TIBCO iWay® Service Manager

Functional Language Reference Guide

Version 8.0 and Higher
March 2021
DN3502113.0321

Contents

Chapter 1

iWay Functions

iWay Service Manager (iSM) provides many functions that support dynamic configuration and logical/dynamic conditional routing. You can use these functions in any expression (simple or complex) that you need to develop.

In this chapter:

iWay Functions Overview

iWay functions enable you to examine the context in which a message is processed and the attributes of that message. You can use the result to determine the subsequent processing steps for the message. You can select, bypass, or modify the steps in the defined process.

For example, you can examine an XML message with _XPATH() to extract specific elements of the message that serve as execution parameters. You can check for a schema error, in which case you might route the message differently.

Conditional routing uses functions that either return a Boolean value (true or false) or can, as a result of a relationship test, result in a Boolean condition. If the result is true, the step governed by the conditional routing expression is performed. Otherwise, it is bypassed. For example, if you use an emitter only if a document is in error, you might use the _ISERROR() conditional routing expression.

Runtime Functions

iWay Service Manager (iSM) supports a large number of runtime functions that can be used to make routing decisions and supply configuration parameters to iWay components, such as services and listeners. These functions can be combined into expressions, which consist of one or more of the functions that are discussed in this chapter.

Functions fall into several categories:

- Functions that reference the system environment.
- Functions that reference the server's operational context.
- Functions that reference the current document flowing through the server.

Syntax and Usage

Functions are written using the following common computer language format:

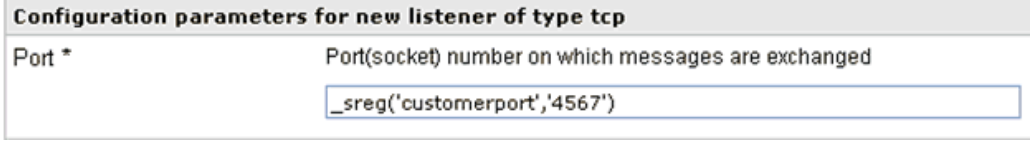
```
_name(parameters)
```

Functions return strings that can be used directly or tested for routing purposes.

For example, the following function returns the value of a specific special register (context value):

```
_sreg(<name>,<default>)
```

This value can be used directly as a configuration parameter. For example:



This value can also be tested in a route, such as:

```
_sreg('iway.config') yields 'base'
```

More sophisticated checking methods, including mathematical conditions, existence and case insensitive compares, are available through the following special function:

```
_cond(<value>,operator,<operand>)
```

For more information on how to use and configure this function, see *_cond(): Perform Conditional Test* on page 118.

Parameters can also include literals, which are sequences of characters, usually enclosed in single quotes. A quote character can be included by escaping it, such as:

```
_sreg('playwrite','O\'Casey')
```

The compiler attempts to recognize literals, avoiding the need to use the enclosing quotes. However iWay strongly recommends that functions be written using the enclosing quotes.

All iWay functions begin with the underscore character. Some commonly used functions such as _sreg(), _xpath(), _ldap(), and _file() can often be written without the leading underscore. Using the underscore helps prevent ambiguity and is strongly recommended.

Functions can be combined in literal statements. For example, a data path can be:

```
_sreg('iway.home')/_sreg('iway.config')/etc/data
```

which will be evaluated to the data subdirectory in the etc subdirectory of the current configuration.

Simple mathematical operations can also be performed by using functions. Supported operations include add, subtract, multiply, divide, and modulus. For example, assume that the special register timeout holds a number of seconds that you want to use in a parameter that requires milliseconds. Specifying the following accomplishes the needed conversion:

```
_mul(_sreg('timeout',0),1000)
```

To divide a timeout period in half, use the following:

```
_div(_sreg('timeout',0),2)
```

Division can be accomplished faster by multiplying by the reciprocal of the divisor. For example, to obtain one half of the timeout value in the above example, use the following to yield the desired result:

```
_mul(_sreg('timeout',0),0.5)
```

The _div and _mul functions are used in place of the traditional slash and asterisk to avoid confusion in file names and unique patterns.

Integer math, especially suitable for date arithmetic functions such as adding a duration to a base date, are available through the _iadd, _isub, _imul, and _idiv functions. Integer math is supported to 16 places of precision. For more information on date arithmetic, see *_dateof(): Return the Timestamp for a Passed Time* on page 67.

Functions that return Boolean values, such as _isroot() can be combined with AND and OR conjunctions. For example, the following expression returns as *true* if the current root element name of the document is either a or b:

```
_isroot('a') or _isroot('b')
```

A more complicated test might be:

```
_isroot('a') or sreg('a')==55
```

Unary not (!) is supported for tests. For example, if a document is in error, !_iserror() will return as false. The not can be used in combination with other tests, such as _isxml() and !iserror() combines two tests with a logical and condition.

Example: Routing an Output Document

The following example uses the _COND() function to route an output document if it is larger than 1000 characters:

```
_COND(_LENGTH(_FLATOF()),GT,1000)
```

Example: Testing the Result of an Attribute

The following example uses the _ALL() function to test the result of each attribute returned from an XPath expression and to determine if the document is in an error state. If the attribute, ABC, is either TOM or HARRY, or the document is in an error state, routing occurs.

```
_ALL(_XPATH(//XX/@ABC), EQ, _OR("TOM", "HARRY")) OR _ISERROR()
```

Note: When preceded without an underscore character, OR is used as a predicate.

Parameter Evaluation

Parameters are evaluated in the standard manner, so that any operators in the parameters are compiled. Evaluation of math symbols can be avoided by enclosing the constant parameters in literal quotes. However, for reasons of simplicity of upward compatibility, this is not often performed. As a convenience, certain functions for which math evaluation of parameters makes little sense do not apply such compilation to their parameters. These include the following:

- _sreg()
- _atthdr() and _atthdric()
- _concat()
- _lit()
- _sql()

- _xml()
- _xpath(), _xpath1(), and _iwxpath()
- _exist(), and _iwexist()
- _base64, _frombase64(), _encode64(), and _decode64()

Conjunctions

The AND and OR conjunctions allow you to compose expressions that include multiple phrases. For example:

```
_if(_sreg('a') < 15 and sreg('b') = 'hello', 'true', 'false')
```

AND and OR conjunctions are used only in COND(), ALL(), and ANY() functions, where a list of comparands are being evaluated. For example, consider the following sample document:

```
<root>
<child>4</child>
<child>2.3</child>
<child>1</child>
</root>
```

The following expression returns as *true*:

```
_ANY(_XPATH(//child),EQ,_OR(5.5,1,7,8))
```

Programmers refer to this type of comparison as a *lazy or*. It applies only to the _COND(), _ALL(), and _ANY() functions.

The Functions

Each of the available functions is discussed, along with comments on their syntax and common purpose. Some functions are available only in specific server configurations or with optional components installed. Where such is the case this is stated in the Function Support Table.

Additional functions can be written and included in the function repertory by following the instructions in the *iWay Service Manager Programmer's Guide*.

The function categories are often arbitrary and are used only for description purposes. The richness of the function may enable it to be used for several purposes. For example, the _sreg() function returns the value of a context special register. Some registers are defined during configuration and some are only defined as a specific document passes through the system.

Automatic Concatenation

The interpreter attempts to construct values by evaluating the entire string for iFL. It will concatenate the value returned by an evaluated iFL at the point of the function. For example, if the special register partname has a value of xyz, then the string c:/w_sreg(partname)/file.txt will be evaluated to c:/wxyz/file.txt.

In some cases, automatic concatenation can lead to unexpected results. For example, suppose you have a value myfile(fname). Evaluating this can result in an attempt to read load the contents of the fname file, which will result in an error. This can be avoided by treating the statement as a literal and having iFL evaluate it as such. One solution is to place the text in a _concat() function. For example:

```
_concat('myfile(fname)')
```

In this case, the desired value is returned. The literal marks (in quotes) prevent the evaluation of the literal itself.

Environmental Functions

Environmental functions return information about or from the environment in which the document is being processed.

_sreg(): Lookup a Special Register

The _sreg() function uses the following format:

```
_sreg(name [,default])
```

name	string	Name of the register
default	string	Default value

Special registers contain context information. The information can be configured on the runtime system, reflect the processing state of the current message, or be explicitly set by services in a process flow.

Special registers exist in scopes, and are looked up from the local scope to the outermost scope (the system context). The value returned is that of the nearest scope in which the register is found. If the register is not found in any scope, the default is returned. If the default is not specified, null is returned.

The _sreg() function searches in a case sensitive manner. Consider the following examples:

Register Name	Value	_sreg('a','def1')
a	hello	hello
A	hello	defl

_property(): Retrieve a Value from a Java Property Object File

The _property() function loads the value for the desired property from a Java properties object file. It uses the following format:

```
_property(file, attribute [, default [,control] [,evaluate]] )
```

file	file name	Path to the properties file.
attribute	string	Name of the desired property.
default	string	Default if property does not exist.
control	string	Keyword to control operation: ❑ **check.** Check for modification. ❑ **keep.** Do not check for modification.
evaluate	keyword	The keyword to control the result evaluation: ❑ **evaluate.** Evaluate the result as iFL. Allows iFL to be held in a property value. ❑ **constant.** Do not evaluate. (default)

Load the value for the desired property from a Java properties object file. The check control option causes the properties file modification timestamp to be examined on each request. This is to determine whether the file has changed since the last load. If so, the properties file is reloaded. The *keep* control option prevents this check for cases in which it is known that changes will not be made or should not be loaded. Avoiding the check can result in file read time savings. The *keep* option is the default.

For example, assume file test.properties contains:

```
one=first
two=second
```

The following function call causes the value *first* to be returned:

```
_property('test','one','notfound','check')
```

Next, change the properties file to the following:

```
one=next
two=second
```

In this case, the following function call returns next:

```
_property('test','one','notfound','check')
```

If the control option is omitted or *keep* is used, the following function call returns first as before:

```
_property('test','one','notfound','keep')
```

The value of the property can also by stored using the Advanced Encryption Standard (AES), by configuring the *set property* command. For more information on using the *set property* command, see the *iWay Service Manager User's Guide*.

The value can be decrypted during a read using the _aes() function applied to the result of the _property() function. For example:

```
_aes('decrypt',_sreg('mykey'),_property('file',key'))
```

_propertymatch(): Match a String Against a File of Regular Expression Patterns

A properties file contains *key=value* pairs consisting of a regular expression and a value. An input string is matched against the patterns in the properties file, and the value associated with the first pattern to match the input is returned.

The properties file is loaded once, cached for the channel, and is not reloaded for each use. If any changes are made in the properties file, the channel must be restarted to reflect these changes.

The _propertymatch() function uses the following format:

```
_propertymatch(file,input,[,default [,control [,encoding]]]
```

file	path	Path to the properties file. The suffix is optional.
input	string	The candidate to be matched against the regular expressions in the properties file.

default	string	The value to be returned if none of the patterns are matched by the candidate input string.
control	keyword	Keyword to control operation: ❑ **check.** Check for modification. ❑ **keep.** Do not check for modification.
encoding	string	IANA encoding of the properties file.

The properties file consists of one or more regular expressions (keys), each with an associated value. Standard properties file comments (lines starting with #) and blank lines are allowed. Continuation lines are not supported.

The *check* control option causes the properties file modification timestamp to be examined on each request to determine whether the file has changed since the last load, and if so, the properties file is reloaded. The *keep* control option prevents this check for cases in which it is known that changes will not be made or should not be loaded. Avoiding the check can result in file read time savings. The *keep* option is the default.

In the following example, an input ZIP code must be matched to select an appropriate subflow to handle the location. The file stored at */appdata/zip.properties* might be structured as follows:

```
# zip code flow selection file
00.*=zip00
01.*=zip01
10.*=zipnyc
20500=zipwhitehouse
20.*=zipwashdc
```

Since the patterns are matched in the order specified in the file, the ZIP code for the White House will match before the general Washington, D.C. ZIP codes are matched.

Assume that the ZIP code is located in an input document and stored in a Special Register (SREG) called INZIP. The following function call does the matching:

```
_propertymatch('/appdata.zip',sreg(INZIP),'zipother')
```

If the INZIP SREG holds 10121, then the function returns *zipnyc*. If the INZIP SREG holds 11570, then the function returns *zipother*.

The value of the property can also by stored using the Advanced Encryption Standard (AES), by configuring the *set property* command. For more information on using the *set property* command, see the *iWay Service Manager User's Guide*.

The value can be decrypted during a read using the _aes() function applied to the result of the _propertymatch() function. For example:

```
_aes('decrypt',_sreg('mykey'),_propertymatch('file',key'))
```

_inlist(): Check Value in a List

The _inlist() function checks whether a key value is in a list. This function uses the following format:

```
_inlist(list, key [,control])
```

list	string or path	Source of the list, which is under control of the *control* operand. This can either be the path to the file containing the list values, or a string directly in the function.
key	string	The value to be checked.
control	keyword	Specifies how the list is interpreted. The following controls are supported: ❑ **keep. (default)** The list is a file path. Do not check for modification. ❑ **check.** The list is a file path. Check for modification. ❑ **string.** The list is a direct list.

The _inlist() function facilitates checking whether a value is in a list of valid values (for example, a set of medical codes). If the control is set to *string*, then the list is direct (for example, 1987,4567,3334). The *check* control option causes the list file modification timestamp to be examined on each request. This is to determine whether the file has changed since the last load. If this is the case, then the list is reloaded. The *keep* control option prevents this check for cases where it is known that changes will not be made or should not be loaded. Avoiding the check can decrease file read times. The *keep* control option is set by default. For more detailed examples using the *keep* and *check* control options, see *_property(): Retrieve a Value from a Java Property Object File* on page 15 and *_propertymatch(): Match a String Against a File of Regular Expression Patterns* on page 16.

If a file is being used, then it takes the form of an iWay list file consisting of single tokens delineated by a separator or an end of line. A line that starts with a hash character (#) is considered to be a comment and is ignored. For example, listfile.txt may look like the following example:

```
# example list file, first lines have one token each
1234
2345
# now a line with multiple entries
5678,8764
```

Examples:

The following _inlist() function evaluates as *true*:

```
_inlist(listfile.txt,2345,keep)
```

The following _inlist() function evaluates as *false*:

```
_inlist("washington,adams,jefferson","cohen",string)
```

_setreg(): Set a Special Register

The _setreg() function sets the specified Special Register (SREG) to the value that is entered and returns the previous value, if any. This function uses the following format:

```
_setreg(name,value[,type [,scope [,action] ]])
```

name	string	The name of the register.
value	string	The value to be set.
type	keyword	The register type. The following types are supported: ❑ **user (default).** The user register, which is a simple value. ❑ **doc.** The document-related value. ❑ **hdr.** The header value, which is serialized by appropriate protocols when emitted. ❑ **delete.** Eliminates the specified register.

scope	keyword	The scope of the specified register. The query on the former value is performed at this scope. The following scopes are supported: ❏ **local (default).** The scope is the local register context. ❏ **flow.** The scope is the head of the flow. ❏ **message.** The scope is the message.
action	keyword	Determines how to interpret the value operand. The following actions are supported: ❏ **string (default).** The value is a string. ❏ **parse.** Deprecated. ❏ **json.** The string is parsed to a JSON object. ❏ **xml.** The string is parsed into an XML tree.

Use caution when using the _setreg() function, since this function can change the contents of the local special register manager.

The *action* operand controls how the value is handled. Normally, it is a string and inserted into the register. If set to *xml*, then the string is treated as a flat XML tree, which is parsed and set into the register. The register can be used in an _xpath() function to allow XPath evaluation of the XML.

Similarly, *json* sets the string into JSON form, making it available for the _jsonpath() and _jsonptr() functions.

The parsed object can also be set into the current document by using the _restoredoc() function.

The selected scope must be appropriate to the context in which the function is used. For example, a flow scope is not present in non-flow contexts.

When deleting a register, the delete takes place at the identified scope. Registers are normally looked up by the nearest scope, so the register may continue to exist at lower scopes. This function can be used effectively as an operand of the _if() function.

If you need to set multiple registers in parallel, you can consider using the _concat() function. The following example sets registers *ra* and *rb* if register *rt* is true:

```
_if(sreg('rt')='true',_concat(_setreg('ra','1'),_setreg('rb','2'))
```

If you are setting multiple registers using this technique, then you may need to consider the _lock() function as a part of the setting clause. For example:

```
_lock('lock1',_concat(,_concat(_setreg('ra','1'),_setreg('rb','2')))
```

The value parameter is not recommended to be a literal value, but rather a function to obtain the value, such as _xpath() or _jdbc().

The _setreg() function can also be used to increment a counter. The counter is the value of a register defined above the update. For example, if a counter for each message handled by the channel is required, then use a register at the channel level and update it in each worker. Reviewing the _lock() function is recommended, which implements a standard software technique for ensuring the integrity of the counter. An example is also provided in the _lock() function topic. For more information, see *_lock(): Obtain Value Under Lock* on page 124.

Document Functions

Document functions access information pertaining to the current document being processed. A document contains a payload and context information about the payload, such as type, encoding, and whether errors have been previously detected. Parsed payloads include XML and JSON.

Document functions require a current document for execution. Consequently, document functions apply only to channel components, such as preparsers and process flow services. They do not apply to the configuration of server components, such as resource providers or listeners, as these components are configured before any document is passed through. Other iWay Functional Language (iFL) functions, such as _property(), are applicable to this usage scenario, as they do not require a document.

_docinfo(): Information About the Current Document

The _docinfo() function returns information about the current document (message) that is passing through the process flow. Some document information can also be obtained using other functions, such as _iserror(). The _docinfo() function uses the following format:

```
_docinfo(type)
```

type	String	Determines what information is required from the document. The following types are supported: ❑ **encoding.** Returns the current encoding that is being used for the message. If the document arrived with a declared encoding, then that encoding is returned. Otherwise, the default encoding of the channel is returned. ❑ **format.** Type of the current payload. Can be *empty*, *flat*, *bytes*, *xml* or *json*. ❑ **xmlversion.** The XML version of the document, which is applicable to XML only. Returns *1.0* or *1.1*. ❑ **suffix.** The *type* of the current document. When used as a file suffix, this often triggers a visualize association with the data type (for example, pdf). The default, if not set deliberately, is the file type of the loaded payload format. ❑ xml –> xml ❑ Json –> json ❑ lines –> txt ❑ bytes –> dat

Example 1

In the following example, the message being received in an arbitrary encoding is to be converted to Base64 encoding:

```
_base64(_docinfo('encoding'))
```

Example 2

In a process flow, a File Read Service (com.ibi.agents.XDFileReadAgent) loads a PDF as a bytes file. The Set Document State Service (com.ibi.agents.XDDocAgent) is used to assign a suffix of `pdf`. The file emit (either in the process flow or as an emit following the process flow) might use the following output name:

```
/myfiles/file1._docinfo('suffix')
```

A file named `file1.pdf` is returned as the output.

_isroot(): Tests Element Root

The _isroot() function returns true if the root of the current document matches the parameter. It uses the following format:

```
_isroot(name)
```

name	string	Name to be compared to the root element

The _isroot() function is often used to drive a routing decision, such as _root('a') or _root('b') to cause the route to be selected for any document with a root of 'a' or 'b'.

_root(): Returns the Root Element Name

The _root() function returns the root element name of the current document. It uses the following format:

```
_root()
```

This is equivalent to the _xpath function _xpath(/*/name()). It might be used in a routing decision, such as _cond(_root(),eqc,'a') which selects the route for documents with roots of 'a' or 'A'.

_isxml(): Test for Parsed XML Content

The current document can hold parsed or non-parsed (flat) information. The _isxml function returns *true* if the current document holds parsed XML. If the document has no contents, then this function returns *false*. The _isxml function uses the following format:

```
_isxml()
```

_isjson(): Test for Parsed JSON Content

The current document can hold parsed or non-parsed (flat) information. The _isjson function returns *true* if the current document holds parsed JavaScript Object Notation (JSON). If the document has no contents, then this function returns *false*. The _isjson function uses the following format:

```
_isjson()
```

_isflat(): Test for Non-Parsed Content

The current document can hold parsed or non-parsed (flat) information. The _isflat function returns *true* if the current document holds non-parsed data. If the document has no contents, then this function returns *false*. The _isflat function uses the following format:

```
_isflat()
```

_iserror(): Is the Document in Error State?

