
Notes for Sirius Mods Interim Release 7.0

February, 2007



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 Debugger*
- *Janus SOAP*
- *Sirius Debugger*
- *SirTune*

are proprietary products of Sirius Software, Inc.:

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

Model 204® is a proprietary product of Computer Corporation of America, a wholly-owned subsidiary of Rocket Software, Inc., which owns the trademark:

Rocket Software Corporate Office
M204 Division
275 Grove Street
Suite 3-410
Newton, Massachusetts 02466-2272
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
MAXBG system parameter	5
Initial clause for Longstring variables	5
FUNCOPTS system parameter X'02' bit	6
Chapter 4: Janus Debugger and Sirius Debugger	7
Assembler language replaces User Language	7
SIRDEBUG file no longer needed	7
Workstation client available from Sirius Web site	7
Licensing simplified	7
Set multiple breakpoints with one command	8
Maximum number of breakpoints increased	8
New mappable commands for Client GUI	8
breaks	8
breaksAt	8
runWithoutDaemons	8
stepOut	9
Ability to display FEO Loop value	9
Simplified SIRIUS DEBUG ON syntax	9
SDEBGUIP system parameter	9
SDEBWRKP system parameter	9
SIRIUS DEBUG ON examples	9
Chapter 5: Janus SOAP ULI	11
Enhancements not part of any Janus SOAP ULI System class	11
Is [Not] Defined Visible tests	11
Stringlist class enhancements	12
Named Parameters for Print Method	12

RegexSplit method	13
Enhancements to support for regex methods	14
Daemon class enhancements	16
RunAsynchronously subroutine	18
RunIndependently subroutine	20
WaitAsynchronous function	21
Enhancements to File classes	23
Request cancelled if group field reference nonsensical for file	23
Exact rules of group field cancellation	25
Using Is Defined/Visible to avoid cancellation	27
Clear subroutine	28
New CursorState enumeration value: NoRecord	28
Chapter 6: Janus SOAP Xml Classes	29
Null string SelectionNamespace value now means "no namespace"	29
IsSelectionPrefix: Check if prefix has XPath selection association	30
DeleteSelectionPrefix: Delete any XPath selection association for prefix	30
Next and Previous methods	30
XPathNodeID method	31
ExclCanonical and WithComments options on Serial method	32
AddTrailingDelimiter argument of Serial method	35
LinefeedNoTrailingTabs option of deserialization methods	35
Support for numeric comparisons in XPath predicates	37
Highlights	37
Examples	41
Examples: Direct numeric comparison	41
Examples: Direct numeric comparison request cancellation	42
Examples: number([locPath]) function for non-numeric data	42
Examples: number(locPath) != n, locPath result is empty node- set	43
Examples: number() request cancellation	45
Example: number() default argument is "."	45
Document deserialization relinquishes CPU	45
LoadXml now supports Stringlist items longer than 6K	46
XmlDoc restored after parse errors	46
XML parsing cancels request if CCATEMP full	46
InvalidCharacterPosition function (actually part of 6.9)	47
Chapter 7: SirTune	49
SYSPARM data	49
System method quads	49
Compatibility	49

Chapter 8: Compatibility/Bug fixes	51
Backwards incompatibilities	51
Janus SOAP XML processing	51
XPath with “!=” and Element with more than 1 child	51
Fixed XPath “following” and “xx-sibling” axes	52
Corrected XPath results with unusual axes	52
Performance considerations: Document order, certain axes	54
LoadXml support of Stringlist items longer than 6K	58
Indent option limited to 254	58
Null string SelectionNamespace doesn't mean "no association"	58
XmlDoc not reinitialized after parse errors	59
Janus SOAP ULI File classes	60
Request cancellation due to nonsense group field usage	60
Fixes in Sirius Mods 7.0 but not in 6.9	60
Version corequisites	60

CHAPTER 1 ***Introduction***

This document lists the enhancements and other changes contained in *Sirius Mods* version 7.0, which was released in February, 2007. The previous version of the *Sirius Mods*, 6.9, was available in August, 2006.

