JP1 Version 11

JP1/IT Desktop Management 2 - Asset Console Creating an Access Definition File Guide

3021-3-B57-30(E)

Notices

■ Relevant program products

JP1/IT Desktop Management 2 - Manager

P-2A42-78BL JP1/IT Desktop Management 2 - Manager version 11-50

The above product includes the following:

P-CC2A42-7ABL JP1/IT Desktop Management 2 - Manager version 11-50 (for Windows Server 2016, Windows Server 2012, and Windows Server 2008 R2)

P-CC2A42-7BBL JP1/IT Desktop Management 2 - Agent version 11-50 (for Windows Server 2016, Windows 10, Windows 8.1, Windows 8, Windows Server 2012, Windows 7, and Windows Server 2008 R2)

P-CC2A42-7CBL JP1/IT Desktop Management 2 - Network Monitor version 11-50 (for Windows Server 2016, Windows 10, Windows 8.1 Enterprise, Windows 8.1 Pro, Windows 8 Enterprise, Windows 8 Pro, Windows Server 2012, Windows 7 Enterprise, Windows 7 Professional, Windows 7 Ultimate, and Windows Server 2008 R2)

P-CC2A42-7DBL JP1/IT Desktop Management 2 - Asset Console version 11-50 (for Windows Server 2016, Windows Server 2012 and Windows Server 2008 R2)

■ Trademarks

■ Restrictions

■ Issued

Nov. 2017: 3021-3-B57-30(E)

■ Copyright

Summary of amendments

The following table lists changes in this manual (3021-3-B57-30(E)) and product changes related to this manual.

Changes	Location
None.	--

Legend:

--: Not applicable

In addition to the above changes, minor editorial corrections were made.

Preface

This manual describes how to use scripts of JP1/IT Desktop Management 2 - Asset Console (abbreviated hereafter to *Asset Console*) to add your own original processes.

■ Intended readers

This manual is intended for the following users:

- System administrators who create an asset management system that uses Asset Console
- Asset administrators who manage asset information
- Those who have basic knowledge of object-oriented techniques

■ Organization of this manual

This manual is organized into the following chapters and an appendix:

1. Overview
: Chapter 1 explains the aims of creating an access definition file and the work flow for doing so.

2. Creating Access Definition Files
: Chapter 2 explains how to create access definition files and provides two examples of access definition files.

3. Executing Access Definition Files
: Chapter 3 explains how to execute an access definition file.

4. Tags Used in Access Definition Files
: Chapter 4 provides detailed explanations of the tags that can be used in access definition files.

5. Embedded Functions Used in Access Definition Files
: Chapter 5 explains the embedded functions that can be used in an access definition file.

A. Version Changes
: Appendix A describes changes in each version.

B. Reference Material for This Manual
: Appendix B provides reference material for readers of this manual.

Contents

1 Overview

This chapter explains the purpose of creating an access definition file and the flow of operations to do so.

1.1 Purpose of creating an access definition file

This section explains the purpose of creating an access definition file on an asset management system that uses Asset Console.

1.1.1 Ability to input and output data in any format

An asset management system built with *Asset Console* uses an *asset management database* to provide integrated management of hardware asset information, including information on network devices, software asset information, and maintenance contract information. Using a database to provide integrated management of asset information helps rationalize and reduce management costs of IT asset management activities that accompany jobs such as stocktaking and adding or transferring equipment.

To import and export information registered to the asset management database of an asset management system, normally you use commands provided by Asset Console (`jamimport` and `jamexport`), or Asset Console's job menu (**Import** and **Export**). Use of the commands results in batch import and export processing with a fixed format, while use of the job menu results in batch import and export processing with items specified.

However, by using a special script (called an *access definition file*) provided by Asset Console to define processing, you can specify detailed conditions and import and export asset management database information without having to know the format.

1.1.2 Ability to use asset information for a variety of jobs

You can use without change a set of execution results produced by the access definition file. In addition, you can combine such execution results with, for example, email transmission processing created in Windows Script, so that you can create a task that sends an email to the asset manager whenever an asset information item exceeds a predetermined criterion.

You can also register in Windows Task Scheduler a task for executing a process designed to monitor asset information on a regular basis, significantly reducing the asset manager's workload.

1.1.3 Ability to access directory service information

You can use Asset Console's special scripts to define access to *directory information* managed by any directory service that uses LDAP (lightweight directory assistance protocol), which provides an open DAP based on the X.500 data model running over TCP/IP. This enables operations to acquire user and organizational information from directory information.

1.2 What is an access definition file?

An access definition file is used to define methods of importing and updating data in an asset management database.

Access definition files are text format files, which means that you do not need any special application to create one. Neither are there special limitations to file naming.

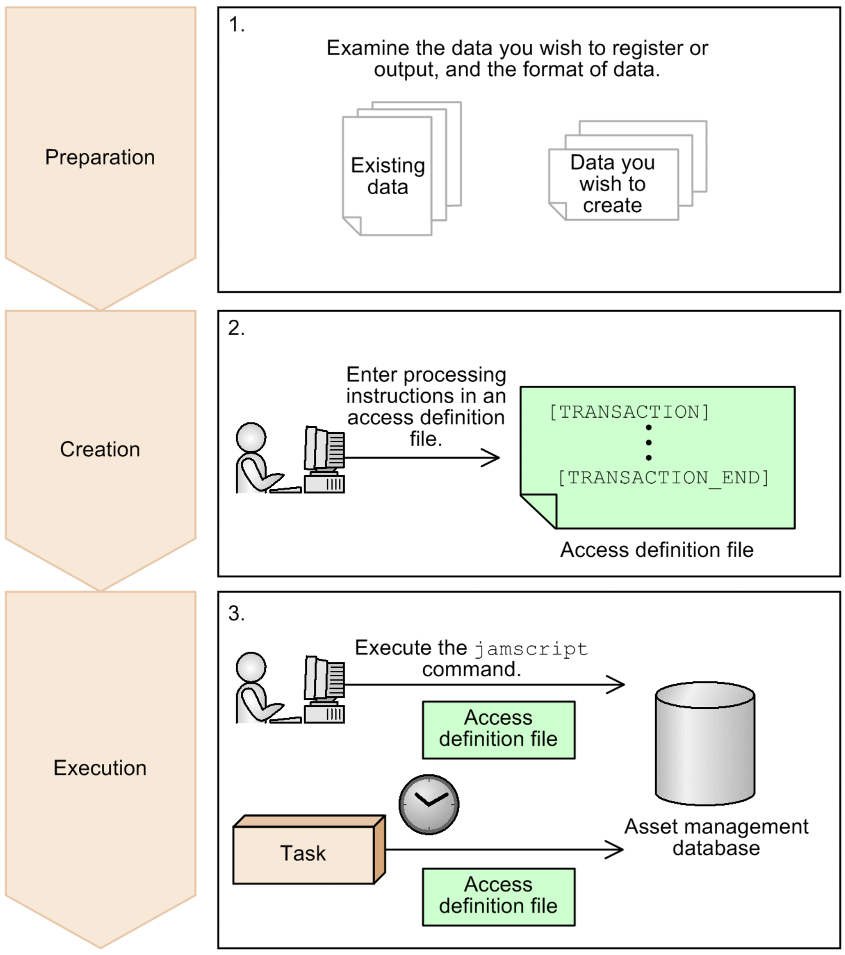
Special tags and embedded functions provided by Asset Console, along with variables and operators, can be combined in access definition files to define various processes.

The process defined in an access definition file is executed using the `jamscript` command or by means of a task defined using Task Scheduler.

1.3 Creating an access definition file to extend functionality

This section explains the operation flow of creating an access definition file to add your own process.

Figure 1–1: Operation flow of creating an access definition file to add your own process



Preparation
1.
Examine the data you wish to register or output, and the format of data.
Existing data
Data you wish to create
Creation
2.
Enter processing instructions in an access definition file.
[TRANSACTION]
[TRANSACTION_END]
Access definition file
Execution
3.
Execute the jamscript command.
Access definition file
Task
Access definition file
Asset management database

1. Examine the data registered in the asset management database, a list of the asset management database information you wish to output and create, and other details pertaining to the process you wish to add. For details about the classes and properties of information managed by an asset management database, see the *JP1/IT Desktop Management 2 - Asset Console Configuration and Administration Guide*.

   At this time, you must also prepare the data needed for processing. Access definition files can import and export CSV format files.

2. In the access definition file, code the procedure for registering and updating the information in the asset management database, and for importing or exporting the required information from or to a CSV format file.

   For details about how to create an access definition file, see *2. Creating Access Definition Files*.

3. Execute the `jamscript` command (execute access definition file). Processing is executed under the conditions specified in the access definition file.

   Note also that the process coded in the access definition file can be registered in Windows Task Scheduler for execution on a regular basis. Differentiate which way to execute by the type of job each process is performing.

   For details about how to execute the `jamscript` command, see *3.1 Executing from a command line*. For details about registering tasks, see *3.2 Executing using a task*.

2 Creating Access Definition Files

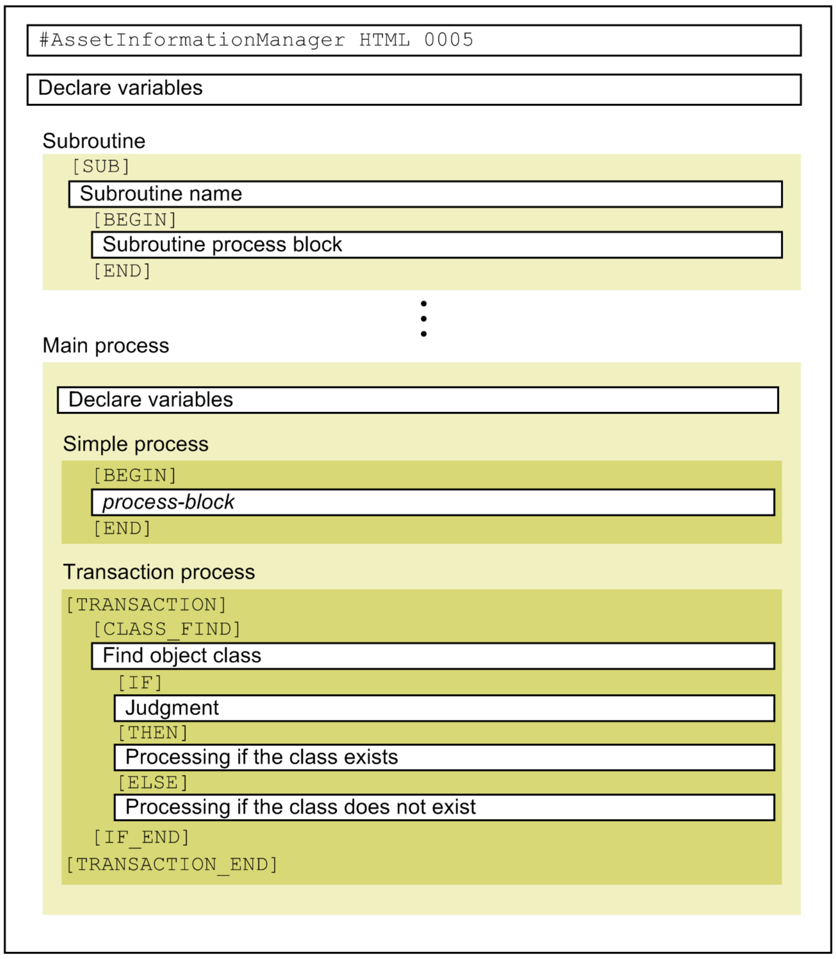
This chapter explains how to create access definition files and provides two examples of access definition files.

2.1 Basic format of access definition files

Access definition files are created using tags, variables, operators, and embedded functions.

The following figure shows the *basic format* of an access definition file. Always make sure to code any subroutines before the main process.

Figure 2–1: Format of access definition files



2.2 Coding method

This section explains the coding rules for access definition files and describes directory information operations as processes that can be described in an access definition file.

2.2.1 Coding rules

The following lists the coding rules for access definition files:

- Do not create an access definition file that contains more than 2,097,152 bytes (2 megabytes).
- Use the ASCII code to enter text in an access definition file.
- Use CRLF (`\r\n`) as the carriage return code.
- The first line must contain the script header and cannot have a space or tab character at the beginning or end.
- On the second and subsequent lines, you can use a space or a tab at the beginning or end of the line. You can also enter a comment following a hash mark (`#`).
- Enclose constants in single quotation marks (`'`).
- Use two single quotation marks (`''`) to enclose a character constant.
- You can include the contents of a file by specifying its file name following `#%include` on the second and subsequent lines. However, only access definition files can be included.

Remarks

Data integrity may be lost if the `jamscript` command is executed for the same database from more than one computer. Take this into account in your access definition file code.

2.2.2 Specifying a script header

A script header is specified on the first line of the access definition file. You must always specify a script header. If you do not, the `jamscript` command does not recognize the file as an access definition file, and no processing is performed.

Enter the script header beginning from the first column as follows:

```
#AssetInformationManagerΔHTMLΔ0005
```

The four-digit numeric in the script header (`0005`) indicates the version of the script.

2.2.3 Coding rules for variables

This subsection explains the variables used in access definition files. You can use the following types of variables in access definition files:

- Variables
- Array variables

If not otherwise required, these variables may both be referred to collectively as *variables*.

(1) Characters that can be used in variable names

Alphanumeric characters and the underscore (_) can be used in variable names, with the exception that a variable name cannot begin with a number. In addition, note that case is treated as significant in variable names.

(2) Character strings that cannot be used as variable names

The following reserved character strings cannot be used as variable names:

`NORMAL`, `ERROR`, `NODATA`, `MULTI`, `FLUSH`, `RENEW`, `NEW`, `ADD`, `CRLF`

(3) Declaring variables

Before you use a variable or an array variable in an access definition file, you must first declare it. Use a `[VAR]` tag to declare a variable, and an `[ARRAY]` tag to declare an array variable. For details about how to use the `[VAR]` and `[ARRAY]` tags, see *[VAR] (declare variable)* and *[ARRAY] (declare array variable)*, respectively, in *4. Tags Used in Access Definition Files*.

(4) Assigning a value to a variable

Use the assignment operator (=) to assign a value to a variable.

(a) Variables

You can assign the following types of values to a variable:

- Character strings
- Variable values
- Class.property values

You can code assignment statements in either of the following tags, except that assignment statements for class.property values can only be coded in the `[GET_VALUE]` tag.

- `[SET_VALUE]` tag
- `[GET_VALUE]` tag in the `[CLASS_FIND]`, `[ASSOC_FIND]` and `[JOIN_FIND]` tags

(b) Array variables

For array variables, use the embedded function `$SETARRAY` to code an assignment statement.

(5) Referencing variables

Use the following format to reference array variables:

```
DATA=$GETARRAY(array-variable-1,array-variable-2)
```

(6) Valid range of variables

Variables and array variables are valid when coded between their corresponding `[BEGIN]` and `[END]` tags.

2.2.4 Operators

This subsection lists and explains the operators that are available for use. It also explains how to specify *class*.*property* values to perform fuzzy searches.

(1) Available operators

There are three types of operators that you can use. A character string concatenation operator, which is a binary operator that joins two character strings; an assignment operator, which assigns the operation result on its right side to the value on its left side; and relational operators, which compare a *class*.*property* value or variable on the operator's right side to the one on its left side. The following table lists and describes the operators that are available.

Table 2–1: Available operators

Operator		Description
Binary operator	+	Joins the character string immediately following it to the character string immediately preceding it. You can specify a constant, a variable, or an embedded function as the character string. If you specify an embedded function, the join operation is performed on the character string that the embedded function returns. You can also join several character strings by using more than one + operator, as in `A+B+C`.
Assignment operator	=	Assigns the constant or operation result on its right side to the value on its left side. If you specify an embedded function, this operator assigns the value returned by the embedded function.
Relational operators	=	Compares the value on its right side to the value on its left side, and returns a logical true if the values match.
	!=	Compares the value on its right side to the value on its left side, and returns a logical true if the values do not match.
	>	Compares the value on its right side to the value on its left side, and returns a logical true if the value on its left side is greater than the value on its right side.
	>=	Compares the value on its right side to the value on its left side, and returns a logical true if the value on its left side is greater than or equal to the value on its right side.
	<	Compares the value on its right side to the value on its left side, and returns a logical true if the value on its left side is less than the value on its right side.
	<=	Compares the value on its right side to the value on its left side, and returns a logical true if the value on its left side is less than or equal to the value on its right side.
	<>	This operator produces the same results as the != operator. It compares the value on its right side to the value on its left side, and returns a logical true if they differ.
	`LIKE`#	This operator is used to specify condition expressions for performing fuzzy searches. It compares a character string on its right side that contains one or more wildcard characters to a *class*.*property* value on its left side, and returns a logical true if they match. For details about the wildcard characters that can be specified, see *(2) Wildcard characters that can be specified in a condition expression of a fuzzy search*, below.
	`NOT_LIKE`#	This operator is used to specify condition expressions for performing fuzzy searches. It compares a character string on its right side that contains one or more wildcard characters to a *class*.*property* value on its left side, and returns a logical true if they do not match. For details about the wildcard characters that can be specified, see *(2) Wildcard characters that can be specified in a condition expression of a fuzzy search*, below.

#
Cannot be used in an `[IF]` tag.

(2) Wildcard characters that can be specified in a condition expression of a fuzzy search

Using the relational operators `LIKE` and `NOT_LIKE`, you can perform fuzzy searches on *class*`.`*property* values by specifying the `%` (percent) and _ (underscore) wildcard characters in the character string on the right side of the operator.

The following bullets explain how to use these wildcard characters.

- `%` (percent)
  Finds character strings that contain zero or more instances of any character at the position specified. A few specification examples are given below:
  - To find character strings in *class*`.`*property* values that begin with `ABC` (leading match search), specify `ABC%`.
  - To find character strings in *class*`.`*property* values that end with `ABC` (trailing match search), specify `%ABC`.
  - To find character strings in *class*`.`*property* values that include `ABC`, specify `%ABC%`. This search condition finds character strings that include `ABC` anywhere in the character string.
- _ (underscore)
  Finds character strings that contain a single instance of any character at the specified position. For example, to find five characters that contain `ABC` beginning from the third character, such as `AAABC`, specify two contiguous underscores, as in `__ABC`. This specification also finds character strings such as `BBABC` and `CCABC`.

If you do not use either wildcard character `%` (percent) or _ (underscore) following `LIKE` or `NOT_LIKE`, the system searches for perfect matches. In this case, `LIKE` is equivalent to =, and `NOT_LIKE` is equivalent to `!=`.

Note

To find character strings that contain a percent (`%`) or underscore (_) character, specify a forward slash (`/`) immediately preceding the character, such as `/%` or `/_`. To indicate a forward slash (`/`), specify two contiguous forward slashes, such as `//`.

2.2.5 Coding rules for embedded variables

The following shows the format of arguments that can be specified in embedded variables.

- Variables
  For variables and array variables, specify names that have been defined in an access definition file.
- Constants
  For constants, specify a character string enclosed in single quotation marks (`''`).
- Numeric characters
  For numeric characters, specify a character string consisting of numeric characters between 0 and 9 inclusive.

2.2.6 Manipulating directory information

Using the `$LDAPACS` embedded function, you can perform operations such as authenticating connections to a directory service, searching directory information, and acquiring entries and attributes.

This subsection lists the functions that can be used by the `$LDAPACS` embedded function and explains the rules for manipulating objects.

(1) Functions that can be used by the $LDAPACS embedded function

The following table lists the functions that can be used by the `$LDAPACS` function, with their descriptions.

Table 2–2: List of functions that can be used by the $LDAPACS embedded function

Function name	Description
`CONNECT`	Authenticates a connection to a directory service.
`CONVERT`	Converts a character string to one that can be used for searching directory information.
`DISCONNECT`	Releases a connection to directory services.
`FIRSTENTRY`	Acquires the first entry found.
`FREEENTRY`	Releases an entry.
`FREERESULT`	Releases the search results.
`GETDN`	Acquires the DN of an entry.
`NEXTENTRY`	Acquires the second and subsequent entries found.
`SEARCH`	Searches a directory service.
`SELECTVALUE`	Acquires an attribute value.

For details about the descriptions, formats, parameters, and return values of these functions, see *$LDAPACS (access directory)* in *5. Embedded Functions Used in Access Definition Files*.

(2) Rules for manipulating objects

Acquiring entries

- You cannot acquire the second and subsequent found entries without first acquiring the first found entry.
- You cannot acquire a particular entry more than once.
- You cannot modify, delete, or add an entry using a script.

Acquiring attribute values

- You can acquire the value of an attribute by specifying the name of the attribute.
- You cannot modify, delete, or add attribute values using a script.
- In cases for which more than one value with the same attribute name is assigned, you can use multiple instances of `SELECTVALUE` to sequentially specify the values.

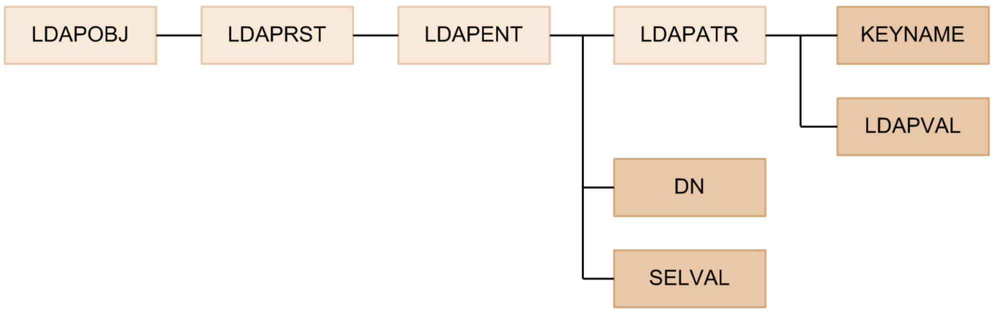
Using and referencing object parameters

- Once an object parameter has been released, it cannot be used or referenced again.

(3) Memory management structure of objects

The following figure shows the memory management structure of objects.

Figure 2–2: Memory management structure of objects



LDAPOBJ, LDAPRST, LDAPENT, and LDAPATR manage all the objects below themselves. Accordingly, releasing LDAPOBJ, LDAPRST, LDAPENT, and LDAPATR releases all the objects below them. Note, however, that KEYNAME, LDAPVAL, DN, SELVAL, and the other lowermost character strings cannot be released as objects.

(4) Example

An example using the $LDAPACS embedded function:

```
[VAR]
  STATUS
  MSG
  HOST
  PORT
  FILTER
  BASE
  SCOPE
  FIRST
  LDOBJ
  LDRST
  LDENT
  DN
  NAME

[SET_VALUE]
  HOST = 'localhost'
  PORT = '389'
  BASE = 'ou=people,o=xxxxxxx.co.us'
  SCOPE= 'LDAP_SCOPE_ONELEVEL'

[SET_VALUE]
  $LDAPACS('CONNECT',LDOBJ,HOST,PORT,'','')                # CONNECT
  STATUS = $GETSTATUS()

  [SET_VALUE]
    FILTER = '(&(objectclass=*)(title;lang-ja='
    FILTER = FILTER+$LDAPACS('CONVERT','Supervisor')          # CONVERT
    FILTER = FILTER+'))'
    # FILTER=(&(objectclass=*)(title;lang-ja=\E4\B8\BB\E4\BB\BB))

    $LDAPACS('SEARCH',LDRST,LDOBJ,BASE,FILTER,SCOPE)     # SEARCH
    FIRST = 1

    [DO]
      [IF]
        FIRST = 1
        [THEN]
          [SET_VALUE]
            $LDAPACS('FIRSTENTRY',LDENT,LDRST)          # GET FIRST ENTRY
            STATUS = $GETSTATUS()
            FIRST = 0
        [ELSE]
          [SET_VALUE]
            $LDAPACS('NEXTENTRY',LDENT,LDRST)           # GET NEXT ENTRY
            STATUS = $GETSTATUS()
      [IF_END]

      [IF]
        STATUS = NORMAL
        [THEN]
          [SET_VALUE]
            $LDAPACS('GETDN',DN,LDENT)                  # GET DN
            $LDAPACS('SELECTVALUE',NAME,LDENT,'cn')     # GET VALUE OF CN
            MSG='DN ['+DN+'] is '+NAME
            $ECHO(MSG)
            $LDAPACS('FREEENTRY',LDENT)                 # FREE ENTRY OBJECT
        [ELSE]
          [SET_VALUE]
            $BREAK()
        [IF_END]
    [DO_END]

  [SET_VALUE]
    $LDAPACS('FREERESULT',LDRST)                        # FREE SEARCH OBJECT

[SET_VALUE]
  $LDAPACS('DISCONNECT',LDOBJ)                          # FREE LDAP OBJECT
```

2.3 Access definition file examples

This subsection provides examples of access definition files for the following two cases:

- For updating and deleting asset information conditioned on asset status
- For listing software assets of installed programs that are not authorized

2.3.1 Example for updating and deleting asset information conditioned on asset status

This subsection provides an example of updating and deleting asset information based on various statuses by evaluating the status of an asset.