The current document can have errors posted to it by the system, or can be set into an error state by the actions of an application process. If the document is in error state, default replies are delivered to the configured error addresses, otherwise they are delivered to the configured reply addresses. The _iserror() function returns true if the document is in error state. It uses the following format:

```
_iserror()
```

_hasruleerr(): Test for Rule Violations

As the document passes through its execution route, a supplied rules-based validation system can be employed. Often the rules relate to eBusiness documents such as SWIFT or EDI. If the rules system has posted a validation error to this document, the _hasruleerr() returns true. It uses the following format:

```
_hasruleerr()
```

_hasschemaerr(): Test for Schema Rule Violations

Certain rules and services can detect schema violations during the life of a document in the system. Whether such a violation has been detected can be tested with this function. A common use of this function is in a process flow test object. This function uses the following format:

```
_hasschemaerr()
```

_iswellformed(): Test for Valid Format

The _iswellformed() function uses the following format:

```
_iswellformed([format])
```

format	boolean	The intended form: ❑ xml (default) ❑ json

The current document can hold parsed or non-parsed information. This function returns *true* if the current document holds flat information that can be parsed into XML or JSON. If it has not been parsed, then a *true* response from this function assures that the message can be parsed. If the message already holds parsed information, then this function returns *true*.

_iseos(): Is Document at End of Stream

Streamable input is used for handling large documents or documents for which the application desires to split the input into sections under the same transaction. Following the completion of the input stream handling, a final pass is made with a special document containing batch information:

```
<batch count='n'/>
```

The end of stream can also be tested with the _iseos() function. A common use of this function is in a process flow test object. This function uses the following format:

```
_iseos()
```

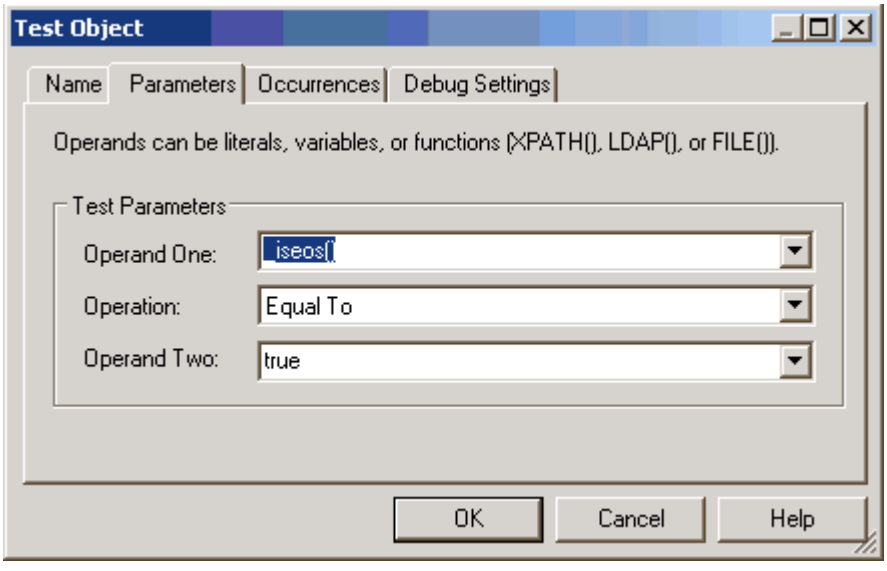
Creating a Process Flow to Test for an End of Stream Document

Splitting and streaming or splitting preparsers generate a special control document at the end of a batch to indicate that this batch was completely processed. This End of Stream document is a non-data XML document that is passed into a channel after all data documents. The _iseos() function will return as true if the current document contains an End of Stream message, else it will return as false. The End of Stream document may also contain additional statistics about the splitting process, such as how many iWay documents were generated from the batch.

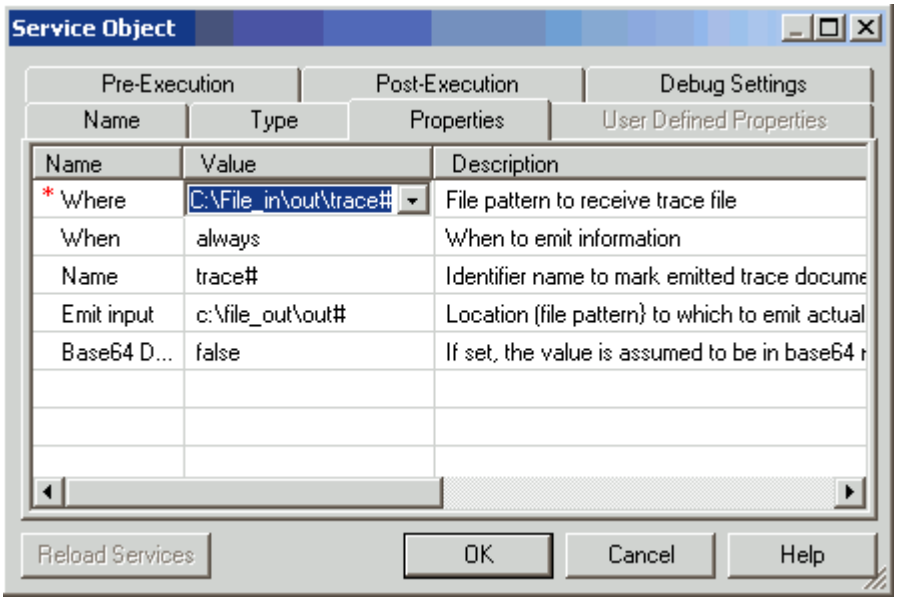
To create a process flow using iWay Designer to test for an End of Stream document:

1. Create a process flow and add a new Test object.

2. Configure the parameters for this Test object, as shown in the following image.

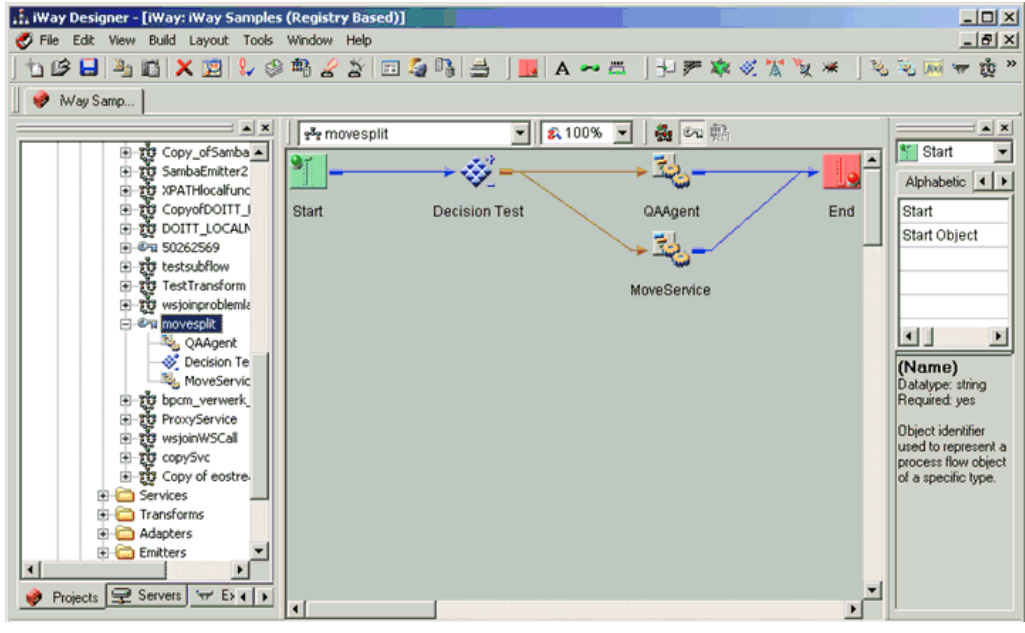


3. Add a new Service object to the process flow that uses the QAAgent.
4. Configure the parameters for this Service object, as shown in the following image.



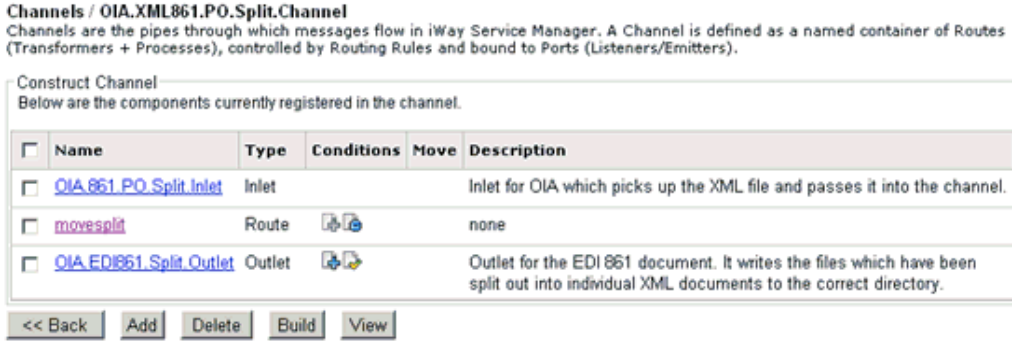
5. Connect the Service object to the True edge of the Test object.

The new process flow should have a structure that resembles the example in the following image.



6. Using the iWay Service Manager Administration Console, create a channel that consists of an XMLSplitpreparser as an inlet, which would return an End of Stream document when the last pass is reached.
7. Add a route to the channel that consists of the process flow, which was created using iWay Designer.
8. Add an outlet to the channel.

   The new channel should have a structure that resembles the example in the following image.



9. Use the following input document to test your channel with /a/b as the level string:

```
<a>
  <b name="b1">
    <c>value of input element a/b@name=b1/c is 1</c>
    <c>value of input element a/b@name=b1/c is 2</c>
  </b>
  <b name="b2">
    <c>value of input element a/b@name=b2/c is 3</c>
    <c>value of input element a/b@name=b2/c is 4</c>
  </b>
</a>
```

The _iseos() functions returns as true at the third (last) pass and a trace file is created which displays the End of Stream document.

_flatof(): Flatten the Payload

The current document is flattened to a string. If the current document is in XML format, then the *decl* parameter determines whether an XML declaration will be included in the flattened output. The default value is set to *true*.

The _flatof() function uses the following format:

```
_flatof([decl])
```

decl	boolean	Determines if an XML declaration should be included. Applies only to XML and is ignored for currently flat or JSON payloads.

_attcnt(): Index Attachments

Documents can hold attachments. This function has two forms. When used with no parameters, it returns the number of attachments associated with the document. When used with two parameters, it returns the index of the first attachment having the named header equal to the specified value. The _attcnt() function uses the following format:

```
_attcnt([name,value [,notfound]])
```

name	string	Name of an attachment header
value	string	Value of the named header

notfound	string	What to return if the named attachment header is not found

_atthdr(): Attachment Header Value

The _atthdr() function returns the value of a specific header in the attachment. The IC (independent case) form tests the attachment header names in a case independent fashion. Number 0 is the first attachment. The _atthdr() function uses the following format:

```
_atthdr(index, name, default)
```

```
_atthdric(index, name, default)
```

index	integer	Index of the attachment, base 0
name	string	Name of the header desired
default	string	Default if header not found

When the _atthdr() function is used in a conditional expression in comparison against a string value, the default value must be specified.

For example, the following conditional expression would not compile, since it requires a default value in the expression:

```
COND(_atthdr(1,'mysample'),eq,'a')
```

Instead, the expression must be specified as follows:

```
COND(_atthdr(1,'mysample','defaultval'),eq,'a')
```

For example, to return the first header in the attachment, `_atthdr(0,'mysample')` could be used.

_attbyfname(): Locate an Attachment By File Name

This function accesses the Content-Disposition header of each attachment, and returns the index (base 0) of the first attachment with the requested filename. It returns *-1* if the attachment is not found.

```
_attbyfname(filename)
```

filename	string	Name for testing.

Example:

The following sample document contains two attachments:

```
------=_Part_30_1258108044.1487787594926
Content-Type: application/xml
<?xml version="1.0" encoding="ISO-8859-1" ?><base/>
------=_Part_30_1258108044.1487787594926
Content-Type: application/xml
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename=walter.txt
Content-Length: 32
<part1>attachment data one</part1>
------=_Part_30_1258108044.1487787594926
Content-Type: application/txt
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename=fred.txt
Content-Length: 32
<part2>attachment data two</part2>
------=_Part_30_1258108044.1487787594926--
```

The function `_attbyfname('fred.txt')` will return the following value:

```
'1'
```

_attflatof(): Make the Content of an Attachment Available

The _attflatof() function returns the content of an attachment as a string. This function uses the following format:

```
_attflatof(index [,encoding])
```

index	integer	Index of the attachment, base 0.
encoding	string	The type of encoding used during the conversion. The default is UTF-8.

Example:

The following sample document contains two attachments:

```
------=_Part_30_1258108044.1487787594926
Content-Type: application/xml
<?xml version="1.0" encoding="ISO-8859-1" ?><base/>
------=_Part_30_1258108044.1487787594926
Content-Type: application/xml
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename=walter.txt
Content-Length: 32
<part1>attachment data one</part1>
------=_Part_30_1258108044.1487787594926
Content-Type: application/txt
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename=fred.txt
Content-Length: 32
<part2>attachment data two</part2>
------=_Part_30_1258108044.1487787594926--
```

The function `_attflatof(_attbyfname('fred.txt'))` will return the following value:

```
<part2>attachment data two</part2>
```

_attbyfname(): Locate an Attachment by File Name

The _attbyfname() function accesses the Content-Disposition header of each attachment, and returns the index (base 0) of the first attachment with the requested file name. A value of -1 is returned if the attachment is not found. This function uses the following format:

```
_attbyfname(filename)
```

filename	string	The file name of the attachment for testing.

Example:

The following sample document contains two attachments:

```
------=_Part_30_1258108044.1487787594926
Content-Type: application/xml
<?xml version="1.0" encoding="ISO-8859-1" ?><base/>
------=_Part_30_1258108044.1487787594926
Content-Type: application/xml
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename=walter.txt
Content-Length: 32
<part1>attachment data one</part1>
------=_Part_30_1258108044.1487787594926
Content-Type: application/txt
Content-Transfer-Encoding: 7bit
Content-Disposition: attachment; filename=fred.txt
Content-Length: 32
<part2>attachment data two</part2>
------=_Part_30_1258108044.1487787594926--
```

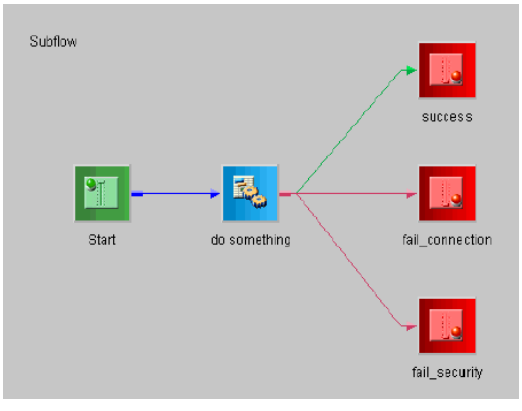
The function `_attbyfname('fred.txt')` will return a value of 1, which indicates that the attachment is found.

_srcname(): Source Name from Subflow

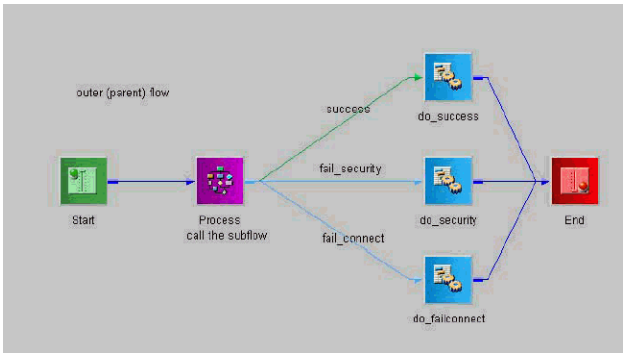
The _srcname() function returns the name of the terminate (end node) of the subflow through which the document reached the calling process flow. This allows the subflow to communicate information back to the parent flow.

Imagine a subflow that performs some operation that results in one of three possible situations:

- Success.
- Failure by a security violation.
- Connection error.



The parent flow calls the subflow, and wishes to deal with the result from that flow. Within the subflow, the document that is produced is returned through a terminate or end node, as shown below. The name of that terminate node becomes the edge name followed by the returned document. As this example shows in the image below, there are three activities (called do_<something>) in the outer flow, each of which handles the result returned from the inner subflow.



If the subflow returns multiple documents, each follows the edge of its own name. If more than one of any name is returned, the documents are associated as siblings, and can be split later using the sibling iterator. The source name can be used in a switch statement for more detailed routing.

_treehash (): Generate an MD5 Hash

The _treehash ([SREG]) function generates an MD5 hash of an XML tree. The hash can be used to compare two documents.

sreg	string	Register name of a register holding an XML document or an XML payload as saved using the _savedoc() function.

If no parameters are used, the current document's hash is returned. If you have saved a document or an XML payload, the hash for that tree is returned. The function does not address non-xml documents.

The hash is an MD5 hash computation including all elements, attributes, and element values. It also takes into account the structure of the document. The probability of two different documents having the same hash is vanishingly small.

Example:

Assume that a document has been saved into register *save*. You can compare it to the current document with:

```
_if(_treehash() == _treehash('save'),true,false)
```

Similarly, you can save the hash into a database for later comparison.

Parsed XML Functions

iWay Service Manager (iSM) manages messages in XML format. This section describes parsed XML functions that belong to the iWay Functional Language (iFL) set. Do not use these functions with documents in other (non-XML) formats, since errors will be generated. Parsed XML is a native format that is managed within iSM.

_xpath(): Execute an XPath Expression

The xpath() function is always an alias for one of the supported xpath functions: xpath1() or _iwxpath(). By default, xpath() is an alias for _iwxpath() but through compatibility flags, it can be made to alias xpath1().

The _xpath() function uses the following format:

```
_xpath(expression [,nsmap [,object]])
```

_xpath1(): Execute an XPath Expression

The document is evaluated by the provided xpath expression. The start is always considered to be the root, so all expressions are assumed to begin with a forward slash character (/).

The purpose of the xpath in the context of functions is to extract values from an input document to be used as configuration parameters. This use of xpath is not intended for general XML tree manipulation.

In some servers, xpath is a full xpath version 1 as specified by XML Xpath Language *http://www.w3.org/TR/1999/REC-xpath-19991116*, while others support only the portion of xpath specified in section 2.5, Abbreviated Syntax. iSM Version 7.0 supports full XPath per the specification.

The _xpath1() function uses the following format:

```
_xpath1(expression [,nsmap [,object]])
```

expression	string	Expression in xpath language
nsmap	string	Name of a namespace map from a provider. If omitted, no namespace map is applied.

object	document	A document to which xpath is applied. If omitted, the current document is evaluated.

The nsmap is the name of a namespace provider. Namespace providers relate the namespace portion of the xpath expression to the URI to which it relates. The xpath execution engine will map the URI to the namespace tokens used in the document being evaluated.

If the object parameter is omitted, the xpath is applied against the current document. Otherwise, this must be the name of a special register holding a document or a register itself holding a document.

In some cases, XPath syntax can be confusing. Consider a document in a default namespace:

```
<root xmlns='http://someuri.com'>
    <child name='whoami'/>
</root>
```

A request for the child name would appear to be //child/@name, and if there were no default namespace, this would work as expected. For the anonymous namespace, a solution is one.

The default namespace throws in a twist. The XPath specification states the following:

A node test that is a QName is true if and only if the type of the node (see [5 Data Model]) is the principal node type and has an expanded-name equal to the expanded-name specified by the QName.

This means that an XPath processor does not deal with plain element names, except those element names for which no namespace has been declared. If there is any namespace declaration at all, including one for the default (unprefixed) namespace, the processor uses the expanded-name.

```
//*[namespace-uri()="http://someuri.com" and local-name()="child"]/@name
```

Alternatively, create a namespace provider name, for example, empty1. Add a specification for a namespace, such as:

none	http://someuri.com

Then use an _xpath statement, such as the following:

```
_xpath(/none:root/none:child/@none:name,empty1)
```

The processor will map the none namespace to the URI in the statement and select the proper nodes from the document.

_iwxpath(): Execute an XPath Expression

The _iwxpath() function uses the following format:

```
_iwxpath(expression)
```

expression	string	Expression in xpath language

The current document is evaluated by the provided xpath expression. The start is always considered to be the root, so all expressions are assumed to begin with forward slash (/).

The purpose of the xpath in the context of functions is to extract values from an input document to be used as configuration parameters. This use of xpath is not intended for general XML tree manipulation.

The xpath language supported is specified in XML Xpath Language *http://www.w3.org/TR/1999/REC-xpath-19991116*, section 2.5, Abbreviated Syntax.

