
Notes for Sirius Mods Release 6.8

November, 2005



Sirius Software, Inc.
875 Massachusetts Avenue, Suite 21
Cambridge, MA 02139

Telephone: (617) 876-6677

FAX: (617) 234-1200

E-mail: support@sirius-software.com

World Wide Web: <http://sirius-software.com>

August 6, 2010

© 2010 Sirius Software, Inc.

Proprietary Notices

The following products:

- *Janus SOAP*
- *Janus Sockets*
- *Janus Web Server*
- *Sirius Functions*

are proprietary products of Sirius Software, Inc.:

Sirius Software, Inc.
875 Massachusetts Avenue, Suite 21
Cambridge, Massachusetts 02139
USA

Model 204® and **Connect *™** are proprietary products of Computer Corporation of America, a wholly-owned subsidiary of Rocket Software, Inc., which owns the trademarks:

Rocket Software Corporate Office
M204 Division
275 Grove Street
Suite 3-410
Newton, Massachusetts 02466-2272
USA

SoftSpy™ is a proprietary product of Information Technology Systems:

Information Technology Systems
95 Wells Avenue
Newton, Massachusetts 02459-3216
USA

Contents

Proprietary Notices	ii
Contents	iii
Chapter 1: Introduction	1
Chapter 2: Maintenance and Support	3
Model 204 support	3
Documentation	3
Chapter 3: All or Multiple Products	5
New SirMon statistics	5
COMPOPT parameter	6
Sdaemon enhancements	6
Terminal MODEL 6 support	7
Chapter 4: Janus SOAP LDAP Helper	9
BaseObject parameter for Find methods	9
No limit on length of returned data	9
Change of behavior with binary attribute values	9
Chapter 5: Janus SOAP ULI	11
Collection keyword now optional	11
Record, Recordset, and SortedRecordset objects deepCopyable	11
Item method now optional	12
Virtual constructors for system classes	13
This keyword for classnames	14
Daemon object enhancements	14
Transactional daemons	14
Info parameter on Run method	15
MasterNumber and ParentNumber methods	16
Colon allowed in User Language class names	17
Chapter 6: Janus SOAP Xml Classes	19
Validate method	19
InvalidChar XmlDoc property	20
IsValidString function	21

Invalid character error message	21
LoadXml accepts Stringlist, and LoadFromStringlist is obsolete	22
LoadXml for XML fragments	22
Chapter 7: Janus Web Server	25
Keep-Alive support	25
INPUTTIMEOUT port definition parameter	27
RAWINPUTONLY port definition parameter	27
COMPRESS feature is improved	27
New URL encoding/decoding \$functions	28
Chapter 8: Janus Sockets	29
Domain Name Services enhancements	29
Keep-Alive support	32
Chapter 9: SirSafe	35
New APSYSEC parameter	35
Chapter 10: Sirius Functions	37
More callable functions	37
\$GZIP compression is improved	37
\$a2e and \$e2a	37
\$lstr_base64_encode and \$lstr_base64_decode	38
\$lstr_c2x and \$lstr_x2c	38
\$daemonMasterNumber and \$daemonParentNumber	38
Chapter 11: Compatibility/Bug fixes	41
Backwards incompatibilities	41
Janus SOAP XML processing	41
Omit dangling space in null string PI	41
xml:lang handling (compatibility)	42
Declaration of xml prefix and URI (compatibility)	42
Janus SOAP ULI	42
“This” not allowed as class name	42
Fixes in Sirius Mods 6.8 but not in 6.7	42
Binary attribute values in LDAP Helper	42
xml:lang handling	42
Declaration of xml prefix and URI (bug fix)	43
Version corequisites	43

CHAPTER 1 ***Introduction***

This document lists the enhancements and other changes contained in *Sirius Mods* version 6.8, which was released in November, 2005. The previous version of the *Sirius Mods*, 6.7, was available in May, 2005.

CHAPTER 2 ***Maintenance and Support*****2.1 Model 204 support**

Sirius Mods version 6.8 supports *Model 204* V5R1 and V6R1 only.

2.2 Documentation

All the text in the *Sirius Mods* version 6.8 documentation is included in the Sirius Master Index, a searchable database you can download from the Sirius web site.

On the Documentation page of the web site, the download package contains a zipped collection of the Sirius PDF documents along with the Adobe-built, English-only, index (a PDX file and its supporting files), which you view with the Adobe Acrobat Reader.

Note that in addition to changes to the features and functionality of the *Sirius Mods*, the documentation is constantly being improved. Significant documentation changes are shown in a section in the preface of each manual.

3.1 New SirMon statistics

These statistics are added or modified for version 6.8. Most of the new statistics are for monitoring Table E activity, and all are defined only for *Model 204 V6R1* or later.

- The following new statistic is retrievable or viewable with \$FISTAT, \$FISTATL, or *SirMon*:

NPTE Size of Table E in pages. Same as ESIZE.

- The following new statistic is retrievable or viewable with \$USSTAT, \$USSTATL, or *SirMon*:

FSCBSW Full screen reads issued or text web responses sent, including Janus Web and Connect* transactions (rate or total).

- The following new statistics are retrievable or viewable with \$SYSTAT or *SirMon*:

BUFPAGL The number of Table E pages in the buffer pool (like BUFPAGD).

MODPAGL The number of modified Table E pages in the buffer pool (like MODPAGD).

DKSTBLE Number of waits on Table E pages (like DKSTBLD).

- The following system statistics are deleted: DKSRW, PFLI, PFLO, SRBT, and SRVC.
- Several name server statistics are added. See [“Domain Name Services enhancements” on page 29](#).

3.2 COMPOPT parameter

As of *Sirius Mods* version 6.8, the system parameter COMPOPT facilitates migration to mixed-case User Language. COMPOPT is a *Model 204* bitmask parameter that must be set in the CCAIN (User 0 input) stream. The bits in COMPOPT have the following meanings:

- X'01'** If on, all procedures start out in Sirius Case ToUpper mode, whether or not they begin with a mixed-case Begin statement. Sirius Case ToUpper mode translates all unquoted tokens to uppercase, so User Language statements, keywords, variable names, etc. may be written in mixed case. By setting the COMPOPT X'01' bit, a site is essentially enabling mixed-case User Language almost everywhere.
- X'02'** If on, Sirius Case Leave compiler directives are to be ignored: if Sirius Case ToUpper is in effect, it remains in effect even if a Sirius Case Leave directive is encountered. Setting the COMPOPT X'02' bit along with the X'01' bit enables mixed-case User Language everywhere, thus ensuring consistent language processing throughout an Online.
- X'04'** If on, image or image-item names, either literal or in variables, are to be automatically converted to uppercase before being used in methods or \$functions. Since mixed-case User Language is accomplished by translating unquoted tokens to uppercase, this case conversion for image or image-item names is the runtime equivalent of the compiler mixed-case support.