- Remove a hardware asset whose status is erase, along with all related information.
- Perform the following updates and deletions on assets whose status is scrap (code range `500` to `719`):
  - Clear network information IP addresses.
  - Delete installed software information.

```
#AssetInformationManager HTML 0005

#==========================================================================
======
# Variable definitions
#==========================================================================
======
[VAR]
  STATUS
  DUMMY
  ECHOMSG
  MSG
  ASSET_ID
  ASSET_NO
  ASSET_ST
  NETWORK_ID
  IPADDR
  IPKIND
  WORK

#==========================================================================
======
# Output message routine
#==========================================================================
======
[SUB]
  ECHO

  [IF]
    MSG = '1'
    [THEN]
    [SET_VALUE]
      $ECHO(ECHOMSG)
  [IF_END]
```

```
[SUB_END]

#=========================================================================
======
# Deletion of hardware information (main)
#=========================================================================
======
[BEGIN]

  [SET_VALUE]
    MSG = $GETSESSION('MSG')             # Determine if message must be
output.

  [TRANSACTION]
    [ASSET_ITEM_LOOP]
      [CLASS_FIND]
        AssetInfo
      [FIND_DATA]
        (AssetInfo.AssetKind   = '001') and     # Hardware asset
        (AssetInfo.AssetStatus >= '500')         # Assets whose hardware
status is not active.
      [GET_VALUE]
        ASSET_ID = AssetInfo.AssetID
        ASSET_NO = AssetInfo.AssetNo
        ASSET_ST = AssetInfo.AssetStatus

      [IF]
        (ASSET_ST = '999')
        [THEN]                      # Delete asset information whose
asset status is erase.
          [DELETE]
            AssetInfo
          [DATA]
            AssetInfo.AssetID           = ASSET_ID

          [SET_VALUE]
            ECHOMSG = 'asset number ['+ASSET_NO+' (ID:'+ASSET_ID+')]
deleted.'
            $GOSUB(ECHO)

        [ELSE]                      # If asset status is neither erase
nor active.

          [ASSET_ITEM_LOOP]         # Clear network information IP
address.
            [CLASS_FIND]
              NetworkInfo
            [FIND_DATA]
              (NetworkInfo.AssetID = ASSET_ID)
            [GET_VALUE]
              NETWORK_ID = NetworkInfo.NetworkID
              IPADDR     = NetworkInfo.IPAddress
              IPKIND     = NetworkInfo.IPAddressKind

            [UPDATE]
              NetworkInfo
            [DATA]
```

```
              NetworkInfo.AssetID       = ASSET_ID
              NetworkInfo.NetworkID     = NETWORK_ID
              NetworkInfo.IPAddressKind = '002'
              NetworkInfo.IPAddress     = ''
            [SET_VALUE]
              ECHOMSG = '['+ASSET_ID+'] network information
                         ['+NETWORK_ID+' '+IPADDR+'] cleared.'
              $GOSUB(ECHO)

              $SETSTATUS('NORMAL')
          [ASSET_ITEM_LOOP_END]

          [ASSET_ITEM_LOOP]         # Delete all installed software
information.
            [CLASS_FIND]
              InstalledInfo
            [FIND_DATA]
              (InstalledInfo.AssetID = ASSET_ID)
            [GET_VALUE]
              WORK = InstalledInfo.InstalledID
            [DELETE]
              InstalledInfo
            [DATA]
              InstalledInfo.AssetID           = ASSET_ID
              InstalledInfo.InstalledID       = WORK
              InstalledInfo.CreationClassName = 'InstalledInfo'
            [SET_VALUE]
              ECHOMSG = '['+ASSET_ID+'] installation information ['+WORK
+'] deleted'
              $GOSUB(ECHO)

              $SETSTATUS('NORMAL')
          [ASSET_ITEM_LOOP_END]

      [IF_END]

    [SET_VALUE]
      $SETSTATUS('NORMAL')
    [ASSET_ITEM_LOOP_END]

  [SET_VALUE]
    $SETSTATUS('NORMAL')
  [TRANSACTION_END]

[END]
#==========================================================================
======
```

2.3.2 Example for listing software assets of installed programs that are not authorized

This subsection provides an example of listing the assets of software not authorized to be installed, along with the unauthorized programs, and outputting the list to a file. To create the list, this example performs a search that uses the `[JOIN_FIND]` tag to join multiple classes.

- Sample output file
  The following are the contents output to file by the example in this subsection:

```
"Software name","Version","Asset No.","Group information","User"
"SoftwareA","0100","0000000001","Head Office/Sales Dept./Section 1","user1"
"SoftwareA","0101","0000000001","Head Office/Sales Dept./Section 1","user1"
"SoftwareB","0100","0000000001","Head Office/Sales Dept./Section 1","user1"
"SoftwareA","0100","0000000002","Head Office/Sales Dept./Section 2","user2"
```

```
#AssetInformationManager HTML 0005

#============================================================================
====
# Variable definitions
#============================================================================
====
[VAR]
  STATUS
  DUMMY
  WORK
  ECHOMSG
  MSG
  FILENAME

[SET_VALUE]
  MSG = $GETSESSION('MSG')

#============================================================================
====
# Output of information on unauthorized software to CSV file (main)
#============================================================================
====
[BEGIN]

  [VAR]
    ASSET_NO
    GROUP
    USER
    SOFTNAME
    SOFTVR
    FH
    LINE
  [ARRAY]
    OUTLINE

  [SET_VALUE]
    FILENAME = $GETSESSION('CSV')
    FH = $FILEOPEN(FILENAME, RENEW)

    $SETARRAY(OUTLINE,'Software name')
    $SETARRAY(OUTLINE,'Version')
    $SETARRAY(OUTLINE,'Asset No.')
    $SETARRAY(OUTLINE,'Group name')
    $SETARRAY(OUTLINE,'User')

    $FILEARRAY(FH, OUTLINE)
    $CLEARARRAY(OUTLINE)
    LINE = 0

  [TRANSACTION]
    [ASSET_ITEM_LOOP]
      [JOIN_FIND]
      [JOIN]
        joinassoc;InstalledListLink;
        joinfrom;InstalledList,InstalledInfo;
        jointype;INNER;
      [JOIN]
        joinassoc;InstalledLink;
        joinfrom;InstalledInfo,AssetInfo;
        jointype;INNER;
      [JOIN]
        joinfrom;AssetInfo,GroupInfo;
        jointype;OUTER;
        joinkey;AssetInfo.GroupID,GroupInfo.GroupID;
      [FIND_DATA]
        (InstalledList.InstalledPermit = '2')
      [GET_VALUE]
        SOFTNAME = InstalledList.InstalledName
        SOFTVR   = InstalledList.InstalledVersion
        ASSET_NO = AssetInfo.AssetNo
        GROUP    = GroupInfo.FullPathName
        USER     = AssetInfo.UserName

      [SET_VALUE]
        $SETARRAY(OUTLINE,SOFTNAME)
        $SETARRAY(OUTLINE,SOFTVR)
        $SETARRAY(OUTLINE,ASSET_NO)
        $SETARRAY(OUTLINE,GROUP)
        $SETARRAY(OUTLINE,USER)

        $FILEARRAY(FH, OUTLINE)
        $CLEARARRAY(OUTLINE)

        LINE = $ADD(LINE, 1)
    [ASSET_ITEM_LOOP_END]

  [SET_VALUE]
    $SETSTATUS('NORMAL')
  [TRANSACTION_END]

  [SET_VALUE]
    $FILECLOSE(FH)

  [IF]
    (LINE = 0)
    [THEN]
      [SET_VALUE]
        ECHOMSG = 'No data found.'
        $ECHO(ECHOMSG)
        $EXIT(1)
    [ELSE]
      [SET_VALUE]
        ECHOMSG = LINE+' items of data output'
        $ECHO(ECHOMSG)
  [IF_END]

[END]
```

3 Executing Access Definition Files

This chapter explains how to execute an access definition file that you have created.

3.1 Executing from a command line

This section explains how to execute the processing defined in an access definition file from a command line.

3.1.1 Before executing

This subsection explains what you need to know before using operation commands on an asset management server.

(1) Command execution procedure

To execute a command:

1. Log on as a user with administrator permissions.
2. Open a Command Prompt window, enter a command, and press **Enter**.
   The command executes.

(2) Notes on executing commands

If you specify a character string that includes a space in a command option argument, you must enclose the character string in double quotation marks (").

Example:

```
jamscript -f "c:\example\accessdef.txt" -s "CSV =c:\temp\data.csv"
```

(3) Location of command execution files

Command execution files are located in the following folder:

Asset-Console-installation-folder`\exe`

(4) If processing stops

If one of the following errors occurs, processing of the access definition file stops.

- An error exists in the syntax of the access definition file coding.
- When a class was newly registered or deleted, a key property was not set in the assignment statement.
- When a class was newly registered or deleted, a property# that must be specified in the assignment statement when registering a new class was not specified.
- A value specified in an assignment statement is outside the range of the property's attributes.
- No `[TRANSACTION]` tag was specified for a tag that accesses an asset management database.
- A syntax error exists in an argument of an embedded function.
- A variable type that cannot be specified exists in an argument of an embedded function, or a character or numeric value outside the specifiable range exists in an argument of an embedded function.
- There was insufficient memory when an embedded function was executed.
- An asset management database access error other than the above occurred.

#

For details about properties that must be specified when registering a new class, see the *JP1/IT Desktop Management 2 - Asset Console Configuration and Administration Guide*.

3.1.2 jamscript (execute access definition file) command

(1) Function

The `jamscript` command registers, updates, and deletes asset information in batch mode according to the definitions in the access definition file. As defined in the access definition file, this command can import and process information from a CSV file into an asset management database, and find information in an asset management database to export into a CSV file.

(2) Syntax

```
jamscript -f access-definition-file
          (-s variable-name=value (-s variable-name=value))
          (-bp basepath-name)
          (-c)
```

(3) Options

`-f` *access-definition-file*

Specifies the path to the desired access definition file. You can specify either its full path or a relative path. When specifying a relative path, reference the specification to the folder specified by the base path name. This option cannot be omitted.

`-s` *variable-name=value*

Specifies the variable name and value to be used as session information. You can also use this option to specify a value when you wish to modify the processing conditions (search conditions, for example) defined in the access definition file. You can also assign a value to this variable when the `$GETSESSION` embedded function is used in the access definition file.

For details on the `$GETSESSION` embedded function, see *$GETSESSION (get session information)* in *5. Embedded Functions Used in Access Definition Files*.

`-bp` *basepath-name*

Specifies the reference path name of the access definition file. You must specify a full path name for *basepath-name*. This option can be omitted. If omitted, *Asset-Console-installation-folder*`\scriptwork` is assumed.

`-c`

Use to analyze the syntax of the `jamscript` command, without executing it.

(4) Return values

This command has the following return values:

Return value	Description
`0`	Normal end.
`11`	An error was found in the syntax of a command option.

Return value	Description
21	The specified access definition file does not exist.
31	Memory is insufficient.
32	The environment needed to execute the access definition file is not configured properly.
34	An error was found in the access definition file.
52	The user cancelled execution.
101 or greater	The command ended in an error other than the above.

(5) Execution example

```
jamscript -f "c:\example\accessdef.txt" -s "CSV =c:\temp\data.csv"
```

(6) Notes

If you use the -bp option to specify a base path name, make sure that you create the CSV file in the first level of the specified folder. If no CSV file exists in the specified folder, an error occurs when the command is executed.

3.2 Executing using a task

This section explains how to execute the processing defined in an access definition file using a task defined in Windows Task Scheduler.

To register a task in Windows Server 2008 R2:

1. In the Windows task scheduler, double-click **Create Basic Task**.
2. Specify the settings as instructed by the task wizard that appears.
   Specify the `jamscript` command execution file (`jamscript.exe`) for the program to be executed.
   The `jamscript` command execution file is located in the following folder:
   Asset-Console-installation-folder`\exe`
   For the user name, specify a user that has Administrators privileges.
3. Select the **Open the Properties dialog for this task when I click Finish** check box, and then click the **Finish** button.
4. Click the **Task** tab in the dialog box that appears.
5. Under **Run**, enter the location of the access definition file as specified by the `-f` option.
   For details about the other options, see *3.1.2 jamscript (execute access definition file) command*.
6. Click the **OK** button.
   The dialog box closes, registering the task for executing the specified access definition file.

4 Tags Used in Access Definition Files

This chapter provides detailed explanations of the tags that can be used in an access definition file.

List of tags used in access definition files

The following table lists and describes the tags that can be used in access definition files.

Table 4–1: List of tags

Type	Description	Tags

Processing flow control	Begin or end CSV file information acquisition.	`[CSV_FILE_LOOP]`
		`[CSV_COLUMN_NAME]`
		`[CSV_FILE_LOOP_END]`

	Process block.	`[BEGIN]`
		`[END]`

	Conditionally execute a group of statements (`IF`).	`[IF][THEN]([ELSEIF][THEN])([ELSE])[IF_END]`
	Conditionally execute a group of statements (`SWITCH`).	`[SWITCH][CASE]([DEFAULT])[SWITCH_END]`

	Class search loop.	`[ASSET_ITEM_LOOP]`
		`[ASSET_ITEM_LOOP_END]`

	Define loop block.	`[DO][DO_END]`
	Define subroutine.	`[SUB][SUB_END]`
	Re-evaluate (dynamic generation).	`[EVALUATE][EVALUATE_END]`

Defining transactions	Define transaction.	`[TRANSACTION]`
		`[TRANSACTION_END]`

Variables and counters	Declare variable.	`[VAR]`
	Declare array variable.	`[ARRAY]`
	Substitute value.	`[SET_VALUE]`

Searching of classes and acquisition of values set to properties	Find object class.	`[CLASS_FIND]`
		`[FIND_DATA][GET_VALUE]`
		`([ORDER_ASC][ORDER_DESC])`

	Find association class.	`[ASSOC_FIND]`
		`[CLASS1]`
		`[CLASS2]`
		`[FIND_DATA][GET_VALUE]`
		`([ORDER_ASC][ORDER_DESC])`

	Find joined class.	`[JOIN_FIND]`
		`[JOIN][FIND_DATA][GET_VALUE]`
		`([ORDER_ASC][ORDER_DESC])`
		`([DUPLICATE])`

Data creation processing on search results from referenced class	Create object class.	`[APPEND]`
		`[DATA]`

	Update object class.	`[UPDATE]`
		`[DATA]`

	Delete object class.	`[DELETE]`
		`[DATA]`

Type	Description	Tags

Data creation processing on search results from referenced class	Add association class.	`[APPEND_ASSOC]`
		`[CLASS1][DATA]`
		`[CLASS2][DATA]`

	Delete association class.	`[DELETE_ASSOC]`
		`[CLASS1][DATA]`
		`[CLASS2][DATA]`

Legend:

Tags in parenthesis ( ) can be omitted.

Detailed explanation of tags used in access definition files

This section provides detailed explanations of the tags that can be used in access definition files, generally in the following format. The tag explanations appear in alphabetical order by tag name.

Tag

This subsection provides the name and a short description of the tag.

Syntax

This subsection provides the tag's syntax.

Values

This subsection explains the values that can be specified in the tag.

Status

This subsection explains the status of the processing coded in the tag and explains what each status means.

Example

This subsection provides a coding example using the tag.

[APPEND] (Create object class)

`[APPEND]` creates a new object class.

Syntax

```
[APPEND]
  object-class-name
  [DATA]
    assignment-statements
```

Values

- *object-class-name*

  Codes the name of the new class object to be created.

- *assignment-statements*

  Codes the values to be assigned to the properties.

  Always specify the properties and key properties that must be specified when registering a new object class. Note, however, that `CreationClassName` can be omitted.

  For details about the properties that must be specified when a new object class is registered, see the *JP1/IT Desktop Management 2 - Asset Console Configuration and Administration Guide*.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end.
`NODATA`	—
`ERROR`	The key has already been registered.
`MULTI`	—

Legend:

—: Not applicable

Example

The following example creates a new object class named `AssetInfo`:

```
[APPEND]
  AssetInfo
[DATA]
  AssetInfo.AssetID       = '10000'
  AssetInfo.AssetNo       = 'R11111'
  AssetInfo.AssetWorkKind = '001'
  AssetInfo.AssetStatus   = '002'
  AssetInfo.AssetKind     = '001'
  AssetInfo.AssetBranchNo = '0'

[SET_VALUE]
  STATUS = $GETSTATUS()
[IF]
  (STATUS != NORMAL)
  [THEN]
    [SET_VALUE]
      MSG = 'APPEND (' + STATUS + ')'
      $ECHO(MSG)
[IF_END]
```

[APPEND_ASSOC] (add association class)

`[APPEND_ASSOC]` uses an association class to link two object classes.

Syntax

```
[APPEND_ASSOC]
  association-class-name
  [CLASS1]
    object-class-name
    [DATA]
       assignment-statements
  [CLASS2]
    another-object-class-name
    [DATA]
       assignment-statements
```

Values

- *association-class-name*
  Codes the name of an association class to be created.
- *object-class-name*
  Codes the name of an object class to be linked by the newly created association class.
- *assignment-statements*
  Codes the information for linking the object classes by assigning values to the desired properties.
- *another-object-class-name*
  Codes the name of another object class to be linked by the association class.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end.
`NODATA`	—
`ERROR`	Indicates one of the following: • The key has already been registered. • No data exists for an object class to be linked.
`MULTI`	—

Legend:
  —: Not applicable

Remarks

To set up an association, the object classes to be linked must already be registered.

Example

The following example uses the association class `MemberLink` to link object classes `UserInfo` and `GroupInfo`, and to add a user to a group:

```
[APPEND_ASSOC]
  MemberLink
[CLASS1]
  UserInfo
[DATA]
  UserInfo.UserID = 'user1'
[CLASS2]
  GroupInfo
[DATA]
  GroupInfo.GroupID = '11000000'

[SET_VALUE]
  STATUS = $GETSTATUS()
[IF]
  STATUS != NORMAL
[THEN]
  [SET_VALUE]
    MSG = 'APPEND_ASSOC(' +STATUS+ ')'
    $ECHO(MSG)
[IF_END]
```

[ARRAY] (declare array variable)

`[ARRAY]` declares the variable name of an array.

Syntax

```
[ARRAY]
  variable-name
```

Values

- *variable-name*
  Specifies the variable name of an array that is being declared.

Example

The following example declares the array variable `ARY`:

```
[ARRAY]
  ARY
```

[ASSET_ITEM_LOOP] (class search loop)

`[ASSET_ITEM_LOOP]` specifies a class search loop. One class search loop is specified for each information item to be searched. The class search loops iterate the number of times that the specified conditions are matched.

A combination of the `[CLASS_FIND]`, `[ASSOC_FIND]`, and `[JOIN_FIND]` tags can be used for the class search conditions. For details about these tags, see *[CLASS_FIND] (find object class)*, *[ASSOC_FIND] (find association class)*, and *[JOIN_FIND] (find joined class)*.

Syntax

```
[ASSET_ITEM_LOOP]
  [CLASS_FIND]
  search-conditions-for-class
   . . .
  [BEGIN]
   processing-on-search-results
   . . .
  [END]
[ASSET_ITEM_LOOP_END]
```

Values

- *search-conditions-for-class*
  Codes the conditions for searching a class.
- *processing-on-search-results*
  Codes the processing performed on classes that match the search conditions.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	—
`NODATA`	End of data.
`ERROR`	—
`MULTI`	—

Legend:
—: Not applicable

Remarks

Whether or not processing in an `[ASSET_ITEM_LOOP]` is terminated is determined according to the status referenced by the `[ASSET_ITEM_LOOP_END]` loop end tag. If the status is `NORMAL`, processing continues. If the status is other than `NORMAL`, processing terminates. This last fact is useful when an error that occurs during processing interrupts processing at the point the error occurred.

When processing continues until no data remains, before determining whether or not to end processing according to the `[ASSET_ITEM_LOOP_END]` tag, we recommend that you use the `$SETSTATUS` embedded function to explicitly specify `NORMAL` as the status.

The status of the access definition file is updated on an ongoing basis by execution of tags and embedded functions. If processing that updates the status is specified more than once in an `[ASSET_ITEM_LOOP]` tag, you must determine whether or not to terminate processing based on the status that was last specified.

Example

The following example outputs a list of installed software names for asset ID `10000`:

```
[ASSET_ITEM_LOOP]
  [ASSOC_FIND]
    InstalledListLink
  [FIND_DATA]
    InstalledInfo.AssetID = '10000'
  [CLASS1]
    InstalledInfo
  [CLASS2]
    InstalledList
  [GET_VALUE]
    INSTNAME = InstalledList.InstalledName
  [SET_VALUE]
    MSG = 'Installed Software Name :' +INSTNAME
    $ECHO(MSG)
    $SETSTATUS('NORMAL')
[ASSET_ITEM_LOOP_END]
```

Execution result:

```
Installed Software Name : InstalledSoftware A
Installed Software Name : InstalledSoftware B
```

[ASSOC_FIND] (find association class)

`[ASSOC_FIND]` uses an association class to search classes.

If the class being searched matches the specified conditions, its property values are substituted into the declared variables. You can specify the `[ASSOC_FIND]` tag the number of times of required to find the base object class.

Syntax

```
[ASSOC_FIND]
  association-class-name
   ([FIND_DATA])
    condition-expression
  [CLASS1]
    object-class-name
  [CLASS2]
    another-object-class-name
  [GET_VALUE]
    assignment-statement
   ([ORDER_ASC] or [ORDER_DESC])
    sort-key
```

Values

- *association-class-name*
  Codes the name of the association class to be searched.
- *object-class-name*
  Codes the name of an object class that is linked by the association class.
- *another-object-class-name*
  Codes the name of another object class that is linked by the association class.
- *condition-expression*
  Codes the condition expression. To specify multiple search conditions, join them with an operator. For details about the operators that can be used in condition expressions, see *2.2.4 Operators*.
  You can omit the `[FIND_DATA]` tag if you do not use a condition expression.
- *assignment-statement*
  Codes an assignment statement into which property information from found classes is substituted. To acquire a display name, add the at mark (`@`) to the end of *class*`.`*property*.
- *sort-key*
  To sort the results, specify a sort key, in the format *class*`.`*property*. The `[ORDER_ASC]` tag sorts results in ascending order, and the `[ORDER_DESC]` tag sorts results in descending order. If the tag is omitted, the results are sorted in the order of properties specified by the `[GET_VALUE]` tag.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end.
`NODATA`	No data satisfies the search conditions.