In those servers not providing full xpath support, both xpath and _iwxpath are identical. In other servers, it can be expected at _iwxpath should operate faster than full xpath. iWay recommends use of full xpath if _iwxpath does not provide the required services or results.

_xflat(): Generate a Subtree

The _xflat() function is always an alias for one of the supported _xflat() functions:

- _xflat1()
- _iwxflat()

By default, xflat() is an alias for _iwxflat(). However, by using compatibility flags, it can be made to alias xflat1().

_xflat1(): Generate a Subtree

The _xflat1() function uses the following format:

```
_xflat1(expression [,nsmap [,object]])
```

expression	string	An XPath expression.
nsmap	string	The name of a namespace map from a provider. If omitted, no namespace map is applied.

object	document	A document to which XPath is applied. If omitted, the current document is evaluated.

The current document is evaluated by the provided XPath expression. As with XPath, the root of the current document is the starting point. The subtree addressed by this expression is returned in a flattened form that is suitable for reparsing.

The specific parameters are discussed under BAD XREF HERE "_xpath: Execute an XPATH Expression. The _xflat1() function is often used to provide document segments in the constant value service leading into a join. For example:

```
<a>
   <top>
      <b>one</b>
      <b>two</b>
      <x>
         <xroot>
            <xchild>rootchildvalue</xchild>
         </xroot>
      </x>
   </top>
```

The expression _xflat1(//xroot) yields the subdocument:

```
    <xroot>
      <xchild>rootchildvalue</xchild>
    </xroot>
```

_iwxflat(): Generate a Subtree

The _iwxflat() function uses the following format:

```
_iwxflat(expression [,nsmap [,object]])
```

expression	string	An XPath expression.
nsmap	string	The name of a namespace map from a provider. If omitted, no namespace map is applied.
object	document	A document to which XPath is applied. If omitted, the current document is evaluated.

The current document is evaluated by the provided XPath expression. As with XPath, the root of the current document is the starting point. The subtree addressed by this expression is returned in a flattened form that is suitable for reparsing.

The specific parameters are discussed under BAD XREF HERE "_xpath: Execute an XPATH Expression. As is the case with _xpath(), this function is faster but not as fast as the standard _xflat1() function. The _iwxflat() function is often used to provide document segments in the constant value service leading into a join. For example:

```
<a>
   <top>
      <b>one</b>
      <b>two</b>
      <x>
         <xroot>
            <xchild>rootchildvalue</xchild>
         </xroot>
      </x>
   </top>
```

The expression _iwxflat(//xroot) yields the subdocument:

```
<xroot>
       <xchild>rootchildvalue</xchild>
</xroot>
```

Understanding XML Path Language (XPath)

Support for XML Path Language (XPath) is an important feature of iWay and is used in a number of areas within the server. XPath is a non-procedural language used to access and manipulate sections of an XML document. The XPath expression gathers information from the document, as if the XML document is a self-contained hierarchical database. The XPath expression specifies levels (segments or fields), filter predicates, and functions on the XML document data. The result of the XPath can be one or more values, a set of XML nodes, or a particular location in the XML structure. Using these XPath results, iWay Service Manager (iSM) can control the behavior of service agents, conditional routing, and decision-making inside of process flows. iSM supports multiple XPath processors ranging from a fast, internal processor supporting a subset of the XPath language, to full support of the entire language. The complete XPath language specification and information on the abbreviated syntax can be found at:

http://www.w3.org/TR/xpath.html

Navigating XML With Location Steps

In iWay Service Manager, the node context from which location steps begin is always the root of the document. Furthermore, only the child axis is implemented and is implicit in all iWay XPath location steps.

Reference: Location Steps

Expression (phrase)	Action
/name	Locate down one level, selecting children of the specified name.
//name	Locate down, selecting children of the specified name regardless of the depth.
/*	Locate down, selecting all children.
//*	Locate down, selecting all children.
/.	Locate all nodes that are already selected.
/..	Locate upward one level, selecting the parent of each node in the node-set.

The result of the location step phrase can be a set of XML document nodes consisting of zero, one, or many nodes. This set of nodes is referred to as a node-set. This node-set is provisional and may not be the final node-set returned by the XPath expression, depending on subsequent predicates.

XPath Best Practices

iWay supports two XPath processors with different characteristics. Selecting the right one for any specific situation can greatly improve the performance of an application.

_iwxpath(). This is an XPath processor provided by iWay. It supports a limited subset of the XPath language. If your document and query are appropriate to using _iwxpath, you can expect to achieve significant performance improvements. However, the limitations of this process are:

- Very limited function support.
- No advanced XPath features such as math.
- Surrounding functions such as count() are not supported.
- Conjunctions (such as OR and AND) and complex tests are not supported.
- Namespaces are not supported.

Note: For most applications, these omitted features are not used. If _iwxpath() does not return the results you desire, use _xpath1().

_xpath1(). This is the full XPath. It may be slower for certain operations, but it does provide the complete language support of XPath version 1.

_xpath(). This is the default xpath call. Whether it calls _iwxpath() or _xpath1() depends on a configuration setting on the iSM Administration Console.

In general, it is best practice to write your xpath expressions as simply as possible. Avoid expressions that may return multiple results when you are not expecting them. Selecting any (*) can slow down any XPath processor, as well as the //. Use these language constructs if they are needed, but understand that there may be a penalty in performance.

XPath Predicates

Predicates are written after the location step, and are enclosed in square brackets. There can be one or more predicates in a step, each of which is applied left to right to control the membership of the node-set.

A predicate filters the node-set implied by a location step, to produce a new node-set. For each node in the node-set to be filtered, the predicates are evaluated with that node. If predicate expression evaluates to true for that node, the node is included in the new node-set. Otherwise, it is not included.

Multiple predicates are written as sequential predicate terms and are applied left to right, behaving as if connected by a logical AND:

```
/Book[Author='Smith'][Price<10]
```

Each predicate term consists of a single integer, a filter function, or a relation of the form.

```
<left-value> operator <right-value>
```

A predicate can hold any number of terms, logically connected by AND and OR. The specification calls for left to right precedence, with AND taking higher precedence than OR. Terms may be grouped by parenthesis to force a particular order of evaluation. For example, the predicate [a=b OR c=d AND e=f] is evaluated as [a=b OR (c=d AND e=f)].

Integer predicates of the form /Tag[2] imply a numeric index into the current node-set (in this case, selecting the second node).

Reference: Comparison and Logical Operators

Symbol	Description	Example
=	Equal	//*[local-name(CustomerRecord/RecordDate)='OrderRecord']

Symbol	Description	Example
!=	Not Equal	//*[local-name(CustomerRecord/RecordDate)! ='OrderRecord']
<	Less than	/OrderRecord/LineItems[position() < 5]
<=	Less than or equal	/OrderRecord/LineItems[position() <= 5]
>	Greater than	/OrderRecord/LineItems[position() > 5]
>=	Greater than or equal	/OrderRecord/LineItems[position() >= 5]
Or	Logical Or	OrderRecord[salesman = 'Jones' or salesman='Scott]
And	Logical And	OrderRecord[salesman = 'Jones' and salesman='Scott]
Not	Logical Not	/a//*[not(starts-with(name(),'d'))]

Reference: **Predicate (filter) Functions**

Expression	Description
starts-with(string1, string2)	Returns true if the first string starts with the second string, otherwise returns false.
ends-with(string1, string2)	Returns true if the first string ends with the second string, otherwise returns false.
contains(string1, string2)	Returns true if the first string is contained within the second string, otherwise returns false.

In each of the preceding filter functions, string1 may be name(), @attribute, @* (all attributes) or the name of the child node.

XPath Functions

A collection of functions is provided to operate on nodes and node-sets. A function operates on the current context (for example, the current node) or on characteristics of the current node-set, and returns a single value.

Reference: Functions

Function	Description
count(<node-set>)	Returns the number of nodes in the specified node-set.
last()	Returns the position number of the last node in the context node-set.
position()	The position of a node in the node-set relative to its parent.
count()	The number of children of each node in the node-set.
text()	Returns the value of each node in the node-set. Also used to return the value of CDATA.
name()	Returns the name of the nodes in the selected node-set. Example //sql/*/name() returns the names of the grandchildren of each sql node.
local-name	Returns the local name of each node in the node-set.
sreg(name[,default])	The value of a named special register (iWay extension to XPath).
@attrib, @*	Returns the values of the specified attribute, * returns the set of all attribute values.
concat(string1, string2)	Returns the concatenation of all its arguments. Two or more strings may be concatenated.
Substring(string, position, length)	Returns the substring of the first argument starting at the position specified in the second argument with length specified in the third argument.
Substring-before(string1, string2)	Returns the substring of string1 that precedes the first occurrence of string2 in string1, or the empty string if string1 does not contain string2.
Substring-after(string1, string2)	Returns the substring of string1 that follows the first occurrence of string2 in string1, or the empty string if string1 does not contain string2.

Function	Description
namespace-uri()	Returns the namespace URI of each node in the node-set.

Parsed JSON Functions

iWay Service Manager (iSM) manages messages in JavaScript Object Notation (JSON) format. This section describes parsed JSON functions that belong to the iWay Functional Language (iFL) set. Do not use these functions with documents in other (non-JSON) formats, since errors will be generated. Parsed JSON is a native format that is managed within iSM.

To demonstrate the functionality of these parsed JSON function description, consider the following sample JSON message:

```
{"db":{"vec":[1,2,3],"str":"abc","obj":{"a":1,"b":2}}}
```

_jsonptr(): Execute a JSON Pointer Expression

The _jsonptr() function uses the following format:

```
_jsonptr(jexpression [,object])
```

expression	string	An expression to be evaluated against the JSON.
object	document	A document to which JSON Pointer is applied. If omitted, then the payload of the current document is evaluated.

The JSON Pointer expression is evaluated against the JSON document. JSON Pointer defines a string syntax for identifying a specific value or section within a JSON document. The expression meets the criteria of *RFC 6901 – JavaScript Object Notation (JSON) Pointer*.

Expression	Result
`_jsonptr(/db/str)`	`abc`
`_jsonptr(/db/vec)`	`[1,2,3]`
`_jsonptr(/db/obj)`	`{"a":1,"b":2}`

The optional object can be the name of a Special Register (SREG) holding a value in JSON format, or a value in JSON format. If present, then the expression is evaluated against this value.

_jsonpath(): Execute a JsonPath Expression

The _jsonpath() function uses the following format:

```
_jsonpath(jexpression [,object])
```

expression	string	An expression to be evaluated against the JSON document.
object	document	A document to which JsonPath is applied. If omitted, then the payload of the current document is evaluated.

The JsonPath expression is evaluated against the JSON document. JsonPath defines a string syntax for identifying a specific value or section within a JSON document. The expression meets the criteria of JsonPath, which has no official RFC. It is reputed to be an analog of XPath, having considerably more power than JSON Pointer. It supports, for example, the use of predicates.

Expression	Result
`_jsonpath($.db.str)`	`abc`
`_jsonpath($.db.vec)`	`[1,2,3]`
`_jsonpath($.db.obj)`	`{"a":1,"b":2}`

The optional object can be the name of a Special Register (SREG) holding a value in JSON format, or a value in JSON format. If present, then the expression is evaluated against this value.

String Functions

String functions operate on information that is in string form. Such strings can be returned from other functions such as _flatof() or can be literals with operation parameters taken from other operations such as xpath(). This section lists and describes the various string functions that you can use in iWay Service Manager (iSM).

_left(): Leftmost Substring

The _left() function extracts the leftmost substring from the input string. This function is equivalent to using the _substr() function from the left position (0), but is more convenient to use for many cases.

```
_left(input, length)
```

input	string	The string value to be operated upon.
length	integer	The length of the desired substring.

_right(): Rightmost Substring

The _right() function extracts the rightmost substring from the input string. This function is equivalent to using the _substr() function, but avoids the need to compute the starting position.

```
_right(input, length)
```

input	string	The string value to be operated upon.
length	integer	The length of the desired substring.

_lcase(): Convert to Lower Case

The _lcase() function converts the input string to lower case. The _lcase() function uses the following format:

```
_lcase(input)
```

input	string	The string value to be converted to lower case.

_ucase(): Convert to Upper Case

The _ucase() function converts the input string to upper case. The _ucase() function uses the following format:

```
_ucase(input)
```

input	string	The string value to be converted to upper case.

_trim(): Eliminate Whitespace

The _trim() function removes leading and trailing white spaces from a string. This includes blanks, tabs, and carriage returns. Whitespace includes blanks, tabs, carriage returns, and any characters defined for the current locale as a whitespace character. The _trim() function uses the following format:

```
_trim(input)
```

input	string	The string value to be trimmed.

_normalizespace(): Eliminate Whitespace

The _normalizespace() function removes leading and trailing white spaces from a string. This includes blanks, tabs, carriage returns, and any characters defined for the current locale as a whitespace character. Within the string multiple blanks are converted to a single blank. For example, if the dot character represents a space, _normalizespace('ab…c') yields 'ab.c'.

The _normalizespace() function uses the following format:

```
_normalizespace(input)
```

input	string	The string value to be normalized.

_entity(): Entity Encoding

The _entity() function replaces XML characters with entities, for example, A&B becomes A&B.

The _entity() function uses the following format:

```
_entity(input)
```

input	string	The string value to be operated upon.

_deentity(): Entity Decoding

The _deentity() function replaces XML entities with characters, for example, A&B becomes A&B.

The _deentity() function uses the following format:

```
_deentity(input)
```

input	string	The string value to be operated upon.

_substr(): Substring

The _substr() function extracts a substring from the input string. The substring begins at the starting position and extends to the ending position, which is not included in the substring. If a specific ending position is omitted, then the substring extends to the end of the input string. For example, _substr('abcde',1,3) returns bc.

The _substr() function uses the following format:

```
_substr(input, start [,end])
```

input	string	The string value to be operated upon.
start	integer	Starting position, base 0.
end	integer	Ending position, base 0.

_before(): Substring

The _before() function extracts a substring from the input string. The substring begins at the start of the string and extends to the ending string, which is not included in the substring. If omitted, the substring extends to the end of the input string. For example, _before('abcde','de') returns abc.

The _before() function uses the following format:

```
_before(input, pattern)
```

input	string	The string value to be operated upon.

pattern	string	Termination of the string.

_after(): Substring

The _after() function extracts a substring from the input string. The substring begins at the start trigger and extends to the end of the string. For example, _after('abcde','bc') returns de.

The _after() function uses the following format:

```
_after(input, pattern)
```

input	string	The string value to be operated upon.
pattern	string	Trigger of the string.

_pad(): Pad to Desired Length

```
_pad(input, length [,type [,character [,control]]])
```

input	string	The string value to be operated upon.
length	integer	Length of desired output.
type	keyword	Direction of the padding. ❏ **Right.** Pad on the right of the input (default). ❏ **Left.** Pad on the left of the input (right justification).
character	char	One character to be used for padding. Default is blank.
control	keyword	Final disposition of the output. ❏ **asis.** Take no action following padding (default). ❏ **cut.** Cut output to length if input is too long.

The input string is padded and optionally cut to fit an exact length. Padding can be done to the right or left of the string.

For example, given the input abc, the following will be the result, using the character b for blank:

Length	Direction	Character	Control	Result
5	right	omit	omit	**abc**bb
5	left	omit	omit	bb**abc**
2	right	omit	cut	**ab**

_concat(): Concatenate Strings

The _concat() function joins together to generate a single string. For example, if special register x holds the letter 'b', _concat('a',sreg(x),'c') returns abc.

The compiler attempts automatic concatenation when functions are recognized in input strings. For example, the special register iway.home holds the root of the iWay Server installation. So, if you want to address a file named myfile.txt in the mydir subdirectory off the iWay root, use of sreg('iway.home')/mydir/myfile.txt will return the correct path name.

The _concat() function uses the following format:

```
_concat(input*)
```

input	string	The string value to be operated upon.

_length(): String Length

The _length() function returns the number of characters in the supplied input string. It uses the following format:

```
_length(input)
```

input	string	The string value to be operated upon.

_count(): String Element Count

The _count() function returns the number of items in the supplied input string. It uses the following format:

```
_count(input [,action][,delim])
```

input	string	The string value to be operated upon.
action	keyword	What to include in the count ❏ **value.** Only nodes containing values. ❏ **empty.** Only nodes with empty values. ❏ **all.** Total of nodes.
delim	character	The field delimiter.

Returns the number of items in the supplied input string. Usually this is one (1). Sometimes an xpath expression will identify several values meeting a test. In such a case this is the number of matches that the xpath expression located in the current document.

A common use is to determine how many elements of a given type are contained in the document. For example, given the document:

```
<a>
     <top>
     <b>one</b>
     <b>two</b>
      <x>sreg(iwayhome)</x>
     </top>
</a>
```

The expression _count(xpath(//b)) yields 2, while the expression _count(xpath(//b),empty) yields 0.

The delimiter is not specified when the first parameter (input) is an xpath expression. For other types of input, it may be needed. For example:

```
_count(“a,b,,c,d”,empty,’,’) yields 1.
```

_contains(): String Contents

The _contains() function returns true if the input string contains the value. The search is case sensitive. The _contains() function uses the following format:

```
_contains(input,value)
```

input	string	The string value to be operated upon.
value	string	The search value.

_startswith(): String Contents

The _startswith() function returns true if the input string starts with the value. For example, _startswith('iWay Software','iW') yields true. The _startswith() function uses the following format:

```
_startswith(input,value)
```

input	string	The string value to be operated upon.
value	string	The search value.

_endswith(): String Contents

The _endswith() function returns true if the input string ends with the value. It uses the following format:

```
_endswith(input,value)
```

input	string	The string value to be operated upon.
value	string	The search value.

_regex(): Replace Portions of a String

The _regex() function searches an input string using the regular expression. Any matches are replaced with the value that is specified for the replacement parameter. Regular expressions are specifications of what is to be located and where (for example, locating the first carriage return). The _regex() function uses the following format:

```
_regex(input, pattern, replacement)
```

input	string	The string to be operated upon.

pattern	string	Regular expression pattern.
replacement	string	The replacement text.

For example, to change *scat, cat* to *scat, dog*, you can use the following _regex() function call:

```
_regex('scat, cat','\\bcat','dog')
```

Note: The backslash character in the pattern was doubled. This is due to the use of a single backslash as an escape character. The double backslash characters (\\) causes a single backslash character to be inserted in the pattern.

Double backslash characters are used because the pattern must be \bcat and the iWay Functional Language (iFL) handles a single backslash character as an escape character. The \b pattern character is used to specify a word boundary.

Regular expressions are a standard means of searching data, and there are many books on this topic. Some commonly used online references include:

- *http://docs.oracle.com/javase/tutorial/essential/regex/*
- *http://www.javamex.com/tutorials/regular_expressions/*

A good book on this topic is:

Hitchens, Ron, "Java NIO", Cambridge, O'Reilly Media, Inc., 2002. [ISBN 0-00288-2]

_reverse(): Reverses a String

Reverses the string value. This function is useful when a value is to be used as an index in a database, where the high order characters are relatively invariant. Using the reverse of the value can improve index hashing for some databases.

For example:

```
_reverse('iWay Software_01')
```

Yields the following:

```
10_erawtfoS iaWi
```

Because the varying part is at the head of the new field, the hash algorithms may generate a wider distribution.

_match(): Perform a String Match Against a Pattern

The _match() function matches an input string against a regular expression. A successful match is returned as *true*, and an unsuccessful match is returned as *false*.

```
_match(input,pattern)
```

input	string	The input string to be matched against the regular expression.
pattern	string	The regular expression pattern.

For more information on regular expression matching, see *_regex(): Replace Portions of a String* on page 51.

For example, to determine whether an attribute value that is contained in a Special Register (SREG) matches a stored SIC code in category 127xx (to be routed to the appropriate subflow), configure the _match() function as follows:

```
_if(_match(sreg(insic),'127.*'),_setreg('flowcall','subflow127'),_setreg('fl
owcall','subflowAny'))
```

The application could then use the subflow SREG as an input to the Process Flow object (XDPFlowAgent) to call the selected subflow.

To check a string against many patterns simultaneously, use the _propertymatch() function. For more information, see *_propertymatch(): Match a String Against a File of Regular Expression Patterns* on page 16.

_replace(): Translate Characters in a String

The _replace() function translates characters in a string.

```
_replace(input, chars, replacements [,eoloption])
```

input	string	The string to be operated upon.
chars	string	The characters in the string to be replaced.
replacements	string	The replacement characters.

eoloption	keyword	End of line conversions. You can specify one of the following conversion options: ❑ **NONE.** No conversion is specified. ❑ **LF2CR.** Map the linefeed character to a carriage return. ❑ **LF2CRLF.** Map the linefeed character to a carriage return linefeed. ❑ **CRLF2LF.** Map the carriage return linefeed sequence to a single linefeed. This specification is used most often for Windows/ UNIX conversions.