Setting the COMPOPT X'04' bit enables image and image-item names that appear as literals in User Language programs to be entered in mixed case. The only time this might be a problem is if there are true mixed-case image or image-item names in an application. A true mixed-case image or image-item name is one written in mixed case, either inside quotation marks (image and image-item names can *indeed* be put in quotes) or without Sirius Case ToUpper in effect. In general, neither of these is too likely, so true mixed-case image or image-item names are not likely in most applications.

3.3 Sdaemon enhancements

Sdaemons running on behalf of other threads, via the \$comm functions or Daemon objects, will no longer suffer record locking conflicts with other threads in the same thread *family*. Two threads are in the same family if one is a synchronous child daemon of the other via a \$comm function or a Daemon object. In addition, all families with common threads are considered to be a single family. So, if thread A is a synchronous parent of thread B, which is a synchronous parent of thread C, threads A, B, and C are all considered part of the same family. Furthermore, if in this example, thread B had two other synchronous children (via Daemon objects), threads D and E, then threads A, B, C, D, and E would all be considered part of the same family.

Before *Sirius Mods* 6.8, threads in a family could get record locking conflicts with each other. This greatly limited what could be done on daemon threads. The one condition under which two threads in the same family can still suffer a record locking conflict is if they both try to update the same record, which would require both threads to have an exclusive pending update lock on the record being updated. However, even this possibility of a record locking conflict is eliminated for transactional daemons (“[Transactional daemons](#)” on page 14).

3.4 Terminal MODEL 6 support

Support has been added for terminal models beyond the standard Mod 2 (24 X 80), Mod 3 (32 X 80), Mod 4 (43 X 80), and Mod 5 (27 X 132). The new terminal models are supported by setting the terminal model to 6:

```
RESET MODEL 6
```

There's really no such thing as a Model 6 terminal, but setting the terminal model to 6 tells *Model 204* to issue a Write Structured Field Query to the terminal to have the terminal indicate its geometry (number of rows and columns) to *Model 204*. In this way, *Model 204* can dynamically set a terminal's geometry, whether it's one of the standard geometries (Mod 2, 3, 4, or 5) or not. Many terminal emulators allow alternate 3270 sizes to be set. This makes it possible to set the terminal geometry to match the optimal combination of font size and physical screen size for a particular workstation, rather than trying to set the emulator font size to work well with one of a limited number of screen geometries.

Unfortunately, the standard User Language screen definitions don't allow the defining of fields that extend beyond column 79. However, \$scrHide, \$scrSize, and \$scrWide make it possible for User Language screens to take advantage of columns beyond column 79. In addition, these functions make it possible to dynamically modify screen definitions to allow a single screen definition to work with an arbitrary variety of screen sizes. While these functions are a bit awkward to use and somewhat limited, they're not unreasonable for building dynamic scrolling screens — scrolling screens being particularly suited for larger screen geometries.

To facilitate User Language applications for varying screen sizes, the VIEW command for the MODEL parameter has been enhanced to show the screen geometry in addition to the model number for model 6 terminals:

```
> V MODEL
MODEL      6 34*142    TERMINAL MODEL
```

So \$view issued for the above terminal returns a “6 34*142”, from which a User Language application could readily determine that the screen has 34 rows and 142 columns.

To enable model 6 support, the SIRTERM system parameter must be set in the CCAIN stream. This is a bitmask parameter with the following meanings for the bits:

- X'01'** Enable MODEL 6 support.
- X'02'** Always issue a Write Structured Field Query for terminals connecting to *Model 204* through VTAM. This allows *Model 204* to dynamically determine the screen geometry of any terminal connecting to it through VTAM, without having to issue a RESET MODEL 6 command. The downside of this setting is that it could add a small amount of time to the initial connection process and a slight amount of extra network traffic.

If a terminal is using a non-standard screen geometry via model 6 support, the *Model 204* editor and command line will correctly use the available screen space. Many *UL/SPF* subsystems such as SirScan, SirMon, and SirPro will also take advantage of the additional available screen space.

The following features are new or changed in the LDAP Helper methods (the `Ldap` class):

4.1 BaseObject parameter for Find methods

The `BaseObject` parameter is added to the LDAP Helper Find methods. This optional, `NameRequired`, string contains one or more comma-separated `attribute=value` pairs that direct the Find method search to a particular domain in the target LDAP directory tree. For example:

```
BaseObject='dc=hawaii,dc=edu'
```

Such a string may be required by your target LDAP server to provide an LDAP base “distinguished name,” which ensures that the entries your search string locates are unique.

4.2 No limit on length of returned data

The LDAP Helper can now accommodate as much returned data as an LDAP server sends. Formerly, an approximately 1300-byte limit was enforced on each LDAP request.

4.3 Change of behavior with binary attribute values

In version 6.7 of the *Sirius Mods*, if the result of one of the Find methods contained binary data that was not a valid XML string, a *Model 204* snap was issued and the user was restarted. In version 6.8, such data simply causes a request cancellation, with a message that displays a fragment of the value including the invalid character.

As listed in “[Binary attribute values in LDAP Helper](#)” on page 42, this is actually a bug fix. However, in addition to this fix, you may now use the `XmlDoc InvalidChar` property. If you set the result `XmlDoc`'s `InvalidChar` property to `Allow` before invoking the `Ldap Find*` method, the result will contain the binary data (of course, it will be translated to a corresponding EBCDIC value).

The following sections describe changes in the *Janus SOAP ULI* in this release.

5.1 Collection keyword now optional

The keyword “collection” is now optional on declarations of `ArrayList` and `NamedArrayList` collections. Instead of coding “collection `arrayList`” or “collection `namedArrayList`,” you may now simply use “`arrayList`” or “`namedArrayList`.”:

```
%monkeys      is arrayList of object monkey
```

5.2 Record, Recordset, and SortedRecordset objects deepCopyable

The `Record`, `Recordset`, and `SortedRecordset` objects are now `deepCopyable`. This means that these objects can be contained inside User Language classes and those User Language classes, themselves, could be `deepCopyable`. Perhaps even more important, this means that these objects can be passed back and forth between master and daemon threads, via the `Run`, `GetInputObject`, `ReturnObject`, and `ReturnInfoObject`

methods. Among other things, this makes it easy to dynamically generate Find statements that run on a daemon thread and then to use the resulting found set on the master thread:

```
b

%recs      is object recordSet in file sirfiled
%daem      is object daemon
%program    is object stringList
%criteria1  is string len 32
%criteria2  is string len 32

%criteria1 = 'rectype eq ''FILE''

%program = new
text to %program
  b
    %recs      is object recordSet in file sirfiled
    find records to %recs
      {%criteria1}
      {%criteria2}
    end find
    %(daemon):returnObject(%recs)
  end
end text

%daem = new
%daem:open('FILE SIRFILED', default=true)
%daem:run('*LOWER')
%daem:run(%program, , %recs)

print %recs:count

end
```

5.3 Item method now optional

The Item method name is now optional in any context where it is used for system methods. This has always been true for collections. That is, if %foo is declared as

```
%foo      is collection arrayList of object xmlDoc
```

The following two lines are equivalent:

```
%foo:item(%i):print
%foo(%i):print
```

Now, the method name is also optional in the other two places the Item method appears

in system classes: in the Stringlist class and the XmlNodeList class. That is, if `%list` is a Stringlist object, the following two lines are equivalent:

```
print %list:item(%i)
print %list(%i)
```

And, if `%nodes` is an XmlNodeList object, the following two lines are equivalent:

```
%nodes:item(%i):print
%nodes(%i):print
```

While this capability is not indicative of some larger principle such as “the Item method is always a class's default method, if it's defined,” it's likely that any new system classes with Item methods will also exhibit this behavior.

