, the starting position is relative to the rightmost character in String. If Start is zero, a runtime error occurs.

%Count. Numeric expression. Optional. The number of characters to extract. If Count is greater than the number of characters from Start to the end of String, the extra is ignored. If Count is not specified, Count defaults to the number of characters from Start to the end of String.

Returns: String value. The specified substring.

Notes: N/A

Example:
```
$Var := SUBSTR( $Msg, 4, 3)
$Var := SUBSTR( $First + $Second, 6, 8)
```

See Also: ATSTR, FINDSTR, LEFTSTR, REPSTR, RIGHTSTR, STR, SUBSTR,

SWITCH

Syntax: SWITCH...CASE...[DEFAULT...]ENDSWITCH

SWITCH Expression
CASE SimpleExpr:
[commands]
[BREAK]
[CASE SimpleExpr:]
[commands]
[BREAK]
[DEFAULT:]
[commands]
[BREAK]
ENDSWITCH

Description: Execute command(s) based on the value of an expression.

Action: Each CASE Simple Expression is evaluated and compared (one by one in the listed order) to the Expression.

- If the comparison evaluates to TRUE, the command(s) for that CASE statement are executed.
- If the BREAK statement is not present in the CASE command block, the commands in all subsequent CASE command blocks are also executed (regardless of each subsequent CASE statement as they are not evaluated). Execution continues until a BREAK or ENDSWITCH statement is encountered.
- If no comparison evaluates to TRUE, then, if present, the DEFAULT command statements are executed.

Parameters: **Expression**. Numeric or string expression. The evaluated expression to compare to each CASE statement.

SimpleExpr. Numeric or string simple expression. A numeric or string literal or variable compared to the expression.

DEFAULT. Keyword. Optional. The command block that is executed when none of the comparisons matched.

Returns: N/A.

Notes:

1. The expression type that each expression evaluates to must match the variable type with which it is being compared.
2. Once execution of commands in a CASE statement has begun, the BREAK statement ends it and prevents the statements in the next CASE block from being executed. No other statement ends execution.
3. DEFAULT, if specified, must be the last case statement.
4. The maximum number of CASE statements per SWITCH command is unlimited.
5. The maximum number of nested **SWITCH()** commands is 256.

Example:

```
SWITCH $Msg[ 3]
      CASE “IOS000I”:
            //call the IOS000 script to handle
            IOS000( $Msg[ 4], $Msg[ 5])
            BREAK
      CASE “IOS050I”:
            //call the IOS050 script to handle
            IOS050( $Msg[ 4], $Msg[ 5])
            BREAK
      CASE “IOS060I”:
      CASE “IOS070I”:
      CASE “IOS080I”:
            //call the IOS067 script to handle
            IOS0678( $Msg[ 4], $Msg[ 5])
            BREAK
      DEFAULT:
            LOG( LOG_FLT, “OTHER MESSAGE”)
            BREAK
ENDSWITCH
SWITCH %Index
      CASE 0:
            //call the IOS000 script to handle
            DoThisNow( $Msg[ 4], $Msg[ 5])
            BREAK
      CASE 1:
            //call the IOS050 script to handle
            DoThatNow( $Msg[ 4], $Msg[ 5])
            BREAK
      CASE 2:
      CASE 3:
      CASE 4:
            //call the IOS067 script to handle
            IOS0678( $Msg[ 4], $Msg[ 5])
            BREAK
      DEFAULT:
            LOG( LOG_FLT, “OTHER MESSAGE”)
            BREAK
ENDSWITCH
```

See Also: IF

SYSEXEC

Syntax: SYSEXEC($String) ==> %Return

Description: Executes a Unix command on the MCC host system with parameters.

Action: The string is executed as a system command by passing it to the Unix system. Script execution waits until the system command completes execution and gives a return value.

Parameters: **$String**. String expression. The system command and any parameters.

Returns: Numeric Value. %Return. The exit value of the executed command, as follows:

Return Code	Meaning
0	Success
-1	Permission Denied
2	Syntax Error
127	Not Found

If the command is successful, the return code is the same value as a shell script would return for the same command. The value is obtained from the Unix command line program.

Notes:

1. SYSEXEC can be used to play a sound file on an X-term, if the X-term has a sound card installed and the appropriate software is installed on the MCC server. To determine the name of an X-term, compare its IP address to the entries in /etc/hosts. Refer to the second example below.
2. Any program, such as a Unix shell script, started from the SYSEXEC command must be stopped in the reserved script #SHUTDN.SCR. If it is stopped in any other way, restarting the MCC may result in multiple copies of the same script, causing unexpected results.

Example:

```
//This example creates a file named /usr/home/ics/files
//from the contents of the directory /usr/isc/script
$Command := “ls –1 /usr/ics/script > /usr/home/ics/files”
%Return := SYSEXEC($Command)
//This example plays the sound ‘godzilla.au’ on the X-
term
//named todds_x-term. The command is broken into 4
//strings for readability.
$Command := “/usr/tekxp/bin/AlphaAXP_OSF1/xpsh -display ”
$Target := “todds_x-term”
$Flags := “:0.0 aplay -p -a NFS -f “
$Soundfile := “/usr/ics/audio/godzilla.au”
%Return := SYSEXEC($Command + $Target + $Flags + $Soundfile)
```

See Also: N/A

TEMP

Syntax: TEMP(%Port) ==> %Temp

Description: Reads the current temperature from a sensor unit.

Action: The temperature value is read from the sensor unit connected to the specified port.

Parameters: **%Port**. Numeric expression. The assigned sensor port number to which the sensor is connected. Refer to *Ports* on page 24 for more information.

Returns: Numeric value. The current temperature reading.

Notes: N/A

Example:

```
%Temp := TEMP( 2)
LOG( LOG_EXEC, "TEMP 2 = " + STR( %Temp))
```

See Also: HUMID

TIME

Syntax: TIME([$TimeString]) ==> %MidnightSeconds

Description: Converts a time string to a time value.

Action: The time string is converted to midnight seconds. Midnight seconds can be used in time calculations.

Parameters: **$TimeString**. String expression. Optional. The time string to convert. Must be in the 24-hour clock format "HH:MM:SS". "HH" represents the hours (00-23), "MM" represents the minutes (00-59), and "SS" represents the seconds (00-59). ":" is the separator. If TimeString not specified, the current system time is used.

Returns: Numeric value. The time expressed as the number of seconds past midnight. Zero is 00:00:00 (midnight) and 86,399 is 23:59:59.

Notes:
1. Refer also to the description of *Date/Time* on page 40.
2. Do not feed the results of the **TIME()** command into **TIMESTR();** use the **SECONDS()** command instead.

Example:
```
%Time := TIME()
%Time := TIME( "04:08")
$Time := "14:32"
%Time := TIME( $Time)
%Time := #14:32#        // literal time value
```

See Also: DATE, SECONDS, TIMESTR, WAITFOR, WAITUNTIL

TIMESTR

Syntax: TIMESTR(%EpochSeconds, $Format) ==> $Formatted

Description: Formats epoch seconds into a date/time string.

Action: The time value expressed as epoch seconds is formatted into a date/time string as specified by the Format parameter.

Parameters: **%EpochSeconds**. Numeric value. Date/time expressed as the number of seconds past the epoch.

$Format. String expression. The template for the resultant date/time string. Format consists of literal characters (such as a colon for a time separator) and special formatting codes that determine the date and time formats in the resultant string.

Each code specification begins with a percent sign "%" and ends with a special code character. Non-code characters in the template are copied "as is" into their relative position in the resultant string.

Refer to *Date Related Codes for TIMESTR()* following for the special code characters and their definitions.

Returns: String value. The time value formatted as a string.

Notes:

1. Refer also to the description of *Date/Time* on page 40.
2. In simple date and time format strings, the basic time codes are in uppercase characters (%H:%M:%S) and the basic date codes are in lowercase characters (%m/%d/%y).
3. In codes that pertain to the week number of the year, if the week containing January 1 has four or more days in the new year, it is considered week 1 of the new year. If not, it is week 53 of the previous year, and the next week then becomes week 1.
4. The %S code permits values of up to 61, rather than up to 59, to allow leap seconds that are sometimes added to years to keep clocks in correspondence with the solar year. It poses no problem if this feature is ignored.
5. The **TIME ()** command returns the number of seconds since midnight, so handing the result of a **TIME()** command to **TIMESTR()** returns incorrect results.

Example:

```
$Msg := TIMESTR( SECONDS(), "Current date / time is %D %T")
LOG( LOG_FLT, $Msg)
```

See Also: DATE, SECONDS, TIME, WAITFOR

Date Related Codes for TIMESTR()

Code	Description	Notes/Examples
a	Weekday short name	Mon, Wed
A	Weekday long name	Monday, Wednesday
b	Month short name	Jan, Feb
B	Month long name	January, February
C	Century number with a leading zero filler	Between 00 and 99
d	Day of the month number with a leading zero filler	Between 01 and 31
e	Day of the month number with a leading space filler	Between 1 and 31
j	Julian day number of the year with leading zero fillers	Between 001 and 366
m	Month number with a leading zero filler	Between 01 and 12
u	Weekday number	Between 1 and 7, Mon=1
U	Week number of the year (Sun as first day of week)	Between 00 and 53
V	Week number of the year (Mon as first day of week)	Between 01 and 53
w	Weekday number	Between 0 and 6, Sun=0
W	Week number of the year (Mon as first day of week)	Between 00 and 53
y	Year without the century with a leading zero filler	Between 00 and 99
Y	Year with the century with leading zero fillers	Between 0000 and 9999

Table 16. Date-related codes for TIMESTR()

Time Related Codes

Code	Description	Notes/Examples
H	Hour number for a 24-hour clock with a leading zero filler	Between 00 and 23
I	Hour number for a 12-hour clock with a leading zero filler	Between 01 and 12
M	Minute number with a leading zero filler	Between 00 and 59
p	AM or PM indicator	AM, PM
S	Second number with a leading zero filler	Between 00 and 61
T	Time shortcut for "%H:%M:%S"	16:08:02
Z	Time zone name, if defined to the system (TZ env var)	CDT, CST, PST

Table 17. Time-related codes for TIMESTR()

Shortcut Codes

Code	Description	Notes/Examples
c	Date and time format "%a %b %d %H:%M:%S %Y"	Mon Jan 01 00:00:00 1990
D	Date format "%m/%d/%y"	06/20/90
r	Time format "%I:%M:%S p"	04:08:02 PM
R	Time format "%H:%M"	16:08
T	Time format "%H:%M:%S"	16:08:02

Table 18. Shortcut codes for TIMESTR()

Miscellaneous Codes

Code	Description	Notes/Examples
n	Newline character (causes a blank line)	
t	Tab character (causes tabbing to next screen field)	
%	Percent character	

Table 19. Miscellaneous codes for TIMESTR()

TRIMSTR

Syntax: TRIMSTR($String[, %Where]) ==> $Trimmed

Description: Removes leading and trailing spaces from a string.