Because 7.0 was an interim version not made available to all customers, all customers were required to upgrade to the next commercial release of *Sirius Mods* when it became available. This manual is still available as a historical record of the changes introduced by 7.0.

2.1 Model 204 support

As of version 7.0, *Sirius Mods* no longer supports *Model 204* version V5R1.0; only V6R1 is supported.

2.2 Documentation

- The ***Sirius Mods Command and Parameter Reference Manual*** is now available. It describes all *Model 204* commands and parameters and *Model 204* Editor commands which are implemented as part of the *Sirius Mods*.
- The “XPath” appendix in the ***Janus SOAP Reference Manual*** has been slightly reorganized and extensively updated, including the new “Comparison tests in predicates” section.
- The descriptions of the \$functions and methods that perform regular expression processing, as well as the discussion of the rules for *Sirius* regex, have been extensively updated. The ***Sirius Functions Reference Manual*** and ***Janus SOAP Reference Manual*** include new “Regex Processing” sections.

 CHAPTER 3 *All or Multiple Products*

3.1 MAXBG system parameter

The MAXBG parameter is a user0 parameter that must be set in the CCAIN stream. It specifies the maximum number of background or independent daemon requests that may be running at once. Before *Sirius Mods 7.0*, the only way to create a background daemon request was with the \$commbg function. Under *Sirius Mods 7.0* and later, independent daemon requests can also be created using the Daemon class RunIndependently method. Daemons running either as a result of RunIndependently or \$commbg are both counted the same way against the MAXBG limit.

If a \$commbg request would exceed the MAXBG limit (or there are no daemon threads immediately available to run the request) the request is enqueued and run later when daemon threads are available and the number of running background/independent requests drops below MAXBG. If a RunIndependently request would exceed the MAXBG limit, the request is cancelled.

The default value of MAXBG is the number of daemon (SDAEMDEV) threads divided by two or minus 10, whichever is greater. For example, if there are 10 daemon threads defined in the **ONLINE**, MAXBG defaults to 5. If there are 40 daemon threads, MAXBG defaults to 30. While there was no explicit MAXBG parameter before *Sirius Mods 7.0*, the default MAXBG was used “under the covers” by \$commbg, anyway.

Setting MAXBG to a value greater than or equal to the number of daemon threads allows background/independent requests to use up every single available daemon thread, if they wish. Setting MAXBG to 0 indicates that the default value of MAXBG, calculated from the number of daemon threads is to be used. This means that the lowest explicit value of MAXBG that will be honored is 1.

3.2 Initial clause for Longstring variables

The Initial clause may now be used for Longstring variables. The length of the initial value must be 255 or less. For example:

```

Begin
%aLottaDots Longstring Initial($Lstr_Left('.', 250, '.'))
%aLottaDots = %aLottaDots With %aLottaDots
Print %aLottaDots
End
  
```

3.3 FUNCOPTS system parameter X'02' bit

The X'02' bit of the FUNCOPTS system parameter causes all \$list functions that encounter CCATEMP full conditions to act as if the LISTFC \$sirparm were set to 1, that is, to cancel the request with a CCATEMP full condition.

This feature was also implemented via maintenance zaps in versions 6.8 and 6.9 of the *Sirius Mods*.

Janus Debugger and Sirius Debugger

The *Janus Debugger* is a tool designed for software developers who create and maintain Janus Web Server applications. The *Sirius Debugger* is designed to debug classical screen-based 3270 applications as well as BATCH2 applications. Though the two debuggers use different thread types on the host, they can run concurrently, and they both use the same Windows-based GUI client. For more information about the debuggers, see the ***Janus/Sirius Debugger User's Guide*** on the Sirius web site Documentation page.

The debuggers were first available with Version 6.8 of the *Sirius Mods* and have been steadily enhanced since. Some enhancements are delivered solely by new releases of the GUI client, while other enhancements depend upon features added to new releases of the *Sirius Mods*. Each of the GUI releases is capable of working with all versions of the *Sirius Mods*. The features described below require both release 7.0 or later of the *Sirius Mods* and the latest release of the client GUI.