5.4 Virtual constructors for system classes

A shared method in a class that returns an instance of the class is sometimes referred to as a “factory method” or “virtual constructor,” because it behaves very much like a constructor. In *Sirius Mods* 6.8, system methods that are virtual constructors can be invoked without specifying the class name or a class variable before the method name, much as constructors can be invoked without the class name. Currently, the only virtual constructor in any system class is the CurrentRecord method in the Record class. Before *Sirius Mods* 6.8, CurrentRecord would have to be invoked like this:

```
b
%rec    is object record in file foo

in file foo fr
    %rec = %(record in file foo):currentRecord
    ...
end for
    ...
```

or perhaps:

```
b
%rec    is object record in file foo

in file foo fr
    %rec = %rec:currentRecord
    ...
end for
    ...
```

The new virtual constructor enhancement allows this to be written as:

```
b
%rec    is object record in file foo

in file foo fr
  %rec = currentRecord
  ...
end for
  ...
```

5.5 This keyword for classnames

The `This` keyword can be used to indicate the current class for shared member references inside the class. For example, in the following, the word `This` inside parentheses means the current class, that is, class `Foo`:

```
class foo
  public
    subroutine increment
  end public
  public shared
    variable counter is float
  end public
  subroutine increment
    %(this):counter = %(this):counter + 1
  end subroutine
end class
```

As part of this enhancement, the name `This` is no longer allowed as a class name.

5.6 Daemon object enhancements

5.6.1 Transactional daemons

The New constructor for Daemon objects has a new, boolean, named parameter called `Transactional`:

```
%pantalaimon  is object daemon
...
%pantalaimon = new(transactional=true)
```

If `Transactional` is set to `True`, the master and daemon thread will share a single updating transaction. This means that if updates are performed by both the master and daemon thread, these updates are all either committed or backed out when a commit or

backout is performed on either the daemon or master thread. If a thread has multiple transactional daemons, then all those daemons share a single updating transaction with the master thread. In addition, if a transactional daemon thread itself has transactional daemon threads, those threads also share a single updating transaction with the ultimate master thread. Effectively, all of a thread's transactional daemons, and all the transactional daemons of those transactional daemons, and so on, make up a *transactional family* that shares a single updating transaction.

There are a few differences in updating-transaction processing between a transactional daemon thread and other threads:

- Transactional daemon threads will never do an implied commit. Other threads typically do implied commits at end of request, unless inside a non-autocommit subsystem. In addition, certain operations such as a procedure update will cause an implied commit on non-transactional daemon threads. Of course, an explicit Commit or Backout statement run on a transactional daemon thread will result in the transaction being committed or backed out, including those updates performed on the master thread or some other thread in the transactional family.
- As an outgrowth of the previous item, transactional daemon threads can perform procedure updates in the middle of an updating transaction, as well as other file maintenance operations such as field definitions.
- Transactional daemon threads cannot perform User Language updates against files that their ultimate master thread does not have open in update mode. The ultimate master thread is the thread's master or the master's master if that thread is, itself, a transactional daemon, and so on. This is because the actual commit or backout is always performed on the highest level master thread in a transactional family, so that thread must have update privileges for all files in an updating transaction. One benefit of this approach is that it's possible for a daemon thread to perform some updates, go away (log off), and for the update to still not be committed.

As noted in [“Sdaemon enhancements” on page 6](#), transactional daemons cannot suffer record locking conflicts with other threads in the family, and they can even update the same records as other threads in the transactional family.

5.6.2 Info parameter on Run method

The Run method has a new, named, output parameter called *Info*. This parameter allows a second output object for the Run method (in addition to the standard output object, which is the third parameter). While the Info object can be used for any kind of output object, its intent is to separate the “true” output from a daemon request from informational output (return codes, error messages, diagnostics, etc.). While this could be accomplished by making the output object for the Run method be of a class that extends the true output class and some informational class, this solution adds a lot of

extra complexity that is best avoided. In the following example, a Stringlist object is used as the Info output from a Run method — presumably error messages would go on that Stringlist:

```
%errors    is object stringList
%result    is object myClass
%daem      is object daemon
...
%daem:run('MYSUBSYS', , %result, info=%errors)
```

The daemon thread running the request would return the info object using the `ReturnInfoObject` method, which is a shared method that has a single required parameter: the object being returned:

```
%errors    is object stringList
...
%(daemon):returnInfoObject(%errors)
```

Since, like any other objects passed between master and daemon, the info object is passed via deep copy, the class of the info object must be `deepCopyable`.

5.6.3 MasterNumber and ParentNumber methods

The Daemon class has new shared methods (functions) called `MasterNumber` and `ParentNumber`.

The `ParentNumber` function returns the user number of the thread that created the daemon thread, either via a `Daemon` object or even via a `$comm` function. If the thread issuing the `ParentNumber` method is not an `sdaemon`, or is not performing work for another thread via a `Daemon` object or a `$comm` function, the `ParentNumber` function returns a value of -1. The following code audits a thread's parent user number:

```
audit 'My parent''s user number is ' %(daemon):parentNumber
```

The `MasterNumber` method returns the same value as the `ParentNumber` method, except in two cases:

- The parent thread, itself, has a parent. In such a case, the `MasterNumber` method follows the chain of parents to the highest level, that is, to the parent that does not, itself, have a parent.
- The issuing thread is an asynchronous `sdaemon`. In this case, the `MasterNumber` value is -1.

The `ParentNumber` and `MasterNumber` methods have `$function` equivalents: `$daemonParentNumber` and `$daemonMasterNumber` respectively. The `$functions` and methods can be used interchangeably regardless of whether the daemons were created with `$comm` functions or `Daemon` objects, so the decision of which to use is largely a matter of taste.

5.7 Colon allowed in User Language class names

The colon character (:) is now allowed in User Language class names. The following, for example, is a valid class declaration:

```
class customer:order
```

No colon-demarcated part of a class name can be the word `System`. For example, `System`, `System:foo`, `Foo:system`, and `Foo:system:bar` are all *invalid* class names.