Status	Description
ERROR	—
MULTI	—

Legend:

—: Not applicable

Example

The following example acquires the group to which user ID `user1` belongs:

```
[ASSOC_FIND]
  MemberLink
[FIND_DATA]
  UserInfo.UserID = 'user1'
[CLASS1]
  UserInfo
[CLASS2]
  GroupInfo
[GET_VALUE]
  FULLPATH = GroupInfo.FullPathName

[SET_VALUE]
  STATUS = $GETSTATUS()
[IF]
  STATUS = NORMAL
[THEN]
  [SET_VALUE]
    MSG = 'FullPathName :' +FULLPATH
    $ECHO(MSG)
[ELSE]
  [SET_VALUE]
    MSG = 'ASSOC_FIND (' +STATUS+ ')'
    $ECHO(MSG)
[IF_END]
```

[BEGIN] (process block)

`[BEGIN]` specifies a process block.

Syntax

```
[BEGIN]
   processing
[END]
```

Values

- *processing*
  Defines the processing performed by the process block.

Example

The following example shows a process block:

```
[BEGIN]
  [SET_VALUE]
    $ECHO('Hello world')
[END]
```

[CLASS_FIND] (find object class)

`[CLASS_FIND]` searches an object class.

If the class being searched matches the specified conditions, its property values are substituted into the declared variables. You can specify the `[CLASS_FIND]` tag the number of times required to find the base object class.

Syntax

```
[CLASS_FIND]
  name-of-object-class-to-search
   ([FIND_DATA])
    condition-expression
  [GET_VALUE]
    assignment-statement
   ([ORDER_ASC] or [ORDER_DESC])
    sort-key
```

Values

- *name-of-object-class-to-search*
  Codes the name of the object class to be searched.
- *condition-expression*
  Codes the condition expression. To specify multiple search conditions, join them with an operator. For details about the operators that can be used in condition expressions, see *2.2.4 Operators*.
  You can omit the `[FIND_DATA]` tag if you do not use a condition expression.
- *assignment-statement*
  Codes an assignment statement. To acquire a display name, add the at mark (`@`) to the end of *class*`.`*property*.
- *sort-key*
  To sort the results, specify a sort key, in the format *class*`.`*property*. The `[ORDER_ASC]` tag sorts results in ascending order, and the `[ORDER_DESC]` tag sorts results in descending order. If the tag is omitted, the results are sorted in the order of properties specified by the `[GET_VALUE]` tag.

Status

The following table lists and describes the possible statuses:

Status	Description
NORMAL	Normal end.
NODATA	No data satisfies the search conditions.
ERROR	—
MULTI	—

Legend:
—: Not applicable

Example

The following example outputs the asset number of asset ID `10000`:

```
[CLASS_FIND]
  AssetInfo
[FIND_DATA]
  (AssetInfo.AssetID  = '10000')
[GET_VALUE]
  ASSETNO     = AssetInfo.AssetNo
  ASSETSTATUS = AssetInfo.AssetStatus@

[SET_VALUE]
  STATUS = $GETSTATUS()
[IF]
  STATUS = NORMAL
  [THEN]
    [SET_VALUE]
      MSG = 'ASSETNO = '+ASSETNO+'('+ASSETSTATUS+')'
      $ECHO(MSG)
  [ELSE]
    [SET_VALUE]
      MSG = 'CLASS_FIND ('+STATUS+')'
      $ECHO(MSG)
[IF_END]
```

Execution result:

```
ASSETNO = R11111(active)
```

[CSV_FILE_LOOP] (get CSV file data)

`[CSV_FILE_LOOP]` specifies the beginning and end of processing to acquire data from a CSV file.

Syntax

```
[CSV_FILE_LOOP]
  CSV-file-name
  [CSV_COLUMN_NAME]
    column-name=column-number
  [BEGIN]
    processing-on-acquired-data
  [END]
[CSV_FILE_LOOP_END]
```

Values

- *CSV-file-name*

  Specifies the name of the CSV file from which data is to be acquired in the form of a variable. Specify using a relative path name referenced to the base path specified with the `-bp` option of the `jamscript` command. If the `-bp` option was omitted, *Asset-Console-installation-folder*`\scriptwork` is assumed to be the reference folder.

- *column-name=column-number*

  Maps the name for referencing the data to the column number in the CSV file. Specify a column number assuming `1` as the first column.

- *processing-on-acquired-data*

  Specifies the processing to perform on the data acquired from the CSV file.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	—
`NODATA`	End of data.
`ERROR`	There is a row that contains more than 32 kilobytes of data in a single column.
`MULTI`	—

Legend:
—: Not applicable

Remarks

Processing stops if the specified CSV file does not exist.

Whether or not importing of data from the CSV file is terminated is determined according to the status referenced by the `[CSV_FILE_LOOP_END]` loop end tag. If the status is `NORMAL`, processing continues. If the status is other than `NORMAL`, processing terminates. This last fact is useful when an error that occurs during processing interrupts processing at the point the error occurred.

When processing continues until no data remains, before determining whether or not to end processing according to the `[CSV_FILE_LOOP_END]` tag, we recommend that you use the `$SETSTATUS` embedded function to explicitly specify `NORMAL` as the status.

The status of the access definition file is updated on an ongoing basis by execution of tags and embedded functions. If processing that updates the status is specified more than once in a `[CSV_FILE_LOOP]` tag, you must determine whether or not to terminate processing based on the status that was last specified.

Example

The following example acquires data from rows in the CSV file `input.csv`, and outputs the acquired data by row number:

```
[SET_VALUE]
  FILENAME = 'input.csv'
  CNT = 1
[CSV_FILE_LOOP]
  FILENAME
  [CSV_COLUMN_NAME]
    COLUMN1 = 1
    COLUMN2 = 2
    COLUMN3 = 3
    [SET_VALUE]
      MSG = ' LINE('+CNT+')  ['+COLUMN1+']['+COLUMN2+']['+COLUMN3+']'
      $ECHO(MSG)
      CNT = $ADD(CNT,1)

  [SET_VALUE]
    $SETSTATUS('NORMAL')
[CSV_FILE_LOOP_END]
```

Execution result:

```
LINE(1) [0000000001][R11111][active]
LINE(2) [0000000002][R22222][restore]
LINE(3) [0000000003][R33333][scrap]
```

[DELETE] (delete object class)

`[DELETE]` deletes an object class. If multiple searches for information are being performed in the applicable classes, this tag deletes all classes.

Syntax

```
[DELETE]
  object-class-name
  [DATA]
    assignment-statement-for-property
```

Values

- *object-class-name*
  Codes the name of the object class to be deleted.
- *assignment-statement-for-property*
  Codes the assignment statement for the key property.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end.
`NODATA`	—
`ERROR`	No applicable data exists in the object class.
`MULTI`	—

Legend:
  —: Not applicable

Example

The following example deletes asset information from asset ID `10000`:

```
[DELETE]
  AssetInfo
[DATA]
  AssetInfo.AssetID = '10000'

[SET_VALUE]
  STATUS = $GETSTATUS()
[IF]
  STATUS != NORMAL
[THEN]
  [SET_VALUE]
    MSG = 'DELETE (' +STATUS+ ')'
    $ECHO(MSG)
[IF_END]
```

[DELETE_ASSOC] (delete association class)

`[DELETE_ASSOC]` deletes the association between two object classes.

Syntax

```
[DELETE_ASSOC]
  association-class-name
  [CLASS1]
    object-class-name
    [DATA]
      assignment-statement-for-property
  [CLASS2]
    another-object-class-name
    [DATA]
      assignment-statement-for-property
```

Values

- *association-class-name*
  Codes the name of the association class to be deleted.
- *object-class-name*
  Codes the name of an object class that is linked by the association class to be deleted.
- *assignment-statement-for-property*
  Codes all the assignment statement for the properties that are linked as keys by the association.
- *another-object-class-name*
  Codes the name of another object class that is linked by the association class to be deleted.

Example

The following example deletes the association between user ID `user1` and group ID `11000000`:

```
[DELETE_ASSOC]
  MemberLink
[CLASS1]
  UserInfo
[DATA]
  UserInfo.UserID = 'user1'
[CLASS2]
  GroupInfo
[DATA]
  GroupInfo.GroupID = '11000000'

[SET_VALUE]
  STATUS = $GETSTATUS()
[IF]
  STATUS != NORMAL
[THEN]
  [SET_VALUE]
    MSG = 'DELETE_ASSOC (' +STATUS+ ')'
    $ECHO(MSG)
[IF_END]
```

[DO] (loop block)

`[DO]` repeats processing that is not dependent on specific information. To terminate the loop, specify the `$BREAK` embedded function.

Syntax

```
[DO]
  data-processing
  $BREAK()
[DO_END]
```

Values

- *data-processing*
  Codes the processing to be performed on various data. Processing is performed based on this code.

Example

The following example shows a loop block. This example ends the loop block once `COUNT` has been output ten times within the loop block:

```
[SET_VALUE]
  CNT = 1
[DO]
  [IF]
    CNT > 10
    [THEN]
      [SET_VALUE]
        $BREAK()
  [IF_END]

  [SET_VALUE]
    MSG = ' COUNT = ' + CNT
    $ECHO(MSG)
    CNT = $ADD(CNT,1)
[DO_END]
```

Execution result:

```
COUNT = 1
COUNT = 2
COUNT = 3
COUNT = 4
COUNT = 5
COUNT = 6
COUNT = 7
COUNT = 8
COUNT = 9
COUNT = 10
```

[EVALUATE] (re-evaluate)

`[EVALUATE]` defines a process block. This process block analyzes the syntax of code during execution. By enclosing a variable in percent signs (`%`), you can resolve the value held by the variable during execution. This tag is used when you are dynamically generating search conditions.

Syntax

```
[EVALUATE]
  process-block
[EVALUATE_END]
```

Values

- *process-block*

  Codes the process block by which data is to be processed. The syntax of the tags in the process block must be complete. You can also specify a syntax that includes tags within variables enclosed by percent signs (`%`).

Examples

Example 1

This example determines the search conditions during execution by changing the search condition section to variables. This example indicates an asset number `R11111` and an asset ID with an asset status of `002`:

```
[SET_VALUE]
  STATEMENT = '(AssetInfo.AssetNo = ''R11111'') and' + CRLF
  STATEMENT = STATEMENT + '(AssetInfo.AssetStatus = ''002'')'

[EVALUATE]
  [CLASS_FIND]
    AssetInfo
  [FIND_DATA]
    %STATEMENT%
  [GET_VALUE]
    ASSETID = AssetInfo.AssetID

  [SET_VALUE]
    STATUS = $GETSTATUS()
[EVALUATE_END]
  [IF]
    (STATUS = NORMAL)
    [THEN]
      [SET_VALUE]
        MSG = ' ASSETID = ' + ASSETID
        $ECHO(MSG)
    [ELSE]
      [SET_VALUE]
        MSG = 'CLASS_FIND (' + STATUS + ')'
        $ECHO(MSG)
  [IF_END]
```

Execution result:

```
ASSETID = 10000
```

Example 2

This example stores the processing used to acquire a user name from a user ID into variables and searches the results:

```
[SET_VALUE]
  STATEMENT =             '[CLASS_FIND]' + CRLF
  STATEMENT = STATEMENT + 'UserEntry'    + CRLF
  STATEMENT = STATEMENT + '[FIND_DATA]'  + CRLF
  STATEMENT = STATEMENT + 'UserID = '    + USERID + CRLF
  STATEMENT = STATEMENT + '[GET_VALUE]'  + CRLF
  STATEMENT = STATEMENT + 'USERNAME = UserEntry.UserName' + CRLF
[EVALUATE]
  %STATEMENT%
[EVALUATE_END]
```

[IF] (conditionally execute a group of statements)

`[IF]` conditionally executes a group of statements based on specified conditions.

If the first condition is true, the processing beginning with `[THEN]` is executed. If the first condition is false, the `[ELSEIF]` tag can be used to conditionally execute another group of statements. If neither of the conditions is true, processing beginning with `[ELSE]` is executed; if `[ELSE]` is not defined, no processing is executed.

Although you can specify any number of `[ELSEIF]` tags, it is better to use the `[SWITCH]` tag to specify constants when you need to compare a condition value to a large number of constants, because you do not need to nest the condition branches so deeply, thus producing code that is easier to read. For a coding example of the `[SWITCH]` tag, see *[SWITCH] (conditionally execute a group of statements)*.

Syntax

```
[IF]
  condition-1
  [THEN]
    processing-if-condition-1-is-true
([ELSEIF])
    condition-2
  ([THEN])
    processing-if-condition-1-is-false-and-condition-2-is-true
   ([ELSE])
    processing-if-all-conditions-are-false
[IF_END]
```

Values

- *condition-1*, *condition-2*
  Specifies the conditions used to determine whether to branch.
- *processing-if-condition-1-is-true*
  Specifies the processing to be executed when *condition-1* is true.
- *processing-if-condition-1-is-false-and-condition-2-is-true*
  Specifies the processing to be executed when *condition-1* is false and *condition-2* is true. This coding is optional. If you omit this coding, do not specify a condition for it.
- *processing-if-all-conditions-are-false*
  Specifies the processing to be executed when all the conditions are false. This specification is optional.

Example

Example 1

The following example searches for the asset information for asset number `10000`. If data is found (`[THEN]`), that data is deleted; if no data is found (`[ELSE]`), the termination status is displayed.

```
[CLASS_FIND]
  AssetInfo
[FIND_DATA]
  (AssetInfo.AssetNo  = '10000')
[GET_VALUE]
  AssetID = AssetInfo.AssetID
```

```
[SET_VALUE]
  STATUS = $GETSTATUS()
[IF]
  STATUS = NORMAL
  [THEN]
    [DELETE]
      AssetInfo
    [DATA]
      AssetInfo.AssetID = AssetID
    [SET_VALUE]
      MSG = 'CLASS_FIND (' + STATUS + ')'
      $ECHO(MSG)

  [ELSE]
    [SET_VALUE]
      MSG = 'CLASS_FIND (' + STATUS + ')'
      $ECHO(MSG)
[IF_END]
```

Example 2

The example below performs the following processing:

- If the data is updated normally (`[THEN]`), the termination status is displayed.
- If the data has already been updated by another control process (`[ELSEIF]`, `[THEN]`), a message is output.
- If no data exists (`[ELSE]`), data is added.

```
[UPDATE]
  AssetInfo
[DATA]
  AssetInfo.AssetID = '10000'
  AssetInfo.AssetNo = '10000'
  AssetInfo.AssetKind = '001'
  AssetInfo.AssetBranchNo = 0
  AssetInfo.UpdateTime = _UpdateTime
[SET_VALUE]
  STATUS = $GETSTATUS()
[IF]
  STATUS = NORMAL
    [THEN]
      [SET_VALUE]
        MSG = 'UPDATE (' +STATUS+ ')'
        $ECHO(MSG)
    [ELSEIF]
      STATUS = MULTI
        [THEN]
          [SET_VALUE]
            MSG = 'Asset number [10000] is updated already.'
            $ECHO(MSG)
    [ELSE]
      [APPEND]
        AssetInfo
      [DATA]
        AssetInfo.AssetID = '10000'
        AssetInfo.AssetNo = '10000'
        AssetInfo.AssetKind = '001'
        AssetInfo.AssetBranchNo = 0
      [SET_VALUE]
        MSG = 'UPDATE (' +STATUS+ ')'
        $ECHO(MSG)
[IF_END]
```

[JOIN_FIND] (find joined class)

`[JOIN_FIND]` joins multiple object classes and searches them.

Syntax

- Joining through use of an association class

```
[JOIN_FIND]
  [JOIN]
  joinassoc;association-class-name;
  joinfrom;object-class-name,another-object-class-name;
  jointype;(OUTER|INNER)
  ([FIND_DATA])
  condition-expression
[GET_VALUE]
  assignment-statement
([ORDER_ASC] or [ORDER_DESC])
    sort-key
([DUPLICATE]#)
```

- Joining by keying on a property

```
[JOIN_FIND]
  [JOIN]
  joinfrom;object-class-name,another-object-class-name;
  jointype;(OUTER|INNER);
  joinkey;join-key-class-property-names
  ([FIND_DATA])
  condition-expression
[GET_VALUE]
  assignment-statement
 ([ORDER_ASC] or [ORDER_DESC])
    sort-key
 ([DUPLICATE]#)
```

#

Specify the `[DUPLICATE]` tag to suppress output of the second and subsequent result when the values of all properties of the search results are identical.

Values

- Join information

  Codes the classes to be joined, by using either an association class or key properties.

- *condition-expression*

  Codes the condition expression. To specify multiple search conditions, join them with an operator. For details about the operators that can be used in condition expressions, see *2.2.4 Operators*.

  You can omit the `[FIND_DATA]` tag if you do not use a condition expression.

- *assignment-statement*

  Codes an assignment statement into which property information from found classes is substituted. When acquiring a property defined in a code table, you can acquire the code table value or the display name. To acquire a display name, add the at mark (`@`) to the end of *class*`.`*property*.

- *sort-key*

To sort the results, specify a sort key, in the format *class*.*property*. The `[ORDER_ASC]` tag sorts results in ascending order, and the `[ORDER_DESC]` tag sorts results in descending order. If the tag is omitted, the results are sorted in the order of properties specified by the `[GET_VALUE]` tag.

Status

The following table lists and describes the possible statuses:

Status	Description
NORMAL	Normal end.
NODATA	No data satisfies the search conditions.
ERROR	—
MULTI	—

Legend:

—: Not applicable

Example

The following examples search multiple classes.

Example 1

The following example joins asset information, installed software information, and a list of installed software, and outputs a list of installed software information for asset number `1000`:

```
[ASSET_ITEM_LOOP]
  [JOIN_FIND]
  [JOIN]
    joinassoc;InstalledLink;
    joinfrom;AssetInfo,InstalledInfo;
    jointype;INNER;
  [JOIN]
    joinassoc;InstalledListLink;
    joinfrom;InstalledInfo,InstalledList;
    jointype;INNER;
  [FIND_DATA]
    (AssetInfo.AssetNo = '1000')
  [GET_VALUE]
    INSTALLNAME = InstalledList.InstalledName
    VERSION     = InstalledList.InstalledVersion
    PERMIT      = InstalledList.InstalledPermit@

  [SET_VALUE]
    MSG = INSTALLNAME+'['+VERSION+'] ('+PERMIT+')'
    $ECHO(MSG)

  [SET_VALUE]
    $SETSTATUS('NORMAL')
[ASSET_ITEM_LOOP_END]
```

Execution result:

```
SoftwareA[0100] (authorized)
SoftwareA[0101] (authorized)
SoftwareB[0000] (authorized)
SoftwareC[0000] (unauthorized)
```

Example 2

The following example joins asset information and group information, then outputs the group name of asset ID `10000`. This example specifies an outer join (`[OUTER]`) so, if no group ID has been specified, `" "` is output.

```
[JOIN_FIND]
[JOIN]
  joinfrom;AssetInfo,GroupInfo;
  jointype;OUTER;
  joinkey;AssetInfo.GroupID,GroupInfo.GroupID;
[FIND_DATA]
  (AssetInfo.AssetID = '10000')
[GET_VALUE]
  GROUP_NAME = GroupInfo.FullPathName

[SET_VALUE]
  STATUS = $GETSTATUS()
[IF]
  STATUS = NORMAL
  [THEN]
    [SET_VALUE]
      MSG = 'GROUP = ' + GROUP_NAME
      $ECHO(MSG)
  [ELSE]
    [SET_VALUE]
      MSG = 'JOIN (' + STATUS + ')'
      $ECHO(MSG)
[IF_END]
```

Execution result:

```
GROUP = Head Office/Sales Dept./Section 1
```

[SET_VALUE] (substitute value)

`[SET_VALUE]` enumerates assignment statements for variables. You can then use an embedded function to call the variables.

Syntax

```
[SET_VALUE]
  assignment-statement-for-variable-1
  assignment-statement-for-variable-2 . . .
```

Values

- *assignment-statement-for-variable*
  Codes an assignment statement for a variable that has been declared using the `[VAR]` tag.

Example

The following example assigns `Hello world` to the variable `MSG`, and outputs the result:

```
[SET_VALUE]
  MSG = 'Hello world'
  $ECHO(MSG)
```

[SUB] (subroutine)

`[SUB]` defines processing that is reused as a single process block. Subroutines are not executed, even if they go through the processing path; subroutines are executed using the `$GOSUB` embedded function. This means that the subroutine must be defined before the `$GOSUB` embedded function is specified.

Syntax

```
[SUB]
  subroutine-name
  . . .
[SUB_END]
[BEGIN]
  . . .
  [SET_VALUE]
    $GOSUB(subroutine-name)
[END]
```

Values

- *subroutine-name*

  Specifies the subroutine name. For the subroutine name, you can use alphanumeric characters and the underscore (_). However, a numeric character cannot be used as the first character of the subroutine name. Note also that case is considered significant in subroutine names.

Example

The following example defines a subroutine that outputs data to a Command Prompt window.

In this example, by specifying `-s MSG=1` as an option in the `jamscript` command, the text set for `ECHOMSG` is output to a Command Prompt window:

```
[SUB]
  ECHO
  [IF]
    MSG = '1'
  [THEN]
    [SET_VALUE]
      $ECHO(ECHOMSG)
  [IF_END]
[SUB_END]

[BEGIN]
  [SET_VALUE]
    MSG = $GETSESSION('MSG')

    ECHOMSG = 'Hello world'
    $GOSUB(ECHO)
[END]
```

Executed command:

```
jamscript -f Example.txt -s MSG=1
```

Execution result:

```
Hello world
```

[SWITCH] (conditionally execute a group of statements)

`[SWITCH]` executes processing when a condition value matches a specified constant.

The `[SWITCH]` tag is more convenient to use than the `[IF]` tag when you wish to compare a condition to a large number of constants, because the statements are nested less deeply. When the condition value matches a constant, the processing following the associated `[CASE]` tag is executed. When the condition value does not match any of the specified constants, the processing following `[DEFAULT]` is executed; if `[DEFAULT]` is not defined, no processing is executed.

If the same constant is specified under more than one `[CASE]` tag, processing is executed beginning with the `[CASE]` tag in which the constant was first specified (beginning from the left if all of the relevant coding is specified on one line), and continues until all processing specified for the condition value matching those constants has executed.

Syntax

```
[SWITCH]
  condition-value
  [CASE]
    constant
      processing-if-condition-value-matches-the-constant
([CASE])
    constant[,constant[,constant...]]
      processing-if-condition-value-matches-any-of-the-constants
([DEFAULT])
    processing-if-no-constant-is-matched
[SWITCH_END]
```

Values

- *condition-value*
  Specifies a condition value.
- *constant*
  Specifies a constant; when this constant matches the condition value, the associated processing is executed. Multiple constants can be specified by separating them with commas.
- *processing-if-condition-value-matches-the-constant*
  Specifies the processing to be executed when the condition value matches the associated constant. If you want to execute only the processing specified for the first constant that the condition value matches, specify `$BREAK()` at the end of the `[CASE]` tag block.
- *processing-if-condition-value-matches-any-of-the-constants*
  Specifies the processing to be executed when the condition value matches any one of the constants. If you want to execute only the processing specified for the first constant that the condition value matches, specify `$BREAK()` at the end of the `[CASE]` tag block. This specification is optional.
- *processing-if-no-constant-is-matched*
  Specifies the processing to be executed when the condition value does not match any of the constants. You can specify the `[DEFAULT]` tag before the `[CASE]` tags if you wish. This specification is optional.