Action: Returns a character string with leading and/or trailing space characters removed, depending on the %Where parameter.

Parameters: **$String**. String expression. The string to remove leading and/or trailing spaces from.

%Where. Numeric expression. Optional. Where to remove the spaces from. Valid constants are BOTH, LEFT, and RIGHT. If not specified, the default %Where is BOTH.

- BOTH removes leading and trailing spaces. Equivalent to using both LEFT and RIGHT on the same string: TRIMSTR(TRIMSTR($String, LEFT), RIGHT)
- LEFT removes leading spaces only.
- RIGHT removes trailing spaces only.

Returns: String value. The string with leading and/or trailing spaces removed.

Notes: N/A

Example:

```
//create a string of spaces
$Spaces := REPSTR( “ ”, 10)
//create a string containing leading and trailing spaces
$String := $Spaces + “string” + $Spaces
%Len := LEN( $String)                    // 26
%Len := LEN(TRIMSTR( $String))           //  6
$Trimmed := TRIMSTR( $String, LEFT))
%Len := LEN( $Trimmed)                   // 16
```

See Also: STR

UPPER

Syntax: UPPER($String) ==> $UpperString

Description: Converts lowercase characters to uppercase.

Action: The lowercase characters of an evaluated string expression are converted to uppercase characters.

Parameters: **$String**. String expression. The character string to convert to uppercase.

Returns: String value. A copy of the evaluated string expression with all alphabetic characters in uppercase. All of the other characters remain the same as in the original expression.

Notes:
1. The original string expression is not changed.
2. Only the alphabetic characters are converted; numeric and special characters are not changed.

Example:
```
$UpperCaseText := UPPER( $Original)
$Text := UPPER( $Text)
IF UPPER( $Name) == “FRED”
      //some commands
ENDIF
```

See Also: LOWER

VAL

Syntax: VAL($String) ==> %Number

Description: Converts a string expression to a number.

Action: The first number characters in a string expression are returned as a number.

Parameters: **$String**. String expression. The string expression to convert to a number.

Returns: Numeric value. The first number characters in String formatted as a numeric value. If the first characters in String are not numeric, VAL() returns zero.

Notes: N/A

Example:

```
%Var := VAL( “4”)       // 4
%Var := VAL( “123abcd”) // 123
%Var := VAL( “abcd123”) // 0
```

See Also: STR

VERSION

Syntax: VERSION() ==> $VersionStr

Description: Returns a string giving the product and script language version levels.

Action: none

Parameters: none

Returns: String value of the following form:

V.RP(B)-v.r.p

where

V product version

R product revision

P product patch level

B product build number

v script language version

r script language revision

p script language patch level

Example:

```
$vstr := VERSION()
LOG( LOG_FLT, "Running version " + $vstr, 5 )
```

See Also: none

WAITFOR

Syntax: WAITFOR(%Seconds)

Description: Pauses script execution for the specified number of seconds.

Action: Execution of the current script is paused for the specified number of seconds.

Parameters: **%Seconds**. Numeric expression. The number of seconds to pause.

Returns: N/A.

Notes: N/A

Example:

```
DOUNIT( 1, ON)
WAITFOR( 2)
DOUNIT( 2, ON)
WAITFOR( 15)
CPUPOWER( 1, ON)
SCANP( 1, “PSW 00000000 00000000”, 1800, *IMLERR)
```

See Also: WAITUNTIL

WAITUNTIL

Syntax: WAITUNTIL(%MidnightSeconds)

Description: Pauses current script execution until the specified time is reached.

Action: Execution of the current script pauses until the specified time is reached. Execution then continues with the next statement. If the specified time has already passed, no pause occurs—execution continues immediately with the next statement.

Parameters: %**MidnightSeconds**. Numeric expression. The time expressed as the number of seconds past midnight. Zero is 00:00:00 (midnight) and 86,399 is 23:59:59.

Returns: N/A.

Notes: Refer also to *Date/Time* on page 40.

Example:
```
%Time := TIME( “20:30”)       //8:30 PM
WAITUNTIL( %Time)             //wait here until 8:30pm
START( SHUTDOWN())            //run the script named “SHUTDOWN”
```

See Also: DATE, SECONDS, TIME, TIMESTR, WAITFOR

WHILE

Syntax: WHILE...ENDWHILE

```
WHILE Expression
    Commands
ENDWHILE
```

Description: Repeats a sequence of commands while an expression evaluates to TRUE.

Action: The commands in the WHILE block are continually executed in sequence while the expression evaluates to TRUE.

Parameters: Expression Boolean expression. The expression to evaluate that determines whether to continue looping. Refer also to *Boolean* on page 50.

Returns: N/A.

Notes:

1. The commands in the WHILE block may never execute because the expression is evaluated at the beginning of the loop.
2. The number of nested WHILEs is unlimited.
3. The number of commands allowed within the WHILE block is unlimited.

Example:

```
%Num := 1
WHILE %Num < 10
      INC %Num
ENDWHILE
```

See Also: REPEAT

Chapter 6 Obsolete Material

This chapter:

- Lists discontinued intrinsic manifest constants and their replacements.
- Describes obsolete scripting commands.

Overview

As the Master Console Center has evolved, some constants and commands have been superseded. This chapter contains information on constants and commands that are no longer supported in current releases. While these methods may still function, Visara strongly recommends migrating to the new methods described for each command.

Manifest Constants

The following table lists discontinued intrinsic manifest constants and their replacements:

Obsolete Constant	Value	Replaced by
IO	4	UNIT
SYS	2	CPU
APPL	7	SW
FLT	8	LOG_FLT
IO	4	LOG_UNIT
CON	1	LOG_EXEC
SYS	2	LOG_CPU
APPL	7	LOG_SW
GRP	16	GROUP
RESET	2	[none], except in QUEUE() where RESET is still used.
STATUS_POWEROFF	8	
STATUS_POWERON	9	
STATUS_STARTING	10	

Table 20. Obsolete Manifest Constants

Commands

KEY Command (Date and Time Formats)

The time and date KEY command formats have been deprecated. The TIMESTR command should be used instead (see page 201).

> *Note*: Other KEY commands formats may still be used (see page 138).

Table 21 lists deprecated date formats, the KEY commands, and the equivalent TIMESTR format:

Date Format	Key Command	TIMESTR replacement
yymmdd	[DT1]	$dt = TimeStr(Seconds(), "%y%m%d")
yy.mm.dd	[DT2]	$dt = TimeStr(Seconds(), "%y.%m.%d")
yyddd	[DT3]	$dt = TimeStr(Seconds(), "%y%j")
yy.ddd	[DT4]	$dt = TimeStr(Seconds(), "%y.%j")
yy/mm/dd	[DT5]	$dt = TimeStr(Seconds(), "%y/%m/%d")
mmddyy	[DT6]	$dt = TimeStr(Seconds(), "%m%d%y")
mm.dd.yy	[DT7]	$dt = TimeStr(Seconds(), "%m.%d.%y")
mm/dd/yy	[DT8]	$dt = TimeStr(Seconds(), "%m/%d/%y")
dddyy	[DT9]	$dt = TimeStr(Seconds(), "%j%y")
ddd.yy	[DTA]	$dt = TimeStr(Seconds(), "%j.%y")

Table 21. Date Formats and Key Command Equivalents

Table 22 lists deprecated time formats, the KEY commands, and the equivalent TIMESTR format:

Time Format	Key Command	TIMESTR replacement
HHMMSS	[TM1]	$tm = TimeStr(Seconds(), "%H%M%S")
HH.MM.SS	[TM2]	$tm = TimeStr(Seconds(), "%H.%M.%S")
HH:MM:SS	[TM3]	$tm = TimeStr(Seconds(), "%H:%M:%S")
HHMM	[TM4]	$tm = TimeStr(Seconds(), "%H%M")
HH.MM	[TM5]	$tm = TimeStr(Seconds(), "%H.%M")
HH:MM	[TM6]	$tm = TimeStr(Seconds(), "%H:%M")

Table 22. Time Formats and Key Command Equivalents

EVENTCLOSE

Syntax: EVENTCLOSE(%QueueID) ==> %Status

Description: Close an event queue.

Action: Contacts the outside event daemon, and closes the specified queue. Events are no longer available from the specified queue.

Parameters: **%QueueID.** Numeric expression. The ID of the queue to close. This is the unique queue ID returned by EVENTOPEN().

Returns: Numeric value, as follows:

Value	Meaning
0	Success
-1	Error

If %Status is error, the ERRORNUM function will return a value according to the following table:

Error	Manifest Constant	Description
11002	ERR_OED_QUEUE_NOT_OPEN	The queue is not open
11003	ERR_OED_REQUEST_FAILED	The script could not communicate with the outside event daemon. Make sure gwOed is running.
11004	ERR_OED_RESPONSE_FAILED	The script did not receive the correct response from the outside event daemon. Make sure gwOed is running.

Notes: Only use EVENTCLOSE() on queues opened with EVENTOPEN().