While there is no special meaning to the colon-demarcated parts of a class name, the intent is that the colons be used to group classes, and that higher-level qualifiers come before the lower-level ones. For example, in class name `MiddleEarth:rohan:rider` there is a `Rider` class that falls under a grouping of classes qualified by `Rohan`. This grouping, itself, falls under the larger grouping of `MiddleEarth`. Using the colon in this way is likely to make it easier to take advantage of future class management features that are likely to depend on a hierarchical, colon-based, class naming scheme.

Janus SOAP Xml Classes

The following sections describe changes in the *Janus SOAP Xml** classes (the XML document API) in this release.

6.1 Validate method

A major new enhancement for the *Xml** classes is the ability to validate an *XmlDoc* using a schema, according to the **XML Schema Recommendation**. This can be accomplished with the following method in the *XmlDoc* class:

```
%rc = doc:Validate(schemaInput, [options])
```

Where:

%rc The return code, which indicates the success of validation. Possible values are:

- 0** if valid
- 10** if validation failed (request canceled if **ErrVal** or **ErrRet** option not specified)
- 100** if schema is invalid (request canceled if **ErrRet** option not specified)
- +n** if schema parsing error near character position *n* (request canceled if **ErrRet** not specified)

`Validate` is callable, so the assignment of the return code can be omitted.

doc The method object, which is an “instance” *XmlDoc* to be validated.

schemalInput A string or *Stringlist* that is the serialized schema document for validating the instance.

options A list of option words. In addition to the options available to the `LoadXml` method (**ErrRet**, **DTDIgnore**, three whitespace options) the following option, unique for `Validate`, is available:

ErrVal[idation] Return on validation errors (but not schema errors) (instead of cancelling); mutually exclusive with **ErrRet**.

Note: Contrary to most options, this option may be abbreviated (to the first six or more letters).

The meaning of the ErrRet option for Validate is similar to its meaning for LoadXml: it indicates that Validate should return (with return code `-100` or `+n`) even if given an invalid schema document.

Note: At the present time, the support for Validate is preliminary; to access this feature requires a temporary authorization of the XML Schema Prototype. Contact Sirius if you want to try this feature.

If you want a reference for XML Schema, there is a wonderful book: *Definitive XML Schema*, by Priscilla Walmsley.

6.2 InvalidChar XmlDoc property

In previous versions of *Janus SOAP*, it is illegal to add to an XmlDoc a string that contains a “control character,” for example, `X'01'`. This restriction is specified by the XML Recommendation: if you presented a document containing such a character to a W3C-compliant XML processor, it would be rejected as erroneous.

However, you can use an XmlDoc to contain data that is not exchanged with other applications, yet whose content cannot be controlled by your application. The Find* methods in the Ldap class are examples of this (see “[Change of behavior with binary attribute values](#)” on page 9).

There is now a mechanism to allow invalid characters in an XmlDoc — the InvalidChar property:

```
doc:InvalidChar = Allow | Disallow
```

```
%XmlInvalidChar = doc:InvalidChar
```

This property is set to a value of the XmlInvalidChar enumeration:

Allow An XmlDoc with this InvalidChar setting allows invalid characters.

Disallow An XmlDoc with this setting does **not** allow invalid characters (this is the default setting).

The InvalidChar property applies to updates to Element or Attribute values. If InvalidChar is `Allow`, an invalid XML string is allowed in these cases:

- As the second argument of the AddElement, InsertElementBefore, and AddAttribute methods

- As the right-hand side of an assignment to the Value property of an Element or Attribute node

Note that the above operations imply that the string contains an EBCDIC value.

This property does not affect updates to Comment or PI nodes. It also does not affect deserialization operations, such as the LoadXml method, although in future versions of *Janus SOAP* it may be extended to allow invalid Attribute and Element values created with deserialization methods.

If InvalidChar is set to `Disallow` in the method object of the AddSubtree or InsertSubtreeBefore methods, the request is cancelled if the source XmlDocument “possibly” has any invalid characters. That is, it is cancelled if the XmlDocument containing the argument XmlNode has been updated (in any form) while its InvalidChar property was set to Allow. This is **any** form of update, including, for instance, deserialization, all the Add* and Insert* methods, assigning to the Value property, and assigning a non-null string to the SelectionNamespace property.

You may also check that a string is a valid XML string using the [“IsValidString function”](#).

6.3 IsValidString function

You may use the IsValidString function to determine whether a string is a valid XML Element or Attribute value, prior to updating an XmlDocument:

```
%Boolean = %(XmlDoc):IsValidString(string)
```

If the result is `True`:

- The AddAttribute, AddElement, and InsertElementBefore functions may use the string *string* as their second argument.
- The Value property may use the string *string* as the right side of an assignment (subject to restrictions on null strings).

Note that the above operations imply that *string* contains an EBCDIC value.

6.4 Invalid character error message

Normally, changes to error messages are not described in the Release Notes, but it is worth noting that the error message produced by AddElement, for example, if you attempt to add an invalid character, now contains a fragment of the value that includes the invalid character.

6.5 LoadXml accepts Stringlist, and LoadFromStringlist is obsolete

The LoadXml method now accepts a Stringlist as its first argument, thus replacing the purpose of the LoadFromStringlist method.

As a result, the LoadFromStringlist method will be obsolesced in the *Janus SOAP* documentation; the description of the operation of the LoadXml method with a Stringlist argument will apply to the operation of the LoadFromStringlist method.

6.6 LoadXml for XML fragments

The LoadXml method, previously only available in the XmlDoc class, is now available in the XmlNode class. This allows for deserializing an “XML fragment” as the child or children of a node in an XmlDoc.

For example, you can create an XHTML document:

```
%d Object XmlDoc Auto New
%n Object XmlNode
%n = %d:AddElement('html')
%n = %n:AddElement('body')
%s1 Object Stringlist Auto New
Text To %s1
    <h1>Test</h1>
    This is a test
End Text
%n:LoadXml(%s1)
%d:WebSend
```

In the above example, note that:

- The method object (%n) is an XmlNode, not an XmlDoc, and it refers to an Element, here named `body`.
- The serialized input is not a valid XML document. In this example, it does not have a “top-level element.” It consists of an element (named `h1`) followed by character content (`This is a test`) that is not within element content.

An **XML fragment** is a substring of a serialized XML document, such that

- The fragment, if “wrapped” within a simple element start tag and end tag (such as `<w>` and `</w>`, respectively), is a legal XML document.
- The fragment may contain undeclared prefixes. Any such prefixes must be declared at the Element node referred to by the method object of LoadXml; these declarations (along with the default namespace) are inherited by the inserted fragment.

Prior to version 6.8, the only “serialize and insert” technique uses an XmlDocument Load* method followed by AddSubtree. Besides handling an XML fragment that is not a “proper subtree” (that is, has leading or trailing character content), the inheritance of namespace declarations is a significant advantage, because you do not need to respecify the declarations in the Load* method, and there are no “extra” namespace declarations in the result. For example:

```
%d Object XmlDocument Auto New
%n Object XmlNode
%n = %d:AddElement('html', , 'http://www.w3.org/1999/xhtml')
%n:LoadXml('<body><h1>Hello</h1>world</body>')
```

The body and h1 elements above will both be in the http://www.w3.org/1999/xhtml namespace, even though that was not specified in the LoadXml input.