4.1 Assembler language replaces User Language

In order to hasten initial delivery, the *Model 204* (server-side) support for the debuggers was originally provided by User Language programs in conjunction with *Janus Sockets* and a special set of dollar functions. This User Language code has been converted to assembler and integrated within the *Sirius Mods* as of Version 7.0.

4.1.1 SIRDEBUG file no longer needed

Downloading a User Language dump file (SIRDEBUG) to install the server-side support for the debuggers is no longer necessary.

4.1.2 Workstation client available from Sirius Web site

The Windows Installer program (setup.exe) for the Debugger workstation client, which was distributed within the SIRDEBUG procedure file for *Sirius Mods* prior to version 7.0, is now downloadable from the Sirius web site.

4.1.3 Licensing simplified

As of version 7.0 of the *Sirius Mods*, users of *Janus Debugger* and *Sirius Debugger* are no longer required to license *Janus SOAP* and *Janus Sockets*. Instead, a new Debugger Server port type (DEBUGGERSERVER) replaces the SRVSOCK type that is

required for earlier versions of the *Sirius Mods*, and a new port type (DEBUGGERCLIENT) replaces the CLSOCK type that is required for the *Sirius Debugger* for earlier versions of the *Sirius Mods*.

Using the new STATUS option of the SIRIUS DEBUG command produces a simple status report of the Debugger worker threads.

4.2 Set multiple breakpoints with one command

Options are added to set multiple breakpoints at once: on all lines that match a search string or regex, or on executable statements that follow comment lines that begin with *Break.

4.3 Maximum number of breakpoints increased

The breakpoint maximum is increased from 40 to 1000 per User Language request.

4.4 New mappable commands for Client GUI

The debugger Client GUI allows the end user to customize the assignment of commands for common functions to buttons and hot keys. The following new GUI commands require Version 7.0 of the *Sirius Mods*:

4.4.1 breaks

Sets breakpoints on lines after comments that have the form *break.

4.4.2 breaksAt

Sets breakpoints on lines that match a search string.

4.4.3 runWithoutDaemons

Runs until end-of-request, or until an error or breakpoint is encountered; does not pause at daemon code.

4.4.4 stepOut

Steps out of a subroutine or method.

4.5 Ability to display FEO Loop value

In FEO loops, you can display the current occurrence value.

4.6 Simplified SIRIUS DEBUG ON syntax

Invocation of the *Sirius Debugger* has been simplified. Two new User 0 parameters may be used to set system-wide default values for two operands required by the **SIRIUS DEBUG ON** command.

4.6.1 SDEBGUIP system parameter

This User 0 parameter may be reset by a System Manager. If set to a non-zero value, it provides a default for the workstation port number listened to by the Debugger Client. If SDEBGUIP is set to 0 (its default), then there is no default workstation port value, and every SIRIUS DEBUG ON command must specify a port number for connecting with the debugger GUI on the client's workstation.

4.6.2 SDEBWRKP system parameter

This User 0 parameter may be reset by a System Manager. If set to a non-zero value, it provides a default for the port number listened to by the current Online for spawning Debugger worker threads. If SDEBWRKP is set to 0 (its default), then there is no default port value, and every SIRIUS DEBUG ON command must specify a port number for spawning *Model 204* Debugger worker threads.

4.6.3 SIRIUS DEBUG ON examples

Given that the Debugger Client port is likely to be standardized at an installation, and that for a given Online there is probably only one port for spawning Debugger worker threads, providing system-wide defaults for these port values lets you start debugging just by providing the IP address for the workstation being used by the programmer to run the debugger GUI client.