Example

The example below executes the following processing:

- If the data is updated normally (`STATUS = 'NORMAL'`), the termination status is displayed.
- If the data has already been updated by another control process (`STATUS = 'MULTI'`), a message is output.
- If no data exists (`[DEFAULT]`), data is added.

```
[UPDATE]
  AssetInfo
[DATA]
  AssetInfo.AssetID = '10000'
  AssetInfo.AssetNo = '10000'
  AssetInfo.AssetKind = '001'
  AssetInfo.AssetBranchNo = '0'
  AssetInfo.UpdateTime = _UpdateTime
[SET_VALUE]
  STATUS = $GETSTATUS()
[SWITCH]
  STATUS
  [CASE]
    'NORMAL'
    [SET_VALUE]
      MSG = 'UPDATE (' +STATUS+ ')'
      $ECHO(MSG)
      $BREAK()
  [CASE]
    'MULTI'
    [SET_VALUE]
      MSG = 'Asset number [10000] is updated already.'
      $ECHO(MSG)
      $BREAK()
  [DEFAULT]
    [APPEND]
      AssetInfo
    [DATA]
      AssetInfo.AssetID = '10000'
      AssetInfo.AssetNo = '10000'
      AssetInfo.AssetKind = '001'
      AssetInfo.AssetBranchNo = '0'
    [SET_VALUE]
      MSG = 'UPDATE (' +STATUS+ ')'
      $ECHO(MSG)
      $BREAK()
[SWITCH_END]
```

[TRANSACTION] (define transaction)

`[TRANSACTION]` specifies the range to be handled as a transaction when an access definition file is being used to customize an operation window.

If the processing status is `NORMAL` when the processing defined in the transaction ends, the transaction is committed; if the processing status is not `NORMAL`, the transaction is rolled back.

Syntax

```
[TRANSACTION]
  . . .
  [BEGIN]
    . . .
  [END]
[TRANSACTION_END]
```

Values

Codes the transaction processing.

Remarks

Whether or not transaction processing is complete is determined according to the status referenced by the transaction processing end tag `[TRANSACTION_END]` or by the embedded function `$BREAK`, which breaks out of the transaction processing. If the status is `NORMAL`, the transaction is committed; if the status is not `NORMAL`, it is not committed (it is rolled back).

The status of the access definition file is updated on an ongoing basis by execution of tags and embedded functions. If processing that updates the status is specified more than once in a transaction, whether to commit or roll back is based on the status that was last specified.

This means that if a transaction is committed immediately before a `[TRANSACTION_END]` tag or a `$BREAK` embedded function, we recommend that you use the `$SETSTATUS` embedded function to explicitly specify `NORMAL` as the status.

Note that transaction processing cannot be nested.

Example

The following shows an example of transaction processing:

```
[TRANSACTION]
  [APPEND]
    AssetInfo
  [DATA]
    AssetInfo.AssetID = '10000'
    AssetInfo.AssetNo = '10000'
    AssetInfo.AssetWorkKind = '001'
    AssetInfo.AssetKind = '001'
    AssetInfo.AssetBranchNo = '0'
  [SET_VALUE]
    STATUS = $GETSTATUS()
  [IF]
    STATUS = NORMAL
  [THEN]
    ##Process 1
    [APPEND]
```

```
      InstalledInfo
    [DATA]
      InstalledInfo.InstalledID = '10000'
      InstalledInfo.AssetID = '10000'
    [SET_VALUE]
      STATUS = $GETSTATUS()
    [IF]
      STATUS = NORMAL
    [THEN]
      ##Process 2
      [SET_VALUE]
        MSG = 'TRANSACTION : COMMIT'
        $ECHO(MSG)
    [ELSE]
      ##Process 3
      [SET_VALUE]
        MSG = 'TRANSACTION : ROLLBACK'
        $ECHO(MSG)
    [IF_END]
  [ELSE]
    ##Process 4
    [SET_VALUE]
      MSG = 'TRANSACTION : ROLLBACK'
      $ECHO(MSG)
  [IF_END]
[TRANSACTION_END]
```

In this processing, the following exception processing is performed.

- If addition of asset ID `10000` fails:

  Control is transferred to the processing indicated by comment `##Process 4`. Because the termination status of `[APPEND]` is not `NORMAL`, the transaction processing between `[TRANSACTION]` and `[TRANSACTION_END]` is rolled back and processing terminates. This means that all requests to the database are invalidated.

- If addition of asset ID `10000` is successful:

  Control is transferred to the processing indicated by comment `##Process 1`. Asset ID `10000` and installed software ID `10000` are registered in the installed software information.

- If addition of asset ID `10000` succeeds, and addition of installed software ID `10000` fails:

  Control is transferred to the processing indicated by comment `##Process 3`.

  Because the termination status of `[APPEND]` is not `NORMAL`, the transaction processing between `[TRANSACTION]` and `[TRANSACTION_END]` is rolled back and processing terminates. This means that all requests to the database are invalidated.

- If addition of asset ID `10000` and of installed software ID `10000` is successful:

  Control is transferred to the processing indicated by comment `##Process 2`. Because the termination status of `[APPEND]` is `NORMAL`, all requests to the database are validated.

[UPDATE] (update object class)

`[UPDATE]` finds and updates an object class. If multiple object classes are found, only the object class found first is updated. If the specified object class is not found, no processing is performed.

Syntax

```
[UPDATE]
  object-class-name
  [DATA]
    assignment-statement-for-property
```

Values

- *object-class-name*
  Codes the name of the object class to be updated.
- *assignment-statement-for-property*
  Specifies all key properties to be updated as assignment statements.

Status

The following table lists and describes the possible statuses:

Status	Description
NORMAL	Normal end.
NODATA	—
ERROR	The data of the object class to be updated does not exist.
MULTI	Updating has already been performed by another control.

Legend:
—: Not applicable

Example

The following example updates the asset status of asset ID `10000` to `301`:

```
[UPDATE]
  AssetInfo
[DATA]
  AssetInfo.AssetID     = '10000'
  AssetInfo.AssetStatus = '301'

[SET_VALUE]
  STATUS = $GETSTATUS()
[IF]
  STATUS != NORMAL
  [THEN]
    [SET_VALUE]
      MSG = 'UPDATE (' + STATUS + ')'
      $ECHO(MSG)
[IF_END]
```

[VAR] (declare variable)

`[VAR]` declares a variable.

Syntax

```
[VAR]
  variable-name
```

Values

- *variable-name*
  Codes the variable name being declared.

Example

The following example declares the `STATUS` and `MSG` variables:

```
[VAR]
  STATUS
  MSG
```

5 Embedded Functions Used in Access Definition Files

This chapter explains the embedded functions that can be used in access definition file tags.

List of embedded functions

This section explains the embedded functions that can be defined in assignment statements.

The following table lists, in order of purpose, the embedded functions that can be used in access definition files.

Table 5–1: List of embedded functions

Embedded function		Description
Purpose	Name of embedded function	
Array operations	`$GETARRAY`	Reads data from an array.
	`$CLEARARRAY`	Initializes an array.
	`$SETARRAY`	Sets data into an array.
	`$SETARRAYBYKEY`	Sets keyed data into an array.
	`$GETARRAYBYKEY`	Reads keyed data from an array.
	`$GETKEYFROMARRAY`	Reads the corresponding key from an array.
	`$UPDARRAY`	Updates array data.
	`$UPDARRAYBYKEY`	Updates keyed array data.
Block processing operations	`$BREAK`	Interrupts processing in a process block.
	`$GETSTATUS`	Acquires the status of a process block.
	`$SETSTATUS`	Sets the status of a process block.
Session information operations	`$GETSESSION`	Reads session information.
	`$GETTEMPNAME`	Specifies the name of the work file for downloading that is created for a session.
	`$SETSESSION`	Sets session information.
File operations	`$FILEARRAY`	Outputs information stored in an array to a CSV file.
	`$FILECLOSE`	Ends editing of a file.
	`$FILECOPY`	Copies a file (always overwrites any existing file).
	`$FILEDEL`	Deletes a file.
	`$FILEOPEN`	Begins editing of a file.
	`$FILEPUT`	Outputs data to a file.
	`$FILEPUTLN`	When data is being output to a file, adds a CRLF at the end.
	`$FINDFILE`	Finds files.
Directory information operations	`$LDAPACS`	Performs confirmation of connections to directory service, searching, entry acquisition, attribute acquisition, and other directory service operations.
	`$GETROLE`	Stores user roles in an array.
Subroutine	`$GOSUB`	Executes a subroutine.
Arithmetic operations	`$ADD`	Acquires the result of an add operation.
	`$CALCDATE`	Calculates the date and time.

Embedded function		Description
Purpose	Name of embedded function	
Arithmetic operations	`$DIV`	Acquires the result of a division operation.
	`$MOD`	Acquires the remainder of a division operation.
	`$MUL`	Acquires the result of a multiplication operation.
	`$SUB`	Acquires the result of a subtraction operation.
Information extraction	`$SUBSTR`	Extracts a substring.
	`$TOKEN`	Extracts a token.
Information acquisition	`$DATACOUNT`	Acquires the result lines of the previous search.
	`$DATETIME`	Acquires the current date and time.
Numbering	`$NUMBER`	Acquires sequential numbers from a database.
Acquisition of server environmental settings	`$ENVIRONMENT`	Acquires the environmental setting information of a server.
	`$GETREGVALUE`	Acquires registry information for the server environment.
Process end	`$EXIT`	Ends processing of an access definition file.
	`$SETOPTION`	Sets end options to handle errors.
Message output	`$ECHO`	Outputs a message to standard output.
	`$FORMATMSG`	Formats a message text.
	`$LOGMSG`	Outputs a message to a log file.
Conversion	`$LOWER`	Converts an alphabetic character string to lowercase.
	`$UPPER`	Converts an alphabetic character string to uppercase.
Windows initialization file operations	`$GETPROFILEDATA`	Reads the keys and values of a Windows initialization file.
`NULL` value evaluation	`$ISNULL`	Determines whether or not a value is `NULL`.
Character string operations	`$LENGTH`	Acquires the length of a character string.
	`$MATCH`	Evaluates a character string.
	`$STRCMP`	Compares character strings.
DLL operations#	`$DLLLOAD`	Loads a DLL.
	`$DLLEXEC2`	Executes a user function installed on a user-provided DLL.
	`$DLLFREE`	Frees a DLL.
	`$DLLMSG`	Acquires DLL messages.

#

In embedded functions that perform DLL operations, always use DLLs that have been created as 32-bit applications. Even if the OS you are using is Server 2008 R2, you must not use a DLL created as a 64-bit application.

Detailed explanations of embedded functions

This section provides detailed explanations of the embedded functions, generally in the following format. The embedded function explanations appear in alphabetical order by embedded function name.

Embedded function

This subsection provides the name and a short description of the embedded function.

Syntax

This subsection provides the embedded function's syntax.

Values

This subsection explains the values that can be specified in the embedded function.

DLL interface to be used

This subsection explains the interface with the DLL required to specify an embedded function that uses the DLL.

Status

This subsection explains the status of the processing coded by the embedded function and explains what each status means.

Example

This subsection provides a coding example using the embedded function.

$ADD (Addition)

`$ADD` performs addition, treating character strings as numeric values, and returns the arithmetic result.

Syntax

return-value=`$ADD(`*character-string*`,`*numeric-character*`)`

Values

- *return-value*

  Specifies the name of the variable into which the arithmetic result is set. Valid results range from 0.0001 to 999,999,999,999,999 (15 digits).

- *character-string*

  Specifies a numeric value to be added, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`). Specified values can range from 0.0001 to 999,999,999,999,999 (15 digits).

- *numeric-character*

  Specifies a numeric value to be added, either as a constant or a variable. A numeric value specified as a constant that includes a decimal point must be enclosed in single quotation marks (`''`). Specified values can range from 0.0001 to 999,999,999,999,999 (15 digits).

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	Indicates one of the following: • An invalid value was specified in a character string or numeric value. • The arithmetic result is a value outside the range of representable values.
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Remarks

If a value that cannot be specified in *character-string* or *numeric-character* is encountered, or if the arithmetic result is a value outside the representable range, `0` is returned to *return-value*.

Example

The following example calculates 10 + 5 and outputs the result:

```
[SET_VALUE]
  VAL1 = 10
  VAL2 = $ADD(VAL1, 5)

  MSG = 'ADD: ' +VAL1+ '+ 5 =' +VAL2
  $ECHO(MSG)
```

Execution result:

```
ADD: 10 + 5 = 15
```

$BREAK (interrupt process block)

`$BREAK` interrupts a process block or exits it during processing.

While a class is being created, exception processing uses evaluation statements to evaluate the statuses of various requests, and then uses the `$BREAK` embedded function to interrupt the process and to perform post-processing when an error occurs.

Syntax

```
$BREAK()
```

Values

There are no values to specify.

Example

For an example, see *[TRANSACTION] (define transaction)* in *4. Tags Used in Access Definition Files*.

$CALCDATE (date calculation function)

`$CALCDATE` adds a specified number of days or hours to a date or time, then returns the arithmetic result. You can acquire relative dates and times, such as two days before or three hours after a specified date or time, without having to mentally calculate the carryover of months, days, or hours.

Syntax

return-value=`$CALCDATE(`*date/time*`,`*unit*`,`*value-to-add*`,`*output-format*`)`

Values

- *return-value*

  Specifies the name of the variable into which the acquired date or time is to be set.
- *date/time*

  Specifies a constant or a variable. A constant must be enclosed in single quotation marks (`'`).

  date/time can be specified in either of the following formats:
  - *yyyy*`/`*mm*`/`*dd* *hh*`:`*mm*`:`*ss* or *yyyy*`-`*mm*`-`*dd* *hh*`:`*mm*`:`*ss*

    These formats specify both a date and a time; however, the *hh*`:`*mm*`:`*ss* part can be omitted.
  - No specification

    The date and time the embedded function is executed is assumed.
- *unit*

  Specifies a constant or a variable. A constant must be enclosed in single quotation marks (`'`).

  The following lists the variables that can be specified:
  - *d*

    Specifies that a number of days is to be calculated.
  - *h*

    Specifies that a number of hours is to be calculated.
  - *n*

    Specifies that a number of minutes is to be calculated.
  - *s*

    Specifies that a number of seconds is to be calculated.
- *value-to-add*

  Specifies a constant or a variable. A constant must be enclosed in single quotation marks (`'`).

  An integer must be specified as the value.
- *output-format*

  Specifies the output format of the date and time, either as a constant or a variable. For details about the output format, see *$DATETIME (get date/time)*.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end

Status	Description
Script execution interrupted	Indicates one of the following: • An invalid value was specified for *value-to-add*, such as a non-numeric value. • An invalid argument was specified, or an error other than the above occurred.

Example

The following example outputs the date that is 10 days before 2005/04/11:

```
DATE = $CALCDATE('2005/04/11','d','-10','%Y/%m/%d')
$ECHO(DATE)
```

Execution result:

```
2005/04/01
```

$CLEARARRAY (initialize array)

`$CLEARARRAY` deletes all information stored in an array and reinitializes the array.

Syntax

`$CLEARARRAY(`*array-name*`)`

Values

- *array-name*
  Specifies the name of the array to be initialized.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Example

See the example for *$SETARRAY (set value to array)*.

$DATACOUNT (get result lines)

`$DATACOUNT` acquires the number of data items that were found by the most recent search that was executed.

When a search is defined within the `[ASSET_ITEM_LOOP]` and `[ASSET_ITEM_LOOP_END]` tags, the status following the `[ASSET_ITEM_LOOP_END]` tag is always `NODATA`. To determine whether or not applicable data exists, you must use the `$DATACOUNT` embedded function to acquire and evaluate the number of data items resulting from the search.

Syntax

return-value=`$DATACOUNT()`

Values

- *return-value*
  Specifies the name of the variable into which the search results lines are set.

Status

The following table lists and describes the possible statuses:

Status	Description
NORMAL	Normal end
NODATA	—
ERROR	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Example

The following example outputs the number of asset information data items (100) whose status is active (`002`):

```
[CLASS_FIND]
  AssetInfo
[FIND_DATA]
  (AssetInfo.AssetStatus = '002')AND
  (AssetInfo.AssetKind = '001')
[GET_VALUE]
  WORK = AssetInfo.AssetNo

[SET_VALUE]
  STATUS = $GETSTATUS()
  TOTAL = $DATACOUNT()
[IF]
  STATUS = NORMAL
[THEN]
  [SET_VALUE]
    MSG = 'DataCount :' +TOTAL
    $ECHO(MSG)
[ELSE]
```

```
  [SET_VALUE]
    MSG = 'CLASS_FIND (' +STATUS+ ')'
    $ECHO(MSG)
[IF_END]
```

Execution result:

```
DataCount : 100
```

$DATETIME (get date/time)

`$DATETIME` acquires the current date and time according to the specified output format.

Syntax

return-value=`$DATETIME(`*output-format*`)`

Values

- *return-value*

  Specifies the name of the variable into which the acquired date and time are set.

- *output-format*

  Specifies an output format for the date and time, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

  output-format is specified as combinations of *format specifiers* and character constants.

  Format specifiers are symbols indicating the information to be acquired. These specifiers are expressed as a percent sign (`%`) paired with a single alphabetic character. Lower and upper case alphabetic characters in format specifiers are significant, with different information acquired by each. Note that, if an alphabetic character other than one defined as a format specifier character follows the percent sign (`%`), the specified alphabetic character is acquired.

  To use a double quotation mark (`"`) as a character constant, specify `\"`.

  The following lists the format specifiers that can be specified.

  - `%a`

    Acquires the abbreviated day of the current date as follows:

    `Sun`, `Mon`, `Tue`, `Wed`, `Thu`, `Fri`, `Sat`

  - `%A`

    Acquires the conventionally written day of the current date as follows:

    `Sunday`, `Monday`, `Tuesday`, `Wednesday`,

    `Thursday`, `Friday`, `Saturday`

  - `%b`

    Acquires the abbreviated month of the current date as follows:

    `Jan`, `Feb`, `Mar`, `Apr`, `May`, `Jun`,

    `Jul`, `Aug`, `Sep`, `Oct`, `Nov`, `Dec`

  - `%B`

    Acquires the conventionally written month of the current date as follows:

    `January`, `February`, `March`, `April`, `May`, `June`,

    `July`, `August`, `September`, `October`, `November`, `December`

  - `%d`

    Acquires the date as a numeric value from `01` to `31`.

  - `%H`

    Acquires the hour on the 24-hour clock as a numeric value from `00` to `23`

  - `%I`

    Acquires the hour on the 12-hour clock as a numeric value from `01` to `12`.

  - `%j`

Acquires the number of days that have elapsed since January 1 of the current year as a numeric value from `001` to `366`.

- `%m`

  Acquires the month as a numeric value from `01` to `12`.

- `%M`

  Acquires the minute as a numeric value from `00` to `59`.

- `%p`

  Acquires `AM` (morning) or `PM` (afternoon) as a character string.

- `%S`

  Acquires the second as a numeric value from `00` to `59`.

- `%w`

  Acquires a one-digit number corresponding to the day of the week as a numeric value from `0` to `6`, beginning with Sunday as `0`, as follows:

  Sunday: `0`, Monday: `1`, Tuesday: `2`, Wednesday: `3`,

  Thursday: `4`, Friday: `5`, Saturday: `6`

- `%Y`

  Acquires the four-digit calendar year.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:

—: Not applicable

Example

The following example acquires the date and time (01:05:36 on 2015/04/01) in the format *year*/*month*/*date* *hour*:*minutes*:*seconds*:

```
[SET_VALUE]
  DATE = $DATETIME('%Y/%m/%d %H:%M:%S')
  MSG = 'DATE = ' + DATE
  $ECHO(MSG)
```

Execution result:

```
DATE = 2015/04/01 01:05:36
```

$DIV (division)

`$DIV` performs division, treating character strings as numeric values, and returns the arithmetic result.

For example, the Asset Console standard memory and disk sizes are managed in megabyte units. To register information managed in gigabyte units in the asset management database, you need to change the units to megabytes. To do so, you can use the `$DIV` embedded function to convert from gigabytes to megabytes.

Syntax

return-value`=$DIV(`*character-string*`,`*numeric-character*`)`

Values

- *return-value*

  Specifies the name of the variable into which the arithmetic result is set. Valid results range from 0.0001 to 999,999,999,999,999 (15 digits).

- *character-string*

  Specifies a dividend, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`). Specified values can range from 0.0001 to 999,999,999,999,999 (15 digits).

- *numeric-character*

  Specifies a divisor, either as a constant or a variable. A numeric value specified as a constant that includes a decimal point must be enclosed in single quotation marks (`''`). Specified values can range from 0.0001 to 999,999,999,999,999 (15 digits).

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	Indicates one of the following: • An invalid value was specified in a character string or numeric. • The arithmetic result is a value outside the range of representable values.
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Remarks

If a value that cannot be specified in *character-string* or *numeric-character* is encountered, or if the arithmetic result is a value outside the representable range, `0` is returned to *return-value*.

Example

The following example calculates 10 ÷ 5 and outputs the results:

```
[SET_VALUE]
  VAL1 = 10
  VAL2 = $DIV(VAL1, 5)

  MSG = 'DIV: ' +VAL1+ ' / 5 = ' +VAL2
  $ECHO(MSG)
```

Execution result:

```
DIV: 10 / 5 = 2
```

$DLLEXEC2 (execute DLL)

`$DLLEXEC2` executes a user-created user function. An array variable is used to transfer data between the user function executed by `DLLEXEC2` and the access definition file. To access an array variable within the user function, use the macros provided by Asset Console.

Syntax

`$DLLEXEC2(`*DLL-object*`,`*function-name*`,`*array-name-1*`(,`*array-name-2*`(,...)))`

Values

- *DLL-object*
  Specifies the variable name of the DLL object that was acquired by `$DLLLOAD`.
- *function-name*
  Specifies the name of the function to be executed, as either a constant or a variable. A constant must be enclosed in single quotation marks (`''`).
- *array-name*
  Specifies the name of the array variable containing the information to be passed to the function, or the name of the array variable for acquiring the execution result of the function.

DLL interface to be used

The following shows the format of the function that is called by `$DLLEXEC2`:

```
int function-name(int argc       //number of array variables
                              //specified in the script
            ,void**  argv     //start address of the group of array
                              // variables specified in the script
           )
```

The array variables specified in the script are stored sequentially (from left to right) beginning at array 0 of `argv`. Therefore, the array name 1 and array name 2 are processed internally as `argv[0]` and `argv[1]`, respectively, by the function.

If the function returns a negative value, the function executes the `aim_getmessage` function (provided in the same DLL by the user). If a message has been specified, the function writes that message in the Asset Console log, and then cancels the script execution.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end (when the function returns `0`)
`NODATA`	Termination with warning (when the function returns `1`)
`ERROR`	Abnormal termination (when the function returns a positive value)
Script execution interrupted	Indicates one of the following: • Abnormal termination (when the function returns a negative value)# • An invalid argument was specified, or an error other than the above occurred.

#

Indicates that the variable name of the specified DLL object is not the DLL object acquired by the `$DLLLOAD` embedded function.

Note

The user must have previously provided the DLL containing the user functions that are called by the `$DLLEXEC2` embedded function.

Example

This example loads `sample.dll` and executes the summing function `FunctionSum`. It sets arguments `10`, `20`, and `30` in the loaded `sample.dll`, executes `FunctionSum`, and then outputs the result.

```
[SET_VALUE]
  DLLOBJ = $DLLLOAD('sample.dll')
  STATUS = $GETSTATUS()
[IF]
  STATUS != NORMAL
  [THEN]
    [SET_VALUE]
      $ECHO('DLL LOAD ERROR')
      $EXIT(3)
[IF_END]

[SET_VALUE]
  $SETARRAY(INPUT, '10')
  $SETARRAY(INPUT, '20')
  $SETARRAY(INPUT, '30')
  $DLLEXEC2(DLLOBJ,'FunctionSum',INPUT,OUTPUT)
  STATUS = $GETSTATUS()
[IF]
  STATUS = NORMAL
  [THEN]
    [SET_VALUE]
      VAL = $GETARRAY(OUTPUT, 1)
      MSG = 'OUTPUT = ' + VAL
      $ECHO(MSG)
  [ELSE]
    [SET_VALUE]
      VAL = $DLLMSG(DLLOBJ)
      MSG ='DLLEXEC FunctionSum ERROR (' + VAL + ')'
      $ECHO(MSG)
[IF_END]

[SET_VALUE]
  $DLLFREE(DLLOBJ)
```

Creating a user function to be executed by $DLLEXEC2

If the user function to be executed by `$DLLEXEC2` does not need to transfer data with the script, you can create a DLL with the general procedure.