Note that an XML fragment **does not** provide for inserting an Attribute into an Element node; for example, the following would not achieve it:

```
%d Object XmlDocument Auto New
%n Object XmlNode
%n = %d:AddElement('top')
%n:LoadXml('foo="bar"')
%d:Print
```

The deserialized input to LoadXml above is simply character content of the Element containing the fragment, so the result is:

```
<top>foo="bar"</top>
```

Note that if the method object refers to the Root node of an XmlDocument, the LoadXml method in the XmlNode class behaves exactly as the LoadXml method in the XmlDocument class. For example:

```
%d Object XmlDocument Auto New
%n Object XmlNode
%n = %d:SelectSingleNode
%n:LoadXml('<?xml version="1.0"?><top><inner/></top>')
```

When the Root node is the method object, the serialized input must be a legal XML document (for example, the XmlDocument must be Empty, and the serialized input must contain exactly one top-level element).

Unless the method object refers to the Root node, the input to the XmlNode LoadXml method is always an EBCDIC string or a Stringlist of EBCDIC strings. This differs from the XmlDocument deserialization methods, which allow several forms of input string: EBCDIC, "ASCII", UTF-8, and UTF-16.

CHAPTER 7 *Janus Web Server*

The following features are new or changed in *Janus Web Server*.

7.1 **Keep-Alive support**

A new *Janus Web Server* port definition parameter, `KEEPALIVE`, tells *Janus Web Server* to keep an HTTP connection open for a certain number of seconds to allow a client (often a browser) to send another request on the same connection. This can reduce network traffic and, more significantly, HTTP request latency. Both of these benefits are magnified for SSL connections where each HTTP request requires a TCP/IP and SSL connection establishment handshake.

The `KEEPALIVE` parameter must be followed by a single number between 1 and 32767 that indicates the number of seconds a TCP/IP connection is to be held open after an HTTP request on that connection. For the connection to be held open, the client/browser must indicate that it supports HTTP keep-alives. Most modern browsers support and take advantage of keep-alives.

A keep-alive TCP/IP connection uses a Janus web thread, so it is counted against both the maximum connections for a port and against a site's licensed thread limits. Because of the latter, it is probably impractical for a site with a low (10 thread) licensed Janus Web connection limit to take advantage of keep-alives unless *Janus Web Server* is used exclusively to communicate with a handful of servers (that act as clients to *Janus Web Server*). For sites with higher licensed limits, it's probably a good idea to increase the connection limit for any port that is to take advantage of keep-alives.

For a very rough estimate of how many more threads might be required on a port, simply multiply the maximum requests per second by the `KEEPALIVE` time. So, if at the busiest time of day, 10 requests come in per second on a port, and `KEEPALIVE` is set to 10, increasing the thread limit for the port by 10×10 , or 100, should ensure that no requests are rejected because all available threads are held by keep-alive sessions. Of course, this is probably a gross overestimate of the real requirement, since one would assume that many of the requests would be coming from the same client, so they would be transmitted over an existing keep-alive connection rather than sent over a new connection (otherwise, there'd be little benefit to using keep-alives).

Notes:

- A keep-alive connection does **not** use a server or sdaemon thread while between requests. This means that it's quite conceivable that a port might have many more connections held open at a time than there are sdaemon threads in the Online.

Because of this, the former restriction that the thread limit for a WEBSERV port could not exceed that number of sdaemon threads in an Online has been dropped in *Sirius Mods 6.8*.

- The keep-alive facility only preserves a TCP/IP connection. It does not preserve any application context. Other mechanisms such as persistent session support via `$web_form_done` or session context maintained via `$session` functions are required if there is a need to maintain application context between HTTP requests. However, there is no reason that keep-alives couldn't be used to improve the performance of web applications that maintain context between requests.
- While the keep-alive mechanism is useful for browser-to-web server requests, especially when embedded content such as style sheets or images are served from the same web server as the HTML pages, keep-alives can be even more useful for server-to-server HTTP requests. In an environment where one server is sending requests to another, there could be a steady stream of requests from one server, and so one IP address, to another. In such an environment, keep-alives could be a huge win — a single or a handful of TCP/IP connections could be used to service all the requests between the servers, eliminating thousands of server to server connection establishments.

Unfortunately, taking advantage of this is dependent on the ability of threads on the server to act as a client to reuse other threads' TCP/IP connections. That is, a server typically has many threads running at a time. If these threads cannot reuse each others' TCP/IP connections, then the benefit of keep-alives might be significantly reduced. Since a given thread might never issue more than a single HTTP request, and since it could not use another thread's connection, each HTTP request would end up being a new TCP/IP connection.

- Because the potential benefits of using keep-alives in a server-to-server application are so great, it is worth investigating the extent to which any server that communicates with a *Janus Web Server* can take advantage of keep-alives. If both sides of the server-to-server application are *Model 204*, then the use of the `KEEPALIVE` setting on a *Janus Sockets* port is likely to provide tremendous benefit — *Janus Sockets* client requests easily share connections to the same web server.
- The lack of support for keep-alives in *Janus Web Server* before *Sirius Mods 6.8* was probably one of the biggest stumbling blocks preventing *Janus Web Server* from reporting that it is an HTTP 1.1 web server — HTTP 1.1 web servers are expected to support keep-alives. With this support in place, the `HTTPVERSION` port definition parameter could probably be safely set to 1.1, enabling *Janus Web Server* applications to take advantage of other HTTP 1.1 features.

7.2 **INPUTTIMEOUT port definition parameter**

A new *Janus Web Server* port definition parameter, `INPUTTIMEOUT`, tells *Janus Web Server* to use a different timeout for input (receiving the web request) than for output. This might be desirable because, while some browsers sometimes delay receiving web output until some user interaction is completed, there should never be a delay between connection establishment and the sending of the HTTP request. As such, it is perfectly safe to set a very aggressive input timeout while maintaining a less aggressive output timeout. Once a complete HTTP request is received, *Janus Web Server* switches the connection to use the port's `TIMEOUT` value for the output timeout.

While, in theory, no client requests would ever exceed a reasonable (like 5 second) `INPUTTIMEOUT` value, there is anecdotal evidence that some web crawlers and, occasionally, browsers establish connections only to “lose their way” and either not send a request on the connection at all, or not send one for a long time. In addition, unsophisticated denial of service attacks often do nothing more than establish large numbers of connections to a web server, attempting to tie up resources on the web server. Finally, if someone mistakenly connects to a web server port using the wrong protocol (like telnet or HTTP), the connection will often hang until it times out. All these cases can be dealt with handily with a relatively low `INPUTTIMEOUT` such as 5.