The input string is searched for the designated characters. Each located character in the chars operand is replaced with the replacements operand.

The characters in the chars and replacements operands are replaced on a one-for-one basis. The first character in the chars operand is replaced by the first character in the replacements operand, and so forth. Any ASCII character can be entered into either operand. Additionally, escaped sequences, each representing one character, can be included in the operands.

\n	New line
\t	Tab
\r	Line feed
\\	Backslash
\"	Double quote
\'	Single quote
\xcc	Hex character, where cc is the hex representation. For example, \x01 is hex 01.

Example 1. Replace all bar characters with commas. This example is important, as the _xpath functions return lists of values separated by bars.

```
_replace(_xpath(/root/childval),'|',',')
```

Example 2. Replace all new lines with tabs.

```
_replace(sreg(values),\n,\t)
```

Example 3. In this example, four backslashes are used to represent a paired backslash in the second parameter. This is because the input handlers see the backslashes as two single backslashes. The function then sees the escaped backslash (\\) as a single character. This allows any backslash in the value in the sreg() to be replaced with an ampersand.

```
_replace(sreg(abc),'\\\\','&')
```

Example 4. In this example, a hexadecimal value is replaced. A common use is to replace separator characters in an EDI document with a printable value.

```
_replace('ab\0x85c','\0x85', '!')
```

Example 5. A unicode value can be used. In this example, `ab~c` becomes `ab$c`.

```
_replace('ab~c','~','\u0024')
```

Note: The specific Unicode value need not be printable, but the value for $ was selected for this example as a convenience.

_isnumber(): Is a Value Number

The _isnumber() function tests whether the input is a valid number. A value of true indicates that the input is numeric and false indicates that it is not numeric or is not present. Use this to test the results from functions that are expected to return numeric output but may return NAN (not a number) or another value.

The _isnumber() function uses the following format:

```
_isnumber (input)
```

input	string	The string value to be operated upon.

_lit(): Literal String Concatenation

The _lit() function treats the input string as a literal for parsing purposes. Functional replacement is performed, but math operations are ignored. If the input string is in literals, then the entire string is treated as a single literal, and no further operation takes place. However if the literal marks (single quotes) are omitted, then the parameter can be evaluated.

In the following example, assume that a special register (SREG) named *portno* has been set to 3284.

Statement	Result
_lit(<root>sreg(portno)</root>)	<root>3284</root>
_lit('<root>sreg(portno)</root>')	<root>sreg(portno)</root>

The _lit() function uses the following format:

```
_lit(input)
```

input	string	The string value to be operated upon.

_sql(): SQL Concatenation

The input string is treated as a string for parsing purposes. Functional replacement is performed but math operations are ignored. This operates identically to _lit().

The _sql() function uses the following format:

```
_sql(input)
```

input	string	The string value to be operated upon.

_xml(): XML Concatenation

The input string is treated as a string for parsing purposes. Functional replacement is performed but math operations are ignored. This operates identically to _lit().

The _xml() function uses the following format:

```
_xml(input)
```

input	string	The string value to be operated upon.

_qval(): Quote/Null a String

This function generates a quoted string of the input operand. Optionally, the string NULL can be produced. This function is useful when generating values for an SQL insert or update DML statement.

The _qval() function uses the following format:

```
_qval(input [,char [,action]]))
```

input	string	The string value to be operated upon.
char	string	A keyword for the type of quote to be used. ❑ **single.** Use single quote characters. ❑ **double.** Use double quote characters.
action	string	A keyword describing the action to perform for nulls or empty input values. ❑ **none.** Null input is quoted empty string. ❑ **null.** Null input is the word NULL (unquoted). ❑ **empty.** Null or empty input is the word NULL (unquoted). ❑ **spaces.** Maps strings that are filled with spaces to NULL (unquoted).

For example, consider the following SQL expression fragment:

```
values(_qval(a))
```

This might be represented in iWay Designer as:

```
values(?a)
```

with the user field of:

a	`xpath(/root/field)`

In this example, the input document field addressed in the xpath() returns a value or a null result. Depending on what is returned and the action parameter, the results would be:

Returned	Action Parameter	Result
iway	any	iway

Returned	Action Parameter	Result
empty value	none	‘’
empty value	null	‘’
empty value	empty	‘’
empty value	spaces	‘’
null value	none	‘’
null value	null	NULL
null value	empty	NULL
null value	spaces	NULL

_token(): Tokenize a String

Given a delimited string, this function splits the string on the specified delimiter and returns the token at the requested index. The delimiter can be a regular expression and the index of the first token is 1. This function returns an empty string if:

- The delimiter is an invalid regular expression.
- The index parameter can not be parsed as an integer.
- The index parameter is less than 1 or greater than the number of tokens after splitting.

The _token() function uses the following format:

```
_token(input, delimiter, index)
```

input	string	The string value to be operated upon.
delimiter	string	The splitter delimiter, which can be a [Java] regular expression.
index	integer	The index of the token desired, base 1. The default is 1.

Examples:

```
_token('a,b,c,d,e', ',', 3) = c
_token('a/b/c/d/e', '/', 3) = c
_token('a/b/c/d/e', '/', 18) = empty
_token('a_split_b_split_c_split_','_split_',3)=c
```

The next example accepts x followed by exactly three percent signs, followed by an x or a p:

```
_token("ax%%%xbx%%%pcx%%%xdx%%%pe",'x\\%{3}[xp]', 3)=c
```

Note: iFL uses a single backslash to escape the special meanings most characters have in a string of code. To include a single backslash in a string, it must be escaped as well. Use \\ in a string to represent a single backslash. Another example of how to use a single backslash to escape special character meanings is seen in the following expression:

```
 _token(B2BIT.reshma,'\\.', 1)
```

The second parameter is a regular expression. You need the backslash to look for the actual dot character. Because iFL is parsing, and it uses the backslash as an escape, you need to escape it to get the backslash into the regular expression. For a more detailed explanation, see *_regex(): Replace Portions of a String* on page 51.

_indexof(): Return Offset to a Substring

The _indexof() function uses the following format:

```
_indexof(input, pattern [,startat])
```

input	string	The string value to be operated upon.
pattern	string	The characters to be located.
startat	integer	Offset to start the search (base 0).

The input string is searched for the contained pattern. The index of the pattern is returned, base 0. The function is useful when you are parsing or extracting elements of an input string.

_printable(): Mask Nonprintable Characters

The _printable() function uses the following format:

```
_printable(input [,keyword])
```

input	string	The string value to be operated upon.

keyword	string	A control word. If whitespace, carriage returns and tabs are not translated.

Sometimes a string that is to be printed may contain unprintable characters. For example, this can result from the _inflate() or _frombase64() function. This function converts non-printable characters to period characters.

_murmurhash(): Hash a String Value

The _murmurhash() function implements the MurmurHash standard, a non-cryptographic hash function suitable for general hash-based lookup purposes.

Note: The _murmurhash() function is not a cryptographic hash, since MurmurHash is not specifically designed to be difficult to reverse by an adversary.

MurmurHash is useful, among other purposes, for database key generation and lookup, and for use in Bloom filters.

The _murmurhash() function uses the following format:

```
_murmurhash(input [,algorithm [, seed [, encoding]]])
```

input	string	Specifies the value to be hashed. If omitted, then the current document payload is hashed.
algorithm	keyword	Specifies the type of algorithm to be used: ❏ **H16**. Yields a 16-bit hash value. ❏ **H32**. Yields a 32-bit hash value. ❏ **H64**. Yields a 64-bit hash value. The default is H64.
seed	number	Specifies the initial seed in integer or 0x hex form. ❏ H16 (0x3175467) ❏ H32 (0x9747b28c) ❏ H64 (0xe17a1465) The default is H64 (0xe17a1465).

encoding	string	The Internet Assigned Numbers Authority (IANA) encoding value of the input. The default IANA value is ISO-8859-1.

The output of the _murmurhash() function is expressed as a base-10 integer.

While the H32 and H64 algorithms are generally suitable for database key use, the H16 algorithm is optimized for use in short Bloom filters. The H16 algorithm is calculated using the MurmurHash approach, but is not part of the standard.

Time Service Functions

Time services are used to enter parameters requiring time of day operations or to test for values relating to the current time. For example, a process flow can be configured to perform one operation on Friday and another operation on every other day.

Time services are provided by three functions that all provide formatted access to the current time. This section lists and describes the various time service functions that you can use in iWay Service Manager.

_now(): Get Current Timestamp

The _now() function returns the current time based on a provided pattern. It uses the following format:

```
_now([pattern])
```

pattern	string	Format pattern

The default pattern (MM/dd/yyyy) returns the date in month/day/year format. So, for example, the date might return as 06/15/2006 if the function _now() is entered with no pattern.

Pattern characters can be assembled to provide the desired return. The following table lists the characters and the expected result. Characters are case-sensitive. All examples are based on a time of June 15, 2006 at 13:02:08 PM.

Character	Use	Return type	Example
y	year	Digit	yyyy=2006, yy=06
M	Month of year	Depends on length	MM=06, MMM=Jun, MMMM=June

Character	Use	Return type	Example
w	Week in year	Digit	26
W	Week in month	Digit	2
D	Day in year (Julian)	Digit	175
d	Day in month	Digit	25
E	Day of week	Text	Sun
F	Day of week in month	Digit	0
a	Division of day	Text	pm
H	Hour (24 hr clock, 0-23)	Digit	12
h	Hour (12 hr clock 1-12)	Digit	01
K	Hour (24 hr clock, 1-24)	Digit	13
k	Hour (12 hr clock, 0-11)	Digit	00
m	Minute in hour	Digit	02
s	Second in minute	Digit	08
S	Milliseconds	Digit	HH:mm:ss.SSS
z	Time zone	Text	EDT. The local time zone.
u	Day number of week (1 = Monday, ..., 7 = Sunday)	Number	3 (Wednesday)
Z	RFC-822 time zone	Number	-0500 (EST)

The function call must provide sufficient pattern characters to fill the field for those where length is specified. The following table provides a few examples:

cond(_now('E'),eqc,'Tue')	Does work on Tuesday only.

_now('hh:mm')	The current time. For example, 3:37 is expressed as 03:37.
_now('D')	For example, for July 21, 2006, the day of year is 202.
_now('dd/MM/yy')	European date format. For example: 25/06/06
_now('z')	EDT
_now("yyyy-MM-dd'T'HH:mm:ss.SSS'Z'")	RFC 3339 (ISO 8601) time. For example: 2012-02-23T11:41:34.793Z

Note: Special characters can be included in the pattern by using an apostrophe (‘) character. To avoid compiler confusion, surround the pattern string using double quote (“) characters.

In releases running under Java Version 1.6, the pattern character 'u' was used by iWay for advanced timing capabilities, and will continue to provide that service. The special iWay pattern characters are now supported in the _timer() function.

_timer(): Return Unix Epoch Time

The _timer() function returns time values based on the request. It uses the following format:

```
_timer(type)
```

The _timer() function replaces the use of the _now() function with special iWay-specific patterns.

type	keyword	Control

The supported types are listed and described in the following table.

Keyword	Use	Example	Note	Alias*
seconds	Unix epoch time in seconds.	1377179943046	Accurate	u

Keyword	Use	Example	Note	Alias*
milliseconds	Unix epoch time in milliseconds.	1377179943046678	Accurate	U
nanoseconds	Unix epoch time in nanoseconds.	1377179943046678945	To the best resolution available in the system clocks. This will depend on the hardware and JVM being used.	Un

If no *type* value is specified for the _timer() function, then the U (milliseconds) pattern character is used by default.

Note: Aliases are provided for the keywords, which reflect prior support for iWay pattern characters in the _now() function. iWay strongly discourages their use, and cannot guarantee that they will continue to be supported beyond iSM Version 7.0.

The pattern character 'Un' represents the best time available for the system at the nanosecond level. This method can only be used to measure elapsed time and is not related to any other notion of system or wall-clock time. The value returned represents nanoseconds since some fixed but arbitrary. This method provides nanosecond precision, but not necessarily nanosecond accuracy. No guarantees are made about how frequently values change.

_tstamp(): Return the Current Timestamp

The _tstamp() function returns a timestamp (accuracy to seconds) in ISO 8601 format.

This is a standard time function that is commonly used in many XML and web applications.

The _tstamp() function uses the following format:

```
_tstamp([locale])
```

locale	Determines the time zone used for the returned timestamp, which can be set to one of the following values: ❑ **gmt.** The timestamp is returned based on Greenwich Mean Time (GMT). This is the default value. ❑ **local.** The timestamp is returned based on the current time zone that is configured for iWay Service Manager (iSM).

_ftstamp(): Return the Current Timestamp to Milliseconds

The _ftstamp() function returns a fine timestamp (accuracy to milliseconds) in ISO 8601 format. This function uses the following format:

```
_ftstamp([compression] [locale])
```

compression	Determines the format of the returned timestamp, which can be set to one of the following values: ❑ **standard.** ISO 8601 format. This is the default value. ❑ **compressed.** The formatting characters are removed.
locale	Determines the time zone used for the returned timestamp, which can be set to one of the following values: ❑ **gmt.** The timestamp is returned based on Greenwich Mean Time (GMT). This is the default value. ❑ **local.** The timestamp is returned based on the current time zone that is configured for iWay Service Manager (iSM).

For example:

`_ftstamp()` yields `2017-07-21T19:42:44:609Z`

The special pattern of the word compressed eliminates the separator characters, yielding:

```
20170721194244609Z
```

The compressed option simplifies using the output in file names.

Users are cautioned that applications that employ local time may encounter issues across time zones or around time changes, such as daylight saving time (DST).

_fmtdate(): Format a Date/Time from a Millisecond Time Value

The _fmtdate() function formats a date according to a pattern when a date is provided in milliseconds since January 1, 1970, 00:00:00 GMT. It uses the following format:

```
_fmtdate(pattern,value [,language, [,country]])
```

pattern	literal	A type as described for the _timer() function. The specified type controls the formatting of the input value.
value	date/time	A date/time in a format to be formatted based on the pattern.
language	literal	The language code. By default, the language code is set according to the locale setting of the system where iSM is installed.
country	literal	The country code. By default, the country code is set according to the locale setting of the system where iSM is installed.

For example:

`_fmtdate('yyyy.MM.dd', _dateof('MM/dd/yyyy','06/25/2009'))` yields the following:

```
2009.06.25
```

It is sometimes desirable to obtain the date for some moment in relation to the current moment. For example, you may want the current date/time for a date one week in the past:

```
_fmtdate('yyyy/MM/dd hh:mm:ss',_imul(_isub(_timer(),_imul(86400,7)),1000))
```

Replace the 7 in this example with the number of days desired. 86400 is the number of seconds in a day.

For example, to return local time:

```
_fmtdate('dd-MMM-yyyy HH:mm:ss.SSS z',_timer(milliseconds))
```

The language argument is a valid ISO Language Code. These codes are the lowercase, two-letter codes as defined by ISO-639. You can find a full list of these codes at a number of websites, such as *http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt*.

The country argument is a valid ISO Country Code. These codes are the upper case, two-letter codes as defined by ISO-3166. You can find a full list of these codes at a number of websites, such as *http://www.iso.org/iso/country_codes.htm*.

The language and country codes affect the language of the result, not the time zone offset specified by the pattern. The country code is required only in cases in which the language code is ambiguous.

For example, one hour from this moment:

```
_fmtdate('EEE, dd-MMM-yyyy HH:mm:ss zzz', _iadd(_timer(),3600000), 'en')
```

yields the following:

```
Wed, 31-Dec-2012 19:04:00 EST
```

For example:

```
_fmtdate('EEE, dd-MMM-yyyy HH:mm:ss zzz', _iadd(_timer(),3600000), 'fr')
```

yields the following:

```
mer., 31-déc.-2012 19:04:00 EST
```

Note: Date operations can be performed by doing arithmetic on the values with appropriate patterns. The _timer() function returns the current time for this function.

_dateof(): Return the Timestamp for a Passed Time

The _dateof() function parses a date and returns the corresponding number of milliseconds since January 1, 1970, 00:00:00 GMT. It uses the following format:

```
dateof(pattern, value [,zone][,default])
```

pattern	literal	A type as described for the _timer() function. The specified type controls the formatting of the input value.
value	date/time	A date and time in a format to be formatted based on the pattern.
zone	literal	A time zone, such as Zulu.
default	string	Value to be returned if the date and time value (operand 2) is not successfully parsed as a date. Failure to enter this operand will result in an error if a bad date value is detected.

For example:

`_dateof('MM/dd/yyyy', '06/25/2009')` yields the following:

```
1245902400000
```

To create a duration suitable for date arithmetic, use the Zulu zone. This avoids any offset, so that a pattern, such as *ss* will give absolute seconds.

Note: The _iadd() and _isub() functions are preferred for date arithmetic because they operate on long integers with more precision than _add() and _sub().

Compute and display the time ten minutes ago. This example was run at 09/08/27 05:13:54.

```
_fmtdate('yy/MM/dd hh:mm:ss',_isub(_timer(),_dateof('mm',10,gmt)))
```

Results in 09/08/27 05:03:54.

_dateadd(): Add Offset to a Date and Return a Timestamp

The _dateadd() function adds the *offset* parameter to the *date* parameter. If the date and time is not provided, then the current date/time of iWay Service Manager (iSM) is used. This function uses the following format:

```
_dateadd(offset [,date])
```

offset	literal	A number pattern. This pattern describes the number of days, hours, minutes, seconds, or microseconds to add to the date. This pattern has the following definition: `[xxd][xxh][xxm]xx[s]` The value of `xx` represents a numeric value and the suffix is defined as follows: ❏ `d` (Days) ❏ `h` (Hours) ❏ `m` (Minutes) ❏ `s` (Seconds) **Note:** If the value of the offset parameter is entered without the `d`, `h`, `m`, or `s` suffix, then the offset is assumed to be milliseconds. For example: ❏ `1d1h` - Adds one day and one hour. ❏ `15m` - Adds 15 minutes. ❏ `1d30s` - Adds one day and 30 seconds. ❏ `3600000` - Adds 3600000 milliseconds. ❏ `1h10m` - Adds one hour and 10 minutes. ❏ `1h10` - Is a valid pattern (it is missing the suffix on the second numeric value) the seconds [s] are assumed because it is the last numeric entry.
date	date/time	Optional date timestamp. If this value is not provided, then the current date and time of iSM is used.

For example, `_dateadd(1d)` yields the following:

```
1477627200000
```

Note: This is assuming that the date and time of iSM is 10/27/2016 12:00:00 AM.

You may have a use case where you need to perform and display a date calculation on a fixed date (for example, adding a week to 10/27/2016). This may be accomplished by using several iSM functions. For example:

```
_fmtdate('yyyy-MM-dd hh:mm:ss a',_dateadd(7d,_dateof('MM/dd/
yyyy','10/27/2016')))
```

Results in:

```
2016-11-03 12:00:00 AM
```

_datesub(): Subtract Offset to a Date and Return a Timestamp

The _datesub() function subtracts the *offset* parameter from the *date* parameter. If the date and time is not provided, then the current date/time of iWay Service Manager (iSM) is used. This function uses the following format:

```
_datesub(offset [,date])
```

offset	literal	A number pattern. This pattern describes the number of days, hours, minutes, seconds, or microseconds to subtract from the date. This pattern has the following definition: `[xxd][xxh][xxm]xx[s]` The value of `xx` represents a numeric value and the suffix is defined as follows: ❏ `d` (Days) ❏ `h` (Hours) ❏ `m` (Minutes) ❏ `s` (Seconds) **Note:** If the value of the offset parameter is entered without the `d`, `h`, `m`, or `s` suffix, then the offset is assumed to be milliseconds. For example: ❏ `1d1h` - Subtracts one day and one hour. ❏ `15m` - Subtracts 15 minutes. ❏ `1d30s` - Subtracts one day and 30 seconds. ❏ `3600000` - Subtracts 3600000 milliseconds. ❏ `1h10m` - Subtracts one hour and 10 minutes. ❏ `1h10` - Is a valid pattern (it is missing the suffix on the second numeric value) the seconds [s] are assumed because it is the last numeric entry.
date	date/time	Optional date timestamp. If this value is not provided, then the current date and time of iWay Service Manager (iSM) is used.

For example, `_datesub(1d)` yields the following:

```
1477454400000
```

Note: This is assuming that the date and time of iSM is 10/27/2016 12:00:00 AM.