In some cases, it might be desirable to let user function receive information from the script, or to set the processing result of the user function in an array of the script. To do this, you must use a special function provided by Asset Console to access the script's array variables. Asset Console provides a function for accessing such array variables as a macro.

The macros are defined in the header file for C and C++ language `jamScriptAPI.h`. By including this header file, you can manipulate arrays with the same interface you use for the functions in the access definition file.

Note that from `$DLLEXEC2` you can execute only those functions that use array manipulation macros that are included in the DLL.

`jamScriptAPI.h` is stored in the following folder:

```
Asset-Console-installation-folder\sdk\include
```

The following table lists and describes the macros defined in `jamScriptAPI.h`:

Table 5–2: List of macros available to DLL

Macro		Function
Purpose	Macro name	
Manipulation of arrays	`$GETARRAY`	Gets value of array data
	`$CLEARARRAY`	Initializes an array
	`$SETARRAY`	Sets value to an array
	`$SETARRAYBYKEY`	Sets value to an array with a key
	`$GETARRAYBYKEY`	Gets value from an array with a key
	`$GETKEYFROMARRAY`	Reads the corresponding key from an array
	`$GETARRAYLENGTH`	Gets the number of array elements
	`$GETARRAYNAME`	Gets the name of an array
	`$UPDARRAY`	Updates array data
	`$UPDARRAYBYKEY`	Updates value of an array with a key
Acquisition of instances	`$GETINITAREA`	Gets the return value of the `aim_init` function
Acquisition of status	`$GETSTATUS`	Gets details of macro termination status

The following subsections describe each macro available during DLL creation.

■ $CLEARARRAY (initialize array)

`$CLEARARRAY` deletes all information stored in the specified array and initializes the array.

Format (equivalent function format)

```
int $CREARARRAY(void** argv,void* array);
```

Arguments

- `argv`
  Specifies the start address of the group of array variables that were specified in the script.
- `array`
  Specifies one of the elements of the group of array variables (argument of the user function).

Return values

Return value	Description
`0`	Normal end
`-1`	Error#

#

You can acquire detailed error information using *$GETSTATUS (get details of macro termination status)*.

Coding example

This example initializes array 1 specified in the script:

```
int DllFunc8(int argc,void** argv){
  int rc;
  rc = $CLEARARRAY(argv,argv[0]);
  if(rc)return-1;
  return 0;
}
```

■ $GETARRAY (get value of array data)

`$GETARRAY` acquires information from the array at the specified location.

Syntax

```
char* $GETARRAY(void** argv,void* array,int pos);
```

Arguments

- `argv`

  Specifies the start address of the group of array variables that were specified in the script.
- `array`

  Specifies one of the elements of the group of array variables (argument of the user function).
- `pos`

  Specifies the position in the array from which information is to be acquired (begins at 1).

Return values

Return value	Description
Specified array information (character string)	Normal end
`NULL`	Error# or there is no information with the specified location number.

#

You can acquire detailed error information using *$GETSTATUS (get details of macro termination status)*.

Coding example

This example acquires information 1 in array 1 specified in the script:

```
int DllFunc3(int argc,void** argv){
  void *data;
  data = $GETARRAY(argv,argv[0],1);
  if(!data) if($GETSTATUS(argv) != JAM_SCRIPTAPI_NORMAL &&
  $GETSTATUS(argv) != JAM_SCRIPTAPI_NODATA) return -1;
  return 0;
}
```

■ $GETARRAYBYKEY (get value from array with key)

`$GETARRAYBYKEY` uses a specified key to acquire information from an array created using the `$SETARRAYBYKEY` embedded function in the script, or from an array created using the `$SETARRAYBYKEY` macro in the user function. If there is more than one element with the same key, specify the array number in the key to identify the location of the applicable element.

Syntax

```
char* $GETARRAYBYKEY(void** argv,void* array,char* key,int pos);
```

Arguments

- `argv`
  Specifies the start address of the group of array variables that were specified in the script.
- `array`
  Specifies one of the elements of the group of array variables (argument of the user function).
- `key`
  Specifies the key.
- `pos`
  Specifies the array number in the key (begins at 1).

Return values

Return value	Description
Array information specified by key (character string)	Normal end
`NULL`	Error# or there is no information about the specified array number in the key.

#

You can acquire detailed error information using *$GETSTATUS (get details of macro termination status)*.

Coding example

From array 1 specified in the script, this example acquires information about array 1 in the key stored by the `key1` key:

```
int DllFunc4(int argc,void** argv){
  void *data;
  data = $GETARRAYBYKEY(argv,argv[0],"key1",1);
  if(!data) if($GETSTATUS(argv) != JAM_SCRIPTAPI_NORMAL &&
  $GETSTATUS(argv) != JAM_SCRIPTAPI_NODATA) return -1;
  return 0;
}
```

■ $GETARRAYLENGTH (get number of array elements)

`$GETARRAYLENGTH` acquires the number of elements in the specified array.

Syntax

```
int $GETARRAYLENGTH(void** argv,void* array);
```

Arguments

- `argv`
  Specifies the start address of the group of array variables that were specified in the script.
- `array`
  Specifies one of the elements of the group of array variables (argument of the user function).

Return values

Return value	Description
Number of arrays (`0` or a greater numeric value)	Normal end
`-1`	Error#

#
You can acquire detailed error information using *$GETSTATUS (get details of macro termination status)*.

Coding example

This example acquires the number of elements in array 1 specified in the script:

```
int DllFunc10(void* obj,void* functbl,int argc,void** argv){
  int len;
  len = $GETARRAYLENGTH(argv,argv[0]);
  if (len<0) return -1;
  return 0;
}
```

■ $GETARRAYNAME (get name of array)

`$GETARRAYNAME` acquires the array name of a specified array.

Syntax

```
char* $GETARRAYNAME(void** argv,void* array);
```

Arguments

- `argv`
  Specifies the start address of the group of array variables that were specified in the script.
- `array`
  Specifies one of the elements of the group of array variables (argument of the user function).

Return values

Return value	Description
Array name (character string)	Normal end
`NULL`	Error#

#
You can acquire detailed error information using *$GETSTATUS (get details of macro termination status)*.

Coding example

This example acquires the array name of array 1 specified in the script:

```
int DllFunc9(int argc,void** argv){
  char* name;
  name = $GETARRAYNAME(argv,argv[0]);
  if(!name) return -1;
  return 0;
}
```

■ $GETINITAREA (get return value of aim_init function)

`$GETINITAREA` acquires an instance, which is the return value of the `aim_init` function.

Syntax

```
void * $GETINITAREA(void** argv);
```

Arguments

- `argv`

  Specifies the start address of the group of array variables that were specified in the script.

Return value

Return value of the `aim_init` function (instance)

Coding example

This example acquires the return value of the `aim_init` function:

```
int DllFunc(int argc, void** argv){
  USER_HANDLE*  obj;
  obj = $GETINITAREA(argv);
  return 0;
}
```

■ $GETKEYFROMARRAY (read corresponding key from array)

`$GETKEYFROMARRAY` acquires the key information having the specified array number from the information stored in the array with the key.

Syntax

```
char* $GETKEYFROMARRAY(void** argv,void* array,int pos);
```

Arguments

- `argv`

  Specifies the start address of the group of array variables that were specified in the script.

- `array`

  Specifies one of the elements of the group of array variables (argument of the user function).

- `pos`

  Array number of the array element for which information is to be acquired (begins at 1).

Return values

Return value	Description
Key information (character string)	Normal end[#1]

Return value	Description
`NULL`	Error[#2] or there is no value having the specified array number.

#1

If the specified array element has no key, the macro returns a character string of 0 bytes.

#2

You can acquire detailed error information using *$GETSTATUS (get details of macro termination status)*.

Coding example

This example acquires key information stored in array element 1 of array 3 specified in the script:

```
int DllFunc7(void* obj,void* functbl,int argc,void** argv){
  void *key;
  key = $GETKEYFROMARRAY(argv,argv[2],1);
  if(!key) if($GETSTATUS(argv) != JAM_SCRIPTAPI_NORMAL &&
  $GETSTATUS(argv) != JAM_SCRIPTAPI_NODATA) return -1;
  return 0;
}
```

■ $GETSTATUS (get details of macro termination status)

`$GETSTATUS` acquires the termination status of the processing.

Syntax

```
int $GETSTATUS(void** argv);
```

Arguments

- `argv`

  Specifies the start address of the group of array variables that were specified in the script.

Return value

Return value	Description
`JAM_SCRIPTAPI_NORMAL`	Normal end
`JAM_SCRIPTAPI_DUPLICATE`	Duplicate key value
`JAM_SCRIPTAPI_NODATA`	No corresponding array data
`JAM_SCRIPTAPI_INSUFFICIENTMEMORY`	Insufficient memory for processing
`JAM_SCRIPTAPI_ILLEGAL`	Illegal call interface
`JAM_SCRIPTAPI_ERROR`	Other internal error

Coding example

See the coding example in *$GETARRAY (get value of array data)*.

■ $SETARRAY (set value to array)

`$SETARRAY` adds information to an array.

Syntax

```
int $SETARRAY(void** argv,void* array,char* value);
```

Arguments

- `argv`

  Specifies the start address of the group of array variables that were specified in the script.

- `array`

  Specifies one of the elements of the array variable group (argument of the user function to which information is to be added).

- `value`

  Specifies the information to be added (character string).

Return values

Return value	Description
`0`	Normal end
`-1`	Error#

#

You can acquire detailed error information using *$GETSTATUS (get details of macro termination status).*

Coding example

This example adds an array element using information `data1` to array 2 specified in the script:

```
int DllFunc1(int argc, void** argv){
  int rc;
  rc = $SETARRAY(argv,argv[1],"data1");
  if(rc) return -1;
  return 0;
}
```

■ $SETARRAYBYKEY (set value to array with key)

`$SETARRAYBYKEY` adds information to an array with a key.

Syntax

```
int $SETARRAYBYKEY(void** argv,void* array,char* key,char* value);
```

Arguments

- `argv`

  Specifies the start address of the group of array variables that were specified in the script.

- `array`

  Specifies one of the elements of the array variable group (argument of the user function to which information is to be added).

- `key`

  Specifies the key.

- `value`

  Specifies the information to be added (character string).

Return values

Return value	Description
0	Normal end
-1	Error#

#

You can acquire detailed error information using *$GETSTATUS (get details of macro termination status)*.

Coding example

This example adds an array element using the key value `key1` and the information `data1` to array 2 specified in the script:

```
int DllFunc2(int argc,void** argv){
  int rc;
  rc = $SETARRAYBYKEY(argv,argv[1],"key1","data1");
  if(rc) return -1;
  return 0;
}
```

■ $UPDARRAY (update array data)

`$UPDARRAY` updates the value of an array element of an array variable. The array element is specified by its array number.

Syntax

```
int $UPDARRAY(void** argv,void* array,int pos,char* value);
```

Arguments

- `argv`
  Specifies the start address of the group of array variables that were specified in the script.
- `array`
  Specifies one of the elements of the array variable group (argument of the user function whose array element is to be updated).
- `pos`
  Specifies the array number of the array element whose `Array` information is to be updated (begins at 1).
- `value`
  Specifies the new data to be used (character string).

Return values

Return value	Description
0	Normal end
-1	Error#

#

You can acquire detailed error information using *$GETSTATUS (get details of macro termination status)*.

Coding example

This example updates the value of array element 2 of array 2 specified in the script to `data2`:

```
int DllFunc5(int argc,void** argv){
  int rc;
  rc = $UPDARRAY(argv,argv[1],2,"data2");
  if(rc) if($GETSTATUS(argv) != JAM_SCRIPTAPI_NORMAL &&
  $GETSTATUS(argv) != JAM_SCRIPTAPI_NODATA) return -1;
  return 0;
}
```

■ $UPDARRAYBYKEY (update value of array with key)

`$UPDARRAYBYKEY` updates the value of an array element of an array variable. The array element is expressed as a key and array number in the key.

Syntax

```
int $UPDARRAYBYKEY(void ** argv,void* array,char* key,int pos,char*
value);
```

Arguments

- `argv`

  Specifies the start address of the group of array variables that were specified in the script.
- `array`

  Specifies one of the elements of the array variable group (argument of the user function whose array element is to be updated).
- `key`

  Specifies the key.
- `pos`

  Specifies the array number in the key (begins at 1).
- `value`

  Specifies the new data to be used (character string).

Return values

Return value	Description
`0`	Normal end
`-1`	Error#

#

You can acquire detailed error information using *$GETSTATUS (get details of macro termination status)*.

Coding example

This example updates the value of array element 1 stored by the key `key1` in array 2 specified in the script to `data1`:

```
int DllFunc6(int argc,void** argv){
  int rc;
  rc = $UPDARRAYBYKEY(argv,argv[1],"key1",1,"data1");
  if(rc) if($GETSTATUS(argv) != JAM_SCRIPTAPI_NORMAL &&
  $GETSTATUS(argv) != JAM_SCRIPTAPI_NODATA) return -1;
  return 0;
}
```

$DLLFREE (free DLL)

`$DLLFREE` terminates the use of DLL (frees DLL) that was loaded by the `$DLLLOAD` embedded function.

Syntax

`$DLLFREE(`*DLL-object*`)`

Values

- *DLL-object*
  Specifies the variable name of the DLL object that was acquired by the `$DLLLOAD` embedded function.

DLL interface to be used

The following shows the format of the function that is called by `$DLLFREE`:

```
void aim_free(void* object)
```

The `$DLLFREE` embedded function executes the `aim_free` function using the DLL object as its argument, and unloads the DLL upon completion of the execution. The `aim_free` function frees the shared memory and other resources that were used by the functions called by the `aim_init` function and the `$DLLEXEC2` embedded function.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	Indicates one of the following: • The variable name of the specified DLL object is not the DLL object acquired by `$DLLLOAD`. • Argument error or other error

Legend:
—: Not applicable

Example

See the coding example in *$DLLEXEC2 (execute DLL)*.

$DLLLOAD (load DLL)

`$DLLLOAD` loads a DLL in which user functions are included, and acquires an object.

When loading is successful, this function returns the acquired DLL object. When loading fails, the function returns a character string of 0 bytes.

Syntax

DLL-object`=$DLLLOAD(`*DLL-name*`)`

Values

- *DLL-object*
  Specifies the name of the variable in which the DLL object is to be set.
- *DLL-name*
  Specifies the name of DLL to be loaded, as either a constant or a variable. A constant must be enclosed in single quotation marks (`''`).
  Express the DLL name as a path relative to the base path specified in the `-bp` option of the `jamscript` command. If the `-bp` option was omitted, *Asset-Console-installation-folder*`\scriptwork` is assumed to be the reference folder.

DLL interface to be used

The following three execution control functions must have been exported previously to the DLL to be called:

- `aim_init` function for initializing instances
- `aim_getmessage` function for sending error message responses
- `aim_free` function for freeing instances

The following shows the format of each execution control function:

```
void* aim_init()
void aim_free(void* dllobj)
int aim_getmessage(void* dllobj, char** msg)
```

When loading of DLL is successful, this function calls the `aim_init` function to create an instance. The `aim_init` function returns no error. In the event of an error, to output detailed error information to a log file, use information such as the error message address as the return value of the `aim_init` function. To output an error message to Asset Console's log file, set the message by using the `aim_getmessage` function that is called after the `aim_init` function.

In the script, when an error is detected by the `$GETSTATUS` embedded function, execute the `$DLLFREE` embedded function and free the DLL. Because the `aim_free` function is called when `$DLLFREE` is executed, free the area for the return value of the `aim_init` function.

If the `aim_init` function has terminated normally, make sure that the `aim_getmessage` function returns `0`. This makes the `$DLLEXEC2` function executable.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	Indicates one of the following: • The specified DLL does not exist. • Loading of DLL failed. • DLL does not contain the required function entry.
Script execution interrupted	Indicates one of the following: • The specified DLL does not exist. • Loading of DLL failed. • Argument error or other error

Legend:
—: Not applicable

Note

The user must have provided previously the DLL that is to be loaded by the `$DLLLOAD` embedded function. Store the header files provided by Asset Console together with the source files in the compilation environment, and then compile them. During the compilation, specify the `/MT` option.

The header files are stored at the following location:

```
Asset-Console-installation-folder\sdk\include
```

Example

See the coding example in *$DLLEXEC2 (execute DLL)*.

Creation of execution control functions

To call a DLL user function from the Asset Console script, the following three execution control functions are required:

- `aim_init`
- `aim_free`
- `aim_getmessage`

■ aim_init

Function

`aim_init` creates an instance and sets its address in the return value. This enables other functions thereafter to acquire the address created by the `aim_init` function. By holding values in instances, you can avoid possible conflict between threads.

Syntax

```
void* aim_init()
```

Return value

- If the processing is successful, the function returns the address of the handle used by this DLL. The address may be `NULL`. Make sure that a value of `0` is returned, because the `aim_getmessage` function is called immediately after control is returned.

- To output a resumable error and details about the error to Asset Console's log file, return the address of the handle with the error information set so that the message can be acquired by the `aim_getmessage` function. At the same time, return the code that determines the termination status of the script. For details about the return code, see *Return value of aim_getmessage function*.

■ aim_free

Function

`aim_free` frees the instance created by the `aim_init` function.

Syntax

```
void aim_free(void* dllobj)
```

Arguments

- `dllobj` (input-information)

  Specifies the instance created by the `aim_init` function.

■ aim_getmessage

Function

`aim_getmessage` enables you to output the messages in the created DLL to Asset Console's log file, and easily acquire them by using a script. The `aim_getmessage` function is called at the following times:

- After execution of the `aim_init` function

  The `aim_getmessage` function outputs the contents of `msg` to Asset Console's log file.

- After execution of the created function

  If a function returns a negative value (in the event of a script execution interrupted error), the `aim_getmessage` function is executed and sets the address in `msg` as required. When a message is set, the contents of `msg` are output to Asset Console's log file.

- During execution of the `$DLLMSG` embedded function

  Executed by the `$DLLMSG` function, the `aim_getmessage` function acquires the message using variables.

Syntax

```
int aim_getmessage(void* dllobj, char** msg)
```

Arguments

- `dllobj` (input-information)

  Specifies the instance created by the `aim_init` function.

- `msg` (output-information)

  Specifies the address where the message is stored.

Return value

The following table lists and describes the return value and the processing that is executed:

Return value	Description	Executable processing
`0`	Normal end	Changes the script status to `NORMAL`.
`1`	Warning	Changes the script status to `NODATA`.
Positive value	Error	Changes the script status to `ERROR`.
Negative value	Forced termination	Forcibly terminates the script.

Coding example

This example presents coding of a DLL source file and its association with a call from the access definition file.

- Source file of DLL (C++)

```
#include <stdio.h>
#include <windows.h>

#include "jamScriptAPI.h"
extern "C" __declspec(dllexport) void*  aim_init();
extern "C" __declspec(dllexport) void   aim_free(void*);
extern "C" __declspec(dllexport) int    aim_getmessage(void*, char**);
extern "C" __declspec(dllexport) int    DllFunc1(int, void**);
extern "C" __declspec(dllexport) int    DllFunc2(int, void**);

typedef struct aimsample{
    int         status;
    int         datalen;
    char**      data;
    char        message[256];
}AIMSAMPLE;
BOOL APIENTRY DllMain( HANDLE hModule,
                        DWORD  ul_reason_for_call,
                        LPVOID lpReserved
                     )
{
    return TRUE;
}
void* aim_init()
{
    AIMSAMPLE* dllobj = NULL;
    dllobj = (AIMSAMPLE*)LocalAlloc(LMEM_FIXED, sizeof(AIMSAMPLE));
    if(!dllobj) {
        /* error handling processing */
        return NULL;
    }
    dllobj->status = 0;
    dllobj->datalen = 0;
    dllobj->data = NULL;
    *(dllobj->message) = '\0';
    return dllobj;
}
void aim_free(void* dllobj)
{
    int i;
    if(dllobj){
        if(dllobj->data){
            for(i=0;i<dllobj->datalen;i++){
                if(*(dllobj->data+i)){
                    LocalFree(*(dllobj->data+i));
                    *(dllobj->data+i) = NULL;
                }
            }
            LocalFree(dllobj->data);
            dllobj->data = NULL;
        }
        LocalFree(dllobj);
    }
```

```
}
int aim_getmessage(void* dllobj, char** message)
{
    if(dllobj){
        *message = ((AIMSAMPLE*)dllobj)->message;
    }
    return 0;
}
int DllFunc1(int argc, void** argv)
{
    AIMSAMPLE* dllobj = NULL;
    int i,status,length;
    char* data;
    if(argc != 1){
       /* error handling processing */
        return -1;
    }
    /* acquire the area returned by aim_init*/
    dllobj = (AIMSAMPLE*)$GETINITAREA(argv);
    if(!dllobj){
        /* error handling processing */
        strcpy(dllobj->message, "Specified argument is invalid.");
        return -1;
    }
    if(dllobj->status != 0){
        strcpy(dllobj->message, "Call sequence is invalid.");
        return -1;
    }
    length = $GETARRAYLENGTH(argv, argv[0]);
    status = $GETSTATUS(argv);
    if(status != JAM_SCRIPTAPI_NORMAL){
        /* error handling processing */
        strcpy(dllobj->message, "Error was detected by $GETARRAYLENGTH
function.");
        return -1;
    }
    dllobj->datalen = length;
    dllobj->data = (char**)LocalAlloc(LMEM_FIXED, sizeof(char*)*length);
    if(!dllobj->data){
        /* error handling processing */
        strcpy(dllobj->message, "Memory allocation failed.");
        return -1;
    }
    ZeroMemory(dllobj->data, sizeof(char*)*length);
    for(i=0;i<length;i++){
        data = (char*)$GETARRAY(argv, argv[0], i+1);
        status = $GETSTATUS(argv);
        if(status != JAM_SCRIPTAPI_NORMAL){
            /* error handling processing */
            strcpy(dllobj->message, "Error was detected by $GETARRAY
function.");
            return -1;
        }
        *(dllobj->data+length-(i+1)) = (char*)LocalAlloc(LMEM_FIXED,
strlen(data)+1);
        if(!*(dllobj->data+length-(i+1))){
            /* error handling processing */
            strcpy(dllobj->message, "Memory allocation failed.");
            return -1;
        }
        strcpy(*(dllobj->data+length-(i+1)), data);
    }
    /* specific processing */
    dllobj->status = 1;
    *(dllobj->message) = '\0';
    return 0;
}
int DllFunc2(int argc, void** argv)
{
    AIMSAMPLE* dllobj = NULL;
    int i;
    if(argc != 1){
        /* error handling processing */
        return -1;
    }
    /* acquire the area returned by aim_init*/
    dllobj = (AIMSAMPLE*)$GETINITAREA(argv);
    if(!dllobj){
        /* error handling processing */
        return -1;
    }
    if(dllobj->status != 1){
        strcpy(dllobj->message, "Call sequence is invalid.");
        return -1;
    }
    if(dllobj->data){
        for(i=0;i<dllobj->datalen;i++){
            if(*(dllobj->data+i)){
                $SETARRAY(argv, argv[0], *(dllobj->data+i));
            }
        }
    }
    dllobj->status = 2;
    *(dllobj->message) = '\0';
    return 0;
}
```

- Association with a call from the access definition file

  This example outputs the array data acquired by `DllFunc1` to `DllFunc2` in reverse order. If `DllFunc1` and `DllFunc2` are not executed in this order, the script will be interrupted.

```
#AssetInformationManager HTML 0005

[VAR]
  DLLOBJ
  DATA
  STATUS
[ARRAY]
  ARY1
  ARY2
[SET_VALUE]
  DLLOBJ = $DLLLOAD('jamsample.dll')
  STATUS = $GETSTATUS()
[IF]
  STATUS != NORMAL
```

```
[THEN]
  [SET_VALUE]
    #error handling processing
    $EXIT(1)
[IF_END]
[SET_VALUE]
  $SETARRAY(ARY1, 1)
  $SETARRAY(ARY1, 2)
  $SETARRAY(ARY1, 3)
  $DLLEXEC2(DLLOBJ,'DllFunc1',ARY1)
  $DLLEXEC2(DLLOBJ,'DllFunc2',ARY2)
  DATA = $GETARRAY(ARY2,1)
  $ECHO(DATA)
  DATA = $GETARRAY(ARY2,2)
  $ECHO(DATA)
  DATA = $GETARRAY(ARY2,3)
  $ECHO(DATA)
  $DLLFREE(DLLOBJ)
```

Output result:

```
3
2
1
```

$DLLMSG (DLL get message)

`$DLLMSG` acquires a message from the `$DLLEXEC2` and `$DLLLOAD` embedded functions that executed immediately beforehand. To acquire a message from the `$DLLMSG` embedded function, set the message address in the `aim_getmessage` function contained in the DLL that is being called.