In the following example, the DEBUGGERSERVER port for the current online is 3226, the Debugger Client is configured to listen to port 8080, the name of the *Sirius Debugger* client port is DEBUGC, and the IP address for the workstation running the debugger GUI is 198.242.244.235.

```
V SDEBGUIP,SDEBWRKP
SDEBGUIP 8080          SIRIUS DEBUG DEFAULT GUI PORT NUMBER
SDEBWRKP 3226          SIRIUS DEBUG DEFAULT WORKER PORT NUMBER

SIRIUS DEBUG ON DEBUGC 198.242.244.235
```

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

The first section in this chapter lists changes which are not changes to any specific class (in the System namespace) provided with the *Janus SOAP ULI*. Other sections list changes specific to the various *Janus SOAP ULI* classes in the System namespace.

5.1 Enhancements not part of any Janus SOAP ULI System class

The following sections list changes that are not changes to any specific system class provided with the *Janus SOAP ULI*.

5.1.1 Is [Not] Defined | Visible tests

Two new forms of tests (and their negations) are now available for any field in a *Model 204* file. These tests are not in any way based on an object in the File classes, but they are closely related to the features described in [“Request cancelled if group field reference nonsensical for file” on page 23](#).

The tests are useful for User Language code which is (or may be) in the context of a *Model 204* group (of files) when that code contains field references which may be nonsensical in one or more members of the group. They allow you to bracket field accesses, and certain updating statements, in group context if the rules described in [“Request cancelled if group field reference nonsensical for file” on page 23](#) would cause undesirable request cancellation.

Examples of the tests are shown below with `If` blocks:

```
If GRPFLD1 Is Defined Then Add GRPFLD1 = %val
End If
If GRPFLD2 Is Visible Then Delete GRPFLD2
End If
```

The tests can be used wherever User Language tests can be made, which is, loosely speaking, as a User Language expression (not including the `Print/Audit` statement, nor “values” in the `Find` and field update statements), but typically the tests will be used in the `If`, `ElseIf`, and `Repeat` statements.

These tests, like the `Is Present` test, can only be used in a record context, such as a `For Each Record` loop. The syntax is:

```
fld Is [Not] Defined
fld Is [Not] Visible
```

Where:

- fld** is either the name of a field or is a fieldname variable (%%var) containing the name of a field.
- Is Defined** This test returns the value “1” if the field is defined in the current file (as given by the `$CurFile` function); otherwise it returns the value “0”.
- Is Not Defined** This test returns the value “0” if the same `Is Defined` test would return 1, and returns the value “1” if `Is Defined` would return 0.
- Is Visible** This test returns the value “1” if the field is defined in the current file (as given by the `$CurFile` function) and does not have the INVISIBLE field attribute in that file; otherwise it returns the value “0”.
- Is Not Visible** This test returns the value “0” if the same `Is Visible` test would return 1, and returns the value “1” if `Is Visible` would return 0.

See [“Using Is Defined/Visible to avoid cancellation” on page 27](#) for more examples of the use of `Is Defined` and `Is Visible`.

Note: If a fieldname variable is used in any of the above tests and it contains a string which is not defined in the current file nor any member of the current group (when the reference is in group context), the request is cancelled. This cancellation is consistent with the handling of fieldname variables throughout User Language and has nothing to do with the request cancellation introduced for *Janus SOAP ULI* protection against nonsensical field references.

5.2 Stringlist class enhancements

5.2.1 Named Parameters for Print Method

The `Print` method now supports named parameters. The new `Print` method template is:

```
%rc = %sl:Print([NumWidth=] itemnumlen,
               [LenWidth=] itemlenlen,
               [Start=] firstitem,
               [MaxItems=] maxitems)
```

where all the parameters are optional. For example, to display the contents of a Stringlist with 6 bytes to be used for the column containing the item width, and limiting the display to 20 items, one can use positional parameters:

```
%list:print(,6,,20)
```

Under *Sirius Mods 7.0*, one can do the same thing with named parameters:

```
%list:print(lenWidth=6, maxItems=20)
```

5.2.2 RegexSplit method

This method repeatedly applies a regular expression, or “regex,” to a given input string until it has tested the entire string. This splits the string into the substrings that are matched by the regex (the “separators”) and the substrings that are not matched. By default, the method saves each *unmatched* substring as a separate item in the Stringlist method object. The leftmost unmatched substring is the first item in the Stringlist, the next leftmost is the second item, and so on.