You may have a use case where you need to perform and display a date calculation on a fixed date (for example, subtracting a week from 10/04/2016). This may be accomplished by using several iSM functions. For example:

```
_fmtdate('yyyy-MM-dd hh:mm:ss a',_datesub(7d,_dateof('MM/dd/
yyyy','10/04/2016')))
```

Results in:

```
2016-09-27 12:00:00 AM
```

Math Functions

Math functions enable you to generate configuration parameters and to assist with tests. This section lists and describes the various math functions that you can use in iWay Service Manager (iSM).

Note: By default, math operations are performed in floating point, which can generate problematic results when a value is provided below the radix (for example, 12 in 10.12). Decimal numbers when operated on by the iFL functions *_ddiv()* and *_dmul()*, while slightly slower, operate upon decimal digits as entered. This means that a decimal type is not more precise than a binary floating point or fixed point type in a general sense (for example, it cannot store 1/3 without loss of precision), but it is more accurate for numbers provided with a finite number of decimal digits, as is often the case for monetary calculations. Users are responsible for understanding the use to which numeric values are specified, and the scale and precision required for the final result.

_add(): Add a List of Terms

The _add() function is used to add a list of terms. It uses the following format:

```
_add(term, term*)
```

term	number	Number to be added.

This function returns the sum of the terms. There is nothing implied by the order of the terms.

_sub(): Subtract

The _sub() function is used to return the difference. It uses the following format:

```
_sub(minuend, subtrahend)
```

minuend	number	Number to be subtracted from.
subtrahend	number	Number to be subtracted from the minuend.

_mod(): Returns the Modulus

The _mod() function produces the remainder of dividing the first term by the second. For example, mod(22,6) = 4 because 22 / 6 = 3 with a remainder of 4. This is often written 22%6 in common arithmetic.

The _mod() function uses the following format:

```
_mod(term, modulus)
```

term	integer	Number to be divided.
modulus	integer	Number to be used for testing.

_mul(): Multiply a Number

The _mul() function returns the product of the first factor multiplied by the second factor. Note that multiplication is commutative and therefore there is nothing implied by the order of the factors. The result is computed as a floating point value.

The _mul() function uses the following format:

```
_mul(multiplicand, multiplier)
```

multiplicand	number	Number to be operated upon.
multiplier	number	The multiplier for the function.

_div(): Divide a Number

The _div() function returns the quotient of the first factor divided by the second factor. The result is computed as a floating point value.

The _div() function uses the following format:

```
_div(dividend, divisor)
```

dividend	number	Number to be operated upon.
divisor	number	The divisor for the function. Must be != 0.

_iadd(): Add a List of Terms, Integer

The _iadd() function returns the sum of the terms. There is nothing implied by the order of the terms. Any term can be an XPATH() function. If the XPATH() returns multiple results, all are added into the sum. All values are assumed to be integers, and the result is an integer. The iadd() function is suitable for date manipulation, as the addition is computed with sufficient precision to maintain the field integrity. See *_dateof(): Return the Timestamp for a Passed Time* on page 67 for examples.

The _iadd() function uses the following format:

```
_iadd(term, term*)
```

term	number	Number to be added.

_isub(): Subtract, Integer

The _isub() function returns the difference with integer arithmetic. The isub() function is suitable for date manipulation, as the subtraction is computed with sufficient precision to maintain the field integrity. See *_dateof(): Return the Timestamp for a Passed Time* on page 67for examples.

The _isub() function uses the following format:

```
_isub(minuend, subtrahend)
```

minuend	number	Number to be subtracted from.
subtrahend	number	Number to be subtracted from the minuend.

_imul(): Multiply a Number, Integer

The _imul() function returns the product of the first factor multiplied by the second factor. Recall that multiplication is commutative and therefore there is nothing implied by the order of the factors.

The _imul() function uses the following format:

```
_imul(multiplicand, multiplier)
```

multiplicand	number	Number to be operated upon.
multiplier	number	The multiplier for the function.

_idiv(): Divide a Number

The _idiv() function returns the quotient of the first factor divided by the second factor. The quotient is an integer, with the fractional part disregarded.

The _idiv() function uses the following format:

```
_idiv(dividend, divisor)
```

dividend	number	Number to be operated upon.
divisor	number	The divisor for the function. Must be != 0.

_int(): Cast to Integer

The _int() function casts a value to an integer. This is done by ignoring the fractional part, as is the standard case for computer languages. To avoid loss of information, the _round() performs a similar operation with half-adjust rounding.

The _int() function uses the following format:

```
_int(value)
```

value	number	Number to be cast.

_intmask(): Inserts a Number into a Character Mask

The _intmask() function inserts a number into a character mask. It uses the following format:

```
_intmask(pattern,input)
```

pattern	string	Mask into which the number is inserted. All characters but # appear as-is, but # characters (one set) are replaced by the value.
input	integer	Math value to be inserted. Must be present, if not, it will cause an error.

_max(): Maximum of a List of Terms

```
_max(term, term*)
```

term	number	Number to be evaluated.

Returns the maximum value of the terms. There is nothing implied by the order of the terms. Any term can be an XPATH() function. If the XPATH() returns multiple results, all are evaluated.

_min(): Minimum of a List of Terms

```
_min(term, term*)
```

term	number	Number to be evaluated.

Returns the minimum value of the terms. There is nothing implied by the order of the terms. Any term can be an XPATH() function. If the XPATH() returns multiple results, all are evaluated.

_random(): Generate a Random Number

The _random() function returns a pseudo-random integer between zero and the specified upper bound value (with default 232-1). All possible values are produced with equal probability.

The _random() function uses the following format:

```
_random([upperbound] [,width])
```

upperbound	integer	Specified upper bounds to be used.
width	integer	The number of digits in the result. Leading zeros may be added.

This function is especially useful as a seed for a cryptographic algorithm. It is also useful for simulating server response time distributions using the Delay parameter of the Document Copy service (com.ibi.agents.XDCopyAgent).

The width parameter specifies the formatted width of the result. For example, to generate a random telephone number, you might configure the _random() function as follows:

```
212-555-_random(9999,4)
```

The following is a sample result that is generated:

```
212-555-1786
```

_floor(): Obtain the Floor of a Number

The _floor() function returns the largest (closest to positive infinity) value that is not greater than the argument and is equal to a mathematical integer. Special cases include:

- If the argument value is already equal to a mathematical integer, then the result is the same as the argument.
- If the argument is NAN (not a number) or an infinity or positive zero or negative zero, then the result is the same as the argument.

For example, the floor of -7.5 is -8. The floor of 7.5 is 7.

The _floor() function uses the following format:

```
_floor(number)
```

number	value	Number to be operated upon.

_ceil(): Obtain the Ceil of a Number

The _ceil() function returns the smallest (closest to negative infinity) double value that is not less than the argument and is equal to a mathematical integer. Special cases include:

- If the argument value is already equal to a mathematical integer, then the result is the same as the argument.
- If the argument is NAN or an infinity or positive zero or negative zero, then the result is the same as the argument.
- If the argument value is less than zero but greater than -1.0, then the result is negative zero.

Note: The value of ceil(x) is exactly the value of -floor(-x).

For example, the ceil of -7.5 is -7. The ceil of 7.5 is 8.

The _ceil() function uses the following format:

```
_ceil(number)
```

number	value	Number to be operated upon.

_round(): Round a Number to an Integer

The _round() function returns the closest integer to the argument value. The result is rounded to an integer by adding ½, taking the floor of the result, and casting the result to type integer. In other words, the result is equal to the value of the following expression:

```
(int)Math.floor(a + 0.5f)
```

The _round() function uses the following format:

```
_round(number)
```

number	value	Number to be operated upon.

Decimal Math Functions

In addition to supporting functions that use integer math and floating point math, iWay Functional Language (iFL) also supports decimal math functions. Floating point math is performed using the Base 16 (Hexadecimal) numbering system, which does not accurately represent decimal vales below the radix (decimal point). This results in an inability to rely on exact values with decimal places, which is important when calculating monetary units, such as dollars and cents. If values are developed during computations that are not in the represented domain of decimal numbers, an iFL error is generated.

_dadd(): Add a Number

The _dadd() function returns the decimal sum of the decimal terms. This function uses the following format:

```
_dadd(term1, term2)
```

term1	number	The first number to be added.
term2	number	The second number to be added.

There is nothing implied by the order of the terms. Any term can also be an XPATH() function. If an XPATH() function returns multiple results, then all of the results are added to the sum.

_dsub(): Subtract a Number

The _dsub() function returns the decimal difference. This function uses the following format:

```
_dsub(minuend, subtrahend)
```

minuend	number	The number to be subtracted.
subtrahend	number	The number to be subtracted from the minuend.

Both terms that are used in this function must be decimal numbers.

_dmul(): Multiply a Number

The _dmul() functions returns the decimal product of the first factor multiplied by the second factor. This function uses the following format:

```
_dmul(multiplicand, multiplier)
```

multiplicand	number	The number to be multiplied.
multiplier	number	The multiplier for the function.

Since multiplication is a commutative operation, there is nothing implied by the order of the factors. Both terms must be decimal numbers.

_ddiv(): Divide a Number

The _ddiv function returns the decimal quotient of the dividend divided by the divisor. Attempting to divide by zero results in an error. This function uses the following format:

```
_ddiv(dividend, divisor)
```

dividend	number	The number to be divided. The default scale of the result will be that of the dividend.
divisor	number	The divisor for the function, which must not be equal to 0.

Both terms must be decimal numbers.

Encoding Functions

Encoding functions enable you to decode and convert strings. This section lists and describes the various encoding functions that you can use in iWay Service Manager.

_mod10(): Mod10 Check Digit Operations

The _mod10() function generates or checks a modulus 10 digit. This function uses the following format:

```
_mod10(action,value)
```

action	keyword	Specifies the action to be performed: ❑ **Append.** Add the valid check digit to the value. ❑ **Check.** Validate the check digit within the value (true or false). ❑ **Generate.** Generate and return the check digit for the value (single integer).
value	string	A numeric value to be used for check digit activities.

A modulus 10 check (also known as Luhn's Algorithm) is a simple checksum used to validate a variety of common identification numbers, such as credit card numbers, National Provider Identifiers in the US, DUNS numbers, and so on. The algorithm is specified in *ISO/IEC 7812-1*. It is designed to protect against accidental errors such as simple transpositions rather than malicious attacks. The formula verifies a number against its included check digit.

- **Append.** Generates a check digit on a passed value, and returns the value with the check digit added to the end. For example:

  ```
  _mod10('append',1234567891234567891)   := 12345678912345678918
  ```

- **Check.** Checks the passed in value for validity. The last digit is assumed to be the check digit. Return true or false. For example:

```
_mod10('check,12345678912345678914)  := false
```

- **Generate.** Returns a single digit as the check digit for the submitted value. The entire number is used to generate the check digit. For example:

```
_mod10('generate',123456789123456789)   := 8
```

_url(): Convert String to MIME Format

The _url() function converts the URL string to the application/x-www-form-urlencoded MIME format. For more information about HTML form encoding, consult the HTML specification. The _url() function uses the following format:

```
_url(URLString [,encoding])
```

URLString	string	The string to convert.
encoding	string	IANA encoding for the string.

When encoding the string, the following rules apply:

- The alphanumeric characters “a” through “z”, “A” through “Z” and “0” through “9”, “.”, “-”, “*”, and “_” remain the same.
- The space character " " is converted into a plus sign (+).
- All other characters are considered unsafe and are first converted into one or more bytes using the specified encoding scheme. Then each byte is represented by the 3-character string %xy, where xy is the two-digit hexadecimal representation of the byte. The recommended encoding scheme to use is UTF-8., which is the default.

For example, using UTF-8 as the encoding scheme the string The string ü@foo-bar would get converted to The+string+%C3%BC%40foo-bar because in UTF-8 the character ü is encoded as two bytes C3 (hex) and BC (hex), and the character @ is encoded as one byte 40 (hex).

Note: The World Wide Web Consortium Recommendation states that UTF-8 should be used. Not doing so may introduce incompatibilities. For this reason, UTF-8 is the default encoding regardless of the encoding under which the listener is running.

If the function determines that the passed string is a valid URL, it encodes only the portion following the '?'. This is called the <query> in the URL specification. Otherwise, it encodes the complete string.

For example, the URL *http://localhost:1456?value=1 test=2* will encode to *http://localhost:1456?value=1+test=2*.

_urlencode(): Convert String to MIME Encoding

The _urlencode() function converts the full passed in string to the application/x-www-form-urlencoded MIME format. For more information about HTML form encoding, consult the HTML specification. The string is not checked for URL format. The _urlencode() function uses the following format:

```
_urlencode(String [,encoding])
```

String	string	The string to convert.
encoding	string	IANA encoding for the string.

When encoding the string, the following rules apply:

- The alphanumeric characters "a" through "z", "A" through "Z" and "0" through "9", ".", "-", "*", and "_" remain the same.
- The space character " " is converted into a plus sign "+".
- All other characters are considered unsafe and are first converted into one or more bytes using the specified encoding scheme. Then each byte is represented by the 3-character string "%xy", where xy is the two-digit hexadecimal representation of the byte. The recommended encoding scheme to use is UTF-8., which is the default.

Note: The World Wide Web Consortium Recommendation states that UTF-8 should be used. Not doing so may introduce incompatibilities. For this reason, UTF-8 is the default encoding regardless of the encoding under which the listener is running.

Unlike the _url() function, no effort is made to validate the input string. Instead, it encodes the complete string. For example, *http://localhost:1456?value=1 test=2* will encode to *http%3A%2F%2Flocalhost%3A1456%3Fvalue%3D1+test%3D2*.

_urldecode():Decode a String in MIME Format

The _urldecode() function decodes a string from the application/x-www-form-urlencoded MIME format into standard format for use as a parameter, inclusion in an XML value, and so on. It uses the following format:

```
_urldecode(URL String [,encoding])
```

URLString	string	The string to convert.
encoding	string	IANA encoding for the string.

The conversion process is the reverse of that used by the _urlencode() function. It is assumed that all characters in the encoded string are one of the following: "a" through "z", "A" through "Z", "0" through "9", and "-", "_", ".", and "*". The character "%" is allowed but is interpreted as the start of a special escaped sequence.

If the encoding is not specified, UTF-8 is assumed, in accord with the recommendations as described by the _urlencode() function.

_hex(): Encode a String to Hexadecimal

The _hex() function encodes a string into hexadecimal notation. This function uses the following format:

```
_hex(value [,charset])
```

value	string	The value to be encoded into hexadecimal.
charset	string	The character set represented by the internal Unicode or the value. The default is the system default character set. It is a good idea to specify the actual character set, which is often ISO-8859-1 for usual byte to character operations.

Example:

```
_hex(_replace("ab~c",'~','\0x85') ,ISO-8859-1)
```

The following value is returned:

```
61628563
```

_fromhex(): Decode a String from Hexadecimal

The _fromhex() function decodes a string in hexadecimal notation into the ASCII character set. This function uses the following format:

```
_fromhex(value [,charset])
```

value	string	The value to be decoded.
charset	string	The character set represented by the internal Unicode or the value. The default is the system default character set. It is a good idea to specify the actual character set, which is often ISO-8859-1 for usual byte to character operations.

Example:

```
_fromhex(69776179)
```

The following value is returned:

```
iway
```

_base64():Encode Into Base64

The _base64() function uses the following format:

```
_base64(value)
```

value	string	The value to encode.
encoding	string	The encoding to be used in creating the base64.

The input may be represented in a non-server encoding. To set the encoding for the conversion, the *encoding* parameter must be used.

For example, if you want to transfer the current message (document payload) to a third-party in base64 form, configure the function as follows:

```
_base64(_flatof(),_docinfo('encoding'))
```

_frombase64():Decode From Base64

The _frombase64() function uses the following format:

```
_frombase64(value)
```

value	string	The string to convert.
encoding	string	The encoding to be used in creating the base64.

The passed value is converted from base64 representation to standard notation.

_encode64():Conditionally Encode Into Base64

The _encode64() function uses the following format:

```
_encode64(value)
```

value	string	The string to convert.
encoding	string	The encoding to be used in creating the base64.

If the value requires base64 encoding it is converted to base 64, else it is returned with no conversion. Examples of values that need base64 conversion include those with values lower than 0x20.

If conversion is required, the converted value is enclosed in base64() functional notation.

_decode64(): Conditionally Decode From Base64

The _decode64() function uses the following format:

```
_decode64(value)
```

value	string	The string to convert.
encoding	string	The encoding to be used in creating the base64.

If the input value is enclosed in base64() functional notation. it is converted. Otherwise, it is not changed.

Example 1:

```
_decode64('base64(YWJj)')
```

In this example, the string is decoded as 'abc'.

Example 2:

```
_decode64('abcd')
```

In this example, the string is not decoded since it is not enclosed in the base64 tag.

_fmtdec(): Insert an Integer Into a Pattern Mask

The _fmtdec() function is useful when a value must be in a specific format. It uses the following format:

```
_fmtdec(pattern,intval)
```

pattern	string	Define the string to be created.
intval	integer	Value to be inserted.

The value is inserted into the pattern mask to form a complete result. The mask consists of alphabetic and numeric characters and special symbols as defined for Java formatting. When the value is inserted, the appropriate pattern characters are replaced with the value. For example _fmtdec('ab##.#x',17.3) yields ab17.3x.

_fmtint(): Insert an Integer Into a Pattern Mask

The _fmtint() function is useful when a value for a control number is read from the trading partner manager or another source. It uses the following format:

```
_fmtint(pattern,intval)
```

pattern	string	Define the string to be created.
intval	integer	Value to be inserted.

The integer is inserted into the pattern mask to form a complete result. The mask consists of alphabetic and numeric characters and special symbols. It also should contain one sequence of # characters. When the integer is inserted, the # characters are replaced with the integer. For example _fmtint('ab###x',17) yields ab017x.

_urlparse() Extract Portions of a URL/URI

```
_urlparse(URL String, component [,query_kw [,default]])
```

URLString	string	The string to parse.
component	string	The name of the desired component.
query_kw	string	A keyword to be located in the query portion of the URL.
default	string	Value returned if the query keyword is not found.

The Uniform resource Locator/Identified is parsed in order to extract useful pieces. The components are as described in RFC 2396 Uniform Resource Identifiers (URI): Generic Syntax (*http://www.ietf.org/rfc/rfc2396.txt*).

The component parameter, which is required, can be one of these RFC-identified words:

- protocol
- host
- port
- path
- file
- query (also allows parsing of the query for keyboards)
- authority
- ref
- userinfo

When parsing for the query component, two additional parameters are supported. The first is a keyword to be located, and the second is a default. The keyword is a URL keyword contained in the query. If the keyword is not found, then the default is returned. If the default is not present, then an empty string is returned.

For example:

```
_urlparse('http://www.url.com/look?q=iway','query','q','hello')
```

yields the following:

```
iway
```

If a URL has triggered a flow in the HTTP listener components, then the URL will be in the `url` special register (SREG). In this case, the above example could be:

```
_urlparse(_sreg(url),'query', 'q', 'hello')
```

In addition, there are two more keywords for simplicity of use:

- Context, which means the directory portion of the URL path, excluding leading and trailing slashes. For example:

URL	Context
http://myhost/something/else	something
http://myhost/something/else/again	something/else
http://myhost/	empty string
http://myhost/something	empty string

- Filename, which means the file portion of the URL path. For example.

URL	Filename
http://myhost/something/else	else
http://myhost/something/else/again	again
http://myhost/	empty string
http://myhost/something	something
http://myhost/something/	empty string

Note that file and filename are not synonyms. File returns the URL path plus any query string.

To extract specific portions of the returned information, the _token() function can be used.

_deflate(): Compress (Deflate) a Value

The _deflate() function uses the following format:

```
_deflate(value [,encoding ][,output type] [,algorithm modifier])
```

value	String	The string value to be compressed.
encoding		The character set of the input string. The default is ISO-8859-1.
output type	keyword	The format of the output resulting string. The default is *leadhex*. For more information on the supported output types, see the table below.
algorithm modifier	keyword	The algorithm to be used. The default is *standard*. For more information on the supported algorithm modifier types, see the table below.

A string value (including a flattened XML or JSON object) is compressed using standard *ZIP* algorithms.

The compression result is expressed as a Unicode string in a designated format. This string is appropriate for database updates into a varbinary column, for transmission, or for other storage.

The compression operation first converts the string to a byte representation based on the provided encoding. It then applies compression algorithms, and once compressed, the result is converted back to a string under encoding ISO-8859-1 in a requested format. The default format is *leadhex* (for example, 0x010203...) appropriate for direct insertion into most databases.