Syntax

variable-name=`$DLLMSG(`*DLL-object*`)`

Values

- *DLL-object*
  Specifies the variable name of the DLL object that was acquired by the `$DLLLOAD` embedded function.

DLL interface to be used

The following shows the format of the function called by the `$DLLMSG` embedded function:

```
int aim_getmessage(void* object          //Return value of the aim_init
function
                     ,char** message   //Reference pointer for the message
                   )
```

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end (when the function returns `0`)
`NODATA`	Termination with warning (when the function returns `1`)
`ERROR`	Abnormal termination (when the function returns a positive value)
Script execution interrupted	Indicates one of the following: • Abnormal termination (when the function returns a negative value)# • An invalid argument was specified, or an error other than the above occurred.

#
Indicates that the variable name of the specified DLL object is not the DLL object acquired by the `$DLLLOAD` embedded function.

Example

See the example for *$DLLEXEC2 (execute DLL)*.

$ECHO (output stdconsol)

`$ECHO` outputs a message to the Command Prompt.

Syntax

`$ECHO(`*`message`*`)`

Values

- *message*
  Specifies a message, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

Example

The following examples output the message `Hello world`:

```
[SET_VALUE]
  $ECHO('Hello world')

[SET_VALUE]
  MSG = 'Hello world'
  $ECHO(MSG)
```

Execution result:

```
Hello world
Hello world
```

$ENVIRONMENT (get environment information)

`$ENVIRONMENT` acquires the environment settings information of the asset management server.

Syntax

return-value=`$ENVIRONMENT(`*section-name*`,`*key-name*`)`

Values

- *return-value*
  Specifies the name of the variable into which the acquired environment information is set.
- *section-name*
  Specifies a section name that corresponds to an environment settings category on the asset management server. A constant must be enclosed in single quotation marks (`''`).
- *key-name*
  Specifies the name of the key that corresponds to the value of an environment setting, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

For details about section name and key names, see the *JP1/IT Desktop Management 2 - Asset Console Configuration and Administration Guide*.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	The specified section name or key name does not exist.
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Example

The following example acquires the contract history acquisition setting specified in the environment settings:

```
[SET_VALUE]
  VAL= $ENVIRONMENT('BASE','CONTRACT_HISTORY')

  MSG = 'BASE CONTRACT_HISTORY = ' +VAL
  $ECHO(MSG)
```

Execution result:

```
BASE CONTRACT_HISTORY = YES
```

$EXIT (exit)

`$EXIT` ends processing on an access definition file.

Syntax

`$EXIT(`*return-code*`)`

Values

- *return-code*

  Specifies a value for the access definition file return code, either as a constant or a variable. The range that can be specified is from 0 to 9.

Remarks

Processing is interrupted if the specified value is out of the range that can be specified for *return-code*.

Example

The following example outputs the number of asset information data items whose status is active (`002`) or, if none are found, terminates processing with return code `3`:

```
[CLASS_FIND]
  AssetInfo
[FIND_DATA]
  (AssetInfo.AssetStatus = '002')AND
  (AssetInfo.AssetKind = '001')
[GET_VALUE]
  WORK = AssetInfo.AssetNo

[SET_VALUE]
  STATUS = $GETSTATUS()
  TOTAL = $DATACOUNT()
[IF]
  STATUS = NORMAL
[THEN]
  [SET_VALUE]
    MSG = 'DataCount : ' +TOTAL
    $ECHO(MSG)
[ELSE]
  [SET_VALUE]
    MSG = 'EXIT : 3'
    $ECHO(MSG)
    $EXIT(3)
[IF_END]
```

Execution result:

```
EXIT : 3
```

$FILEARRAY (output to the array data CSV file)

`$FILEARRAY` outputs the data stored in an array as a single row in a CSV file.

Syntax

`$FILEARRAY(`*file-object*`,`*array-name*`)`

Values

- *file-object*
  Specifies the variable name of the file object requested by the `$FILEOPEN` embedded function.
- *array-name*
  Specifies the name of the array that you wish to output as a record to the CSV file.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	Indicates one of the following: • The value returned does not match the file object requested by `$FILEOPEN`. • An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Example

See the example in *$FILEOPEN (open file)*.

$FILECLOSE (close file)

`$FILECLOSE` declares an end to editing on the file to which data was output.

Syntax

`$FILECLOSE (`*file-object*`)`

Values

- *file-object*

  Specifies the file object of the file for which editing is to be ended. Specify the variable name of the file object requested by the `$FILEOPEN` embedded function.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	Indicates one of the following: • The value returned does not match the file object requested by `$FILEOPEN`. • An error occurred when an attempt was made to close the file. • An invalid argument was specified, or an error other than the above occurred.

Legend:

—: Not applicable

Example

See the example in *$FILEOPEN (open file)*.

$FILECOPY (copy file)

`$FILECOPY` copies a file.

Syntax

`$FILECOPY(`*source-file-name*`,`*destination-file-name*`)`

Values

- *source-file-name*`,` *destination-file-name*

  Specifies a file name, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

  Specify the file name using a relative path referenced to the base path specified with the `-bp` option of the `jamscript` command. If the `-bp` option was omitted, *Asset-Console-installation-folder*`\scriptwork` is assumed to be the reference folder.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	Indicates one of the following: • No source file exists. • The output destination path does not exist. • The file name specification is invalid.
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Example

The following example copies the file `input.csv` to the file `output.csv`:

```
[SET_VALUE]
  SRCFILE = 'input.csv'
  OUTFILE = 'output.csv'
  $FILECOPY(SRCFILE, OUTFILE)
```

$FILEDEL (delete file)

`$FILEDEL` deletes a file.

Syntax

`$FILEDEL(`*file-name*`)`

Values

- *file-name*

  Specifies the name of a file to be deleted, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

  Specify the file name with a relative path referenced to the base path specified with the `-bp` option of the `jamscript` command. If the `-bp` option was omitted, *Asset-Console-installation-folder*`\scriptwork` is assumed to be the reference folder.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	Indicates one of the following: • The file specified by *file-name* does not exist. • An error occurred when an attempt was made to delete the file (for example, the file is locked). • An invalid argument was specified, or an error other than the above occurred.

Legend:

—: Not applicable

Example

The following example copies the file `input.csv` to the file `output.csv`, and deletes the source file `input.csv`:

```
[SET_VALUE]
  SRCFILE = 'input.csv'
  OUTFILE = 'output.csv'
  $FILECOPY(SRCFILE, OUTFILE)
  $FILEDEL(SRCFILE)
```

$FILEOPEN (open file)

`$FILEOPEN` declares the start of editing of a file to which data is to be output.

Syntax

file-object=`$FILEOPEN(`*file-name*`,`*mode*`)`

Values

- *file-object*
  Specifies the name of the variable into which the acquired file on which editing is to begin is set.
- *file-name*
  Specifies a file name, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).
  Specify the file name with a relative path referenced to the base path specified with the `-bp` option of the `jamscript` command. If the `-bp` option was omitted, *Asset-Console-installation-folder*`\scriptwork` is assumed to be the reference folder.
- *mode*
  Specifies one of the following modes as the editing method:
  - `NEW`: Create as a new file. Do not specify this mode for existing files.
  - `RENEW`: Overwrite an existing file. A new file is created if there is no existing file.
  - `ADD`: Append to an existing file. Do not specify this mode for new files.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	Indicates one of the following: • The file specified by *file-object* does not exist. • An invalid mode was specified. • An error occurred when an attempt was made to open the file (for example, the file is locked). • An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Example

The following example creates the file `output.csv`, and outputs to that file `"Asset ID","Asset No","Asset status","Asset type"`:

```
[SET_VALUE]
  FH = $FILEOPEN('output.csv', RENEW)
```

```
$SETARRAY(OUTLINE,'Asset ID')
$SETARRAY(OUTLINE,'Asset No')
$SETARRAY(OUTLINE,'Asset status')
$SETARRAY(OUTLINE,'Asset type')

$FILEARRAY(FH, OUTLINE)
$CLEARARRAY(OUTLINE)

$FILECLOSE(FH)
```

Execution result:

The following is output to `output.csv`:

```
"Asset ID","Asset No.","Asset status","Asset type"
```

$FILEPUT (output data to file)

`$FILEPUT` outputs data to a file.

Syntax

`$FILEPUT(`*file-object*`,`*character-string*`)`

Values

- *file-object*

  Specifies the file object of the file into which the data is to be output. Specify the variable name of the file object requested by the `$FILEOPEN` embedded function.

- *character-string*

  Specifies a character string to be output to a file, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	Indicates one of the following: • *file-object* does not match the file object requested by `$FILEOPEN`. • A file write error occurred. • An invalid argument was specified, or an error other than the above occurred.

Legend:

—: Not applicable

Example

The following example creates the file `output.csv`, and outputs `ABCDEFGHI` as a single row:

```
[SET_VALUE]
  FH = $FILEOPEN('output.csv', RENEW)

  $FILEPUT(FH, 'ABC')
  $FILEPUT(FH, 'DEF')
  $FILEPUT(FH, 'GHI')

  $FILECLOSE(FH)
```

Execution result:

The following is output to `output.csv`:

```
ABCDEFGHI
```

$FILEPUTLN (output CRLF to file)

`$FILEPUTLN` outputs a character string to a file, and adds a CRLF at the end.

Syntax

`$FILEPUTLN(`*file-object*`,`*character-string*`)`

Values

- *file-object*

  Specifies the file object of the file into which the CRLF is to be output. Specify the variable name of the file object requested by the `$FILEOPEN` embedded function.

- *character-string*

  Specifies a character string to be output to a file, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	Indicates one of the following: • *file-object* does not match the file object requested by `$FILEOPEN`. • An error occurred when an attempt was made to close the file. • An invalid argument was specified, or an error other than the above occurred.

Legend:

—: Not applicable

Example

The following example creates the file `output.csv`, and outputs `ABC`, `DEF`, and `GHI` on three separate lines:

```
[SET_VALUE]
  FH = $FILEOPEN('output.csv', RENEW)

  $FILEPUTLN(FH, 'ABC')
  $FILEPUTLN(FH, 'DEF')
  $FILEPUTLN(FH, 'GHI')
  $FILECLOSE(FH)
```

Execution result:

The following is output to `output.csv`:

```
ABC
DEF
GHI
```

$FINDFILE (find files)

`$FINDFILE` searches files in the specified folder.

Syntax

`$FINDFILE(`*folder-name*`,`*array-name*`)`

Values

- *folder-name*

  Specifies a path relative to the base path, as either a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

- *array-name*

  Specifies the name of the array that stores the name of the folder and the names of files in the folder, expressed as a variable.

  For this array, `File` and `Directory` are specified by the key value for file and folder, respectively.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`ERROR`	The specified folder is invalid.
Script execution interrupted	Indicates one of the following • A wildcard or a character string such as `..` or `.` was included in *folder-name*. • An invalid argument was specified, or an error other than the above occurred.

Example

This example searches files in *Asset-Console-installation-folder*`\wwwroot\bin` and outputs the name of the first file:

```
$FINDFILE('bin', FileData)
FILE=$GETARRAY(FileData, 1)
$ECHO(FILE)
```

Execution result:

```
bin\default.htm
```

$FORMATMSG (set a message format)

`$FORMATMSG` sets the format of a character string that is output to a message.

Syntax

message`=$FORMATMSG(`*message-text*`,`*character-string-1*`(,`*character-string-2*`(,...)))`

Values

- *message*

  Specifies a message, expressed as a variable.

- *message-text*

  Specifies the character string whose format is to be set, as either a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

- *character-string*

  Specifies the character string to be inserted in the message text, as either a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

  A maximum of 99 character strings can be specified.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`ERROR`	—
Script execution interrupted	Indicates one of the following: • More than 99 arguments were specified. • An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Example

This example outputs a message with the asset number and the processing inserted:

```
MSG = $FORMATMSG('Device %2 on asset number %1 failed','1001','deletion')
$ECHO(MSG)
```

Execution result:

```
Device deletion on asset number 1001 failed.
```

$GETARRAY (get value of array data)

`$GETARRAY` acquires information of a specified array number from information stored in an array.

Syntax

return-value=`$GETARRAY(`*array-name*`,`*array-number*`)`

Values

- *return-value*
  Specifies the name of the variable into which the acquired information is set.
- *array-name*
  Specifies the variable name of the array from which to read information.
- *array-number*
  Specifies the array number of the array element from which information is to be read, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`). Specified values can range from 1 to 2,147,483,647.
  If no array element of the specified array number exists, a 0-byte character string is returned.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	No information exists for the specified array number.
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Example

See the example for *$SETARRAY (set value to array)*.

$GETARRAYBYKEY (get value from array with key)

`$GETARRAYBYKEY` acquires information corresponding to a specified key from keyed information stored in an array.

Syntax

return-value=`$GETARRAYBYKEY(`*array-name*`,`*key-value*`(,`*array-number-in-key*`))`

Values

- *return-value*
  Specifies the name of the variable into which the acquired value is set.
- *array-name*
  Specifies the name of the array from which to read information.
- *key-value*
  Specifies the key to the information to be acquired. If no array element exists in the specified key, a 0-byte character string is returned.
- *array-number-in-key*
  Specifies the array number in the key when there are multiple data values that correspond to the specified key, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`). Specified values can range from 1 to 2,147,483,647.
  If no array element of the array number specified in the key exists, a 0-byte character string is returned.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	Indicates one of the following: • The specified key does not exist. • No information exists for *array-number* in the specified key.
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Example

See the example in *$SETARRAYBYKEY (set value to array with key)*.

$GETKEYFROMARRAY (get key information from array)

`$GETKEYFROMARRAY` acquires the key of a specified array number from keyed information stored in an array.

Syntax

return-value=`$GETKEYFROMARRAY(`*array-name*`,`*array-number*`)`

Values

- *return-value*
  Specifies the name of the variable into which the acquired key is set.
- *array-name*
  Specifies the variable name of the array variable from which key information is read.
- *array-number*
  Specifies a numeric character, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`). Specified values can range from 1 to 2,147,483,647.
  If no array element exists for the specified array number, a 0-byte character string is returned.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	No information exists for the specified array number.
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Example

The following example acquires the key information and value of the fifth element of the array variable `ARY`:

```
[SET_VALUE]
  $CLEARARRAY(ARY)

  $SETARRAYBYKEY(ARY,'CPU','100')     # ARY[1] CPU[1]
  $SETARRAYBYKEY(ARY,'CPU','200')     # ARY[2] CPU[2]
  $SETARRAYBYKEY(ARY,'HD' ,'40')      # ARY[3] HD[1]
  $SETARRAYBYKEY(ARY,'HD' ,'20')      # ARY[4] HD[2]
  $SETARRAYBYKEY(ARY,'MEM','128')     # ARY[5] MEM[1]
  $SETARRAYBYKEY(ARY,'MEM','256')     # ARY[6] MEM[2]

  KEY = $GETKEYFROMARRAY(ARY,5)
  VAL = $GETARRAY(ARY, 5)
  MSG = 'ARY[5]: KEY=' + KEY + ' VAL=' + VAL
  $ECHO(MSG)
```

Execution result:

```
ARY[5]: KEY=MEM VAL=128
```

$GETPROFILEDATA (get Windows initialization file data)

`$GETPROFILEDATA` reads all keys and values in the specified section of the specified Windows initialization file name into an array variable. The keys in the section are stored as array key values.

Syntax

`$GETPROFILEDATA(`*Windows-initialization-file-name*`,`*section-name*`,`*array-name*`)`

Values

- *Windows-initialization-file-name*

  Specifies the name of the Windows initialization file to be read. Specify the file name as either a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

  The Windows initialization file is stored in *Asset-Console-installation-folder*`\env`.

- *section-name*

  Specifies the name of the section to be acquired. Specify the section name as either a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

  If *section-name* is a 0-byte character string, this function stores all section names of the Windows initialization file in a key array. When this happens, the key values are not stored in the array.

- *array-name*

  Specifies the variable name of the array into which information from the Windows initialization file is read.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	Indicates one of the following: • The Windows initialization file does not exist. • The specified section does not exist.
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Example

The following example acquires values from the section `TITLE` and the key name `HardwareInfo` from the Windows initialization file `Sample.ini`:

```
[SET_VALUE]
  FILENAME = 'Sample.ini'
  SECTION  = 'TITLE'

  $GETPROFILEDATA(FILENAME, SECTION, ARY)
  VAL=$GETARRAYBYKEY(ARY, 'HardwareInfo')
  MSG = 'HardwareInfo = ' + VAL
  $ECHO(MSG)
```

`Sample.ini` file

```
[TITLE]
  AssetInfo    = Asset Information
  HardwareInfo = Hardware Information
  SoftwareInfo = Software Information
```

Execution result:

```
HardwareInfo = Hardware Information
```

$GETREGVALUE (get a registry value)

`$GETREGVALUE` acquires the specified registry value.

Syntax

`$GETREGVALUE('`*registry-name*`')`

Values

- *registry-name*
  Specifies a registry name expressed as a character string.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`ERROR`	Registry acquisition failed.
Script execution interrupted	Indicates one of the following: • The attribute of the specified registry is not a character string or `DWORD`. • An invalid argument was specified, or an error other than the above occurred.

Example

The following example acquires the value of `KEYVERSION`(`1050`):

```
[SET_VALUE]
  AIMVERSION = $GETREGVALUE('KEYVERSION')
  STATUS = $GETSTATUS()
[IF]
  STATUS = NORMAL
[THEN]
  [SET_VALUE]
    MSG = 'AIMVERSION = ' + AIMVERSION
    $ECHO(MSG)
[ELSE]
  [SET_VALUE]
    MSG = '$GETREGVALUE (' + STATUS + ')'
    $ECHO(MSG)
[IF_END]
```

Execution result:

```
AIMVERSION = 1050
```

$GETROLE (get role of user)

`$GETROLE` reads a list of user roles currently executing asset management jobs in operation windows into the specified array.

Syntax

`$GETROLE(`*array-name*`)`

Values

- *array-name*
  Specifies the variable name of an array.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	Indicates one of the following: • Database access error • An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Remarks

If no roles have been set for users, the process ends normally, but no information is set into the array variable.

Example

The following example acquires user roles, then outputs the role (`administrator`) in array number 1:

```
[SET_VALUE]
  $GETROLE(ARY)
  VAL = $GETARRAY(ARY, 1)
  MSG = 'ROLE = ' + VAL
  $ECHO(MSG)
```

Execution result:

```
ROLE = administrator
```

$GETSESSION (get session information)

`$GETSESSION` acquires the value specified with the `-s` option of the `jamscript` command. The syntax of the `-s` option is `-s` *session-variable-name*=*value*.

Syntax

return-value=`$GETSESSION(`*session-variable-name*`)`

Values

- *return-value*

  Specifies the name of the variable into which the acquired session information is set.
- *session-variable-name*

  Specifies the variable name of the session to be acquired, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	The specified session variable does not exist.
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:

—: Not applicable

Remarks

If no information exists for the specified session variable name, a 0-byte character string is returned.

Example

The following example specifies the `-s` option of the `jamscript` command to acquire information (`aaaa`):

```
[SET_VALUE]
  VAL = $GETSESSION('OPTION')
  MSG = 'OPTION = ' + VAL
  $ECHO(MSG)
```

Execution command:

```
jamscript -f C:\Test.txt -s OPTION=aaaa
```

Execution result:

```
OPTION = aaaa
```

$GETSTATUS (get status)

$GETSTATUS acquires the status of a process. Four statuses exist: NORMAL (normal end), NODATA (no data exists), ERROR (an error occurred), and MULTI (another user is currently updating).

Syntax

return-value=$GETSTATUS()

Values

- *return-value*
  Specifies the name of the variable into which the acquired status is set.

Example

The following example searches for information of asset number R11111, and updates the status of the asset to STOCK. In this example, an error occurs if the information of R11111 is updated by another agent after the search has finished.

```
[TRANSACTION]
  [CLASS_FIND]
    AssetInfo
  [FIND_DATA]
    (AssetInfo.AssetNo  = 'R11111')
  [GET_VALUE]
    ASSETID     = AssetInfo.AssetID
    ASSETSTATUS = AssetInfo.AssetStatus
    ASSETNO     = AssetInfo.AssetNo
    UPDCK       = AssetInfo.UpdateTime

  [SET_VALUE]
    STATUS = $GETSTATUS()
  [IF]
    STATUS = NORMAL
    [THEN]
      [UPDATE]
        AssetInfo
      [DATA]
        AssetInfo.AssetID     = ASSETID
        AssetInfo.AssetStatus = '301'
        AssetInfo.UpdateTime  = UPDCK

      [SET_VALUE]
        STATUS = $GETSTATUS()
      [IF]
        (STATUS = NORMAL)
        [THEN]
          [SET_VALUE]
            MSG = 'ASSETNO('+ASSETNO+') status updated.'
            $ECHO(MSG)
      [IF_END]
      [IF]
        (STATUS = MULTI)
        [THEN]
          [SET_VALUE]
```

```
            MSG = 'ASSETNO(' + ASSETNO + ') already updated by other agent.'
            $ECHO(MSG)
      [IF_END]
      [IF]
        (STATUS != NORMAL) and (STATUS != MULTI)
        [THEN]
          [SET_VALUE]
            MSG = 'ASSETNO(' + ASSETNO + ') status cannot be updated.'
            $ECHO(MSG)
      [IF_END]

    [ELSE]
      [SET_VALUE]
        MSG = 'ASSETNO(' + ASSETNO + ') was not registered.'
        $ECHO(MSG)
  [IF_END]
[TRANSACTION_END]
```

To suppress concurrent updating as shown in this example, acquire the search time with *class-name*`.UpdateTime`, and specify the acquired value as is in an `[UPDATE]` tag. Suppression of concurrent updating is valid only when updating; it cannot be performed when adding or deleting information.

$GETTEMPNAME (get temporary file name)

`$GETTEMPNAME` specifies the name of a temporary file that is managed by session.

The file with the file name specified with the `$GETTEMPNAME` embedded function is deleted when a logout, forcible logout, or a session-ending timeout occurs. When downloading a CSV file, by using a name specified by the `$GETTEMPNAME` embedded function, you can automatically delete the created file. When `$GETTEMPNAME` is used with the `jamscript` command, the file is deleted when the command ends.

If you wish to open the file being downloaded with a browser helper application, specify the file name extension assigned to the helper application.

Syntax

return-value=`$GETTEMPNAME(`*unique-character-string*`)`

Values

- *return-value*
  Specifies the name of the variable into which the acquired temporary file name is set.
- *unique-character-string*
  Specifies a character string that is unique within the session, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Example

The following example uses the `$GETTEMPNAME` embedded function to acquire the name of a temporary file. As indicated under *Execution result*, the temporary file name is set automatically by Asset Console.

```
[SET_VALUE]
  FILENAME = $GETTEMPNAME('Sample.csv')
  MSG = 'FILENAME = ' + FILENAME
  $ECHO(MSG)
```

Execution result:

```
FILENAME = csv/$$3FAE0F01000004EC0001$4$Sample.csv
```

$GOSUB (execute subroutine)

`$GOSUB` executes a subroutine defined by the `[SUB]` tag. The subroutine to be executed must be defined before the `$GOSUB` function appears.