Example:

```
// Open a queue for BMC Patrol events.
%queueId := EVENTOPEN( PATROL_EVENTS )

...

// Close the previously open queue.
%status := EVENTCLOSE( %queueId )
```

See Also: EVENTOPEN, EVENTREAD

EVENTOPEN

Syntax: EVENTOPEN(%Source [, $OsNameArray]) ==> %QueueID

Description: Opens a connection to the outside event daemon, and asks for events from a particular source. Optionally filters the events by OS name.

Action: Contacts the outside event daemon, and opens a new queue. Events are received asynchronously from the daemon, and are made available to the script by the EVENTREAD() command. Only those events that come from the specified event source and match any of the given OS names (if specified) are available.

Parameters: **%Source**. Numeric expression. Source of the events; currently only "Patrol" is supported.

$OsNameArray. Normal string array. Each array item contains the name of a MCC OS to monitor for events. If no OS names are specified, an event can come from any OS.

Returns: Numeric value, as follows:

Value	Meaning
-1	Error
Any other value	The queue ID

If %QueueID is error, the ERRORNUM function returns an error according to the following table:

Error	Manifest Constant	Description
11003	ERR_OED_REQUEST_FAILED	The script could not communicate with the outside event daemon. Ensure gwOed is running.
11004	ERR_OED_RESPONSE_FAILED	The script did not receive a correct response from the outside event daemon. Ensure gwOed is running
11005	ERR_OED_INVALID_EVENT_SOURCE	The source of events is not supported.
11006	ERR_OED_OS_DOES_NOT_EXIST	Specified OS does not exist. Check config/system.cfg file.

Notes:

1. Currently, the only Event Source supported is "PATROL_EVENTS".
2. More than one queue may be open on the same event source. These queues will receive duplicate events if the OS name arrays are not disjoint.
3. To create a correspondence between event host names and MCC OS names, specify the host name by a 'hostname=' entry in the corresponding MCC [OS …] section of the config/system.cfg file. An OS name that is the empty string ("") matches any event that has no host name mapping set up.
4. Specifying a zero length list of OS names is the same as omitting the parameter; there is no filtering on OS names. All events from the specified source can be made available with the EVENTREAD() command.

Example:

```
// Open a queue for all BMC Patrol events.
%queueId := EVENTOPEN( PATROL_EVENTS )

// Open a queue for BMC Patrol events on 'myOS'
// and 'yourOS'.
$osNames[ 1 ] := "myOS"
$osNames[ 2 ] := "yourOS"
%queueId2 := EVENTOPEN( PATROL_EVENTS, $osNames )

...

// Close the previously open queues.
%status := EVENTCLOSE( %queueId )
%status2 := EVENTCLOSE( %queueId2 )
```

See Also: EVENTCLOSE, EVENTREAD

EVENTREAD

Syntax: EVENTREAD(%QueueIdArray,$EventArray[,%Wait])==>%Status

Description: Returns the next event from any of the specified event queues.

Action: The next event from any of the specified queues is returned in the event array. If a timeout is specified, the command times out if no event is found within the given time; otherwise it waits indefinitely for the next event.

Parameters: **%QueueIdArray**. Normal numeric array. Contains the queue IDs from which the event must come from. These are the unique queue IDs created by EVENTOPEN().

$EventArray. Associative string array. This parameter is part of the return. Any existing values in this array are first cleared before the new event data is put into it. The array is associatively indexed with an event field name and assigned the event field value. All events contain the following fields:

%Wait. Number of seconds to wait for an event. Less than 0 indicates don't wait, 0 indicates wait forever, and greater than 0 indicates the timeout period. If no timeout is specified, the default is 0.

Field Name	Description
description	A description of the event (string).
hostName	The name of the host computer that generated the event (string).
nativeId	The id of the event. Depending on the source of the event, this could be a simple number, or a complex identifier containing the host, the date and time, an event ID, etc (string).
osName	The MCC OS name that corresponds to the 'hostName' field. The system looks at the 'hostname=' entries in the config/system.cfg file, and sets 'osName' to the corresponding OS. If there is no entry for 'hostName', the field is set to the empty string (string).
queue	The queue that the event arrived on (number).
systemSource	Indicates what system generated the event. This field is returned as a number. Manifest constants should be used for comparison, e.g. PATROL_EVENTS (number).
timeStamp	Indicates when the event was generated. The time is measured as epoch seconds. Refer to *Date/Time* on page *40* for more information.

%Wait. Number of seconds to wait for an event. Less than 0 indicates don't wait, 0 indicates wait forever, and greater than 0 indicates the timeout period. If no timeout is specified, the default is 0.

Returns:

Numeric value for %Status, as follows:

Value	Meaning
0 or event read	Success
-1	Error
2	Timeout

Associative string array for $EventArray, which is first cleared.

If %Status is error, the ERRORNUM function returns one of the following values:

Error	Manifest Constant	Description
11002	ERR_OED_QUEUE_NOT_OPEN	The queue is not open
11008	ERR_OED_QUEUE_ID_MUST_BE_SPECIFIED	At least one queue ID must be specified.

Notes:

1. If the actual time and date when the event was generated is not available, the time that the event enters the outside event daemon is used.
2. The 'nativeId' field is normally the most unique name for the event, as it comes directly from the generating system.
3. When comparing field names, consider translating the names to all upper case or all lower case.
4. Scripts that use any fields other than those listed may not be completely portable between event sources.
5. The order in which event queues are specified does not guarantee the order in which the queues are searched for events.
6. If an error occurs (%Status=error), the queue is closed, and subsequent EVENTREAD() and EVENTCLOSE() commands on the affecting queue ID give an error.

Example:

```
// Open a queue for all BMC Patrol events.
%queueId[ 1 ] := EVENTOPEN( PATROL_EVENTS )

// Open a queue for BMC Patrol events on 'myOS' and
// 'yourOS'.
$osNames[ 1 ] := "myOS"
$osNames[ 2 ] := "yourOS"
%queueId[ 2 ] := EVENTOPEN( PATROL_EVENTS, $osNames )

// Read events.
WHILE( 1 )
%status := EVENTREAD( %queueId, $event, 10 )
      IF( %status == EVENT_READ )
            LOG( LOG_EXEC, "Event:" )
            LOG( LOG_EXEC, " id=" + $event[ "nativeId" ])

            // Send corrective action.
            %port := PORT( OS, $event[ "osName" ] )
            KEY( %port, "Corrective action." )
      ELSE
            GOTO *done
      ENDIF
ENDWHILE
```

```
// Close the previously opened queues.
*done:

%i := 1
WHILE( %i <= ALEN( %queueId ) )
      %status := EVENTCLOSE( %queueId[ %i ] )
      INC %i
ENDWHILE
```

See Also: EVENTCLOSE, EVENTOPEN

MVSCOMMAND

Syntax: MVSCOMMAND(%ObjID, $CmdArray, $Output Array, %ErrArray $ErrorTextArray[, %PortID] [, $JobName] [, $Filter1] [, $Filter2] [, $Filter3] [, %MvsMsgCount] [, %MvsResponseTime]) ==> %ReturnValue

Description: Enters commands to MVS using the GW-MVS agent.

Action: The commands in the $CmdArray parameter are issued to the OS (LPAR) referenced by the ObjID parameter.

Parameters: **%ObjID**. Numeric expression. The unique object identifier of the OS to which to send the commands. Refer to *Object ID* on page 26 for more information.

$CmdArray. Normal string array. The array of MVS commands to execute on the target OS.

Note: Each command string must be less than eighty (80) characters long.

$OutputArray. Associative array. Holds the text outputs of the MVSCOMMANDS. The output is bounded by ===> and <=== characters, allowing the user to determine the output from each command.

%ErrorArray. Normal integer array. Each command in $CmdArray will have a corresponding element in %ErrorArray indicating the error status of that command. A value of zero (0) indicates successful submission to the host.

Note: A success (0) situation does not necessarily guarantee successful processing. The command may have been submitted successfully, but the action could not be completed on the host.

$ErrorTextArray. Normal string array. Each command in the $CmdArray will have a corresponding string in $ErrorTextArray. Each entry will give diagnostic information describing any problems that occurred when submitting the command to the host.

%PortID. Numeric expression. Optional. The port to use to KEY the commands to if the host service is unavailable. MVSCOMMAND will automatically KEY all of the commands to the specified port when necessary. The alternate route uses the **KEY()** command behind the scenes. Specifying zero (0) indicates that an alternate route should not be used if the service is unavailable, instead just returning an error code. If not specified, zero (0) is the default.

$JobName. String expression. Optional. A filter to restrict output from the submitted commands to only those which correspond to this job name. The output is read using the **QREAD()** command.

$Filter1, 2, 3. String expression. Optional. These filters are used to qualify which lines will be output from the submitted commands. $Filter2 will be ignored unless $Filter1 is not empty. Similarly, $Filter3 will be ignored unless $Filter2 is not empty.

Note: There is a 15-character limit on the size of each filter string.

$MvsMsgCount. Numeric expression. Optional. The maximum number of messages (possibly filtered) that will return in the MVS-

generated output from each independently submitted command. Use the **QREAD()** command to process the output.

Note: Valid values are 0 – 999. The recommended (and default) value is 60.

%**MvsResponseTime**. Numeric expression. Optional. The number of seconds that the mainframe agent processes output from each independently submitted command.

Note: Valid values are 0 – 999. The recommended (and default) value is 60.

Returns:

Numeric value, as follows:

Value	Meaning
0	The host agent successfully processed all commands.
-1 (error)	A problem occurred and not all commands were successfully processed.

If an error occurs, each %ErrorArray element must be checked to see which commands were not successfully submitted.

Notes:

1. MVSCOMMAND requires the optional MCC to GW-MVS interface and the installation of the GW-MVS agent on each LPAR with which to communicate. The LU6.2 networking software must be installed and running on the MCC.
2. For details of errors in the range 4000-4025 that may be returned by MVSCOMMAND, refer to the description of the **ERRORNUM()** command.