A simple application of the method default is to extract only the data items from a string of comma-separated data items. If the specified regex is a comma, each of the resulting Stringlist items will contain one of the data items.

The Stringlist that is returned by a default invocation of RegexSplit will contain at least as many items as there are instances of matched substrings. Upon each match, the input string characters preceding the matched ones (and since the previous matched ones) are saved as a Stringlist item. If there are consecutive matching substrings (no unmatched characters between the matching ones), the corresponding Stringlist item for the second matching substring is empty.

RegexSplit also has non-default options that let you save the following in the Stringlist:

- only the matched substrings
- both the matched and unmatched substrings
- only the substrings that are matched by capturing groups in the specified regex
- the unmatched substrings and the substrings matched by capturing groups

```
[%retcode =] %sl:RegexSplit( inStr, regex      -
                             [, Options=%str]  -
                             [, Add=output]    -
                             [, Status=%num] )
```

RegexSplit Function

%retcode, if specified, is a number that is 0 if the regular expression was invalid or no match was found or some error occurred, or it is the number of items added to the method Stringlist *%sl*.

5.2.3 Enhancements to support for regex methods

The following features are added to the Sirius regular expression support. This support is offered by the five regex methods in the `Stringlist` class and by the `$RegexMatch` and `$RegexReplace` \$functions, and it is summarized in the “Regex rules” sections where these methods and \$functions are described in the *Janus SOAP Reference Manual* and *Sirius Functions Reference Manual*.

- The PDL space requirement recommended when using Sirius regex methods and \$functions is reduced from a minimum LPDLST setting of 8000 to one of 3000.
- “Lazy,” or “non-greedy,” matching (the lazy quantifiers like `*?`, `+?`, `??` etc.) is now supported. This support mimics that provided in the Perl language.
- As in Perl, `\w` and `\W` multi-character escape sequences are supported (except within values in replacement-string arguments in those methods and \$functions that have such arguments):

`\w` Matches any letter (uppercase or lowercase), any digit, or the underscore

`\W` The inverse of `\w`: any non-letter or non-digit except the underscore.

This support for “word characters” does *not* include awareness of locale or Unicode considerations, however.

- The handling of square bracket characters (`[`, `]`) is now supported in Sirius regex as it is in Perl, except when Sirius methods or \$functions use the `Options='C'` argument (XML Schema mode).

Previously, square bracket characters (from any of the several character codes that produce a left or right square bracket in EBCDIC) were legal in a Sirius regex only as metacharacters indicating a character class; otherwise, they had to be preceded by an escape character. As of version 7.0 of the *Sirius Mods*, a square bracket character that is not being processed in XML Schema mode does *not* require a preceding escape character if it is:

- A right bracket (`]`) that is outside of, not part of, a character class expression. So, `(1]9)` matches `0001]9zzz`.
- A right bracket that is the first character (or the second, if the first is a caret (`^`)) in a character class expression. So, `[]xxx` and `[^]xxx` are legal.
- A left bracket that occurs anywhere in a character class expression. So, `[abc[]` is legal and matches any of these four characters: `a b c [`

A left bracket that occurs outside of a character class expression must always be escaped.

Although not required, escape characters may be used in the cases cited above. If XML Schema mode is used, however, these cases are compiler errors unless the cited bracket characters are escaped.

- Multi-character escape sequences (for example, `\s`, `\c`) are now legal within a character class — but not as either side in a range.
- A character class may now contain an unescaped hyphen (-) as a simple character, if it occurs as the first character (or the second, if the first is `^`) or as the last character. An escaped hyphen (`\-`) remains legal.

All the following are legal:

```
[-A-Z...]  
[^-A-Z...]  
[...A-Z-]  
[...A-Z\0-]
```

- For those `$functions` and methods that support the XML Schema mode option, “charClassSubtraction” is now supported in that mode only. You can exclude a subset of characters from the characters already designated to be in the class. This is *not* allowed in Perl.