Syntax

`$GOSUB(`*subroutine-name*`)`

Values

- *subroutine-name*

  For the subroutine name, you can use alphanumeric characters and the underscore (_). However, a numeric character cannot be used as the first character of the subroutine name. Note also that case is considered significant in subroutine names.

Example

The following example creates a subroutine to perform the processing for outputting a message according to the value in the session information `MSG`:

```
[SUB]
  ECHO
  [IF]
    MSG = '1'
  [THEN]
    [SET_VALUE]
      $ECHO(ECHOMSG)
  [IF_END]
[SUB_END]

[BEGIN]
  [SET_VALUE]
    MSG = $GETSESSION('MSG')

    ECHOMSG = 'Hello world'
    $GOSUB(ECHO)
[END]
```

$ISNULL (check NULL)

`$ISNULL` determines whether data acquired with the `[CSV_COLUMN_NAME]` tag is `NULL` or a 0-byte character.

Syntax

return-value=`$ISNULL(`*column-name*`)`

Values

- *return-value*

  Specifies the name of the variable into which the evaluated results are set. If the value of the column specified by *column-name* is `NULL`, `1` is returned; if it is a 0-byte character, `0` is returned.

- *column-name*

  Specifies the variable name of the column that is defined by the `[CSV_COLUMN_NAME]` tag.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Example

The following example acquires data from the file `input.csv`; it then outputs `COLUMN1 IS NULL` if `COLUMN1` is the `NULL` character, or outputs `COLUMN1 LENGTH IS 0` if `COLUMN1` is a 0-byte character:

```
[SET_VALUE]
  FILENAME = 'input.csv'
  CNT = 1
[CSV_FILE_LOOP]
  FILENAME
  [CSV_COLUMN_NAME]
    COLUMN1 = 1
  [BEGIN]
    [SET_VALUE]
      LEN = $LENGTH(COLUMN1)
    [IF]
      LEN = 0
      [THEN]
        [SET_VALUE]
          VAL=$ISNULL(COLUMN1)
        [IF]
          VAL = 1
          [THEN]
            [SET_VALUE]
```

```
              MSG = 'LINE('+CNT+') COLUMN1 IS NULL'
          [ELSE]
            [SET_VALUE]
              MSG = 'LINE('+CNT+') COLUMN1 LENGTH IS 0'
        [IF_END]

      [ELSE]
        [SET_VALUE]
          MSG = 'LINE('+CNT+') COLUMN1 LENGTH IS '+LEN
    [IF_END]
    [SET_VALUE]
      $ECHO(MSG)
      CNT = $ADD(CNT,1)
  [END]

  [SET_VALUE]
    $SETSTATUS('NORMAL')
[CSV_FILE_LOOP_END]
```

Contents of `input.csv`

```
,bbb,ccc
"",bbb,ccc
aaa,bbb,ccc
"aaa",bbb,ccc
```

Execution result:

```
  LINE(1) COLUMN1 IS NULL
  LINE(2) COLUMN1 LENGTH IS 0
  LINE(3) COLUMN1 LENGTH IS 3
  LINE(4) COLUMN1 LENGTH IS 3
```

$LDAPACS (access directory)

`$LDAPACS` provides authentication of connections to directory services, searching, entry acquisition, attribute acquisition, and other services that enable access to directory information. To use this embedded function for manipulating directory information, you must learn the methods and functions for accessing directory information.

Syntax

`$LDAPACS(`*function-name*`,`*argument-1*`(,`*argument-2*`(,...)))`

Values

- *function-name*

  Specifies the name of a function, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

- *argument*

  Specifies an argument for the function, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

Remarks

If information acquisition fails, a 0-byte character string is returned.

Detailed descriptions of functions that can be used to access directory information

The following table lists and describes the functions that can be used by the `$LDAPACS` embedded function.

Table 5–3: List of functions that can be used by the $LDAPACS embedded function

Function name	Description
`CONNECT`	Authenticates connection to directory service.
`CONVERT`	Converts data to a character string used in searching directory information.
`DISCONNECT`	Releases a connection to directory service.
`FIRSTENTRY`	Acquires the first entry that was found.
`FREEENTRY`	Releases an entry.
`FREERESULT`	Releases a search result.
`GETDN`	Acquires an entry DN.
`NEXTENTRY`	Acquires the second and subsequent entries that were found.
`SEARCH`	Searches directory service.
`SELECTVALUE`	Acquires an attribute value.

The following subsections provide a description of each of these functions, along with their syntax, arguments, and statuses. Status differs depending on the status acquired with the `$GETSTATUS` embedded function.

■ CONNECT

`CONNECT` authenticates connection to directory service, and returns a directory object.

Syntax

```
$LDAPACS('CONNECT',LDAPOBJ,HOST,PORT,USERDN,PASSWD)
```

Arguments

Argument	Type	Description
`LDAPOBJ`	Directory object	Specifies the name of the variable into which the directory object is set.
`HOST`	Variable or constant	Specifies the host name or IP address of the directory server.
`PORT`	Variable or constant	Specifies the port number of the directory server.
`USERDN`	Variable or constant	Specifies the user DN for authenticating a connection.
`PASSWD`	Variable or constant	Specifies the password for authenticating a connection.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	Invalid argument.
Script execution interrupted	Indicates one of the following: • An error occurred when an attempt was made to connect to the directory. • An authentication error occurred. • An error other than the above occurred.

Legend:

—: Not applicable

■ CONVERT

`CONVERT` converts data to a character string for use in searching the directory service.

Syntax

```
return-value=$LDAPACS('CONVERT',SOURCE)
```

- *return-value*

  Specifies the name of the variable into which the converted character string is set.

Arguments

Argument	Type	Description
`SOURCE`	Variable or constant	Specifies the character string to be converted.

Status

The following table lists and describes the possible statuses:

Status	Description
NORMAL	Normal end
NODATA	—
ERROR	Indicates one of the following: • Conversion failure. • Invalid argument.
Script execution interrupted	Indicates one of the following: • A variable is not defined. • An error other than the above occurred.

Legend:
—: Not applicable

■ DISCONNECT

`DISCONNECT` releases the directory service connection and all objects under it.

Syntax

```
$LDAPACS('DISCONNECT',LDAPOBJ)
```

Argument

Argument	Type	Description
LDAPOBJ	Directory object	Specifies the directory object requested by CONNECT.

Status

The following table lists and describes the possible statuses:

Status	Description
NORMAL	Normal end
NODATA	—
ERROR	LDAPOBJ does not match the directory object requested by CONNECT.
Script execution interrupted	Indicates one of the following: • A variable is not defined. • An error other than the above occurred.

Legend:
—: Not applicable

■ FIRSTENTRY

`FIRSTENTRY` acquires the first entry object found from the search object. To release an acquired object, you must call `FREEENTRY`.

Syntax

```
$LDAPACS('FIRSTENTRY',LDAPENT,LDAPRST)
```

Arguments

Argument	Type	Description
`LDAPENT`	Entry object	Specifies the name of the variable into which the entry object is set.
`LDAPRST`	Result object	Specifies the result object.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	No entry exists.
`ERROR`	Indicates one of the following: • `LDAPRST` does not match the search object requested by `SEARCH`. • Invalid argument.
Script execution interrupted	An error other than the above occurred.

■ FREEENTRY

`FREEENTRY` releases the specified entry object and all objects under it.

Syntax

```
$LDAPACS('FREEENTRY',LDAPENT)
```

Argument

Argument	Type	Description
`LDAPENT`	Entry object	Specifies an entry object.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	Indicates one of the following: • `LDAPENT` does not match the entry object requested by `FIRSTENTRY` or `NEXTENTRY`. • Invalid argument.
Script execution interrupted	An error other than the above occurred.

Legend:
—: Not applicable

■ FREERESULT

`FREERESULT` releases the specified result object and all objects under it.

Syntax

```
$LDAPACS('FREERESULT',LDAPRST)
```

Argument

Argument	Type	Description
`LDAPRST`	Result object	Specifies a result object.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	Indicates one of the following: • `LDAPRST` does not match the search object requested by `SEARCH`. • Invalid argument.
Script execution interrupted	An error other than the above occurred.

Legend:

—: Not applicable

■ GETDN

`GETDN` acquires the indicator (character string) from the entry object. The acquired character string cannot be released. You must use `FREEENTRY` to release its higher object.

Syntax

```
$LDAPACS('GETDN',LDAPDN,LDAPENT)
```

Arguments

Argument	Type	Description
`LDAPDN`	DN	Specifies the name of the variable into which the DN is set.
`LDAPENT`	Entry object	Specifies the entry object.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	Indicates one of the following: • `LDAPENT` does not match the entry object requested by `FIRSTENTRY` or `NEXTENTRY`. • Invalid argument.
Script execution interrupted	An error other than the above occurred.

Legend:

—: Not applicable

■ NEXTENTRY

NEXTENTRY acquires the second and subsequent entry objects found in the result object. This function cannot be called without first calling FIRSTENTRY. To release the acquired object, you must call FREEENTRY.

Syntax

```
$LDAPACS('NEXTENTRY',LDAPENT,LDAPRST)
```

Arguments

Argument	Type	Description
LDAPENT	Entry object	Specifies the name of the variable into which the entry object is set.
LDAPRST	Result object	Specifies the result object.

Status

The following table lists and describes the possible statuses:

Status	Description
NORMAL	Normal end
NODATA	—
ERROR	Indicates one of the following: • LDAPRST does not match the result object requested by SEARCH. • Invalid argument.
Script execution interrupted	An error other than the above occurred.

Legend:

—: Not applicable

■ SEARCH

SEARCH performs a synchronous search on the LDAP server.

To release the result object, you must call FREERESULT to release its higher object.

Syntax

```
$LDAPACS('SEARCH',LDAPRST,LDAPOBJ,BASE,FILTER,SCOPE)
```

Arguments

Argument	Type	Description
LDAPRST	Result object	Specifies the variable name into which the result object is set.
LDAPOBJ	Directory object	Specifies the directory object acquired by CONNECT.
BASE	Variable or constant	Specifies the base object from which the search starts.
FILTER	Variable or constant	Specifies the search filter.
SCOPE	Variable or constant	Referenced to the base object, specifies to search one of the following directory information levels:

Argument	Type	Description
SCOPE	Variable or constant	• LDAP_SCOPE_SUBTREE (Search all objects under the base object) • LDAP_SCOPE_ONELEVEL (Search objects directly below the base object) • LDAP_SCOPE_BASE (Search the base object)

Status

The following table lists and describes the possible statuses:

Status	Description
NORMAL	Normal end
NODATA	No matching data exists.
ERROR	Invalid argument.
Script execution interrupted	An error other than the above occurred.

■ SELECTVALUE

SELECTVALUE specifies an attribute name from an entry object, and acquires the value of the first attribute (character string).

The acquired character string cannot be released. You must use FREEENTRY to release its higher object.

Syntax

```
$LDAPACS('SELECTVALUE',LDAPSEL,LDAPENT,KEYNAME)
```

Arguments

Argument	Type	Description
LDAPSEL	Attribute value	Specifies the name of the variable into which the attribute value (character string) is set.
LDAPENT	Entry object	Specifies the entry object.
KEYNAME	Variable or constant	Specifies the name of the attribute you wish to acquire.

Status

The following table lists and describes the possible statuses:

Status	Description
NORMAL	Normal end
NODATA	No value exists for the specified attribute.
ERROR	Indicates one of the following: • LDAPENT does not match the entry object requested by FIRSTENTRY or NEXTENTRY. • Invalid argument.
Script execution interrupted	An error other than the above occurred.

Example

The following example outputs the DN and name of the user whose attribute `title;lang-ja` is `Supervisor`, from users who are registered to the directory `ou=people,o=xxxxxxx.co.us`:

```
[VAR]
  STATUS
  MSG
  HOST
  PORT
  FILTER
  BASE
  SCOPE
  FIRST
  LDOBJ
  LDRST
  LDENT
  DN
  NAME

[SET_VALUE]
  HOST = 'localhost'
  PORT = '389'
  BASE = 'ou=people,o=xxxxxxx.co.us'
  SCOPE= 'LDAP_SCOPE_ONELEVEL'

[SET_VALUE]
  $LDAPACS('CONNECT',LDOBJ,HOST,PORT,'','')            # CONNECT
  STATUS = $GETSTATUS()

  [SET_VALUE]
    FILTER = '(&(objectclass=*)(title;lang-ja='
    FILTER = FILTER+$LDAPACS('CONVERT','Supervisor')        # CONVERT
    FILTER = FILTER+'))'
    # FILTER=(&(objectclass=*)(title;lang-ja=\E4\B8\BB\E4\BB\BB))

    $LDAPACS('SEARCH',LDRST,LDOBJ,BASE,FILTER,SCOPE)  # SEARCH
    FIRST = 1

    [DO]
      [IF]
        FIRST = 1
        [THEN]
          [SET_VALUE]
            $LDAPACS('FIRSTENTRY',LDENT,LDRST)         # GET FIRST ENTRY
            STATUS = $GETSTATUS()
            FIRST = 0
        [ELSE]
          [SET_VALUE]
            $LDAPACS('NEXTENTRY',LDENT,LDRST)          # GET NEXT ENTRY
            STATUS = $GETSTATUS()
      [IF_END]

      [IF]
        STATUS = NORMAL
        [THEN]
          [SET_VALUE]
```

```
            $LDAPACS('GETDN',DN,LDENT)                # GET DN
            $LDAPACS('SELECTVALUE',NAME,LDENT,'cn')   # GET VALUE OF CN
            MSG='DN ['+DN+'] is '+NAME
            $ECHO(MSG)
            $LDAPACS('FREEENTRY',LDENT)                # FREE ENTRY OBJECT
        [ELSE]
          [SET_VALUE]
            $BREAK()
        [IF_END]
    [DO_END]

  [SET_VALUE]
    $LDAPACS('FREERESULT',LDRST)                       # FREE SEARCH OBJECT

[SET_VALUE]
  $LDAPACS('DISCONNECT',LDOBJ)                         # FREE LDAP OBJECT
```

Execution result:

```
DN [uid=user1, ou=people, o=xxxxxxx.co.us] is Smith
DN [uid=user3, ou=people, o=xxxxxxx.co.us] is Brown
```

$LENGTH (get string length)

`$LENGTH` acquires the length in bytes of the specified character string.

Syntax

return-value=`$LENGTH(`*character-string*`)`

Values

- *return-value*

  Specifies the name of the variable into which the acquired character string length is set.
- *character-string*

  Specifies the character string whose length is to be acquired, either as a constant or an array variable. A constant must be enclosed in single quotation marks (`''`).

  If an array variable is specified, the number of elements set in the array is returned.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:

—: Not applicable

Example

The following example acquires and outputs the length of character string `string length`:

```
[SET_VALUE]
  DATA = 'string length'
  VAL = $LENGTH(DATA)
  MSG = 'LENGTH = ' + VAL
  $ECHO(MSG)
```

Execution result:

```
LENGTH = 13
```

$LOGMSG (output to the log file)

`$LOGMSG` outputs a specified message to the log file `ASTMES`*n*`.LOG`.

Syntax

`$LOGMSG(`*message-type*`,`*message*`)`

Values

- *message-type*
  Specifies one of the following message types:
  - `E` (Error)
    Indicates that a severe problem occurred, and the program must be stopped.
  - `EW` (Warning)
    Indicates that a problem occurred and, for example, some functionality cannot be used but the program does not need to be stopped. `W` is output to the log file as the message type.
  - `EI` (Information)
    Indicates information. `I` is output to the log file as the message type.
- *message*
  Specifies a constant or variable for the message. If you specify a constant, enclose it in single quotation marks (`'`).

Note

If a *message type* other than `E`, `EW`, or `EI` is specified, the script will be interrupted.

Example

The following example shows the settings and the execution result when the message type `E` (error) is specified. As indicated under *Execution result*, text preceding the message is set automatically by Asset Console. For details about how to read the execution results, see the *JP1/IT Desktop Management 2 - Asset Console Configuration and Administration Guide*.

```
[SET_VALUE]
  MSG = 'Destination E-mail address is not specified.'
  $LOGMSG ('E', MSG)
```

Execution result:

```
0040519212834.673 00000594(00000664) KDAM2G14-E E-mail address is not
specified.
```

$LOWER (convert string)

`$LOWER` converts the alphabetic characters specified in the argument character string to lower case.

Syntax

return-value=`$LOWER(`*character-string*`)`

Values

- *return-value*
  Specifies the name of the variable into which the character string that has been converted to lower case is set.
- *character-string*
  Specifies the character string to be converted, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Example

The following example converts the host name acquired from system inventory information to lower case, and outputs the result:

```
[SET_VALUE]
  NAME = 'HOSTNAME'
  VAL = $LOWER(NAME)
  MSG = 'LOWER = ' + VAL
  $ECHO(MSG)
```

Execution result:

```
LOWER = hostname
```

$MATCH (check string)

`$MATCH` evaluates the characters used in a character string, and returns the number of characters up to but not including the first character that does not match.

Syntax

return-value=`$MATCH(`*character-string*`,`*evaluation-format*`)`

Values

- *return-value*

  Specifies the name of the variable into which the number of characters that matched *evaluation-format* is set. If no characters match, a 0-byte character string is returned.

- *character-string*

  Specifies the character string to be evaluated, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

- *evaluation-format*

  Specifies, as a constant, the format for evaluating *character-string*. If you are specifying a range of allowed characters, specify the range in a `[`*a*`-`*b*`]` format.

  In addition to simple character strings, the following regular expressions can be used in *evaluation-format*.

  - `.` (period)

    Finds any one character.

  - `[]` (square brackets)

    Finds any single character enclosed in the brackets, or any single character within the range indicated by the characters surrounding a hyphen (`-`). For example, `R[OAI]M` finds `ROM`, `RAM`, and `RIM`.

    Similarly, `S[AE]+D` finds `SAD`, `SED`, `SEED`, and `SAAD`, but does not find `SAED` or `SEAD`.

    `C[0-9]` finds `C0`, `C1`, `C2`, and so on.

    Specifying a circumflex (^) as the first character in the square brackets negates the meaning, and finds all characters other than the character that follows the circumflex.

  - `[^]`

    Finds any one character that is not the character following the circumflex (^), or is not in the range of characters indicated with the hyphen following the circumflex.

    For example, `x[^0-9]` finds `xa`, `xb`, `xc`, and so on, but does not find `x0`, `x1`, `x2`, or other `x`-*number* pairs.

  - `^`

    Finds the beginning of a character string.

  - `$` (dollar sign)

    Finds the end of a character string.

  - `*` (asterisk)

    Finds zero or more repetitions of the character or regular expression that appears immediately before the asterisk.

    For example, `ba*c` finds `bc`, `bac`, `baac`, `baaac`, and so on.

  - `+` (addition sign)

    Finds one or more repetitions of the character or regular expression that appears immediately before the addition sign.

    For example, `ba+c` finds `bac`, `baac`, `baaac`, and so on.

- \ (backslash)

  Finds the single character that directly follows this escape character specification. It disables the meaning of characters that have special meaning in regular expression character strings, such as asterisk (`*`) and dollar sign (`$`). In addition, note that the backslash followed by a `t` finds a tab character.

- `\t`

  Matches a tab character.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	No character that matches *evaluation-format* was found.
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:

—: Not applicable

Example

Example 1

The following example checks for characters other than alphanumeric characters:

```
[SET_VALUE]
  DATA = 'user$1'
  VAL  = $MATCH(DATA,'[^a-zA-Z0-9]')
[IF]
  VAL = ''
  [THEN]
    [SET_VALUE]
      MSG  = 'MATCH OK'
      $ECHO(MSG)
  [ELSE]
    [SET_VALUE]
      MSG  = 'MATCH NG ('+VAL+')'
      $ECHO(MSG)
[IF_END]
```

Execution result:

```
MATCH NG (4)
```

Example 2

The following example checks date formats:

```
[SET_VALUE]
  DATA = '2015/04/01'
  VAL  = $MATCH(DATA,'^[1-2][0-9][0-9][0-9]/[0-1][0-9]/[0-3][0-9]$')
[IF]
  VAL != ''
  [THEN]
    [SET_VALUE]
      MSG  = 'MATCH OK'
      $ECHO(MSG)
  [ELSE]
    [SET_VALUE]
      MSG  = 'MATCH NG'
      $ECHO(MSG)
[IF_END]
```

Execution result:

```
MATCH OK
```

$MOD (divide and return only the remainder)

`$MOD` performs division, treating character strings as numeric values, and returns the remainder of the arithmetic result.

Syntax

return-value=`$MOD(`*character-string*`,`*numeric-character*`)`

Values

- *return-value*

  Specifies the name of the variable into which the arithmetic result is set. Valid results range from 0.0001 to 999,999,999,999,999 (15 digits). If the result is outside the valid range, 0 is assumed.

- *character-string*

  Specifies a dividend, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`). Specified values can range from 0.0001 to 999,999,999,999,999 (15 digits).

- *numeric-character*

  Specifies a divisor, either as a constant or a variable. A numeric value specified as a constant that includes a decimal point must be enclosed in single quotation marks (`''`). Specified values can range from 0.0001 to 999,999,999,999,999 (15 digits). If `0` is specified, 0 is returned to *return-value*.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	Indicates one of the following: • An invalid value was specified in a character string or numeric value. • The arithmetic result is a value outside the range of representable values.
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Remarks

If a value that cannot be specified in *character-string* or *numeric-character* is encountered, or if the arithmetic result is a value outside the representable range, `0` is returned to *return-value*.

Example

The following example calculates 10 ÷ 3 and outputs the remainder:

```
[SET_VALUE]
  VAL1 = 10
  VAL2 = $MOD(VAL1, 3)

  MSG = 'MOD: ' +VAL1+ ' MOD 3 = ' +VAL2
  $ECHO(MSG)
```

Execution result:

```
MOD: 10 MOD 3 = 1
```

$MUL (multiplication)

`$MUL` performs multiplication, treating character strings as numeric values, and returns the arithmetic result.

Syntax

return-value=`$MUL(`*character-string*`,`*numeric-character*`)`

Values

- *return-value*
  Specifies the name of the variable into which the arithmetic result is set. Valid results range from 0.0001 to 999,999,999,999,999 (15 digits).
- *character-string*
  Specifies a multiplicand, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`). Specified values can range from 0.0001 to 999,999,999,999,999 (15 digits).
- *numeric-character*
  Specifies a multiplier, either as a constant or a variable. A numeric value specified as a constant that includes a decimal point must be enclosed in single quotation marks (`''`). Specified values can range from 0.0001 to 999,999,999,999,999 (15 digits).

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	Indicates one of the following: • An invalid value was specified in a character string or numeric value. • The arithmetic result is a value outside the range of representable values.
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Remarks

If a value that cannot be specified in *character-string* or *numeric-character* is encountered, or if the arithmetic result is a value outside the representable range, `0` is returned to *return-value*.

Example

The following example calculates 10 × 5 and outputs the results:

```
[SET_VALUE]
  VAL1 = 10
  VAL2 = $MUL(VAL1, 5)

  MSG = 'MUL:' +VAL1+ ' * 5 = ' +VAL2
  $ECHO(MSG)
```

Execution result:

```
MUL: 10 * 5 = 50
```

$NUMBER (numbering)

`$NUMBER` uses the object class `FunctionInfo` to acquire a unique number used by the asset management system database. You can retrieve numbers by specified function ID and extend ID ranging from 1 to 4,294,967,295 (10 digits).

If 4,294,967,295 is exceeded, the function returns to 1. If the number is less than ten digits, zeros are inserted into the unfilled digit places.