Example:

```
// Sample MVSCOMMAND() call
ARESET( $Cmds)                 // Clear these arrays
ARESET( %Err)
ARESET( $ErrMsg)

%OsID := OBJID( OS, “SNOW”)   // Get the ObjectID for the host
%PortID := PORT( OS, “SNOW”)  // alternate route is defined

$Cmds[1] := “d t”
$Cmds[2] := “s alpha”
$Cmds[3] := “s beta”

 %Result := MVSCOMMAND( %OsID, $Cmds, %Err, $ErrMsg,
 %PortID, , , , , ,30)   // Send MVSCOMMAND with timeout 30 sec.

IF %Result == ERROR
      %ErrCode := ERRORNUM()
      IF %ErrCode != ERR_MVS_PORTSUCCESS
  LOG( LOG_EXEC, “Error:” + %ErrCode + “ “ + ERRORMSG(%ErrCode))
  // Log error to Execution Log Display
      ENDIF
ENDIF
// The following output is bounded by ==========> and
// <============ so the user can determine the output
// from each command.

          P390     =======>  D T
 14.23.53 P390     STC00019  D T
 14.23.53 P390     STC00019  IEE136I LOCAL: TIME=14.23.53
DATE=1998.345  GMT: TIME=14.23.53 DATE=1998.345
```

```
 14.23.53 P390     STC00019  GWLU62MC08  D T BY WATCHSNA
USERID
          P390     <=======  **DONE** (MAX MSGS OR MAX
TIME OR CMD CANCELED)
          P390     =======>  D T
 14.30.00 P390     STC00019  D T
 14.30.00 P390     STC00019  IEE136I LOCAL: TIME=14.30.00
DATE=1998.345  GMT: TIME=14.30.00 DATE=1998.345
 14.30.00 P390     STC00019  GWLU62MC08  D T BY WATCHSNA
USERID
 14.30.02 P390     STC00004  ERB101I ZZ : REPORT
AVAILABLE FOR PRINTING
          P390     <=======  **DONE** (MAX MSGS OR MAX
TIME OR CMD CANCELED)
```

See Also: TSOEREXX

Possible Error Codes

The following GW-MVS error messages may be returned in the $Result Array.

Manifest Constant	Value	Associated String
ICLErr_MvsRsp_0000	10000	General MVS response message error.
ICLErr_MvsRsp_0001	10001	Storage shortage.
ICLErr_MvsRsp_0002	10002	Temporary failure.
ICLErr_MvsRsp_0003	10003	Invalid userid.
ICLErr_MvsRsp_0004	10004	Invalid password.
ICLErr_MvsRsp_0005	10005	Logon required prior to request.
ICLErr_MvsRsp_0006	10006	Unknown request.
ICLErr_MvsRsp_0007	10007	Not awaiting diagnosis.
ICLErr_MvsRsp_0008	10008	Not done with diagnosis.
ICLErr_MvsRsp_0009	10009	Missing required parameter.
ICLErr_MvsRsp_0010	10010	No session via any known Bxx node.
ICLErr_MvsRsp_0011	10011	No response data available.
ICLErr_MvsRsp_0012	10012	Userid already exists.
ICLErr_MvsRsp_0013	10013	Userid unknown.
ICLErr_MvsRsp_0014	10014	Userid database failed to open.
ICLErr_MvsRsp_0015	10015	Logic error during add.
ICLErr_MvsRsp_0016	10016	Userid has no administration authority.
ICLErr_MvsRsp_0017	10017	Ditto error.
ICLErr_MvsRsp_0018	10018	Suspended userid.

Manifest Constant	Value	Associated String
ICLErr_MvsRsp_0019	10019	Add operator rejected; administrator required.
ICLErr_MvsRsp_0020	10020	Change from administrator to operator not valid.
ICLErr_MvsRsp_0021	10021	Administrator cannot be suspended.
ICLErr_MvsRsp_0022	10022	Cannot delete last administrator.
ICLErr_MvsRsp_0023	10023	Authorization insufficient.
ICLErr_MvsRsp_0024	10024	Userid already logged on.
ICLErr_MvsRsp_0025	10025	Rejected virtual userid.
ICLErr_MvsRsp_0026	10026	Maximum virtual users logged on.
ICLErr_MvsRsp_0027	10027	Surrogate not logged on.
ICLErr_MvsRsp_0028	10028	Log on denied.
ICLErr_MvsRsp_0029	10029	Authorization insufficient.
ICLErr_MvsRsp_0030	10030	MVS command table exhausted.
ICLErr_MvsRsp_0031	10031	MVS command not outstanding.
ICLErr_MvsRsp_0032	10032	NMVT recording is off.
ICLErr_MvsRsp_0033	10033	Wrong release (incompatible).
ICLErr_MvsRsp_0034	10034	Site not authorized for online access.
ICLErr_MvsRsp_0099	10099	Missing or invalid parameters.
ICLErr_MvsRsp_0997	10997	File not found.

QUEUE

Syntax: QUEUE(Operation[, Port])

Description: Starts (stops and resets) a queue of OS printer console messages.

Action: A current message pointer in a script is set, reset, and cleared with the QUEUE command.

Parameters: **Operation**. Numeric Expression. Determines how a current message pointer is manipulated. Refer to *Table 23. Operation parameter options for QUEUE command* following for details.

Port. Numeric expression. Optional. The assigned OS printer console port number the to create a queue for. If not specified, a queue is created for the OS printer console on which the script is executing.

Returns N/A.

Notes:

1. Refer to *Manifest Constants* on page 42 for the constants reference list.
2. The MCC has one large message queue that holds all of the incoming printer console messages. The queue holds the most recent 6500 messages. The oldest message is discarded when a new message is added (a circular message queue).
3. Each script has multiple current message pointers into the message queue. The message pointers are set and reset with the **QUEUE()** command, and advanced with the **READMSG()** command.
4. The next message in a queue is retrieved with the **READMSG()** command.
5. QUEUE only creates a queue for OS printer console messages. If a port is specified, it must be an OS printer console port. If a port is not specified, the script must be executing on an OS icon.
6. Only QUEUE (RESET) a queue that has had the QUEUE(ON) command activated on it.
7. If an OS does not have a printer port defined, attempting to use the QUEUE( ) and READMSG( ) commands on that OS will generate run-time errors, but will not generate compile errors.

Example:

```
QUEUE( ON)
QUEUE( RESET)
QUEUE( OFF)
```

See Also: READMSG

QUEUE command operation parameter options

Operation Constant	Description
ON	Creates a message queue in the script by creating a current message pointer. The current message pointer is set to the end of the messages—the next new message received will be the first message in the script's queue (and the next message read with the READMSG command).
RESET	A current message pointer is set to the end of the messages, thereby quickly skipping any remaining messages (unread with the READMSG command) in the script's queue. This is the same as executing: QUEUE(OFF) QUEUE(ON)
OFF	Turns off a message queue in the script by removing the current message pointer for the queue.

Table 23. Operation parameter options for QUEUE command

READMSG

Syntax: READMSG($Msg, Wait, *Timeout[, Port [, Filter]])

Description: Reads the next message from a script's message queue.

Action: Reads the next message in a queue. The next message is defined by a script's current message pointer created with the **QUEUE()** command.

The entire message is placed in the array specified by the $MSG array parameter. Each word in the message is placed sequentially by position into the array's elements ($Msg[1] contains the 1st word, $Msg[2] contains the 2nd word, $Msg[3] contains the 3rd word, and so on.).

When the queue is empty, the READMSG command acts as specified by the Wait and *Timeout parameters.

Parameters: **$Msg**. Normal string array. The array to populate with each word from the message. Each array element holds one word. The words are split up by "white space"—one or more space characters.

Wait. Numeric expression. The number of seconds to wait before timing out. When the queue is empty, this number specifies how long the command waits for a message.

***Timeout**. Label. The label to jump to when the **READMSG()** command times out.

Port. Numeric expression. Optional. The OS printer console from which to read a message. A queue must already have been created for the specified port with the QUEUE() command. If not specified, a message is read from the default queue (the OS printer console the script is executing on).

Filter. String Expression. Optional. The text in this string acts as filter criteria—the criteria text must be in the console message in order for READMSG to receive it. Messages will only "appear" to READMSG if they contain the criteria text. A Filter value of null string "" is the same as omitting the Filter parameter.

Returns: N/A.

Notes:

1. QUEUE(ON) must be executed before the READMSG command.
2. **READMSG()** only reads OS printer console messages. If a port is specified, it must be an OS printer console port. If a port is not specified, the script must be executing on an OS icon.

Example:

```
//=======================================================
// Example 1
//=======================================================
      QUEUE( ON)        //turn on queuing
*READ:
      READMSG($Msg, 1800, *TIMEOUT)
      //check first word for IO err
      IF $MSG[ 1] == “IOS000I”
            IOERROR()         //call IO err handling script
      ENDIF
      GOTO *READ              //repeat process continually
*TIMEOUT:
      LOG( FLT, “No messages in 3 minutes!”)
      GOTO *READ
//=======================================================
// Example 2
//=======================================================
%PortPrnSys5 := PORT( PRN, “SYS5”)
QUEUE( ON, %QueueSys5)
READMSG( $Msg, 30, *TIMEOUT, %PortPrnSys5)
```

See Also: GOTO, SCANB, SCANP

TSOEREXX

Syntax: TSOEREXX(%ObjID, $CmdArray, %ErrorArray, $ErrorTextArray) ==> %ReturnValue

Description: Enters commands to TSO/E REXX using the GW-MVS agent.

Action: The commands in the $CmdArray parameter are issued to the OS (LPAR) referenced by the %ObjID parameter.

Parameters:

%ObjID. Numeric expression. The object identifier. The unique ID of the OS to which to send the commands. Refer to *Object ID* on page 26 for more information.

$CmdArray. Normal string array. The array of TSO/E REXX commands to launch on the target OS.