The supported output types are listed and described in the following table. The default is *leadhex*.

Output Type	Description
rawhex	The deflated bytes are represented as hexadecimal digits, two per byte.

Output Type	Description

leadhex	The deflated values are represented as hexadecimal digits, two per byte. The result is prepended with the two characters *0x* creating a value appropriate for most database inserts. For example, using the SQL service object (com.ibi.agents.XDSQLAgent) in iIT to generate an insert for a table with two columns, an integer and a varbinary:
	`SQL INSERT INTO MYTABLE (INTCOL, VBCOL) VALUES(%INTX, %VB)`
	might result in the following:
	`SQL INSERT INTO MYTABLE (INTCOL, VBCOL) VALUES(1, 0X1234556788)`
	For more information on setting insert values using the SQL service (com.ibi.agents.XDSQLAgent), see the *iWay Service Manager Component Reference Guide*.

base64	The deflated bytes are represented in base64, with no iSM prefix (for example, 076572dfhe=). Typically, base64 representation results in a shorter string than with hex representation.
func64	The deflated values are represented in base64, encased in iSM base64 marker prefix. For example: `base64(076572dfhe=)`

The supported algorithm modifier types are listed and described in the following table. The default is *standard*.

Algorith m Modifier	Description
standard	The default compression level. This is usually a good match for balancing performance with the size of the compressed result.
fastest	The compression uses fewer resources, but possibly at the expense of compression size.
smallest	The compression results in a smaller result, but may require additional time to complete the operation.

Algorithm Modifier	Description
huffman	An entropic encoding algorithm well oriented to English language text.
none	No compression is performed. This is useful for diagnostic and testing purposes only.

The following is an example of the _deflate function:

```
_deflate (_flatof(),,'base64','smallest')
```

_inflate(): Inflate a Value

```
_inflate(value, type)
```

value	string	The deflated value expressed as a string.
type	keyword	The representation type of the string. The following types are supported: ❑ **string.** Analyze the value looking for type markers (default). ❑ **base64.** The string is encoded in base64, either with or without the base64() markers. ❑ **leadhex.** The value is hex characters (for example, 010a45), starting with *0x*. ❑ **rawhex.** The value is hex characters without the *0x* marker.

The input is assumed to be a string version produced from a deflated message. How the string is created will depend on the input document, but can be expected to be either a base64 value or a string simply made from a byte array. Users are cautioned that if the input is in base64 format, do not attempt to use the _frombase64() function to preconvert the input to string.

The standard representation of a database varbinary column a read in iSM (SQL listeners, XDSQLAgent, and so on) is marked base64. For example:

```
base64(71889875rdj02=)
```

It is therefore in a format that can be recognized without a type operand.

The standard ZIP inflate algorithms are attempted, and if successful the result returned is the inflated string.

Working with BLOBs and Varbinary

Although some databases automatically compress and decompress character data (text or clob columns), others do not. For applications that expect to store large amounts of textual (string) data, the iFL functions _deflate() and _inflate() are available. For example, to store the current document into a nullable varbinary column, the name/value tokens for the insert statement might be:

thetree	_deflate()

The input from a BLOB or varbinary field is returned from iSM readers as framed base64. This can be passed into the _inflate() iFL function, which automatically recognizes the framing and decompresses the information back to the original data string. For more information, see *_deflate(): Compress (Deflate) a Value* on page 88 and *_inflate(): Inflate a Value* on page 91.

File Functions

File functions operate on the file system of iWay Service Manager (iSM). File operations depend on the specifics of the operating system.

_file(): Get File Contents

The _file() function uses the following format:

```
_file(path, [,default[,encoding]])
```

path	string	Path to the desired file.
default	string	Value to be returned if the file does not exist.
encoding	string	IANA encoding.

The specified file is loaded and returned. If encoding is specified, the file is considered to be encoded in the specified IANA encoding. If omitted, the default file system encoding is assumed.

The special encoding of base64 will return the contents of the file encoded in base64. This is especially useful when contents of files, such as PDFs, are to be included as XML values. Placing an _file() function in the XML document and then using the Tree Evaluator Service (com.ibi.agents.EvalWalk) will perform the inclusion.

For example:

```
_file('/mydoc.pdf',,'base64')
```

_filegdg(): Make File Generations

The _filegdg function *pushes down* versions of a file as new versions are created. It is designed to assist in situations in which only a few generations (versions) of a file are required. The _filegdg function uses the following format:

```
_filegdg(sourcefile , generations [,generationpath] [,compare])
```

sourcefile	string	The path to the file to be preserved.
generations	integer	The number of generations to exist.
generation path	string	Path of a directory in which pushed generations will be preserved. The default is the directory in which the source file is found.
compare	keyword	Determines whether the source file should be compared to the prior last generation. The default is *true*. ❑ **true.** If the current version and the prior version are the same, no generation is pushed. ❑ **false.** The new generation is always pushed (created).

Generation Data Groups (GDG) keep identified generations of a file. Frequently this is used to assist during debugging. For example, to keep a limited number of versions of a file to represent the last few changes to the data. The GDG is the *reverse* of the unique file naming facility of iSM in which versions *count up* to the modulus of the unique characters in the name.

For unique naming, a file named *abc####.txt* would create 10000 files and then restart at *abc00001*. In contrast, the GDG with three versions:

```
_filegdg('Abc.txt',3)
```

might create:

`Abc.txt`

`Abc0.txt` (last version)

`Abc1.txt` (prior version)

When the next Abc.txt is written (assuming compress is true), the contents of the existing Abc.txt will be compared to Abc0.txt. If the contents match, then no change will be made. If they do not match, then the existing Abc1.txt will be deleted, Abc0.txt will be renamed to Abc1.txt, Abc.txt will be renamed to Abc0.txt, and a new Abc.txt will be created.

The function returns the source file value, so that your file write component can write into Abc.txt.

The compare option when set to *true* prevents the creation of identical generations, ensuring that only changes are represented. For a debug run in which you need to see the actual results of the specific list runs, you may want to set this option to *false*.

A reasonable use case would be to write the input file using a File Emit service or a QA service, using the _filegdg() as the file name to be written. Thus at a failure (ending the iteration or stopping the channel with a control agent), the output of the debugging service would hold the last three records, given the above example.

Note: The number of generations and the compare option are consistent over all executions. The source file (and backup directory if used) vary by instance execution.

_fileinfo(): Information About a File

The _fileinfo function returns information about a specified file. The information returned depends on the operating system. The _fileinfo function uses the following format:

```
_fileinfo(type , file)
```

type	keyword	The desired information about the file to be returned. Specify one of the following types: ❑ **absolute.** The full, absolute path of the file. A relative path will be converted to an absolute path. ❑ **exists.** Returns *true* if the file exists, else *false*. ❑ **isdir.** Returns *true* if the file is a directory, else *false* if it does not exist or is a file. ❑ **isfile.** Returns *true* if the file is a file, else *false* if it does not exist or is a directory. ❑ **length.** Returns the (approximate) length in bytes of the file. ❑ **locked.** Returns *true* if the operating system reports that the file is locked. ❑ **name.** Returns the file name portion of a full file path specification. ❑ **owner.** ACL owner for the file, if available in the operating system. ❑ **path.** Returns the path portion of a full file path specification. ❑ **moddate.** Returns a Unix date value for the modification date of the file. The _fmtdate() function uses a pattern to convert the Unix time/date to a readable value. For more information on returning timestamps, see *_timer(): Return Unix Epoch Time* on page 63.
file	string	The specified file for which information is to be returned.

As an example, a file called filedata.txt is located in the c:\mydir\ directory. The following _fileinfo() function is executed:

```
_fileinfo(type,' c:\mydir\filedata.txt')
```

The following table lists the returned values for several supported types.

Type	Sample Returned Value
absolute	c:/mydir/filedata.txt
length	55
isdir	false
isfile	true
owner	pc1/pcuser **Note:** The owner information ID is dependent on the operating system.

Time/date values can be converted to more readable formats by use of the _fmtdate() function. For example:

```
_fmtdate('mm/dd/yyyy HH:mm:ss',_fileinfo('moddate','c:/docs/a.xml'))
```

returns:

```
07/20/2015 07:23:44
```

_fileexists(): Does File Exist

The _fileexists() function determines whether a file exists on a specified path. It uses the following format:

```
_fileexists(path [,nametype])
```

path	string	Path to file.
nametype	Keyword	Determines how the file name is being expressed. Specify one of the following types: ❑ **standard.** Name is a file name. ❑ **dos.** Name is a DOS/Windows wildcard specification.

If the file name is a literal or is resolved as a literal, such as the contents of a Special Register (SREG), then that file name is used. If the parameter is a _file() function, then the file name as resolved for the _file() function is used, not the contents of the file. To test for a name in a file, use the _eval() function to evaluate the file operation.

For example, if a SREG called *fname* holds a file name, then the following function is returned as *true*:

```
_fileexists(sreg(fname))
```

Otherwise, this function is returned as *false*.

If the nametype property is set to *dos*, then the file name may contain DOS-type wildcard characters (for example, ? and *). Wildcard characters cannot be used in the path specification itself, only in the file portion (the file name).

For example, the following function returns *true* if any files on the path meet the specification:

```
_fileexists('/lookhere/any*.txt',dos)
```

Otherwise, this function is returned as *false*.

System Information Functions

It is now possible to obtain and work with system information. This section describes the _sysinfo() and _chaninfo() functions.

_sysinfo(): Information About the Server

The _sysinfo() function returns information about the current server. This function uses the following format:

```
_sysinfo(type [,modifier [,modifier2]])
```

type	keyword	The type of information that is to be obtained. You can specify one of the following values: ❑ **processor.** Number of processors available to the server. A common use of the processor value is to regulate parallel operations to avoid over-committing processors and developing a CPU-availability delay. ❑ **version.** Current server version (for example, 7.0.3). ❑ **shortversion.** Current software version that is shortened to the primary release level (for example, 7.0). ❑ **debug.** Determines if the debug trace level is set. Returns *true* or *false*. ❑ **deep.** Determines if the deep trace level is set. Returns *true* or *false*. ❑ **external.** Determines if the external trace level is set. Returns *true* or *false*. ❑ **envvar.** Accesses an environment variable.

Accessing the System Environment Variable (*envvar*)

Most operating systems provide environment variables to pass configuration information to applications. These are not Java system properties, which can be read through the _sreg() function. Instead, these are variables that reside at the operating system level. There are many subtle differences between the ways environment variables are implemented on different operating systems, and variable names are specific to the operating system. For example, variable names on UNIX systems are case-sensitive, while they are case-insensitive on Windows systems. The way in which environment variables are used also varies. For example, Windows systems provide the user name in an environment variable called USERNAME, while UNIX systems might provide the user name in USER, LOGNAME, or both.

To maximize portability, never refer to an environment variable when the same value is available in a system property. For example, if the operating system provides a user name, it will always be available in the system property _sreg('user.name').

To read an environment variable, compose the _sysinfo() function as follows:

```
_sysinfo('envvar',name [,default])
```

The following example is applicable to Windows:

```
_sysinfo('envvar','temp',_concat(_sreg('iwayworkdir'),'/temp'))
```

_chaninfo(): Information About a Channel

The _chaninfo() function returns information about the specified channel. If the specified channel does not exist, an exception is generated when the function is evaluated. This function uses the following format:

```
_chaninfo(name, [,type])
```

where:

name

string

Is the name of the channel. If an asterisk (*) is specified, the current channel in which the function is running will be evaluated.

type

keyword

Is the type of information that is to be obtained. Supported values include:

- **state.** The state of the channel. Possible values include:
  - **active.** The channel is available to process messages.
  - **begin.** The channel is starting.
  - **retry.** The channel is not processing messages and is awaiting a retry cycle.
  - **config.** The channel cannot process messages due to a configuration error.
  - **stopping.** A stop order has been issued. No further messages are being accepted, but the channel has not completed its work.
  - **stopped.** The channel is not active.

- **waiting.** The channel is on a backup server and is not executing.
- **inactive.** The channel was marked inactive in the configuration and awaits an explicit start command.

- **ispassive.** Is the channel passivated. In some cases, such as an internal queue becoming too full, a passivate can be issued to a listener. Many listeners handle passivation automatically. However, some listeners require process flow support. In such a case, the passivate state test will return as true. The process flow may want to issue a sleep loop until the state becomes false, or take another action.
- **workers.** The number of workers (threads) defined for the channel.
- **active.** The number of messages currently being processed by the channel. A common use for this value is to regulate parallel operations to avoid over-committing processor resources.
- **debug.** Determines if the debug trace level is set. Returns *true* or *false*.
- **deep.** Determines if the deep trace level is set. Returns *true* or *false*.
- **external.** Determines if the external trace level is set. Returns *true* or *false*.
- **pending.** Count of messages on the pending queue at the instant that the call is made. For channels that do not support pending, if pending is not configured, or if the specific queue type count is not available, a value of 0 is returned.

  Note: In a multi-threaded channel, the pending count may change unpredictably at any moment as messages are added and removed for execution.

The _chaninfo() function can also be run in a script. For example:

```
if(_chaninfo('ch1','state')='active',stop ch1)
```

Security Functions

Security functions are available to test the state of the current user. As a user logs on, usually through an Authentication Provider, the authority of the current user is encapsulated in a Principal, which identifies the user and the roles (authorities) that the user possesses. For example, an administrative user has the role admin as a standard, but role names are related to services available to the user when that user logs on. Roles are assigned by the security system (Authentication Realm Providers) based on information stored about the user in the appropriate information stores.

_aes(): Encode and Decode a Value Using the Advanced Encryption Standard With Salt

The _aes() function is used to encode and decode a value using the Advanced Encryption Standard (AES) with salt. This function uses the following format:

```
_aes(action,key,data [,keylength,encoding,[,iterations]]
```

action (required)	keyword	The action to be performed. Specify one of the following actions: ❑ **encrypt.** Encrypt the input using a generated salt. ❑ **decrypt.** Decrypt the input by removing the salt.
key (required)	string	The key to be used. Each character encodes 8 bits of the key and therefore must be between 0 and 255 inclusive.
data (required)	string	The input data to be encrypted or decrypted.
keylength (optional)	integer	The size of the key in bits. The following are the sizes that are supported: ❑ **128.** A 128-bit key. This is the default size. ❑ **192.** A 192-bit key, which requires the proper policy files. ❑ **256.** A 256-bit key, which requires the proper policy files.
encoding (optional)	string	The type of encoding that will be used to convert data from characters to bytes. The default is UTF-8.
iterations (optional)	integer	The number of algorithm iterations (1...255). The default is 1.

Advanced Encryption Standard (AES) is an encryption standard adopted by the U.S. government in 2002. Implementations of AES are available in many encryption packages. Details on AES are beyond the scope of this manual but can be found in many sources of cryptographic information.

AES supports key strengths of 128 (the server default), 192, and 256 bits. Due to import-control restrictions imposed by some countries, the default jurisdiction policy files only permit strong cryptography to be used. An unlimited strength version of these files (that is, with no restrictions on cryptographic strength) is available but is not distributed by iWay.

AES is a block cypher that encrypts and then reencrypts. Any number of iterations can be entered, and the more iterations used, the higher the cryptographic strength of the result. However, this must be balanced against the processor overhead.

The key is entered as an iFL string, and can contain up to 16, 24, or 32 characters. Each character must have a value below 256. Use of escaped literals of iRL, such as use of Unicode values or hex values enable entry of complex keys. Keys shorter than the specified lengths will be padded with binary zero.

```
aes('encrypt','iway software','aes')
BtJLII90UBV7wtsrpN8TDw==
_aes('decrypt','iway software','BtJLII90UBV7wtsrpN8TDw==')
aes
```

It may be convenient to store the key in a properties file or a special register. It is recommended that the key not be hard coded in the function call. A common way to do this is to configure a register using the iWay console, or to add to a startup script:

```
set register mykey _concat('secret key\x01')
```

The _concat function is used because the iFL optimizers would not recognize the literal `'secret key\x01'` and would not convert the hexadecimal escape. Using the _concat() function causes the iFL interpreter to evaluate the literal to produce the 11 character key. This will be padded with five binary zeros by the system (assuming 128 bit keys).

Alternatively, you can store the key in the internally masked form used by iSM. You can create a key of this form using the _encr() function, or using the *set property* command. For example:

```
set property keyfile mykey mysecret -encrypt
```

This command will generate a property file that holds the following key:

```
mysecret=ENCR(3237310127231613138296)
```

If this value is loaded into an iSM configuration register during system startup, the value will never appear in its unmasked form. For more information, see the *iWay Service Manager Security Guide*.

_hasrole(): Is This Authority Available

```
_hasrole(name)
```

name	string	The name of an authority to be tested.

The current Principal is tested for the names authority. If the user represented by this Principal has the identified authority the function returns true.

_getprin(): Get Information from This Principal

```
_getprin(keyword)
```

keyword	string	Keyword of which information is to be obtained. **user.** User name. **password.** Password of the user.

The information associated with the current Principal is returned. A common use of this information is to configure an emitter that inherits the login credentials of the current user.

This function returns auto when the principals are not configured on the server and the default user is used. Otherwise, the principal on the channel is returned.

_encr(): Mask the Value

```
_encr(value)
```

value	string	Term to encrypt

iWay Service Manager (iSM) uses a simple cryptographic mechanism to mask passwords stored in its configuration files. The algorithm employs random seeds and salting when generating the encrypted result. The result is marked with functional braces for recognition by the internal decryption services when the value needs to be used.

iWay strongly recommends that this function not be used to protect values in business systems. Facilities to use validated PKI and session key cryptography are readily available for this purpose. The use of this function should be restricted only to password masking and similar purposes. For example:

```
_encr('iway')
```

This command will generate the following masked value:

```
ENCR(3157318131043128321832252993249)
```

Key generation for functions such as AES encryption can use masked values, which are unmasked only when used. A masked value can be written to a property file by using the *set property* command. Often that value is loaded during startup into a general register, where it continues to be carried in masked form. For more information on using the *set property* command, see the *iWay Service Manager User's Guide*.

_md5(): Generate an MD5 Hash

```
_md5 (term [,term*])
```

term	string	A value to be added to the hash.

In cryptography, MD5 (Message-Digest algorithm 5) is a widely used cryptographic hash function with a 128-bit hash value. MD5 confirms to an Internet standard (RFC 1321). MD5 has been employed in a wide variety of security applications, and is also commonly used to check the integrity of files. An MD5 hash is typically expressed as a 32-digit hexadecimal number. Unlike functions, such as _uuid() that generate unique numbers, an MD5 hash will produce the same value given identical input. The iWay functional language enables generation of an MD5 hash of from one to nine terms.

```
_md5('username','realm','password')
```

returns

```
66999343281B2624585FD58CC9D36DFC
```

A standard use of MD5 is in digest authorization in HTTP. In this case, the username, password, a realm name and a set of random values called nonces are used to generate the hash.

Commonly in iWay, it is useful to add a hash value to a message or to check it on receipt. The _md5 function can help with this requirement.

_sha1(): Generate a SHA1 Hash

The Secure Hash Algorithm (SHA) hash1 function is a cryptographic hash function designed by the National Security Agency (NSA) and published by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard.

The _sha1() function uses the following format:

```
_sha1(term [,term*])
```

term	string	Terms to include in the SHA1 computation.

Although some concern has been raised about the absolute cryptographic security of the SHA1 algorithm, it remains a commonly used hash for securing the value of data.

For example:

```
_sha1('name','digest','password','1234567')
```

The following is returned:

```
95e760b78aaa4ccca9ac94b8815e753674bafaa7
```

_sha256(): Generate a SHA256 Hash

The Secure Hash Algorithm (SHA) hash256 function is a cryptographic hash function designed by the National Security Agency (NSA) and published by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard.

The _sha256() function uses the following format:

```
_sha1(term [,term*])
```

term	string	Terms to include in the SHA256 computation.

For example:

```
_sha256('name','digest','password','1234567')
```

The following is returned:

```
01598ead43e67a3f57bb6a899c62b0874406142db2f0039d1aeba2bec39901a9
```

Other Functions

This section lists and describes other functions that you can use in iWay Service Manager (iSM).

_excel(): Get Value From a Workbook Spreadsheet

The _excel() function returns a value from a workbook spreadsheet.

Note: The _excel() function requires the Excel extension to be installed.