The feature lets you specify a character class like the following, which matches anything from A to Z except D, I, O, Q, U, or V:

```
[A-Z-[DIOQUV]]
```

You can also nest subtractions, as in:

```
[\w-[A-Z-[DIOQUV]]]
```

Characters immediately after the right bracket of a subtracted character class are **not** allowed. `[A-Z-[DIOQUV]abc]` is an *invalid* character class.

You can reverse the sense of inclusion or exclusion by using a negation (`^`) immediately after an opening bracket. `[A-Z-[^DIOQUV]]` is *valid*.

- The Status argument return value for a regex expression compilation error is changed from `-1` to `-1nnn`, where the absolute value of the return minus 1000 gives the 1-based position of the character being scanned when the error was discovered. An error occurring at end-of-string will have a 1-based value equal to the length of the string + 1.

5.3 Daemon class enhancements

Support has been added for asynchronous and independent running of daemon requests. This included two new daemon methods, `RunAsynchronously` and `RunIndependently`:

```
%pantalaimon is object daemon
...
%pantalaimon = new
...
%pantalaimon:runAsynchronously(%commands)
...
%pantalaimon:runIndependently(%commands)
```

As the names suggest, both of these methods return immediately, before the daemon has completed processing its input commands. With `RunAsynchronously`, when the daemon commands are all processed, the daemon goes back to waiting for the master to tell it what to do next. To do this, the master thread has to make sure the daemon thread has completed processing by issuing the (also new in *Sirius Mods 7.0*) `WaitAsynchronous` method against the daemon object:

```
%pantalaimon is object daemon
...
%pantalaimon:runAsynchronously(%commands)
...
%pantalaimon:waitAsynchronous
```

Even if a daemon thread running asynchronously has completed processing its commands, a `WaitAsynchronous` method **must** be issued against the daemon object before further commands can be sent to the daemon. Most methods applied to a daemon object that is running asynchronously will result in request cancellation. Once the daemon that had been running asynchronously has been resynched with the master thread via `WaitAsynchronous`, the master thread can treat it just as any other daemon thread and issue further `Open`, `Run`, `RunAsynchronous`, `RunIndependently` or other methods against the daemon.

With `RunIndependently`, when the daemon commands are all processed, the daemon logs off. This means that a daemon thread for which `RunIndependently` is issued can no longer be controlled by its master thread (as the method name would suggest). In fact, upon return from the `RunIndependently` method, the daemon object is set to null, so any further methods invoked against it would result in request cancellation. `RunIndependently` is roughly analogous to the `$commbg` function with some important differences:

- The `RunIndependently` method allows interaction with the daemon before it does its independent processing.
- `RunIndependently` operates on a daemon object that already has a daemon thread associated with it. `$commbg` obtains a daemon thread as part of its processing and,

if one is not available or the maximum independent/background daemons are already running, the request is enqueued. This is a good news bad news thing. If the master thread wants to be sure that the daemon thread is actually running, the `RunIndependently` has the preferred behavior. If the master thread doesn't care whether the request runs immediately and, in fact, would prefer that the request be run later if/when it's convenient, then `$commbg`'s behavior is preferable. Note, however, that there is no guarantee that a request enqueued by `$commbg` will ever be run — it is quite possible that the request remains enqueued until the **ONLINE** is brought down.

Despite their similarities, `$commbg` and `RunIndependently` use different terminology for the daemon threads. `$commbg` refers to the threads running “in the background” while `RunIndependently` refers to them running “independently”. As alluded to above, there is a limit to the number of background or independent requests that may be running at a time. When this limit is exceeded for `$commbg` requests (or when there are simply no daemon threads available to run the requests), the requests are enqueued, until a daemon thread is available to run the request and the limit for background/independent requests would not be exceeded. `RunIndependently`, on the other hand, causes a request cancellation if the limit for background/independent requests is to be exceeded. `RunIndependently` cannot fail because there are no daemon threads available because the daemon thread is assigned to the object at the time it's instantiated (with the `New` method) so `RunIndependently` does not have to allocate a daemon thread for the request.