Note: Each command string must be less than seventy (70) characters in length.

%ErrorArray. Normal integer array. Each command in $CmdArray has a corresponding element in %ErrorArray indicating the error status of that command. A value of zero (0) indicates successful submission to the host.

Note: A success (0) situation does not necessarily guarantee successful processing. The command may have been submitted successfully, but the action could not be completed on the host for any number of reasons. Use the **QREAD()** command to verify the outcome of a submitted command from that command's output.

$ErrorTextArray. Normal string array. Each command in the $CmdArray has a corresponding string in $ErrorTextArray. Each entry gives diagnostic information indicative of any problems that may have occurred while submitting the command to the host.

Returns: Numeric value as follows:

Value	Status
0	Indicates the host agent successfully processed all commands
-1	Symbolic ERROR indicating there was a problem

If an ERROR occurs, each %ErrorArray element must be checked to identify which commands were not successfully submitted.

Notes:

1. TSOEREXX requires the optional MCC to GW-MVS interface, and the installation of the GW-MVS agent on each LPAR with which to communicate.
2. For information on error checking of TSOEREXX, refer to the description of the **ERRORNUM()** command.

Example:

```
// Sample TSOEREXX() call

ARESET( $Cmds)
ARESET( %Err)
ARESET( $ErrMsg)

%OsID := OBJID( OS, “SNOW”)   // Get ObjectID for “Snow" OS
$Cmds[1] := “mvs1”            // Define command array
$Cmds[2] := “mvs2”
$Cmds[3] := “mvs3”
$Cmds[4] := “mvs4”
$Cmds[5] := “mvs5”

%Result := TSOEREXX( %OsID, $Cmds, %Err, $ErrMsg)
                                        // Send commands to
host
IF %Result == ERROR
      %ErrCode := ERRORNUM()
      LOG( LOG_EXEC, “Error: “ + %ErrCode + “ “ +
ERRORMSG( %ErrCode))                    // Log message
if error
ENDIF
```

See Also: MVSCOMMAND

Appendix A ASCII Character Values

(including ISO-8859-1 ANSI "Latin 1" values)

hex	dec	Char	hex	dec	Char	hex	dec	Char	hex	dec	Char		
00	0	***NUL***	20	32	***SP***	40	64	@	60	96	`	***NUL***	Null
01	1	***SOH***	21	33	!	41	65	A	61	97	a	***SOH***	Start of Heading
02	2	***STX***	22	34	"	42	66	B	62	98	b	***STX***	Start of Text
03	3	***ETX***	23	35	#	43	67	C	63	99	c	***ETX***	End of Text
04	4	***EOT***	24	36	$	44	68	D	64	100	d	***EOT***	End of Transmission
05	5	***ENQ***	25	37	%	45	69	E	65	101	e	***ENQ***	Enquiry
06	6	***ACK***	26	38	&	46	70	F	66	102	f	***ACK***	Acknowledge
07	7	***BEL***	27	39	'	47	71	G	67	103	g	***BEL***	Bell
08	8	***BS***	28	40	(	48	72	H	68	104	h	***BS***	Backspace
09	9	***HT***	29	41	)	49	73	I	69	105	i	***HT***	Horizontal Tab
0A	10	***LF***	2A	42	*	4A	74	J	6A	106	j	***LF***	Line Feed
0B	11	***VT***	2B	43	+	4B	75	K	6B	107	k	***VT***	Vertical Tab
0C	12	***FF***	2C	44	,	4C	76	L	6C	108	l	***FF***	Form Feed
0D	13	***CR***	2D	45	-	4D	77	M	6D	109	m	***CR***	Carriage Return
0E	14	***SO***	2E	46	.	4E	78	N	6E	110	n	***SO***	Shift Out
0F	15	***SI***	2F	47	/	4F	79	O	6F	111	o	***SI***	Shift In
10	16	***DLE***	30	48	0	50	80	P	70	112	p	***DLE***	Data Link Escape
11	17	***DC1***	31	49	1	51	81	Q	71	113	q	***DC1***	Device Control 1
12	18	***DC2***	32	50	2	52	82	R	72	114	r	***DC2***	Device Control 2
13	19	***DC3***	33	51	3	53	83	S	73	115	s	***DC3***	Device Control 3
14	20	***DC4***	34	52	4	54	84	T	74	116	t	***DC4***	Device Control 4
15	21	***NAK***	35	53	5	55	85	U	75	117	u	***NAK***	Negative Acknowledge
16	22	***SYN***	36	54	6	56	86	V	76	118	v	***SYN***	Synchronous Idle
17	23	***ETB***	37	55	7	57	87	W	77	119	w	***ETB***	End of Transmission Block
18	24	***CAN***	38	56	8	58	88	X	78	120	x	***CAN***	Cancel
19	25	***EM***	39	57	9	59	89	Y	79	121	y	***EM***	End of Medium
1A	26	***SUB***	3A	58	:	5A	90	Z	7A	122	z	***SUB***	Substitute
1B	27	***ESC***	3B	59	;	5B	91	[	7B	123	{	***ESC***	Escape
1C	28	***FS***	3C	60	<	5C	92	\	7C	124	|	***FS***	File Separator
1D	29	***GS***	3D	61	=	5D	93	]	7D	125	}	***GS***	Group Separator
1E	30	***RS***	3E	62	>	5E	94	^	7E	126	~	***RS***	Record Separator
1F	31	***US***	3F	63	?	5F	95	_	7F	127	***DEL***	***US***	Unit Separator
												SP	Space (CHARACTER)
												DEL	Delete

(continued on next page)

(continued from previous page)

80	128	***PAD***	A0	160	***NS***	C0	192	À	E0	224	à	***HOP***	High Octet Preset
81	129	***HOP***	A1	161	¡	C1	193	Á	E1	225	á	***BPH***	Break Permitted Here
82	130	***BPH***	A2	162	¢	C2	194	Â	E2	226	â	***NBH***	No Break Here
83	131	***NBH***	A3	163	£	C3	195	Ã	E3	227	ã	***IND***	Index
84	132	***IND***	A4	164	¤	C4	196	Ä	E4	228	ä	***NEL***	Next Line
85	133	***NEL***	A5	165	¥	C5	197	Å	E5	229	å	***SSA***	Start Of Selected Area
86	134	***SSA***	A6	166	¦	C6	198	Æ	E6	230	æ	***ESA***	End Of Selected Area
87	135	***ESA***	A7	167	§	C7	199	Ç	E7	231	ç	***HTS***	Character Tabulation Set
88	136	***HTS***	A8	168	¨	C8	200	È	E8	232	è	***HTJ***	Character Tabulation With Justification
89	137	***HTJ***	A9	169	©	C9	201	É	E9	233	é	***VTS***	Line Tabulation Set
8A	138	***VTS***	AA	170	ª	CA	202	Ê	EA	234	ê	***PLD***	Partial Line Forward
8B	139	***PLD***	AB	171	«	CB	203	Ë	EB	235	ë	***PLU***	Partial Line Backward
8C	140	***PLU***	AC	172	¬	CC	204	Ì	EC	236	ì	***RI***	Reverse Line Feed
8D	141	***RI***	AD	173	-	CD	205	Í	ED	237	í	***SS2***	Single-Shift Two
8E	142	***SS2***	AE	174	®	CE	206	Î	EE	238	î	***SS3***	Single-Shift Three
8F	143	***SS3***	AF	175	¯	CF	207	Ï	EF	239	ï	***DCS***	Device Control String
90	144	***DCS***	B0	176	°	D0	208	Ð	F0	240	ð	***PU1***	Private Use One
91	145	***PU1***	B1	177	±	D1	209	Ñ	F1	241	ñ	***PU2***	Private Use Two
92	146	***PU2***	B2	178	²	D2	210	Ò	F2	242	ò	***STS***	Set Transmit State
93	147	***STS***	B3	179	³	D3	211	Ó	F3	243	ó	***CCH***	Cancel Character
94	148	***CCH***	B4	180	´	D4	212	Ô	F4	244	ô	***MW***	Message Waiting
95	149	***MW***	B5	181	µ	D5	213	Õ	F5	245	õ	***SPA***	Start Of Guarded Area
96	150	***SPA***	B6	182	¶	D6	214	Ö	F6	246	ö	***EPA***	End Of Guarded Area
97	151	***EPA***	B7	183	·	D7	215	×	F7	247	÷	***SOS***	Start Of String
98	152	***SOS***	B8	184	¸	D8	216	Ø	F8	248	ø	***SGCI***	Single Graphic Character Introducer
99	153	***SGCI***	B9	185	¹	D9	217	Ù	F9	249	ù	***SCI***	Single Character Introducer
9A	154	***SCI***	BA	186	º	DA	218	Ú	FA	250	ú	***CSI***	Control Sequence Introducer
9B	155	***CSI***	BB	187	»	DB	219	Û	FB	251	û	***ST***	String Terminator
9C	156	***ST***	BC	188	¼	DC	220	Ü	FC	252	ü	***OSC***	Operating System Command
9D	157	***OSC***	BD	189	½	DD	221	Ý	FD	253	ý	***PM***	Privacy Message
9E	158	***PM***	BE	190	¾	DE	222	Þ	FE	254	þ	***APC***	Application Program Command
9F	159	***APC***	BF	191	¿	DF	223	ß	FF	255	ÿ	***NS***	Nonbreaking Space (CHARACTER)

Table 24. ASCII Character Values

Appendix B Command Syntax

Appendix B contains two tables:

- The first table lists information by command types.
- The second table lists information by command.

Both tables contain the following information: commands, command types, syntax, and a description of each command. There is also a page reference for more information about each command.