The _excel() function uses the following format:

```
_excel(workbook, sheetname,keycol, dif, keyval, valcol [,default] [,keep]
[,evaluate])
```

workbook	string	The path to the workbook.
sheetname	string	Name of the sheet in the workbook. If omitted, the default is *Sheet1*.
keycol	sheetcol (see below)	The column containing the desired property key.
dif	boolean	Determines whether this sheet is compliant with DIF format. If set to *true*, then the property key search starts in row 2, else it starts in row 2. **Note:** As a convenience the keyword *dif* is equivalent to *true*. The setting of this parameter cannot vary based on the input document.
keyval	string	The property name in the keycol.
valcol	sheetcol	The column containing the desired value.
default	string	Value returned if the keycol/valcol cannot be determined or located.
keep	keyword	The keyword to control workbook load operation: ❑ **check.** Check for file modification. (default) ❑ **keep.** Do not check for file modification and retain the worksheet in memory. **Note:** The setting of this parameter cannot vary based on the input document.

evaluate	keyword	The keyword to control the result evaluation: ❑ **evaluate.** Evaluate the result as iFL. Allows iFL to be held in a cell of the worksheet. ❑ **constant.** Do not evaluate. (default) **Note:** The setting of this parameter cannot vary based on the input document.

Similar to the _property() function, a key is evaluated and the value returned. The spreadsheet, however, can provide data anywhere on a sheet, and sheet relationships are supported. For example, a formula in a value column might reference another position or sheet position in a logical hierarchy meaningful to an application. For example, separate sheets within a single workbook might hold values for different customers, message types, and so on.

Once a *keycol* is determined, the rows of the sheet are interrogated to locate the row containing the *keyval*. Once the row is determined, the *valcol* on that row is accessed to obtain the desired value.

A *sheetcol* is used to locate the column of interest. It may be an integer, a column ID (if not DIF format) or a meaningful value in the header row (*dif* is set to *true*) such as *dev* or *prod* enabling multiple sets of properties to be stored together. For example, in the sample spreadsheet, formulas are used to construct simple interrelationships. Other application architects might elect to keep values in different spreadsheets, with the relationships between the spreadsheets arranged in a meaning property hierarchy. The _excel() function attempts to resolve formulas, but users are cautioned that it is possible to construct formulas that _excel() cannot evaluate. Constructing valid spreadsheets that can be evaluated is the responsibility of the application architect.

No provision is made to store a value in encrypted form, however encrypted values can be created using the *set property* command and the value copied to the spreadsheet.

Example 1:

A single sheet table with columns representing the current development status is shown in the following image.

	A	B	C	D	E
1	property	dev	uat	prod	
2					
3	portbase	=D3+200	=D3+100	1250	
4	editransform	=D4	=D4	tran850	
5					
6					
23					
24					

app1 app2 Sheet3

Assume a special register *where* is set to *uat*. The transform for *editransform* might be entered as follows:

```
_excel('/excel/users.xlsx','app1','property',dif,
editransform,_sreg(where),'asis')
```

This function assumes an imaginary transform that is used for debugging. The function will search the *app1* spreadsheet to locate the row that has *editransform* in the property column. If found, then it returns the value in the *uat* column. If not, then it returns *asis*.

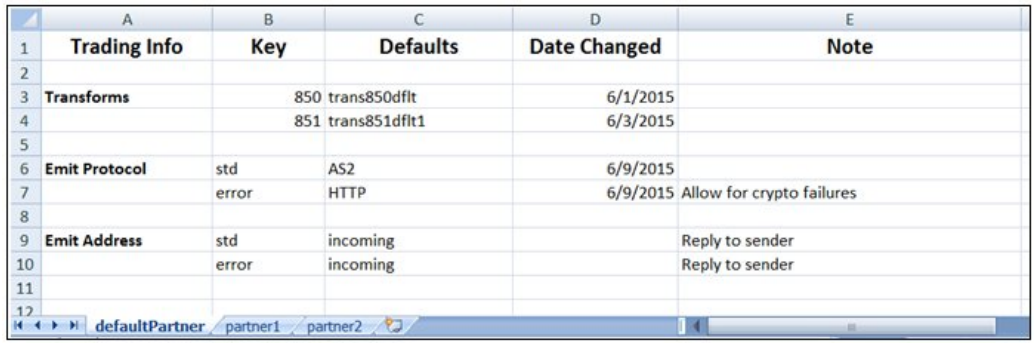
The following is the equivalent for raw addressing:

```
_excel('/excel/users.xlsx','app1','A', false, editransform,'C','asis')
```

The form of addressing used depends upon application needs.

Example 2:

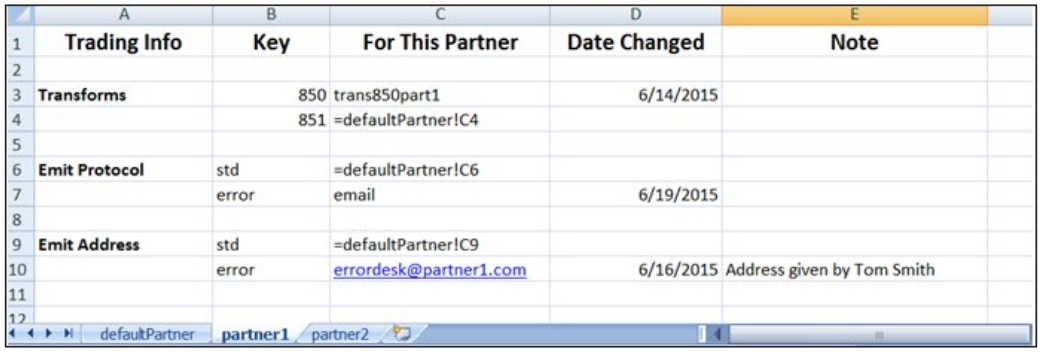
A multicolumn workbook holding information about specific or general EDI trading partners. This example uses the _excelsheets() function described. In iWay Service Manager, the EDI facilities provide a special register *frompartner* holding the name of the source partner in the EDI message. The names of the worksheets in the workbook represent specific trading partner names that do not use the general defaults entered in sheet *defaultPartner*, as shown in the following image.



	A	B	C	D	E
1	**Trading Info**	**Key**	**Defaults**	**Date Changed**	**Note**
2					
3	**Transforms**	850	trans850dflt	6/1/2015	
4		851	trans851dflt1	6/3/2015	
5					
6	**Emit Protocol**	std	AS2	6/9/2015	
7		error	HTTP	6/9/2015	Allow for crypto failures
8					
9	**Emit Address**	std	incoming		Reply to sender
10		error	incoming		Reply to sender
11					
12					

defaultPartner | partner1 | partner2

The sheet for partner1 might be as shown in the following image.



	A	B	C	D	E
1	**Trading Info**	**Key**	**For This Partner**	**Date Changed**	**Note**
2					
3	**Transforms**	850	trans850part1	6/14/2015	
4		851	=defaultPartner!C4		
5					
6	**Emit Protocol**	std	=defaultPartner!C6		
7		error	email	6/19/2015	
8					
9	**Emit Address**	std	=defaultPartner!C9		
10		error	errordesk@partner1.com	6/16/2015	Address given by Tom Smith
11					
12					

defaultPartner | partner1 | partner2

Note that cells of partner1 can refer to other sheets, in this example the *defaultPartner* sheet.

To locate the transform for an EDI 850 message for partner1:

```
_excel('/
info.xslx',_if(_inlist(),_sreg(frompartner),'defaultPartner')),'key','dif',
850,'For This Partner','trans850default')
```

The following table lists and describes this command structure.

Example	Explanation
`_excel('/info.xslx',`	Specify the name of the workbook.
`_if(inlist(_excellist('/ info.xslx',_sreg(frompartner),'string') , sreg(frompartner)), 'defaultPartner')),`	If the incoming partner is in the list of special partners (sheets), then this name is used as the sheet name, else the default is used.
`'key',dif,`	The column in which you look for the specific data row that you require.
`'850'`	The transform name you want to locate.
`'For This Partner',`	Where you get the data that you require.
`'trans850default'`	Just in case.

The result returned will be *trans850part1*.

_excelsheets(): Get the List of Workbook Worksheets

The _excelsheets() function returns a list of the worksheets in a specified workbook. The returned list can be used in the _inlist() iFL function, as shown in the example under _excel().

Note: The _excelsheets() function requires the Excel extension to be installed.

The _excelsheets() function uses the following format:

```
_excelsheets(workbook [,keep])
```

workbook	string	The path to the workbook.

keep	keyword	The keyword to control workbook load operation: ❑ **check.** Check for file modification. (default) ❑ **keep.** Do not check for file modification and retain the worksheet in memory. **Note:** The setting of this parameter cannot vary based on the input document.

_fetch(): Access a Remote Library

The _fetch() function returns specified items (components) from a remote library. The library is a running iSM configuration.

A common use is to access a subflow for a specific customer or situation. The flow is not built into the application, avoiding the need to rebuild as each new customer is added. Some application designers refer to this as an external exit.

The _fetch() function uses the following format:

```
_fetch(category , itemname [,libraryName] [,user] [,password]
[,remotehost:port])
```

category*	string	Category of item to return: ❑ **pflow.** Deployed system process flow. ❑ **xslt.** An XSLT template. ❑ **transform.** An iWay transform.
itemname*	string	Name of the item to return. For a process flow this is the name of a deployed system flow in the configuration. For an iWay transform this is the transform name as published to the library.
libraryName	string	Name of the library. This is the name of a configuration to which the items have been deployed. The default configuration is *base*.

user	string	User name with access to the library. The default is the delivered security profile *iway*.
password	string	Password for the user name with access to the library. The default is the delivered security profile *iway*.
host:port	string	The host name and optionally the port number for a remote library. If the port is omitted, then the default port 9999 is used. If the parameter is omitted, then the access is to a local library configuration. This is the port to the master configuration, usually *base*, on the host. It is not the port for the library configuration should these values differ.

Do not use the _fetch() function to fetch an item from your own application. Use this function only to fetch from a designated library configuration. The component must have been properly deployed to the library.

Example 1:

The desired process flow is deployed to the base configuration in this installation. The configuration is running under the default user credentials.

```
_fetch('pflow','customer456')
```

Example 2:

A configuration named *library* has been deployed on a remote system with modified credentials.

```
_fetch('pflow',_xpath(/root/
customerid),'library'_sreg('libuser'),_sreg('libpswd'),'libhost:9999')
```

Example 3:

The Transform service is configured to run a customer-specific transform deployed to the library in Example 2.

```
_fetch('transform',_xpath(/root/
customerid),'library'_sreg('libuser'),_sreg('libpswd'),'libhost')
```

_manifest(): Read an Attribute From a Java Archive (JAR) Manifest

A Java Archive (JAR) manifest contains information that is added when a .jar file is built. There can be standard or application-specific information contained in a manifest. This function makes the value of a manifest variable available to the application.

The _manifest() function uses the following format:

```
_manifest(jarname, attribute, [,default])
```

jarname	string	Name of the .jar file from which the manifest is to be read. A relative path is resolved to the classpath.
attribute	string	Name (case-sensitive) of the attribute to be read.
default	string	Value to be returned if the attribute is not present or has no value.

For example, you can compose the _manifest() function as follows to determine the build date for this version of the iwcore.jar file:

```
_manifest('iwcore','Built-On')
```

_parmof(): Get Parameter Setting From Another (Component) Parameter

It is sometimes useful to set the parameter of one component (for example, a listener), called the *target*, to reflect the setting of another component, called the *source*. This can, for example, allow a target listener to reflect (shadow) a source listener. The configuration parameter is taken from the settings of the source, not the current runtime value. It is immaterial whether the source component is actually in execution. The setting will be evaluated on the target as if the source setting had directly been present on that target.

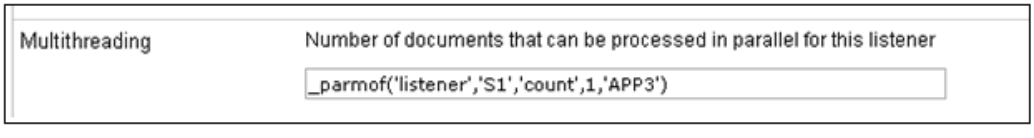
The source can be defined in the runtime dictionary of the same or another configuration (iWay Integration Application (iIA)).

The _parmof() function uses the following format:

```
_parmof(sourcetype, sourcename, key [,default][, configuration] )
```

sourcetype	keyword	The type of component to provide the information (need not be the type of the target component): ❑ **listener.** The value is to be taken from a listener definition. ❑ **system.** The value is to be taken from a system (server) definition.
sourcename	string	The name of the component. Not applicable to *system* requests.
key	string	The (internal) name of the parameter.
default	string	Value to return if the parameter is not found in the source.
configuration	string	Name of the configuration in which the source is defined. The default is the current configuration.

As an example, set the number of workers (threads) for a channel to match the number in a channel named S1 in an iIA named APP3:



_script(): Invoke Scripts

The script iFL allows you to invoke scripts within your process flows and use the results as required. For example, you can compute a configuration value assign the results to a special register to use later on within the process flow. The iFL function's call format is:

```
_script(script[,[scriptFunctionName][,p1[,p2[,...pN]]])
```

Parameter	Description
script	The script file location.
scriptFunctionName	Name of the function within the script to call.
p1 ... pN	The script parameters.

For example:

```
function addit(val1, val2)
{
  return Number(val1) + Number(val2);
}

if(typeof(value1) != "undefined" && typeof(value2) != "undefined")
{
  document.write('the value '+value1+'+'+
value2+'='+addit(value1,value2)+'');
}
JavaScript - f.9
```

The following iFL call:

```
_script(c:/scripts/f9.js,,'value1=100','value2=156')
```

returns the following output:

```
'the value 100+156=256'
```

On the other hand, the following iFL call to the same JavaScript:

```
_script('c:/scripts/f9'.js,addit,1000,24)
```

returns the following output:

```
'1024.0'
```

Notice the differences between the two outputs. The first iFL call did not include the function name, so the script was evaluated and returned the HTML document that would be generated. The second call executed only the function 'addit' and returned the function's results.

_scriptlist(): Generate a Scripting Array

The script iFL also provides the following convenience function:

```
_sciptlist(...)
```

This function is used to generate the scripting array object, for example, `_scriptlist(53000.00,4.5,30)` which produces an array with three elements. The first element being *53000.00*, the second is *4.5*, and finally, *30*.

If the script calls for a stack (Last In First Out array whose elements are pushed in and popped out), the `_sciptlist(...)` function can used to generate it as well. Taking the previous example the first element popped out of the stack would be *30* followed by *4.5* and lastly *53000.00*.

The method `_scriptlist` may only be called within the `_script` function.

_eval(): Evaluate a String

The _eval() function evaluates a string as an expression of the function. The use of this tracing facility is to help debug by reporting the setting of parameters that are set by iFL.

The _eval() function uses the following format:

```
_eval(expression [,tracemsg [,level]])
```

expression	string	The string to be evaluated.
tracemsg	string	A trace message to be issued when the expression is evaluated.
level	keyword	The trace level specified for the tracemsg attribute. The following trace levels are supported: ❑ **none.** Does not return any traces. ❑ **error.** This setting provides error level traces. ❑ **debug.** This setting provides debug level traces (default). ❑ **deep.** This setting provides deep level traces.

A common use of the _eval() function is to store a complex expression in a file. The expression can be used by _eval(_file(<path>)). Assume that the file /myfilescfg.txt contains the simple expression _sreg('iway.config','none'). If the _file('/myfilescfg.txt') function is used alone, then the value will be _sreg('iway.config','none'), the value in the file. However, by using the _eval() function, the _sreg() is evaluated and the result is the name of the configuration in which the server is running.

An optional tracing service adds a message at the specified trace level (if enabled) to the current trace log. This is useful for debugging the value to be output by this function. By including the special token (%v) in the trace message, the expression value can be included in the message. For example:

```
_eval('file(/holdifl.txt)','eval got %v','deep')
```

_log(): Write a Message to the Trace Log

The _log() function writes a message to the trace log. This function uses the following format:

```
_log(tracemsg [,level] [,expression]])
```

tracemsg	string	A trace message to be issued when the expression is evaluated.
level	keyword	The trace level specified for the tracemsg attribute. The following trace levels are supported: ❑ **none.** Does not return any traces. ❑ **error.** This setting provides error level traces. ❑ **debug.** This setting provides debug level traces (default). ❑ **deep.** This setting provides deep level traces. ❑ **info.** This setting provides info level traces.
expression	iFL	An iFL expression (default is true).

The tracing service adds a message at the specified trace level (if enabled). This is useful for debugging the value to be output by this function. By including the special token (%v) in the trace message, the expression value can be included in the message.

For example, the following function produces the specified message:

```
_log('Code reached point one','deep')
```

As an additional example, assume that special register (SREG) A contains the value 12345. Consider the following function:

```
_log('Code reached point one with %v','deep',sreg(a))
```

The output will be:

```
Code reached point one with 12345
```

A use of this function is to display intermediate values in a complex iFL expression. The results of the expression are returned as the value of the function. This allows a _log() function to be cascaded in a complicated expression. When used with a third parameter (the iFL expression), the function is idempotent.

The _log() function differs from the _eval() function in that _eval() is designed to construct the expression to be evaluated. An example is to read an iFL expression from a file and then execute it at runtime. The _log() function does not perform the delayed evaluation, rather the expression is simply part of the overall iFL to be evaluated.

_cond(): Perform Conditional Test

The _cond() function uses the following format:

```
_cond(expression,operator [,operand])
```

function	string	First operand.
operator	string	Operator to be applied.
operand	string	Operand for comparison operators.

The first parameter is a string, often obtained from some other function. This value is operated upon as ordered by the second parameter, possibly using the optional third parameter to complete the test. The result of _cond() is true or false.

Operator	Purpose	Value Used

eq, =, ==	Pure equality. Strings are compared case sensitively.	Yes
eqc	Equality, case-insensitive.	Yes
ne, !=	Not equals.	Yes
lt, <	Less than, numeric, or lexical.	Yes
gt, >	Greater than, numeric, or lexical.	Yes
le, <=	Less than or equals, numeric, or lexical.	Yes
ge, >=	Greater than or equals, numeric, or lexical.	Yes
istrue	Returns true if the result of the expression is true.	No
isfalse	Returns true if the result of the expression is false.	No
isempty	Returns true if the result of the expression exists but has no value. This is useful for testing the return from an xpath operation.	No
isnotempty	Returns true if the result of the expression exists and has a value. This is useful for testing the return from an xpath operation.	No
isnull	Returns true if the result of the expression does not exist.	No
isnotnull	Returns true if the result of the expression exists.	No

_xquery: Evaluate an XQuery Expression

```
_xquery(expression)
```

expression	string	Expression in XQuery language.

The _xquery() function is used to evaluate an XQuery 1.0 expression against the current document. XQuery can be used to select portions of the document and to compute new values in powerful ways. The result is the return value of the function.

The XQuery language is documented in XQuery 1.0: An XML Query Language available at *http://www.w3.org/TR/xquery/*. Notice XQuery 1.0 is a strict superset of XPath 2.0.

The expression argument is treated as a special literal. Functional replacement is performed by iFL but math operations, quotes, and top-level commas are ignored. This makes it easier to pass non-alphanumeric characters to the XQuery interpreter.

The result of evaluating an XQuery expression is a result sequence. The return value of _xquery() is the value of each result sequence item separated by |.

For example, when the expression _xquery(//e[@a="1"]) is applied to the following document:

```
<root>
<e a="1">one</e>
<e a="1">uno</e>
<e a="2">two</e>
</root>
```

the result is “one|uno”. Notice how the XQuery expression does not need special quoting.

Functional replacement is applied before the XQuery interpreter is called. For example, assume the special register reg1 has the value 1, then the expression:

```
 _xquery(//e[@a="sreg(reg1)"])
```

returns the same value as the previous expression when applied to the same document.

An XQuery expression can declare variables to be external. For example, the following expression declares the external variables $v1:

```
_xquery(declare variable $v1 external; //e[@a=$v1])
```

The initial value of an external variable is the value of the special register with the same local name. For example, the last expression returns the same result as the previous examples when the special register v1 has the value 1. An external variable may be declared in a namespace. The namespace is ignored when choosing the special register name. For example, the special register corresponding to the external variable $ns:v2 is simply v2.

_exists(): Does Value Exist

The _exists() function uses the following format:

```
_exists(statement)
```

statement	string	XPath statement of other object.

An attempt is made to determine whether the object exists. If the path parameter starts with a /, this is presumed to be an xpath expression and if the statement locates a value this function returns true.

Otherwise, the object is assumed to be some other internal object such as a special register, and its existence is tested.