7.3 **RAWINPUTONLY port definition parameter**

A new *Janus Web Server* port definition parameter, `RAWINPUTONLY`, tells *Janus Web Server* to save the raw input stream for an HTTP Post, regardless of the mime-type set by the client. This does the same thing as the `RAWINPUTONLY` parameter provided for JANUS WEB ON rules in *Sirius Mods* version 6.7, except that it makes the setting effective for all URLs on the port.

With the introduction of this parameter, the `NOTRAWINPUTONLY` parameter is made available for JANUS WEB ON rules to override a port-wide (JANUS DEFINE) `RAWINPUTONLY` setting on a URL basis.

7.4 **COMPRESS feature is improved**

Improvements to the “deflate” compression algorithm used in the *Janus Web* `COMPRESS` feature provide significantly better compression in *Sirius Mods* version 6.8, without any noticeable increase in CPU requirements. Compression is likely to improve from 5 to 15 percent, especially for streams greater than 6184 bytes, and regardless of whether fixed-code or dynamic-code compression is used. Additional improvements to dynamic compression may yield an additional 2 to 4 percent.

7.5 New URL encoding/decoding \$functions

`$web_url_encode_lstr` is now available. It is identical to `$web_url_encode` with the exception that it is longstring capable. That is, it can take a longstring input and produce the appropriate longstring output.

`$web_url_decode` and `$web_url_decode_lstr` provide the inverse functionality to `$web_url_encode` and `$web_url_encode_lstr`. That is, instead of converting a string to the EBCDIC representation of its URL encoding, they convert an EBCDIC representation of a URL encoding of a string to that string. As one might guess, `$web_url_decode` is not longstring capable, while `$web_url_decode_lstr` is.

`$web_url_encode_lstr` and `$web_url_decode_lstr` are each other's inverse. That is, URL encoding and then decoding a string should produce the original input string. The same is true of `$web_url_encode` and `$web_url_decode`, with the exception that it's much more likely that output from `$web_url_encode` will be truncated. This is because a URL encoded string can be longer than the unencoded string, so the result of a `$web_url_encode` might be truncated at 255 bytes.

`$web_url_decode` and `$web_url_decode_lstr` are particularly useful in processing form fields when using the RAWINPUTONLY form processing (described in [“RAWINPUTONLY port definition parameter” on page 27](#)). For example, the form fields for a post to a URL where RAWINPUTONLY is in effect can be loaded into a Stringlist as follows:

```
%formParms      is object stringList
...
%formParms = new
%formParms:parseLines($web_input_content('TEXT'), ' &')
```

This produces a Stringlist that contains items of the format *fieldname=value*. Assuming that none of the form field names have been URL encoded by the browser (a reasonable assumption for most Latin-character field names), this Stringlist is in a format that can be easily, though not prettily, searched. For example, the following code locates the field named `OrderNumber`:

```
%itemNum = %formParms:locate('OrderNumber=', , 1, 12)
```

Now, because there's no way to ensure that the end user didn't put special characters into the value for field `OrderNumber`, it is necessary to URL decode the found value:

```
if %itemNum then
  %order = $lstr_substr(%formParms:item(%itemNum), 13)
  %order = $web_url_decode_lstr(%order)
```

Before *Sirius Mods* 6.8, one would have to hope that the form field above did not contain any URL encoded data, or would have to write one's own URL decoding function.

The following features are new or changed in *Janus Sockets*:

8.1 Domain Name Services enhancements

The JANUS NAMESERVER command has been enhanced to add support for some new Domain Name Services capabilities. Domain Name Services (sometimes call DNS) are used by *Janus Sockets* to translate host names to IP addresses in client connection requests (\$SOCK_CONN, Socket class New method, or HttpRequest Get or Post method). IP addresses are **always** used to establish connections, but host names are typically used in applications to facilitate changing of hosts' IP addresses as needed.

The new DNS capabilities are:

- Support for multiple name servers
- Internal caching of name server responses
- Faster timeout of name server requests

The format of the JANUS NAMESERVER command is now:

```
JANUS NAMESERVER ip_address port_number -
                  [AND ip_address port_number -
                  [AND ip_address port_number ... ]] -
                  [TIMEOUT numsec] -
                  [CACHE numcache] -
                  [MAXTTL maxsec]
```

JANUS NAMESERVER command syntax

where the AND clause allows specification of alternative backup name servers. This can be useful in providing redundancy on the slight chance there is a failure or shutdown of the primary name server. Typically, all DNS requests go to the primary (first) name server. If a request times out, it is assumed that that name server is having problems, and the second name server is sent the request. If the second name server sends a response, it becomes the primary name server. That is, all subsequent DNS requests are sent to it. If the second name server fails to respond, the third name server is tried; if it responds, it becomes the primary name server. And so on.

The following command tells *Janus Sockets* to use the name server at IP address 198.242.244.9 as the primary name server, and the one at 198.242.244.47 as a backup. No port numbers are specified because the name servers use the default port of 53 (as do almost all name servers):

```
JANUS NAMESERVER 198.242.244.9 AND 198.242.244.47
```

If after being the primary name server for a while, a name server stops responding (times out), the name servers are tried one after the other, starting with the first in the name server list. Since name servers rarely go down, it is probably quite unnecessary to have more than two name servers, but up to eight name servers are supported by Janus.

Note that a negative response from a name server, that is, a response that indicates that the name server does not know the requested host name, does **not** cause a subsequent name server to be tried. If name servers are properly configured, changing name servers should not affect the success of a hostname lookup.

The other new JANUS NAMESERVER command parameters available in *Sirius Mods* 6.8 are:

TIMEOUT numsec This sets the maximum number of seconds to wait for a name server response. The internet standard for this value, and the value used by *Janus Sockets* before *Sirius Mods* 6.8, was 10. While this might be a reasonable value if lookups are going against busy name servers on the other side of the world, and the internet is extremely busy, this value is absurdly high for business applications where the nameserver is local and all host names being sought are local. Even for internet-wide lookups, a 10-second timeout is probably overkill these days.

In any case, the cost of a high timeout value is that if there should be a problem with a name server, it would take the timeout number of seconds to notice the problem and correct for it. 10 seconds is a fairly long time for this to happen, so one might want to set this timeout to a lower value.

CACHE numcache This parameter indicates that *Janus Sockets* is to save hostname-to-IP address mappings in the Online address space. This means that subsequent name server lookups would be extremely fast in-memory lookups, rather than calls to an external name server. This would both save CPU and reduce name lookup latency.

Numcache is the maximum number of hostnames to cache in the Online. Names are cached in virtual storage, and the amount of virtual storage required for the cache is about $100 * \text{numcache}$. If the number of hostnames ever looked up is relatively small, it makes sense to set *numcache* to a little more than this number of

hostnames. Otherwise, the setting of *numcache* is a trade-off between CPU savings and latency and virtual storage — cache entries are saved according to a most recently used algorithm so that frequently requested hostnames tend to remain in cache.