Command Syntax—By Command Type

Command	Command Type	Description	Syntax
ALARM	Alerting	Generates a repetitive alarm tone (beep) at the workstation (Global WorkStation).	ALARM(%Operation)
ALERTCREATE	Alerting	Create a new alert.	ALERTCREATE(%Status,%State,$Source,$MsgText,$User Note)==>%AlertID
ALERTDEL	Alerting	Deletes an existing alert.	ALERTDEL(%AlertID) ==> %ErrCode
ALERTGETACTIVE	Alerting	Retrieves information on all active alerts into an array.	ALERTGETACTIVE($AssocArray) ==>%ErrCode
ALERTMOD	Alerting	Modifies the value of a field in an existing alert.	ALERTMOD(%AlertID,%AlertField,NewValue) ==>%ErrCode
ICON	Alerting	Changes icon characteristics.	ICON(%Status[,$Message[,%Class[,$Name]]])
ICONMSG	Alerting	Returns an icon's current message.	ICONMSG([%Class[,$Name]])==>$Message
ICONNAME	Alerting	Returns an icon's name.	ICONNAME([%Port]])==>$Name
ICONSTATUS	Alerting	Returns an icon's current status.	ICONSTATUS([%Class[,$Name]])==>%Status
ASCRN	Console Message	Fill an array with the full text of a console screen.	ASCRN($Array,%Port)
BLOCKSCAN	Console Message	Enables up to 256 SCANB commands to execute as a group.	BLOCKSCAN(%Wait,*Timeout[,$Array])…ENDBLOCK

Command	Command Type	Description	Syntax
KEY	Console Message	Enters a character string to the specified console.	KEY(%Port,$Keys[,%Timeout])==>%RetCode
PORT	Console Message	Returns the port number for a console definition.	PORT(%Class[,$IconName])==>%Port
QCLOSE	Console Message	Closes a message queue.	QCLOSE(%QueueID)
QOPEN	Console Message	Opens a new queue of OS printer console messages.	QOPEN([%ObjIDArray])==>%QueueID
QPREVIEW	Console Message	Previews the current message on the screen (before it enters the queue).	QPREVIEW(%QueueID,$ResultArray)==>%RetCode
QREAD	Console Message	Reads the next message from a message queue.	QREAD(%QueueID,$MsgArray,%Wait[<$Filter]) ==>$MsgLine
QSKIP	Console Message	Moves a current message pointer for a queue.	QSKIP(%QueueID,%Skip)
SCANB	Console Message	Same as SCANP except for use with the BLOCKSCSN command.	SCANB(%Port,$Text,*Found)
SCANP	Console Message	Searches a console for a specified character string.	SCANP(%Port,$Text,%Wait,*Found[,$Array])
SCRNTEXT	Console Message	A full or partial screen snapshot—returns characters from a console screen.	SCRNTEXT(%Port,%Start,%Length==>$Text

EVENTCLOSE	Event	Closes a specified event queue. Events are no longer available from the specified queue.	EVENTCLOSE(%QueueID)==>%Status
EVENTOPEN	Event	Opens a connection to the outside event daemon, and asks for events from a specified source.	EVENTOPEN(%Source[,$OsNameArray])==>%QueueID
EVENTREAD	Event	Returns the next event from any of the specified event queues.	EVENTREAD(%QueueIdArray,$EventArray[,%Wait] ==>%Status
MVSCOMMAND	Event	Enters commands to MVS using the GW-MVS agent (optional software)	MVSCOMMAND(%ObjID, $CmdArray, $Output Array, %ErrArray $ErrorTextArray[, %PortID] [, $JobName] [, $Filter1] [, $Filter2] [, $Filter3] [, %MvsMsgCount] [, %MvsResponseTime]) ==> %ReturnValue
TSOEREXX	Event	Enters commands to TSO/E REXX.	TSOEREXX(%ObjID, $CmdArray, %ErrorArray, $ErrorTextArray) ==> %ReturnValue
FCLOSE	File	Closes an open file.	FCLOSE(%FileNum)
FDELETE	File	Permanently deletes a file.	FDELETE($FileName)==>%Success
FEXISTS	File	Determines if a file exists.	FEXISTS($FileName)==>%Success
FILENO	File	Obtains the system integer file descriptor from a file handle.	FILENO(%FileHandle)==>%FileDescriptor
FMODTIME	File	Returns the time value of a last modified date/time stamp for a file.	FMODTIME(File)==>%EpochSeconds
FOPEN	File	Opens a file for I/O access.	FOPEN($FileName[,%Mode])==>%FileHandle

FPOS	File	Returns an open file's current record pointer position.	FPOS(%FileNum)==>%Position
FREAD	File	Reads values from an open file into variables.	FREAD(%FielNum,var1[,var2,…,[var*n*]…])==>%QtyRead
FRENAME	File	Renames a file.	FRENAME($CurrentName,$NewName)==>%Success
FREWIND	File	Moves an open file's current record pointer to the beginning.	FREWIND(%FileNum)
FSEEK	File	Moves an open file's current record pointer to a byte offset.	FSEEK(%FileNum,%Position)==>%Success
FWRITE	File	Writes expression to an open file.	FWRITE(%FileNum,expre[,%NEWLINE])==>%Success
LOG	File	Enters a message in a log.	LOG(%LogType,$Message[,%Status])
MKDTEMP	File	Creates a unique temporary directory for I/O access.	MKDTEMP($Pattern)==>$DirectoryName
MKSTEMP	File	Opens a unique temporary file for I/O access.	MKSTEMP($Pattern)==>%FileHandle
MKTEMP	File	Returns a unique file name	MKTEMP($Pattern)==>$FileName
END	Flow Control	Ends the execution of the script thread.	END
EXEC	Flow Control	Executes a script whose name s stored in a string expression.	EXEC($ScriptName[,Parm1,Parm1,…])==>ReturnValue

GOSUB	Flow Control	Immediately transfers script execution to the specified label and waits until the called routine finishes execution.	GOSUB*Label
GOTO	Flow Control	Immediately transfers script execution to the specified label.	GOTO*Label
IF (ENDIF)	Flow Control	Evaluates an expression for TRUE or FALSE. (If…Then statement)	IF…[ELSE…]ENDIF
REPEAT	Flow Control	Repeats a sequence of commands until an expression evaluates to TRUE.	REPEAT…UNTIL
RETURN	Flow Control	Returns execution to the calling routine, passing an optional return value.	RETURN[Expression]
START	Flow Control	Initiates execution of another script for concurrent processing.	STARTScriptName(Parms)[,%Class[,$Name]])
STOP	Flow Control	Halts execution of another script.	STOP(ScriptName[,%Class[<$Name]])
SWITCH	Flow Control	Executes command(s) based on the value of an expression.	SWITCH…CASE…[DEFAULT…]ENDSWITCH
SYSEXEC	Flow Control	Executes a (Unix) command on the MCC host system with parameters.	SYSEXEC($String)==>%Return

WHILE (ENDWHILE)	Flow Control	Repeats a sequence of commands while an expression evaluates to TRUE.	WHILE…ENDWHILE
CLASSNAME	Misc.	Returns the class name for a class number	CLASSNAME(%Class])==>$ClassName
CLASSNUM	Misc.	Returns the class number for a class name string.	CLASSNUM([$ClassName])==>%Class
ERRORMSG	Misc.	Returns the error message associated with the error number.	ERRORMSG(%ErrNum)==>$ErrMsg
ERRORNUM	Misc.	Returns the error number for the most recent command.	ERRORNUM()==>%ErrNum
OBJEXEC	Object Manager	Executes an action on an object.	OBJEXEC(%ObjID,$Action[,Parms…])==>ReturnValue
OBJGET	Object Manager	Returns the current value in an object's field.	OBJGET(%ObjID,$ObjFieldName)==>$CurrentValue
OBJGETARRAY	Object Manager	Populates an associative string array with the current field values from an object.	OBJGETARRAY(%ObjID,$AssocArray)==>%Success
OBJID	Object Manager	Returns the ID (unique only for the particular script thread) for a MCC object.	OBJID(%Class,$ObjKeyExpr)==>%ObjectID
OBJIDARRAY	Object Manager	Returns the number of children of the selected object ID.	OBJIDARRAY(%Class, %ObjIDParent, %AssocArray)==> %Children

OBJSET	Object Manager	Populates an associative numeric array with object IDs from the children of a parent object.	OBJIDARRAY(%Class,%ObjIDParent,%AssocArray)
OBJSET	Object Manager	Sets the current value in an object's field.	OBJSET(%ObjID,$ObjFieldName,$NewValue)==>%ErrCode
OBJSETARRAY	Object Manager	Sets the field values for an object from an associative string array.	OBJSETARRAY(%ObjID,$AssocArray)==>%Success
CPUPOWER	Physical Interface Control	Switches a CPU's power ON or OFF.	CPUPOWER(%Port,%Operation)
DIUNIT	Physical Interface Control	Check the status of a device connected to a DI unit.	DIUNIT(%Port)==>%Status
DOUNIT	Physical Interface Control	Controls the device connected to a DO unit.	DOUNIT(%Port[,%Operation])==>%DOSwitch
HUMID	Physical Interface Control	Reads the current humidity from a sensor unit.	HUMID(%Port)==>%Humidity
TEMP	Physical Interface Control	Reads the current temperature from a sensor unit.	TEMP(%Port)==>%Temp
SCRIPTCANCEL	Scripting	Obtains the system integer file descriptor from a file handle	SCRIPTCANCEL($ScriptName,$Class,$Name)