Syntax

return-value=`$NUMBER(`*function-ID*`,`*extend-ID*`)`

Values

- *return-value*
  Specifies the name of the variable into which the acquired sequential number is set.
- *function-ID*
  Specifies the value of a feature specified by the object class `FunctionInfo`, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).
- *extend-ID*
  Specifies any character string of 1 to 251 bytes to further classify the feature specified by the object class `FunctionInfo`. The extend ID must be specified either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	Indicates one of the following: • Database access error. • An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Remarks

You must also register to the object class `FunctionInfo` the exclusion control lock record that corresponds to the function ID.

The following shows an example of the data file that is imported into `FunctionInfo`, with a function ID of `USER` and an extend ID of `Number`.

```
OP,CreationClassName,FunctionID,ExtendID,UpdateDate,SequenceNo
a,FunctionInfo,USER,Number,2003/1/1,0
a,FunctionInfo,USER,NumberLock,2003/1/1,0
```

Example

The following example uses the function ID `USER` and the extend ID `Number` to acquire a number (`0000000001`):

```
[SET_VALUE]
  VAL = $NUMBER('USER' , 'Number')
  MSG = 'NUMBER = ' + VAL
  $ECHO(MSG)
```

Execution result:

```
NUMBER = 0000000001
```

$SETARRAY (set value to array)

`$SETARRAY` adds information to an array.

Syntax

`$SETARRAY(`*array-name*`,`*character-string*`)`

Values

- *array-name*
  Specifies the name of the array into which a value is to be added.
- *character-string*
  Specifies the value to be added to the array variable, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Example

The following example initializes the array `ARY` and sets to it array data `ARY[1] aaa`, `ARY[2] bbb`, and `ARY[3] ccc`. In this example, the value in array number 2 of the created array variable is then updated to `ddd`, which is then acquired and output.

```
[SET_VALUE]
  $CLEARARRAY(ARY)

  $SETARRAY(ARY,'aaa')
  $SETARRAY(ARY,'bbb')
  $SETARRAY(ARY,'ccc')

  $UPDARRAY(ARY,2,'ddd')

  VAL = $GETARRAY(ARY, 2)
  MSG = 'ARY = ' + VAL
  $ECHO(MSG)
```

Execution result:

```
ARY = ddd
```

$SETARRAYBYKEY (set value to array with key)

`$SETARRAYBYKEY` adds keyed information to an array.

Syntax

`$SETARRAYBYKEY(`*array-name*`,`*key-value*`,`*character-string*`)`

Values

- *array-name*
  Specifies the variable name of the array variable into to which a value is to be added.
- *key-value*
  Specifies the key of the value that is to be added to the array variable.
- *character-string*
  Specifies the value to be added to the array variable, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Example

The following example updates the value of array number 1 in the key `MEM` from `128` to `1024`, and acquires the updated value by specifying its key:

```
[SET_VALUE]
  $CLEARARRAY(ARY)

  $SETARRAYBYKEY(ARY,'CPU','100')     # ARY[1] CPU[1]
  $SETARRAYBYKEY(ARY,'CPU','200')     # ARY[2] CPU[2]
  $SETARRAYBYKEY(ARY,'HD' ,'40')      # ARY[3] HD[1]
  $SETARRAYBYKEY(ARY,'HD' ,'20')      # ARY[4] HD[2]
  $SETARRAYBYKEY(ARY,'MEM','128')     # ARY[5] MEM[1]
  $SETARRAYBYKEY(ARY,'MEM','256')     # ARY[6] MEM[2]

  $UPDARRAYBYKEY(ARY,'MEM',1,'1024')

  VAL = $GETARRAYBYKEY(ARY,'MEM',1)
  MSG = 'ARY MEM[1] = '+VAL
  $ECHO(MSG)
```

Execution result:

```
ARY MEM[1] = 1024
```

$SETOPTION (set run options)

`$SETOPTION` sets options so that script processing will not be interrupted even if an error occurs on the script.

Syntax

`$SETOPTION('`*option-name*`',`*parameter*`)`

Values

- *option-name*
  Specifies the following option (you can specify one option per function):
  - `ErrorFlush`
    Specifies whether or not script processing is to be interrupted when an error occurs in a script.
- *parameter*
  The following parameters can be specified with the `ErrorFlush` option:
  - `0`
    Terminates script processing if an error occurs.
  - `1`
    Continues script processing even if an error occurs.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`ERROR`	An invalid value was specified for *parameter*.
`FLUSH`	Error occurred, but processing continued (when `1` is specified in *parameter*).
Script execution interrupted	Indicates one of the following: • An invalid option was specified. • An invalid argument was specified, or an error other than the above occurred.

Example

If asset number `1000000001` is already in use, this example outputs the contents of `MSG` to the log file without terminating the script:

```
[SET_VALUE]
  $SETOPTION('ErrorFlush', 1)
[APPEND]
  AssetInfo
[DATA]
  AssetInfo.AssetID = 1000000001
  AssetInfo.AssetNo = 1000000001
[SET_VALUE]
  $LOGMSG('E', MSG)
```

$SETSESSION (set session information)

`$SETSESSION` sets a character string as session information.

Syntax

`$SETSESSION(`*session-variable-name*`,`*character-string*`)`

Values

- *session-variable-name*

  Specifies a constant or a variable. A constant must be enclosed in single quotation marks (`''`). Note, however, that any session name that begins with an ampersand (`&`) cannot be registered or updated.

- *character-string*

  Specifies a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:

—: Not applicable

Example

The following example sets the value `123456789` into the session variable `SID`:

```
[SET_VALUE]
  $SETSESSION('SID','123456789')

  VAL = $GETSESSION('SID')
  MSG = 'SID = ' + VAL
  $ECHO(MSG)
```

Execution result:

```
SID = 123456789
```

$SETSTATUS (set status)

`$SETSTATUS` changes the status of a process.

Syntax

`$SETSTATUS(`*status-constant*`)`

Values

- *status-constant*

  Specifies `NORMAL` (normal end), `NODATA` (no data exists), `ERROR` (an error occurred), or `MULTI` (another user is currently updating), either as a constant or a reserved string. A constant (`NORMAL`, `ERROR`, `NODATA`, or `MULTI`) must be enclosed in single quotation marks (`''`).

Status

status-constant becomes the specified status. If an invalid status constant is specified, the script processing is interrupted.

Example

The following example searches all asset information items whose status is stock (`301`), updates their status to active (`002`), and terminates processing:

```
[ASSET_ITEM_LOOP]
  [CLASS_FIND]
    AssetInfo
  [FIND_DATA]
    (AssetInfo.AssetStatus = '301')AND
    (AssetInfo.AssetKind = '001')
  [GET_VALUE]
    ASSETID = AssetInfo.AssetID
  [UPDATE]
    AssetInfo
  [DATA]
    AssetInfo.AssetID = ASSETID
    AssetInfo.AssetStatus = '002'
  [SET_VALUE]
    $SETSTATUS('NORMAL')
[ASSET_ITEM_LOOP_END]
```

$STRCMP (compare strings)

`$STRCMP` compares two character strings.

Syntax

return-value=`$STRCMP(`*character-string-1*`,`*character-string-2*`)`

Values

- *return-value*
  Specifies the name of the variable into which the comparison results are set.
  - If *character-string-1* is smaller than *character-string-2*, `0` is returned to *return-value*.
  - If *character-string-1* is equal to *character-string-2*, `1` is returned to *return-value*.
  - If *character-string-1* is greater than *character-string-2*, `2` is returned to *return-value*.
- *character-string-1*
  Specifies a comparison character string, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).
- *character-string-2*
  Specifies a character string to compare, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Example

The following example compares the characters strings of `DATA1` and `DATA2`, and outputs `STRCMP IDENTICAL` if the strings are identical, and `STRCMP DIFFERENT (`*$STRCMP-return-value*`)` if they are not identical:

```
[SET_VALUE]
  DATA1 = 'Asset Console1'
  DATA2 = 'Asset Console2'
  VAL = $STRCMP(DATA1,DATA2)
[IF]
  VAL = 1
  [THEN]
    [SET_VALUE]
      MSG  = 'STRCMP IDENTICAL'
      $ECHO(MSG)
  [ELSE]
    [SET_VALUE]
      MSG  = 'STRCMP DIFFERENT ('+VAL+')'
      $ECHO(MSG)
[IF_END]
```

Execution result:

```
STRCMP DIFFERENT (0)
```

$SUB (subtraction)

`$SUB` performs subtraction, treating character strings as numeric values, and returns the arithmetic result.

Syntax

return-value=`$SUB(`*character-string*`,`*numeric-character*`)`

Values

- *return-value*

  Specifies the name of the variable into which the arithmetic result is set. Valid results range from 0.0001 to 999,999,999,999,999 (15 digits).

- *character-string*

  Specifies a minuend, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`). Specified values can range from 0.0001 to 999,999,999,999,999 (15 digits).

- *numeric-character*

  Specifies a subtrahend, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`). Specified values can range from 0.0001 to 999,999,999,999,999 (15 digits).

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	Indicates one of the following: • An invalid value was specified in a character string or numeric value. • The arithmetic result is a value outside the range of representable values.
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:

—: Not applicable

Remarks

If a value that cannot be specified in *character-string* or *numeric-character* is encountered, or if the arithmetic result is a value outside the representable range, `0` is returned to *return-value*.

Example

The following example calculates 10 - 5 and outputs the result:

```
[SET_VALUE]
  VAL1 = 10
  VAL2 = $SUB(VAL1, 5)

  MSG = 'SUB:' +VAL1+ ' - 5 = ' +VAL2
  $ECHO(MSG)
```

Execution result:

```
SUB: 10 - 5 = 5
```

$SUBSTR (get substrings)

`$SUBSTR` extracts a portion of a character string from a specified character string as defined by the extraction start position and the length of the character substring being extracted.

Syntax

return-value=`$SUBSTR(`*base-character-string*`,`*extraction-start-position*`,`*length-of-extracted-string*`)`

Values

- *return-value*

  Specifies the name of the variable into which the extracted substring is set.
- *base-character-string*

  Specifies the character string from which data is to be extracted, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).
- *extraction-start-position*

  Specifies the position at which extraction starts, assuming the first character in the base character string is `0`, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

  Specify an extraction start position value of `0` or greater. If a extraction start position that does not exist is specified, a 0-byte character string is returned.
- *length-of-extracted-string*

  Specifies the length of a character string to be extracted (in bytes), as either a constant or a variable. If you specify a length that exceeds the end point of the base character string, only the number of characters to the end of the base character string are extracted. To unconditionally extract the entire character string from the extraction beginning position to the final character, specify a negative value.

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	No character string existed at the specified extraction start position.
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:

—: Not applicable

Example

The following example sets the character string `ABCDEFG` into `NAME`, and then extracts `ABC` from `NAME` and sets it into `VALUE`:

```
[SET_VALUE]
  NAME = 'ABCDEFG'
  VALUE=$SUBSTR(NAME,0,3)
  $ECHO(VALUE)
```

Execution result:

```
ABC
```

$TOKEN (get token)

`$TOKEN` extracts a token from a specified character string as defined by the position of the token to be extracted and a separator character.

Syntax

```
return-value=$TOKEN(base-character-string,extraction-token-
position,separator-character)
```

Values

- *return-value*

  Specifies the name of the variable into which the extracted token is set.

- *base-character-string*

  Specifies the character string from which the token is extracted, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

- *extraction-token-position*

  Specifies the position of the token to be extracted, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`). Specify the sequential number of the token to be extracted, assuming that the first token indicated by the separator character is `0`. For example, with respect to the character string `aaa,bbb,ccc,ddd`, the characters extracted (`aaa`) as delimited by the separator character become the token. In this example, `0` to `3` correspond to the positions of the following token.

  `0`: `aaa`

  `1`: `bbb`

  `2`: `ccc`

  `3`: `ddd`

  If an invalid token position or an out-of-range extraction is specified, a 0-byte character string is returned.

- *separator-character*

  Specifies the character to be used to separate tokens, either as a constant or a variable. Usable characters are alphanumeric characters and symbols. A constant must be enclosed in single quotation marks (`''`).

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	The value specified by *base-character-string* was not found at the position specified by *extraction-token-position*.
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:

—: Not applicable

Example

The following example sets the character string `ABC/DEF/GHI` into `NAME`, and then extracts `DEF` from `NAME` and sets it into `VALUE`:

```
[SET_VALUE]
  DATA = 'ABC/DEF/GHI'
  VAL = $TOKEN(DATA,1,'/')
  MSG = 'TOKEN = ' + VAL
  $ECHO(MSG)
```

Execution result:

```
TOKEN = DEF
```

$UPDARRAY (update array data)

`$UPDARRAY` updates a value in an array element of an array variable. The array element is specified with an array number.

Syntax

`$UPDARRAY(`*array-name*`,`*array-number*`,`*character-string*`)`

Values

- *array-name*

  Specifies the name of an array in which data is to be updated.
- *array-number*

  Specifies the array number of the array element to be updated, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`). Specified values can range from 1 to 2,147,483,647.
- *character-string*

  Specifies the value to be updated, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	No information exists for the specified array number.
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:

—: Not applicable

Example

See the example for *$SETARRAY (set value to array)*.

$UPDARRAYBYKEY (update value of array with key)

`$UPDARRAYBYKEY` updates the value of an array element of an array variable. The array element is specified using a key and the array number in the key.

Syntax

`$UPDARRAYBYKEY(`*array-name*`,`*key-value*`(,`*array-number-in-key*`),`*character-string*`)`

Values

- *array-name*
  Specifies the name of the array in which a value is to be updated.
- *key-value*
  Specifies the key value of the array element to be updated, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).
- *array-number-in-key*
  Specifies the array number in the key when there are multiple data values that correspond to the key to be updated, as a numeric character, a constant, or a variable. A constant must be enclosed in single quotation marks (`''`). Specified values can range from 1 to 2,147,483,647.
  You can omit this value. If it is omitted, the default is `1`.
- *character-string*
  Specifies the value to be updated, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	Indicates one of the following: • The specified key does not exist. • The specified array number in the key does not exist.
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:
—: Not applicable

Example

See the example in *$SETARRAYBYKEY (set value to array with key)*.

$UPPER (convert string)

`$UPPER` converts the alphabetic characters specified in the argument character string to upper case.

Syntax

return-value=`$UPPER(`*character-string*`)`

Values

- *return-value*

  Specifies the name of the variable into which the character string that has been converted to upper case is set.
- *character-string*

  Specifies the character string to be converted, either as a constant or a variable. A constant must be enclosed in single quotation marks (`''`).

Status

The following table lists and describes the possible statuses:

Status	Description
`NORMAL`	Normal end
`NODATA`	—
`ERROR`	—
Script execution interrupted	An invalid argument was specified, or an error other than the above occurred.

Legend:

—: Not applicable

Example

The following example converts the contents of the variable `NAME` to upper case, and outputs the result:

```
[SET_VALUE]
  NAME = 'computer name'
  VAL = $UPPER(NAME)
  MSG = 'UPPER = ' + VAL
  $ECHO(MSG)
```

Execution result:

```
UPPER = COMPUTER NAME
```

Appendix

A. Version Changes

A.1 Changes in version 11-50

- None.

A.2 Changes in version 11-10

- Windows Server 2016 was added as an applicable operating system for the following products:
  - JP1/IT Desktop Management 2 - Manager
  - JP1/IT Desktop Management 2 - Agent
  - JP1/IT Desktop Management 2 - Network Monitor
  - JP1/IT Desktop Management 2 - Asset Console
  - Remote Install Manager

A.3 Changes in version 11-01

- Windows 10 was added as an applicable operating system for JP1/IT Desktop Management 2 - Network Monitor.

A.4 Changes in version 11-00

- Windows 10 was added as a supported operating system for the following products:
  - JP1/IT Desktop Management 2 - Agent
  - JP1/IT Desktop Management 2 - RC Manager
  - Remote Install Manager
- Windows Server 2003 and Windows Server 2008 (excluding Windows Server 2008 R2) are no longer supported by the following products:
  - JP1/IT Desktop Management 2 - Manager
  - JP1/IT Desktop Management 2 - Agent
  - JP1/IT Desktop Management 2 - Network Monitor
  - JP1/IT Desktop Management 2 - RC Manager

B. Reference Material for This Manual

B.1 Related publications

This manual is part of a related set of manuals. The manuals in the set are listed below (with the manual numbers):

- *JP1 Version 11 Asset and Distribution Management: Getting Started* (3021-3-B51(E))
- *JP1 Version 11 JP1/IT Desktop Management 2 Overview and System Design Guide* (3021-3-B52(E))
- *JP1 Version 11 JP1/IT Desktop Management 2 Configuration Guide* (3021-3-B53(E))
- *JP1 Version 11 JP1/IT Desktop Management 2 Administration Guide* (3021-3-B54(E))
- *JP1 Version 11 JP1/IT Desktop Management 2 Distribution Function Administration Guide* (3021-3-B55(E))
- *JP1 Version 11 JP1/IT Desktop Management 2 - Asset Console Configuration and Administration Guide* (3021-3-B56(E))
- *JP1 Version 11 JP1/IT Desktop Management 2 Messages* (3021-3-B58(E))

B.2 Conventions: Abbreviations for product names

This manual uses the following abbreviations for product names:

Abbreviation			Full name or meaning
Asset Console			JP1/IT Desktop Management 2 - Asset Console
Windows 7#1	Windows 7 Enterprise		Microsoft(R) Windows(R) 7 Enterprise
	Windows 7 Home Premium		Microsoft(R) Windows(R) 7 Home Premium
	Windows 7 Professional		Microsoft(R) Windows(R) 7 Professional
	Windows 7 Starter		Microsoft(R) Windows(R) 7 Starter
	Windows 7 Ultimate		Microsoft(R) Windows(R) 7 Ultimate
Windows 8#1	Windows 8		Windows(R) 8
	Windows 8 Enterprise		Windows(R) 8 Enterprise
	Windows 8 Pro		Windows(R) 8 Pro
Windows 8.1#1	Windows 8.1		Windows(R) 8.1
	Windows 8.1 Enterprise		Windows(R) 8.1 Enterprise
	Windows 8.1 Pro		Windows(R) 8.1 Pro
Windows 10#1	Windows 10 Enterprise		Windows(R) 10 Enterprise
	Windows 10 Pro		Windows(R) 10 Pro
Windows Server 2008 R2#1, #2			Microsoft(R) Windows Server(R) 2008 R2 Datacenter
			Microsoft(R) Windows Server(R) 2008 R2 Enterprise
			Microsoft(R) Windows Server(R) 2008 R2 Standard
Windows Server 2012#1, #3	Windows Server 2012#3	Windows Server 2012 Datacenter	Microsoft(R) Windows Server(R) 2012 Datacenter

Abbreviation			Full name or meaning
Windows Server 2012[#1, #3]	Windows Server 2012[#3]	Windows Server 2012 Standard	Microsoft(R) Windows Server(R) 2012 Standard
	Windows Server 2012 R2		Microsoft(R) Windows Server(R) 2012 R2 Datacenter
			Microsoft(R) Windows Server(R) 2012 R2 Standard
Windows Server 2016[#1]	Windows Server 2016 Datacenter		Microsoft(R) Windows Server(R) 2016 Datacenter
	Windows Server 2016 Standard		Microsoft(R) Windows Server(R) 2016 Standard

#1

If there are no functional differences among OSs, *Windows* is used generically when referring to Windows 7, Windows 8, Windows 8.1, Windows 10, Windows Server 2008 R2, Windows Server 2012, or Windows Server 2016.

#2

Does not include Server Core installation.

#3

If Windows Server 2012 R2 is noted alongside Windows Server 2012, the description for Windows Server 2012 does not apply to Windows Server 2012 R2.

B.3 Conventions: Acronyms

This manual also uses the following acronyms:

Acronym	Full name or meaning
CPU	Central Processing Unit
CSV	Comma Separated Value
DLL	Dynamic Linking Library
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
ID	IDentifier
IP	Internet Protocol
IT	Information Technology
JIS	Japanese Industrial Standards
LDAP	Lightweight Directory Access Protocol
OS	Operating System
PC	Personal Computer
RDB	Relational Database
TCP/IP	Transmission Control Protocol/Internet Protocol
URL	Uniform Resource Locator
WS	Work Station
WWW	World Wide Web

B.4 Format used in this manual

(1) Conventions: Fonts and symbols

The following table lists the general font conventions:

Font	Convention
Bold	Bold type indicates text on a window, other than the window title. Such text includes menus, menu options, buttons, radio box options, or explanatory labels. For example, bold is used in sentences such as the following: • From the **File** menu, choose **Open**. • Click the **Cancel** button. • In the **Enter name** entry box, type your name.
Italics	Italics are used to indicate a placeholder for some actual text provided by the user or system. Italics are also used for emphasis. For example: • Write the command as follows: `copy` *source-file target-file* • Do *not* delete the configuration file.
`Code font`	A code font indicates text that the user enters without change, or text (such as messages) output by the system. For example: • At the prompt, enter `dir`. • Use the `send` command to send mail. • The following message is displayed: `The password is incorrect.`

Examples of coding and messages appear as follows (although there may be some exceptions, such as when coding is included in a diagram):

```
MakeDatabase
...
StoreDatabase temp DB32
```

In examples of coding, an ellipsis (...) indicates that one or more lines of coding are not shown for purposes of brevity.

(2) Conventions in syntax explanations for commands and scripts

Syntax definitions appear as follows:

```
StoreDatabase [temp|perm] (database-name ...)
```

The following table lists the conventions used in syntax explanations:

Example font or symbol	Convention
`StoreDatabase`	Code-font characters must be entered exactly as shown.
database-name	This font style marks a placeholder that indicates where appropriate characters are to be entered in an actual command.
`SD`	Bold code-font characters indicate the abbreviation for a command.
<u>`perm`</u>	Underlined characters indicate the default value.
`[ ]`	Square brackets enclose an item or set of items whose specification is optional.

Example font or symbol	Convention
|	Only one of the options separated by a vertical bar can be specified at the same time.
`...`	An ellipsis (...) indicates that the item or items enclosed in ( ) or [ ] immediately preceding the ellipsis may be specified as many times as necessary.
`()`	Parentheses indicate the range of items to which the vertical bar (|) or ellipsis (...) is applicable.

(3) Conventions for permitted characters

The following table lists characters that are permitted as syntax elements (values that can be specified by users):

Type	Definition
Alphabetic characters	`A` to `Z`, `a` to `z`
Upper-case alphabetic characters	`A` to `Z`
Lower-case alphabetic characters	`a` to `z`
Numeric characters	`0` to `9`
Alphanumeric characters	`A` to `Z`, `a` to `z`, `0` to `9`
Symbols	! " # $ % & ' ( ) + , - . / : ; < = > @ [ ] ^ _ { } ~ ? space

B.5 Conventions: Version numbers

The version numbers of Hitachi program products are usually written as two sets of two digits each, separated by a hyphen. For example:

- Version 1.00 (or 1.0) is written as 01-00.
- Version 2.05 is written as 02-05.
- Version 2.50 (or 2.5) is written as 02-50.
- Version 12.25 is written as 12-25.

The version number might be shown on the spine of a manual as *Ver. 2.00*, but the same version number would be written in the program as *02-00*.

B.6 About online help

JP1/IT Desktop Management 2 provides online help in relation to the following subjects:

Window descriptions

This help describes how to use the screen that is currently displayed. You can view these help topics by clicking the **Help** button in the user interface.

B.7 Conventions: KB, MB, GB, and TB

This manual uses the following conventions:

- 1 KB (kilobyte) is 1,024 bytes.
- 1 MB (megabyte) is $1{,}024^2$ bytes.
- 1 GB (gigabyte) is $1{,}024^3$ bytes.
- 1 TB (terabyte) is $1{,}024^4$ bytes.

Index

Hitachi, Ltd.

6-6, Marunouchi 1-chome, Chiyoda-ku, Tokyo, 100–8280 Japan