Several system statistics that can be viewed via *SirMon* make it possible to determine how well the name caching is working. These statistics are described below.

MAXTTL maxsec

This parameter indicates the maximum amount of time *Janus Sockets* is to save a hostname to IP address mapping in its local cache before checking it again with the name server. The name MAXTTL stands for MAXimum Time To Live. This parameter might be useful in highly dynamic environments where host addresses might change somewhat frequently.

When a nameserver sends a hostname to IP address mapping response, it also sends a Time To Live with that response. This indicates that the requestor should revalidate the name after the indicated time-to-live. *Janus Sockets* uses the minimum of the time-to-live sent by the name server and the value of MAXTTL for the effective time-to-live of a name server response.

Janus Sockets does not cache negative responses. That is, if the name server did not know a particular hostname, and that hostname is requested again, *Janus Sockets* will again query the name server for that hostname.

The JANUS NAMESERVER command can be issued at any time, so the name server lookup behavior of *Janus Sockets* can be dynamically changed for extraordinary situations such as name server crashes, name server reconfigurations, wholesale IP address changes on the local network, and so on.

Resets of CACHE and MAXTTL should “do the right thing.” That is, a change of CACHE value should preserve as many as the new CACHE value's number of entries. For example, if CACHE went from 300 to 50, the fifty most recently used entries would be preserved. If it went from 100 to 200, all entries would be preserved, of course. So if one wants to clear the cache, *numcache* should be set down to 0, then back up to the desired cache size.

The following system statistics can be viewed in *SirMon* to determine how name server lookups are faring in an Online:

DNSCACHE The number of entries in the name server cache (the CACHE value on the JANUS NAMESERVER command).

DNSMAXTL The value of the MAXTTL parameter on the JANUS NAMESERVER command.

- DNSCURNS** The current “go to” name server, in dotted IP address format with the port number in parentheses. For example: `198.242.244.9(53)`.
- DNSRTOT** Total number of name lookup requests.
- DNSRFAIL** Number of name lookup requests that did not succeed, that is, did not get an IP address.
- DNSRSUCC** Number of name lookup requests that succeeded.
- DNSRCACH** Number of name lookup requests that found the requested name in the local cache.
- DNSRTIMO** Number of requests to name servers that timed out before they got a response. With multiple name servers, a single request can try several servers and so get several timeouts. But, of course, this is probably rare in practice.
- DNSWTIME** Total time spent waiting for responses from name servers.

8.2 Keep-Alive support

A new *Janus Sockets* client port definition parameter, `KEEPALIVE`, tells *Janus Sockets* to keep an HTTP connection to a web server open for a certain number of seconds so *Janus Sockets* can send another request on the same connection. This can reduce network traffic and, more significantly, HTTP request latency. Both of these benefits are magnified for SSL connections where each HTTP request requires a TCP/IP and SSL connection establishment handshake.

The `KEEPALIVE` parameter must be followed by a single number between 1 and 32767 that indicates the number of seconds a TCP/IP connection is to be held open after an HTTP request on that connection. For the connection to be held open, the web server must indicate that it supports HTTP keep-alives. Most modern web servers can take advantage of keep-alives.

A keep-alive TCP/IP connection uses a *Janus Sockets* thread, so it is counted against both the maximum connections for a port and against a site's licensed thread limits. Since often a *Janus Sockets* application is only used to communicate with a single or handful of web servers, this should not be a problem.

Keep-alives are highly specific to the HTTP protocol (there is a totally different keepalive feature at the TCP protocol level with which HTTP keep-alives should not be confused), so they can only be taken advantage of by *Janus Sockets* when it “knows” that HTTP is being used. The only case where this is so is when HTTP helper objects (`HttpRequest` and `HttpResponse`) are used. Fortunately, this is the easiest way to perform client HTTP requests with *Janus Sockets*, so these ought to be the only, or at least, the most common types of such requests in any Online.

For an HttpRequest object Get or Post, if keep-alives are being used for a port (KEEPALIVE non-zero on the port definition), an idle (keep-alive) connection is sought for the target IP address. If one is found, that connection is used, avoiding connection establishment overhead. If none is found, a new connection is established. After the request is completed, rather than breaking the connection, it is held open for the KEEPALIVE timeout period. This approach reduces connection establishment overhead while in no way reducing parallelism — if a keep-alive connection is in use by one thread, another connection is used; if no unused ones are available, a new one is established.

There may be a timing problem with a keep-alive connection if a request tries to use a keep-alive connection just as the target server is closing the connection (most likely because the server's keep-alive time limit was exceeded). In such a case, the client side must either try to use a different connection to the target server or try to establish a new one. This retry functionality is handled automatically and transparently by *Janus Sockets*. In fact, the use of keep-alives for a *Janus Sockets* HTTP client application should have no effect on application functionality.

The following are new or changed features in *SirSafe*:

9.1 New APSYSEC parameter

The APSYSEC parameter was actually introduced in *Sirius Mods 6.7*, but it inadvertently was not documented then, so it is announced here. Currently, the only use of this parameter is to allow any system manager to START, STOP, DEBUG, or TEST any subsystem, without having to add the system manager to the SCLASS authorized to do these things.

The APSYSEC parameter is a bitmask parameter, where the bits mean:

X'01' System managers are allowed to START, STOP, DEBUG, or TEST any subsystem. This reduces the headache of having to add a system manager to a privileged SCLASS in every subsystem in an Online to enable the system manager to at least start and stop the subsystems — a common thing for system managers to need to do.

In fact, if no users other than system managers need to start or stop subsystems, this can eliminate the need even to have SCLASSES in a subsystem to allow starting or stopping of the subsystem. In some cases, eliminating this requirement can reduce the subsystem definition to a single default SCLASS, which has performance benefits — no SCLASS lookup is required when a user enters a subsystem, and no SCLASS-specific compilations are done for the procedures in the subsystem.

Furthermore, because of the overhead associated with multiple SCLASSES in a subsystem (not huge, but possibly measurable), some sites risk simply adding START and STOP privileges (and perhaps TEST and DEBUG) to the one and only SCLASS for a subsystem. This, of course, means that any user can start and stop the subsystem, which might not be ideal from a control or security perspective. The APSYSEC parameter allows such a site to have it both ways: START and STOP privileges can be removed from the default/only SCLASS for a subsystem, and only system managers (or users running subsystems that give them system manager privileges) can start or stop subsystems.

Of course, if a site wants to continue using *Model 204's* traditional fine-grained control of START, STOP, TEST, and DEBUG privileges, the APSYSEC X'01' bit should not be set.