SCRIPTGETACTIVE	Scripting	Retrieves information on all active scripts into an associative array.	SCRIPTGETACTIVE($AssocArray)==>%ErrCode
SNMP_GET	SNMP	Retrieves the value of a specified MIB object.	SNMP_GET($Alias,$MIBOID)==>$Value
SNMP_GETNEXT	SNMP	Retrieves the value and the object ID of the next logical MIB object to a specified MIB object.	SNMP_GETNEXT($Alias,$MIBOID,$NextMIBOID) ==>$Value
SNMP_GETTABLE	SNMP	Retrieves a complete table of information from the MIB on the specified agent.	SNMP_GETTABLE($Alias,$MIBOID,$TableArray[, $Delimiter])==>%ReturnCode
SNMP_SET	SNMP	Sets the value of a specified MIB object.	SNMP_SET($Alias,$MIBOID,$Value)==>%ReturnCode
SNMP_TRAPSEND	SNMP	Sends a trap to one or more hosts.	SNMP_TRAPSEND($Alias,%TrapNum[[,%EntNum[,$MIB OID[,$Time[,$HostName]]]])==>%ReturnCode
DATE	Time	Converts a date string to a date value.	DATE($DateString])==>%EpochSeconds
SECONDS	Time	Returns the time value for the current time.	SECONDS()==>%EpochSeconds
TIME	Time	Converts a time string to a time value.	TIME([TimeString])==>%MidnightSeconds
TIMESTR	Time	Formats epoch seconds into a date/time string.	TIMESTR(%EpochSeconds,$Format)==>$Formatted
WAITFOR	Time	Pauses script execution for the specified number of seconds.	WAITFOR(%Seconds)

WAITUNTIL	Time	Pauses current script execution until the specified time is reached.	WAITUNTIL(%MidnightSeconds)
AICONNAMES	Variable	Fill an array with all of the icon names in a class.	AICONNAMES($AssocArray,%Class,$ParentIcon)
ALEN	Variable	Returns the number of elements in the array.	ALEN(Array)==>%Elements
ARESET	Variable	Reset the contents of an array to “empty”.	ARESET(Array)
ASCII	Variable	Returns the ASCII value of a character.	ASCII($String)==>%Value
ASORT	Variable	Sort a normal array.	ASORT(NormArray,%Direction)
ASSOCKEYS	Variable	Populate a normal string array with the string index keys of an associative array	ASSOCKEYS(AssocArray,$NormArray)
ATSTR	Variable	Returns the starting position of a substring within a string.	ATSTR($String,$Substring)==>%StartPos
BASEDIRECTORY	Variable	Obtains the base directory for the product.	BASEDIRECTORY()==>$DirectoryString
CHR	Variable	Returns the character whose value is the ASCII code given.	CHR(%Number)==>$String
DEC	Variable	Subtracts one from a numeric variable’s value.	DEC%Variable
DECODE	Variable	Decodes a string.	DECODE ($String[,$Key]) ==>$Result
ENCODE	Variable	Encodes a string.	ENCODE ($String[,$Key]) ==>$Result

FINDSTR	Variable	Searches a string for a regular expression pattern.	FINDSTR($String,$Substring==>$FoundText
FORMATSTR	Variable	Formats a string by combining literal characters with conversion specifications.	FORMATSTR($String[,expr1,[expr2,…,[expr*n*]…]])
GETENV	Variable	Obtains the current value of the given environment variable.	GETENV($Variable)==>$Value
GETPID	Variable	Obtains the system process identifier for this script.	GETPID()==>%ProcessId
HEXSTR	Variable	Converts an integer to a hex string.	HEXSTR(%Number)==>$Hex
INC	Variable	Adds one to a numeric variable's value.	INC%Variable
JOIN	Variable	Combines the elements of an array into a string.	JOIN($Array,$Delimiter)==>$String
LEFTSTR	Variable	Returns the leftmost specified number of characters of a string expression.	LEFTSTR($String,%Count)==>$Substr
LEFTSTR	Variable	Returns the leftmost specified number of characters of a string expression.	LEFTSTR($String,%Count)==>$Substr
LEN	Variable	Returns the number of characters in a string expression.	LEN($StringExpr)==>%Count

LOWER	Variable	Converts uppercase characters to lowercase.	LOWER($String)==>$Lowercase
PARMS	Variable	Receives parameters into the script.	PARMSvar1[,var3,…,[var*n*]…]]]
REPSTR	Variable	Returns a string repeated a specified number of times.	REPSTR($String,%Count) ==>$RepeatedString
RIGHTSTR	Variable	Returns the rightmost specified number of characters of a string expression.	RIGHTSTR($String,%Count) ==>$Substr
SET	Variable	Make the contents of a variable equal to the specified expression.	SETVariable:=Expression
SPLIT	Variable	Populates an array with the fields of a string delimited by a string.	SPLIT($Array,$String,$Delimiter)
STR	Variable	Converts a numeric expression to a string.	STR(%Number) ==>$String
SUBSTR	Variable	Extract a substring from a character string.	SUBSTR($String,%Start[,%Count]) ==>$SubStr
TRIMSTR	Variable	Removes leading and trailing spaces from a string.	TRIMSTR($String [, %Where]) ==>$TrimmedString
UPPER	Variable	Converts lowercase characters to uppercase.	UPPER($String) ==>$UpperString
VAL	Variable	Converts a string expression to a number.	VAL($String) ==>A%Number

VERSION	Variable	Returns a string giving the product and script language version levels.	VERSION()==>$VersionStr

Table 25. Summary of Command Syntax—By Command Type

Command Syntax—By Command

Command	Command Type	Description	Syntax
AICONNAMES	Variable	Fill an array with all of the icon names in a class.	AICONNAMES($AssocArray,%Class,$ParentIcon)
ALARM	Alerting	Generates a repetitive alarm tone (beep) at the workstation (Global WorkStation).	ALARM(%Operation)
ALEN	Variable	Returns the number of elements in the array.	ALEN(Array) ==>%Elements
ALERTCREATE	Alerting	Create a new alert.	ALERTCREATE(%Status,%State,$Source,$MsgText,$UserNote) ==>%AlertID
ALERTDEL	Alerting	Deletes an existing alert.	ALERTDEL(%AlertID) ==> %ErrCode
ALERTGETACTIVE	Alerting	Retrieves information on all active alerts into an array.	ALERTGETACTIVE($AssocArray)==>%ErrCode
ALERTMOD	Alerting	Modifies the value of a field in an existing alert.	ALERTMOD(%AlertID,%AlertField,NewValue) ==>%ErrCode
ARESET	Variable	Reset the contents of an array to “empty”.	ARESET(Array)
ASCII	Variable	Returns the ASCII value of a character.	ASCII($String) ==>%Value
ASCRN	Console Message	Fill an array with the full text of a console screen.	ASCRN($Array,%Port)
ASORT	Variable	Sort a normal array.	ASORT(NormArray,%Direction)
ASSOCKEYS	Variable	Populate a normal string array with the string index keys of an associative array	ASSOCKEYS(AssocArray,$NormArray)
ATSTR	Variable	Returns the starting position of a substring within a string.	ATSTR($String,$Substring)==>%StartPos

Command	Command Type	Description	Syntax
BASEDIRECTORY	Variable	Obtains the base directory for the product.	BASEDIRECTORY()==>$DirectoryString
BLOCKSCAN	Console Message	Enables up to 256 SCANB commands to execute as a group.	BLOCKSCAN(%Wait,*Timeout[,$Array])…ENDB LOCK
CHR	Variable	Returns the character whose value is the ASCII code given.	CHR(%Number)==>$String
CLASSNAME	Misc.	Returns the class name for a class number	CLASSNAME(%Class])==>$ClassName
CLASSNUM	Misc.	Returns the class number for a class name string.	CLASSNUM([$ClassName])==>%Class
CPUPOWER	Physical Interface Control	Switches a CPU's power ON or OFF.	CPUPOWER(%Port,%Operation)
DATE	Time	Converts a date string to a date value.	DATE($DateString])==>%EpochSeconds
DEC	Variable	Subtracts one from a numeric variable's value.	DEC%Variable
DECODE	Variable	Decodes a string.	DECODE ($String[,$Key]) ==>$Result
DIUNIT	Physical Interface Control	Check the status of a device connected to a DI unit.	DIUNIT(%Port)==>%Status
DOUNIT	Physical Interface Control	Controls the device connected to a DO unit.	DOUNIT(%Port[,%Operation])==>%DOSwitch
ENCODE	Variable	Encodes a string.	ENCODE ($String[,$Key]) ==>$Result
END	Flow Control	Ends the execution of the script thread.	END
ERRORMSG	Misc.	Returns the error message associated with the error number.	ERRORMSG(%ErrNum)==>$ErrMsg

Command	Command Type	Description	Syntax
ERRORNUM	Misc.	Returns the error number for the most recent command.	ERRORNUM()==>%ErrNum
EVENTCLOSE	Event	Closes an event queue.	EVENTCLOSE(%QueueID) ==> %Status
EVENTOPEN	Event	Opens a connection to the outside event daemon, and asks for events from a particular source.	EVENTOPEN(%Source [, $OsNameArray]) ==> %QueueID
EVENTREAD	Event	Returns the next event from a specified event queue.	EVENTREAD(%QueueIdArray,$EventArray[, %Wait])==>%Status(%Source [, $OsNameArray]) ==> %QueueID
EXEC	Flow Control	Executes a script whose name is stored in a string expression.	EXEC($ScriptName[,Parm1,Parm1,…])==>ReturnValue
FCLOSE	File	Closes an open file.	FCLOSE(%FileNum)
FDELETE	File	Permanently deletes a file.	FDELETE($FileName)==>%Success
FEXISTS	File	Determines if a file exists.	FEXISTS($FileName)==>%Success
FILENO	File	Obtains the system integer file descriptor from a file handle.	FILENO(%FileHandle)==>%FileDescriptor
FINDSTR	Variable	Searches a string for a regular expression pattern.	FINDSTR($String,$Substring==>$FoundText
FMODTIME	File	Returns the time value of a last modified date/time stamp for a file.	FMODTIME(File)==>%EpochSeconds
FOPEN	File	Opens a file for I/O access.	FOPEN($FileName[,%Mode])==>%FileHandle
FORMATSTR	Variable	Formats a string by combining literal characters with conversion specifications.	FORMATSTR($String[,expr1,[expr2,…,[expr*n*] …]])
FPOS	File	Returns an open file's current record pointer position.	FPOS(%FileNum)==>%Position