_exists1(): Does Value Exist

The _exists1() function uses the following format:

```
_exists1(expression [,nsmap [,object]])
```

expression	string	Expression in xpath language.
nsmap	string	Name of a namespace map from a provider. If omitted, no namespace map is applied.
object	document	A document to which xpath is applied. If omitted, the current document is evaluated.

An attempt is made to determine whether the object exists. If the path parameter starts with a /, this is presumed to be an xpath expression and if the statement locates a value this function returns true.

Otherwise, the object is assumed to be some other internal object such as a special register, and its existence is tested. The use of the parameters is as described in the xpath() function. Exists1() is compatible with xpath1() which is the full xpath(). Use of standard xpath() functions is recommended over use of the _exists1() function.

_isendpoint (): Create Special Registers for Placeholders

This function includes the functionality described in ATE-155 and creates special registers for any placeholders in the endpoint pattern.

For example, if the endpoint pattern is:

```
/pet/{petId}
```

and the incoming URL is:

```
/pet/1234
```

The function will create a special register named *petId* with value *1234*. If the ns argument were set to *pets*, the register would be named *pets.petId*.

Note the following arguments of the function:

- **Action.** The HTTP method of the incoming request. When running with an NHTTP listener, this will normally be `_sreg(reqType)`.
- **Url.** The URL for the incoming request. When running with an NHTTP listener, this will normally be `_sreg(url)`.
- **Routeaction.** The action associated with a particular route, to match with the incoming action.
- **Endpointpattern.** A URL pattern associated with a particular route, to match against the incoming URL.
- **Type.** The type of the URL pattern. Default (and only current option) is RAML.8.
- **Ns.** A namespace prefix to use when creating special registers from placeholders in the URL pattern.

_iwexists(): Does Value Exist

```
_iwexists(statement)
```

statement	string	XPath statement of other object.

An attempt is made to determine whether the object exists. If the path parameter starts with a `/`, this is presumed to be an xpath expression and if the statement locates a value this function returns true.

Otherwise the object is assumed to be some other internal object such as a special register, and its existence is tested.

Use the _fileexists() function to test for the existence of a file. Use the _exists() function to test with full xpath.

_ldap(): Get LDAP Contents

The _ldap() function uses the following format:

```
_ldap(filter, attribute[[,context], provider])
```

filter	string	Search filter in the LDAP.

attribute	string	Attribute to be accessed from the repository.
context	string	Context to be applied to the search.
provider	string	Name of any LDAP provider

The value of the attribute is loaded from the LDAP. Some servers support only a single LDAP directory specification (URL, access ID, password) and this default LDAP directory is used to satisfy this function. Some servers support multiple LDAP providers. In this case, the optional provider name can be supplied.

For example, Dick Beck is a member of the corporate group (in LDAP terms the Organizational Unit). A call to look up his telephone number might be as follows:

```
LDAP('CN=Beck, Dick',phoneNumber,'ou=COR')
```

The actual format of the function call will depend upon the schema used to organize the directory.

_if(): Obtain Value Conditionally

The _if() function uses the following format:

```
_if(test [, trueclause [,falseclause]])
```

test	condition	A conditional test, such as sreg(abc)<6 or _fileexists('c:/abc.txt).
trueclause	string	A value to return if the test condition evaluates to true.
falseclause	string	A value to return if the test condition evaluates to false. If omitted, an empty value is returned.

The test is evaluated, If the test results in a true condition, the true clause is returned, else the false clause is returned. This is useful to set a value in a configuration based on a test.

The clauses can themselves be tests. For example, imagine two special registers, aa and bb each holding a value. The following expression is legal:

```
_if(sreg(aa)<8,_if(sreg(bb)=9,'tt','tf'),'f')
```

If the test is not followed by a trueclause or falseclause, the function returns the tokens true or false. For example, assume that special register t1 contains 15, then the following returns as true:

```
_if(sreg(t1)<20)
```

_lock(): Obtain Value Under Lock

The _lock() function obtains a value that is currently under a lock. This function uses the following format:

```
_lock(lockname, value)
```

lockname	string	The name of the lock under which the value is computed and returned. The specified name is arbitrary, but all users of this lock (for example, any user that wants to update a register) must use the same name. The lock is created under this name if it does not exist. If it exists, then other users of this name are queued until they reach the head of the queue on the name. When the final user is complete (end of the operation under the lock), the named lock is destroyed to free resources.
value	string	The value to be returned.

The value is determined while the named lock is held. This function is designed for use in returning values of high-level register, which may be shared with other threads (for example, metrics that hold statistics). The value for the *lockname* parameter must be the same as the name of the lock under which the higher level registers are set or computed by the SREG service (com.ibi.agents.XDSREGAgent).

Metrics can be referenced as special registers. The lock is desirable when one or more of the registers might be changed by an SREG service (com.ibi.agents.XDSREGAgent) while the value of that register is being accessed to compute the value.

```
_lock('my.lock',_if(sreg(aa)<8,_if(sreg(bb)=9,'tt','tf'),'f'))
```

The lock serializes within the server. For example, if a counter is to be updated across workers (subchannels) then the register used to keep the count must be defined at a higher level (for example, the channel level) and the lock will ensure that the counter remains valid. For example, if the channel register *mycounter* is defined as having a value of zero, then the following statement will safely update the register holding the counter:

```
_lock('lockMyCounter',_setreg('mycounter',_iadd('_sreg('mycounter'),1)
```

The _setreg() function sets the *mycounter* register to the value currently in the register plus one.

As a general rule, locks should be as granular as possible. Avoid using a single lock name for all locks of any purpose. The lock should be unique to the resource being protected under the lock.

_jdbc(): Get A Relational Value from a Table

One value is returned from the JDBC data source identified by the provider. The statement is expected to be an SQL select statement, or a {call} statement. It in turn is expected to return a single value. If the statement returns multiple rows, only the first row is accessed. If the statement returns multiple fields, only the first is returned. These situations are not considered errors.

The _jdbc() function uses the following format:

```
_jdbc(provider, statement [,timeout])
```

This function is useful for extracting a value from a table with keys. For example:

key	value
one	first
two	second

Now assume a special register named sequence which holds ‘one’ or ‘two’. The following function statement will return the correct value:

```
_jdbc(provname,_sql(select value from table where key = _sreg(sequence)))
```

Note the use of the _sql() function to assist in the analysis of the statement. This is not necessary unless iWay functions are used in the expression.

_unq(): Generate a Unique Identifier

The _unq() function uses the following format:

```
_unq(pattern)
```

pattern	string	Descriptive pattern

Returns a unique identifier within the bounds of the supported pattern. A pattern consists of literal characters, such as ID, plus trigger characters that are replaced with values by the server. The supported trigger values are:

Character	Use	Restriction
#	Stored digit	None
^	Unstored digit	None
*	Current timestamp in RFC 1123 format with non-path characters removed.	One per pattern

Stored digits survive the restart of the server, while unstored digits are reset to zero each time the server starts. The number of pattern digits defines the modulus of the generated number. For example, a pattern of ab### returns ab001, ab002...ab999. The function then returns ab001 on the next call.

The pattern support is identical to that used for unique file names in other server configuration parameters.

_uuid(): Generate a Unique Identifier

Returns a UUID (Universally Unique Identifier), also known as GUIDs (Globally Unique Identifier) meeting the requirements of RFC 4122. A UUID is 128 bits long, and is guaranteed to be unique across space and time. The method of generation leverages the unique values of IEEE 802 MAC addresses to guarantee uniqueness. The optional code parameter controls the format of the generated result.

The _uuid() function uses the following format:

```
_uuid([code])
```

Code	Use	Example
none, 0 or standard	Display format.	`d71648c0-1485-11dc-a269-0019b92fe248`
1 or compressed	Compressed format.	`3e2246d0148211dc957b0019b92fe248`

_savedoc(): Save a Document or its Payload for Later Restoration

The _savedoc() function allows the current document and (optionally) its state to be held while the document is changed for some purpose. The current document is the document flowing through the process flow at any moment. The _restoredoc() function is used to return the current document to the value set when it was stored in the register. Registers holding documents will not be marshaled for exchange through the gateway (RVI) or passed to an internal channel. The save and restore sequence should not be used across threads in the process flow.

The _savedoc() function is especially useful during pre- and post-service execution in process flows.

Note: Since saving a document requires memory, this facility should be used with caution to avoid an impact on performance.

The _savedoc() function uses the following format:

```
_savedoc(name, {action} [,scope] [,serialization)
```

name	string	Name of a special register to hold the saved information.
action	keyword	Determines what data should be saved for later restoration. The following actions are supported: ❑ **document.** Saves the entire contents of the input document, including status flags, attachments, and so on. A value of *true* is returned if successful. ❑ **payload.** Saves the contents of the document payload (for example, XML tree). A value of *true* is returned if successful. For more information on how to recover the saved document information, see *_restoredoc(): Restore a Saved Document* on page 129.

scope	keyword	The scope of the specified register. The query on the former value is performed at this scope. The following scopes are supported: ❑ **local.** The scope is the local register context. This scope is set by default. ❑ **flow.** The scope is the head of the process flow. ❑ **message.** The scope is the message (worker).
serialization	keyword	Determines how the saved data is stored. Specify one of the following settings: ❑ **internal.** The saved information is stored in memory. This is the default setting. ❑ **external.** The saved information is stored on disk.

Specifying a *local* scope requires the restore of the data (_restoredoc()) to be on the edge line on which the data was saved, or an edge that descends from the edge. Specifying *flow* or *message* scopes allows the _restoredoc() function to be used on another thread line, or even in a subsequent stage of the message handling. Use of these scopes requires careful management of the edge execution (thread management) to ensure that the data was saved before it is restored. As a best practice, specifying *local* scope is recommended.

Serialization settings provide a balance between memory and performance. Specifying the *internal* setting (default) causes the data to be held in memory. For very large documents that are to be saved, specifying the *external* setting causes the data to be held on a disk. This reduces memory use at the cost of significant use of system resources and time to exchange the data with the disk storage. The system maintains the disk storage and the data is only available to the _restoredoc() function.

Example 1. To save the current message:

```
_savedoc('docpoint1','document')
```

Example 2. To save the payload to disk on the local scope:

```
_savedoc('docpoint1','payload',,'external')
```

Example 3. If you are running more than 10 messages on the channel use external media, otherwise use memory:

```
_savedoc('docpoint1','document',,_if(_chaninfo(*,'active')>10,'external','internal'))
```

_restoredoc(): Restore a Saved Document

The _restoredoc() function is designed to be paired with a _savedoc() function that saves a current document or its payload. Using the _savedoc() and _restoredoc() functions provides an efficient means of holding the current document while other operations change the document, and then restoring the current document to its saved state. The _restoredoc() function will restore the entire document or the payload, depending on how the save was performed by the _savedoc() function.

The _restoredoc() function uses the following format:

```
_restoredoc(name [,disposition])
```

name	string	Name of the register holding the information that is saved by the _savedoc() function.
disposition	keyword	An optional action that can be taken after the document is restored. Select one of the following actions: ❑ **clear.** The register used to hold the saved document is deleted, which releases the memory used to store the information. This action is set by default. ❑ **keep.** The register is not cleared, making it possible to restore the document again at a later point in the process flow.

The _restoredoc() function is especially useful during pre- or post-service execution in process flows, or as a clause of an _if() function.

If the data was saved to external media by the _savedoc() function, specifying the *keep* action prevents the file from being deleted. Specifying the *clear* action causes the file to be deleted after the restoration. In either case, the file is deleted when the server terminates.

Arithmetic Expressions

A function that returns an integer can participate in an arithmetic expression. However, there are limitations, such as an expression of the form

```
Intfunction() opvalue
```

where:

op
Is either plus (+) or minus (-).

value
Is either an integer or a function that returns the expected result.

Example: **Arithmetic Expression With Special Register**

If the value of the special register, X, is 10, then the following function returns 12:

```
SREG(X)+2
```

Function Syntax and Return Values

This topic describes the syntax and return values of the functions supplied with iWay Service Manager. Functions are listed in alphabetical order.

Any function can begin with the underscore character. Many functions must begin with the underscore character. This documentation shows the underscore character when it is required.

The underscore character prevents embedded strings from being evaluated. For example, the SQL statement

```
WHERE COUNT(XXX) < 3
```

would result in an error if it were confused with the iWay _COUNT() function. The underscore in the iWay function distinguishes it from the SQL statement.

Reference: **COND() Operators and Operands**

Monadic means that there is only one operand on the left and if there is an operand on the right, it is ignored. For example, COND(FILE(*xx*),EXISTS).

Dyadic functions require two operands, as do comparisons.

The following table lists and describes the available operators and operands.

Operator	Operand	Description
EQC	Dyadic	Case-insensitive compare. The normal case (EQ) is case sensitive.

Operator	Operand	Description
EXISTS	Monadic	Determines if the first operand exists. If the first operand is a special register SREG(*name*), the value is true if the register exists (is defined). Otherwise, the value is false. If the first operand is an XPath expression such as XPath(//SSS), the result is true if the node identified by the XPath is found in the document. For all other operands, the function tests whether the operand has a length.
ISNULL	Monadic	For XPath, determines if the identified node has a value. Results of this test can be ambiguous and its use is discouraged.
ISEMPTY	Monadic	Returns true if the operand has a value (that is, the node identified by the XPath has a value).
ISNOTNULL	Monadic	Reverse of ISNULL.
ISTRUE	Monadic	Returns true if the value of the first operand is true or yes. Otherwise, returns false.
ISNOTTRUE	Monadic	Reverse of ISTRUE.
=, EQ <, LT <=, LE >, GT >=, GE !=, NE	Dyadic	Case-sensitive lexical compare or arithmetic compare.

IWXPATH Language Support

Support for XML Path Language (XPath) is an important feature of iWay and is used in a number of areas within iWay Service Manager (iSM). XPath is a non-procedural language used to access and manipulate sections of an XML document. The XPath expression gathers information from the document, as if the XML document is a self-contained hierarchical database. The XPath expression specifies levels (segments or fields), filter predicates, and functions on the XML document data. The result of the iwxpath can be one or more values, a set of XML nodes, or a particular location in the XML structure. Using these XPath results, iSM can control the behavior of services (agents), conditional routing, and decision making inside of process flows. The fast iwxpath() function that is provided by iSM implements a subset of the XPath location steps, predicates, and functions, which are expressed using abbreviated syntax.

The iwxpath() function is not intended to be a full implementation of the XPath specification, but rather a very fast subset offering commonly used Xpath() searches. If additional functionality is required, iSM offers full Xpath() using the Xpath1() function. You can configure iSM to have the default xpath() function use iwxpath() if required.

The language always returns a string suitable for use in configuring other components. It is most suitable for locating element values and attributes. When multiple values are selected, they are separated by a vertical bar |, and empty values are denoted by &.

The XPath statement used to step into an XML document is known as a phrase. Phrases support both steps used to descend into the document and predicates used to determine how the step is to be applied. Currently selected XML nodes are called the node-set. In general, XPath phrase support of iSM is based on the formal XPath specification section 2.5, Abbreviated Syntax. The specification is available at *http://www.w3.org/TR/xpath*.

In IWXPATH the node context is always the root of the document, as such, only the child axis is implemented.

The XPath phrase support of iSM is as follows:

Steps

/<name>	Step down one level, selecting children of the specified name.
//<name>	Step down, selecting children of the specified name regardless of the level.
/*	Step down, selecting all children.
//*	Step down, selecting all children.

/.	Select all nodes already selected (used to apply predicates to the current node-set).
/..	Step up one level, selecting the parent of each node in the node-set.

These step specifications are fully covered in the appropriate RFC for XPath, section 2.5.

Predicates

Predicates are written after the step, enclosed in square brackets. There can be one or more predicates in a step, each of which is applied left to right to control the membership of the node-set.

Multiple predicates are written as sequential predicate terms: /x[p1][2]... applied left to right, with the predicate affecting the node-set as returned by the prior predicate. In essence, the predicates used by AND.

Any single predicate can hold any number of terms, separated by AND or OR. Terms can be grouped in parenthesis. Each term consists of a single term or a relation of <left>op<right>. The specification calls for left to right binding, AND taking higher precedence. For example, the predicate [a=b OR c=d AND e=f] is evaluated as [a=b OR (c=f AND e=f)].

Single term predicates, such as /x[2] operate as an index into the node-set so far.

Supported are:

Number, such as 3	Selects the members of the node-set that are the nth child of their parent.
last()	Selects the members of the node-set that are the last child of their parent.
count(parm)	The parm must be an iWay XPath expression. The expression is processed against the original document being evaluated from the root context. The number of elements in the node-set returned is used to select the members of the incoming node-set that are the nth child of their parent.
sreg(name[,default])*	The value of a named special register.

starts-with(p1,p2) ends-with(p1,p2)* contains(p1,p2)	Evaluates the incoming context, testing whether the name, attribute, or child values meet the p2 criterion. P1 can be name(), @attribute, @*any attribute or the name of the child.
not(filter)	Inverts the meaning of a selection filter. For example, /a//*[not(starts-with(name(),'d'))]

Filter functions marked with * are iWay extensions to the XPath specification.

Left terms can be:

<name>	Operate on nodes in the node-set with children of the specified name. The test will be on the value of the children in the node-set.
@<attname>	Operate on nodes in the node-set with attributes of the specified name. The test will be on the value of the attribute in the node-set.
*	Operate on all nodes in the node-set with children of any name.
@*	Operate on all nodes in the node-set with attributes of any name. The test will be on the values of any attribute, such as selecting all nodes with any attribute of value iway.
<function>()	One of a specified set of functions. ❏ **count([ns]).** Number of children of the selected node. ❏ **last().** Position() of the last node in the node-set. ❏ **local-name([ns]).** URI of the namespace within which the node exists. ❏ **position().** The position of the selected node in the children of the parent. Functions shown with optional parameters [ns] indicate that the function optionally operates on a node-set. For example, the localname() function returns the local name of either the current node being tested, or that of the first node in the node-set located by the xpath expression represented in the ns parameter.

Operators are the standard =, !=, <, <=, >, >=. Data is automatically case, such that if both the operators are numeric, a numeric comparison will be performed. Otherwise, a character comparison will be performed.

The right term is a literal, which can be:

String literal enclosed in single or double quotes	Example "IWAY".
Value not in quotes, such as a number. Simple string values can be entered this way for convenience.	Example 3 or xyz.
count(xpath)	The function to return the number of nodes in the node-set returned from the xpath parameter. The xpath of the count() function examines the original document being processed, from the root node context.
sreg(name[,default])	Value of the named special register, or the default value if the register is not defined. This is an iWay extension.
Standard string function to build test	Standard string functions concat, substring, substring-before, substring-after.

Standard groupings are allowed. For example, the following specification is now legal in iSM.

```
//greet/*[*=hello or (@addr=fred@ibi.com or position()<3 and
child1=audit)]
```

which selects all greet nodes with children having themselves children with the value "hello" or with attribute addr having the value of *fred@ibi.com* or those having both a position of one or two (first two children) and having a child with the node name child1 which has a value of "audit". This overly complicated example is intended to demonstrate how grouping is used.

Similarly, the expression /edxax/dest[position() = count(//sql)] selects the single dest node that matches in position the number of sql statements in the document.

Arithmetic

Simple arithmetic is supported in predicates. Only plus and minus are supported. For example

```
//password[count(//user)+1]
```

returns the password tag value related to the number of user tags in the document.

Final Functions

A set of final functions are supported to operate on the node-set being returned, such that the values from the XPath operation reflect the value returned by the function rather than the values of the nodes in the node-set. These functions are not strictly supported by the XPath specification, but are included to further the use of XPath in setting adapter parameters.

name()	Returns the name of the nodes in the selected node-set. Example //sql/*/name() returns the names of the grandchildren of each sql node.
position()	The position of each node in the node-set relative to its parent.
count()	The number of children of each node in the node-set.
text()	Returns the value of each node in the node-set.
localname()	Returns the local name of each node in the node-set.
namespace-uri()	Returns the namespace uri of each node in the node-set.

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH SOFTWARE LICENSE AGREEMENT OR CLICKWRAP END USER LICENSE AGREEMENT, THE LICENSE(S) LOCATED IN THE "LICENSE" FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

TIBCO, the TIBCO logo, the TIBCO O logo, FOCUS, iWay, Omni-Gen, Omni-HealthData, and WebFOCUS are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and/or other countries.

Java and all Java based trademarks and logos are trademarks or registered trademarks of Oracle Corporation and/or its affiliates.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only.

This software may be available on multiple operating systems. However, not all operating system platforms for a specific software version are released at the same time. See the readme file for the availability of this software version on a specific operating system platform.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.