CHAPTER 10 *Sirius Functions*

The following are new or changed features in the *Sirius Functions*:

10.1 More callable functions

The following functions are now callable. Instead of assigning the function result to a %variable, these functions may be invoked without an assignment and with or without a User Language CALL statement:

- \$errSet
- \$fakeEnt
- \$procCls
- \$procOpn
- \$scrWide
- \$scrHide
- \$scrSize
- \$sirMsgP
- \$wakeUp

10.2 \$GZIP compression is improved

Improvements to the “deflate” compression algorithm used by \$GZIP processing provide significantly better compression in *Sirius Mods* version 6.8, without any noticeable increase in CPU requirements. Compression is likely to improve from 5 to 15 percent, especially for streams greater than 6184 bytes, and regardless of whether fixed-code or dynamic-code compression is used. Additional improvements to dynamic compression may yield an additional 2 to 4 percent.

10.3 \$a2e and \$e2a

These two \$functions provide longstring capable ASCII-to-EBCDIC and EBCDIC-to-ASCII translation functions. That is, they can take a longstring input and produce a longstring output. Both of these functions take a single string argument and produce a string result.

While there are other mechanisms available for doing ASCII to EBCDIC and EBCDIC to ASCII translation, such as the TranIn and TranOut methods in the *Janus Sockets* Socket

class, `$a2e` and `$e2a` can be used regardless of the thread type or programming context.

`$a2e` and `$e2a` use the “standard” ASCII-to-EBCDIC and EBCDIC-to-ASCII translation tables provided by Sirius, and they provide no mechanism for overriding these tables.

10.4 `$lstr_base64_encode` and `$lstr_base64_decode`

These two \$functions provide identical functionality to the `$base64_encode` and `$base64_decode` functions, except that the new \$functions are longstring capable. That is, they can take a longstring input and produce a longstring output. Like `$base64_encode` and `$base64_decode`, both these new functions take a single string argument and produce a string result.

10.5 `$lstr_c2x` and `$lstr_x2c`

These two \$functions provide identical functionality to the `$c2x` and `$x2c` functions, except the new \$functions are longstring capable. That is, they can take a longstring input and produce a longstring output. Like `$c2x` and `$x2c`, both these new functions take a single string argument and produce a string result.

10.6 `$daemonMasterNumber` and `$daemonParentNumber`

Two new sdaemon thread \$functions, `$daemonMasterNumber` and `$daemonParentNumber`, are introduced:

- The `$daemonParentNumber` function returns the user number of the thread that created the daemon thread, whether the daemon is associated with a `Daemon` object or with a `$comm` function. If the thread issuing the `$daemonParentNumber` method is not an sdaemon, or is not performing work for another thread via a `Daemon` object or a `$comm` function, or if the issuing thread is an asynchronous daemon, the `$daemonParentNumber` function returns a -1.

For example, the following code audits a thread's parent user number:

```
audit 'My parent's user number is ' $daemonParentNumber
```

- The `$daemonMasterNumber` method returns the same value as the `$daemonParentNumber` method, except in the case where the parent thread, itself, has a parent. In such a case, the `$daemonMasterNumber` method follows the chain of parents to the highest level, that is, to the parent that does not, itself, have a parent.

The \$daemonParentNumber and \$daemonMasterNumber methods have Daemon class method equivalents: ParentNumber and MasterNumber, respectively. The \$functions and methods can be used interchangeably, whether or not the daemons were created with \$comm functions or Daemon objects, so the decision of which to use is largely a matter of taste.

CHAPTER 11 *Compatibility/Bug fixes*

This chapter lists any compatibility issues with previous versions of the *Sirius Mods* and any bugs which have been fixed in this version of the *Sirius Mods* but had not, as of the date of this release, been fixed in the immediately previous version (6.7).

In general, backward incompatibility means that an operation which was previously performed without any indication of error, now operates, given the same inputs and conditions, in a different manner. *Sirius* may not list as backwards incompatibilities those cases in which the previous behaviour, although not indicating an error, was “clearly and obviously” incorrect, and which are introduced as normal bug fixes (whether or not they had been fixed with previous maintenance).

11.1 Backwards incompatibilities

Backwards incompatibilities are described per product in the following sections.

11.1.1 Janus SOAP XML processing

The following backwards compatibility issues have been introduced in the *Janus SOAP* Xml* methods.

11.1.1.1 Omit dangling space in null string PI

Previously, a Processing Instruction (PI) node whose value is the null string serialized with an extra space; for example:

```
<?x ?>
```

This is contrary to the canonical XML recommendation, according to which the above PI should be serialized as:

```
<?x?>
```

This change has also been provided in maintenance to version 6.7.

11.1.1.2 **xml:lang handling (compatibility)**

In previous versions of *Janus SOAP*, the `xml:lang=".."` attribute allowed any value. It now accepts only the language identifier tags specified in IETF RFC 3066.

11.1.1.3 **Declaration of xml prefix and URI (compatibility)**

There is a new bug fix (as mentioned below in [“Declaration of xml prefix and URI \(bug fix\)” on page 43](#)): Previous versions of *Janus SOAP* allowed nonsensical namespace declarations involving the `xml` prefix or the URI it is always associated with (<http://www.w3.org/XML/1998/namespace>). Such declarations are no longer allowed.

11.1.2 **Janus SOAP ULI**

11.1.2.1 **“This” not allowed as class name**

The name `This` is no longer allowed as a class name.

11.2 **Fixes in Sirius Mods 6.8 but not in 6.7**

This section lists fixes to functionality existing in the *Sirius Mods* version 6.7 but which, due to the absence of customer problems, have not, as of the date of the release, been fixed in that version.

11.2.1 **Binary attribute values in LDAP Helper**

In previous versions of *Janus SOAP*, if the result of one of the `Ldap Find` methods contains binary data that is not a valid XML string, a *Model 204* snap is issued and the user is restarted. In version 6.8, such data simply causes a request cancellation, with a message that displays a fragment of the value that includes the invalid character.

11.2.2 **xml:lang handling**

In previous versions of *Janus SOAP*, the `xml:lang=".."` attribute allowed any value. It now accepts only the language identifier tags specified in IETF RFC 3066. This is mentioned as a compatibility issue in [“xml:lang handling \(compatibility\)”](#).

11.2.3 Declaration of xml prefix and URI (bug fix)

Previous versions of *Janus SOAP* allowed nonsensical namespace declarations involving the `xml` prefix or the URI it is always associated with (<http://www.w3.org/XML/1998/namespace>) — either associating `xml` with a different URI, or associating a different prefix with that URI. Such declarations are no longer allowed.

This change applies to the various deserialization methods (for example, `LoadXml` and the `HttpResponse` class `Parse` method), the `AddNamespace` method, and the URI argument of the various XML document `Add/Insert` methods.

This is mentioned as a compatibility issue in [“Declaration of xml prefix and URI \(compatibility\)” on page 42](#).

11.3 Version corequisites

This section lists any restrictions on usage of various products (including *Sirius Mods* itself) that will be imposed by use of version 6.8 of *Sirius Mods*.

- There are no corequisites associated with *Sirius Mods* 6.8.