Command	Command Type	Description	Syntax
FREAD	File	Reads values from an open file into variables.	FREAD(%FielNum,var1[,var2,…,[var*n*]…])==>%QtyRead
FRENAME	File	Renames a file.	FRENAME($CurrentName,$NewName)==>%Success
FREWIND	File	Moves an open file's current record pointer to the beginning.	FREWIND(%FileNum)
FSEEK	File	Moves an open file's current record pointer to a byte offset.	FSEEK(%FileNum,%Position)==>%Success
FWRITE	File	Writes expression to an open file.	FWRITE(%FileNum,expre[,%NEWLINE])==>%Success
GETENV	Variable	Obtains the current value of the given environment variable.	GETENV($Variable)==>$Value
GETPID	Variable	Obtains the system process identifier for this script.	GETPID()==>%ProcessId
GOSUB	Flow Control	Immediately transfers script execution to the specified label and waits until the called routine finishes execution.	GOSUB*Label
GOTO	Flow Control	Immediately transfers script execution to the specified label.	GOTO*Label
HEXSTR	Variable	Converts an integer to a hex string.	HEXSTR(%Number)==>$Hex
HUMID	Physical Interface Control	Reads the current humidity from a sensor unit.	HUMID(%Port)==>%Humidity
ICON	Alerting	Changes icon characteristics.	ICON(%Status[,$Message[,%Class[,$Name]]])
ICONMSG	Alerting	Returns an icon's current message.	ICONMSG([%Class[,$Name]])==>$Message

Command	Command Type	Description	Syntax
ICONNAME	Alerting	Returns an icon's name.	ICONNAME([%Port]])==>$Name
ICONSTATUS	Alerting	Returns an icon's current status.	ICONSTATUS([%Class[,$Name]])==>%Status
IF (ENDIF)	Flow Control	Evaluates an expression for TRUE or FALSE. (If…Then statement)	IF…[ELSE…]ENDIF
INC	Variable	Adds one to a numeric variable's value.	INC%Variable
JOIN	Variable	Combines the elements of an array into a string.	JOIN($Array,$Delimiter)==>$String
KEY	Console Message	Enters a character string to the specified console.	KEY(%Port,$Keys[,%Timeout])==>%RetCode
LEFTSTR	Variable	Returns the leftmost specified number of characters of a string expression.	LEFTSTR($String,%Count)==>$Substr
LEFTSTR	Variable	Returns the leftmost specified number of characters of a string expression.	LEFTSTR($String,%Count)==>$Substr
LEN	Variable	Returns the number of characters in a string expression.	LEN($StringExpr)==>%Count
LOG	File	Enters a message in a log.	LOG(%LogType,$Message[,%Status])
LOWER	Variable	Converts uppercase characters to lowercase.	LOWER($String)==>$Lowercase
MKDTEMP	File	Creates a unique temporary directory for I/O access.	MKDTEMP($Pattern)==>$DirectoryName
MKSTEMP	File	Opens a unique temporary file for I/O access.	MKSTEMP($Pattern)==>%FileHandle
MKTEMP	File	Returns a unique file name	MKTEMP($Pattern)==>$FileName
MONIKER	Variable	Obtains the product name.	MONIKER()==>$Name

MVSCOMMAND	Event	Enters commands to MVS using the GW-MVS agent (optional software).	MVSCOMMAND(%ObjID, $CmdArray, $Output Array, %ErrArray $ErrorTextArray[, %PortID] [, $JobName] [, $Filter1] [, $Filter2] [, $Filter3] [, %MvsMsgCount] [, %MvsResponseTime]) ==> %ReturnValue
OBJEXEC	Object Manager	Executes an action on an object.	OBJEXEC(%ObjID,$Action[,Parms…])==>ReturnValue
OBJGET	Object Manager	Returns the current value in an object's field.	OBJGET(%ObjID,$ObjFieldName)==>$Current Value
OBJGETARRAY	Object Manager	Populates an associative string array with the current field values from an object.	OBJGETARRAY(%ObjID,$AssocArray)==>%Success
OBJID	Object Manager	Returns the ID (unique only for the particular script thread) for a MCC object.	OBJID(%Class,$ObjKeyExpr)==>%ObjectID
OBJIDARRAY	Object Manager	Returns the number of children of the selected object ID.	OBJIDARRAY(%Class, %ObjIDParent, %AssocArray)==> %Children
OBJSET	Object Manager	Populates an associative numeric array with object IDs from the children of a parent object.	OBJIDARRAY(%Class,%ObjIDParent,%AssocArray)
OBJSET	Object Manager	Sets the current value in an object's field.	OBJSET(%ObjID,$ObjFieldName,$NewValue)==>%ErrCode
OBJSETARRAY	Object Manager	Sets the field values for an object from an associative string array.	OBJSETARRAY(%ObjID,$AssocArray)==>%Success
PARMS	Variable	Receives parameters into the script.	PARMSvar1[,var3,…,[var*n*]…]]]
PORT	Console Message	Returns the port number for a console definition.	PORT(%Class[,$IconName])==>%Port
QCLOSE	Console Message	Closes a message queue.	QCLOSE(%QueueID)

QOPEN	Console Message	Opens a new queue of OS printer console messages.	QOPEN([%ObjIDArray])==>%QueueID
QPREVIEW	Console Message	Previews the current message on the screen (before it enters the queue).	QPREVIEW(%QueueID,$ResultArray)==>%Ret Code
QREAD	Console Message	Reads the next message from a message queue.	QREAD(%QueueID,$MsgArray,%Wait[<$Filter]) ==>$MsgLine
QSKIP	Console Message	Moves a current message pointer for a queue.	QSKIP(%QueueID,%Skip)
REPEAT	Flow Control	Repeats a sequence of commands until an expression evaluates to TRUE.	REPEAT…UNTIL
REPSTR	Variable	Returns a string repeated a specified number of times.	REPSTR($String,%Count)==>$RepeatedString
RETURN	Flow Control	Returns execution to the calling routine, passing an optional return value.	RETURN[Expression]
RIGHTSTR	Variable	Returns the rightmost specified number of characters of a string expression.	RIGHTSTR($String,%Count)==>$Substr
SCANB	Console Message	Same as SCANP except for use with the BLOCKSCSN command.	SCANB(%Port,$Text,*Found)
SCANP	Console Message	Searches a console for a specified character string.	SCANP(%Port,$Text,%Wait,*Found[,$Array])
SCRIPTCANCEL	Scriptin g	Obtains the system integer file descriptor from a file handle	SCRIPTCANCEL($ScriptName,$Class,$Name)
SCRIPTGETACTIVE	Scriptin g	Retrieves information on all active scripts into an associative array.	SCRIPTGETACTIVE($AssocArray)==>%ErrCod e
SCRNTEXT	Console Message	A full or partial screen snapshot—returns characters from a console screen.	SCRNTEXT(%Port,%Start,%Length==>$Text
SECONDS	Time	Returns the time value for the current time.	SECONDS()==>%EpochSeconds

SET	Variable	Make the contents of a variable equal to the specified expression.	SETVariable:=Expression
SNMP_GET	SNMP	Retrieves the value of a specified MIB object.	SNMP_GET($Alias,$MIBOID)==>$Value
SNMP_GETNEXT	SNMP	Retrieves the value and the object ID of the next logical MIB object to a specified MIB object.	SNMP_GETNEXT($Alias,$MIBOID,$NextMIB OID) ==>$Value
SNMP_GETTABLE	SNMP	Retrieves a complete table of information from the MIB on the specified agent.	SNMP_GETTABLE($Alias,$MIBOID,$TableArr ay[, $Delimiter])==>%ReturnCode
SNMP_SET	SNMP	Sets the value of a specified MIB object.	SNMP_SET($Alias,$MIBOID,$Value)==>%Return Code
SNMP_TRAPSEND	SNMP	Sends a trap to one or more hosts.	SNMP_TRAPSEND($Alias,%TrapNum[[,%EntN um[,$MIBOID[,$Time[,$HostName]]]])==>%Ret urnCode
SPLIT	Variable	Populates an array with the fields of a string delimited by a string.	SPLIT($Array,$String,$Delimiter)
START	Flow Control	Initiates execution of another script for concurrent processing.	STARTScriptName(Parms)[,%Class[,$Name]])
STOP	Flow Control	Halts execution of another script.	STOP(ScriptName[,%Class[<$Name]])
STR	Variable	Converts a numeric expression to a string.	STR(%Number)==>$String
SUBSTR	Variable	Extract a substring from a character string.	SUBSTR($String,%Start[,%Count])==>$SubStr
SWITCH	Flow Control	Executes command(s) based on the value of an expression.	SWITCH…CASE…[DEFAULT…]ENDSWITCH
SYSEXEC	Flow Control	Executes a (Unix) command on the MCC host system with parameters.	SYSEXEC($String)==>%Return

TEMP	Physical Interface Control	Reads the current temperature from a sensor unit.	TEMP(%Port)==>%Temp
TIME	Time	Converts a time string to a time value.	TIME([TimeString])==>%MidnightSeconds
TIMESTR	Time	Formats epoch seconds into a date/time string.	TIMESTR(%EpochSeconds,$Format)==>$For matted
TRIMSTR	Variable	Removes leading and trailing spaces from a string.	TRIMSTR($String [, %Where])==>$TrimmedString
TSOEREXX	Event	Enters commands to TSO/E REXX.	TSOEREXX(%ObjID, $CmdArray, %ErrorArray, $ErrorTextArray) ==> %ReturnValue
UPPER	Variable	Converts lowercase characters to uppercase.	UPPER($String)==>$UpperString
VAL	Variable	Converts a string expression to a number.	VAL($String)==>A%Number
VERSION	Variable		
WAITFOR	Time	Pauses script execution for the specified number of seconds.	WAITFOR(%Seconds)
WAITUNTIL	Time	Pauses current script execution until the specified time is reached.	WAITUNTIL(%MidnightSeconds)
WHILE (ENDWHILE)	Flow Control	Repeats a sequence of commands while an expression evaluates to TRUE.	WHILE…ENDWHILE

Table 26. Summary of Command Syntax—By Command

Index

visara international

707131-001