Before *Sirius Mods 7.0*, the limit for background/independent requests was one half of, or ten fewer than the number of daemon threads, whichever was greater. That is, if there were 10 daemon threads defined in an **ONLINE**, no more than 5 of them could be running `$commbg` requests. If there were 40 daemon threads defined, no more than 30 could be running `$commbg` requests. Under *Sirius Mods 7.0* and later, there is a system parameter call `MAXBG` that indicates the maximum number of simultaneously running background/independent requests that will be allowed at a time. For backward compatibility with previous *Sirius Mods* versions, the default value for `MAXBG` is one half of, or ten fewer than the number of daemon threads defined to the **ONLINE**, whichever is greater.

Daemon threads running asynchronously and threads that are about to run independently, but aren't yet, are counted against the master thread's `MAXDAEM` limit. When, a thread switches to running independently, it is no longer counted against the master's `MAXDAEM` limit.

`RunAsynchronously` can be used to allow a thread to do parts of large, complex tasks in parallel. The master thread could be doing some of the work at the same time a daemon thread is asynchronously doing some of it. In fact, multiple daemon threads can be running asynchronously at the same time for a given master thread. When the master thread is done with its work, it can then do a `WaitAsynchronous` for each daemon running asynchronously until all the work is completed. If the work being run in parallel is very CPU intensive, there is not likely to be any benefit in running work asynchronously unless the **ONLINE** is using multiple tasks (MP/204 authorized and

AMPSUBS>0). On the other hand, if the work being performed asynchronously is I/O intensive, a significant reduction in the time required to perform the work can be achieved over doing all the I/O serially in the master thread. Work involving network delays is an ideal candidate for running asynchronously, especially if multiple such delays are present in a batch of work.

5.3.1 **RunAsynchronously subroutine**

This method runs on the daemon thread the command or set of commands specified by its first argument. Unlike the Run method, this method returns immediately and the thread issuing the RunAsynchronously method can run in parallel with the daemon thread.

```
%daem:RunAsynchronously(command, [%inputObj])
```

RunAsynchronously syntax

Syntax Terms

- %daem* A previously defined Daemon object.
- command* A string or Stringlist that is the required command or the set of commands executed by the daemon.
- %inputObj* The object passed to the daemon method object. This optional argument is passed by deep copy and not by reference, so *%inputObj* must be deep copyable.

Usage Notes

- Issuing RunAsynchronously against a transactional daemon results in request cancellation.
- If the daemon thread and its daemons hold record locks that conflict with the parent thread's family (excluding the daemon thread and its daemons), RunAsynchronously results in request cancellation.
- This is an example RunAsynchronously call:

```
%strlist = %daem:runAsynchronously(%list2, %x)
```

- After a RunAsynchronously method, the daemon thread is considered to be running asynchronously until a WaitAsynchronous method ([“WaitAsynchronous function” on page 21](#)) is issued against the daemon object. This is the case, even if the daemon thread has completed processing all of the input commands.

- While the daemon thread is running asynchronously, methods that require interaction with the daemon thread cause request cancellation. These methods include Run, RunAsynchronously, RunIndependently, Open, and LastCommandErrorCount.
- The WaitAsynchronous method can be used to retrieve any output from the commands run via RunAsynchronously. This includes the terminal output and any output or info object.
- RunAsynchronously's Stringlist argument can pass multiple commands to the daemon.
- The passing of objects and command to daemons is identical whether the method is a Run or RunAsynchronously.
- If the daemon object associated with an asynchronously running daemon is discarded either explicitly or implicitly, the daemon thread is bumped by the parent thread. An implicit discard can happen at request end for non-global daemon objects or at user logout.
- The inputs and outputs from the combination of RunAsynchronously and WaitAsynchronous are identical to the inputs and outputs from the Run method. As such, it should be a relatively simple task to split a Run into a RunAsynchronously and WaitAsynchronous, allowing the daemon processing to be performed in parallel with parent thread processing or the processing on another daemon thread. For example, if a request has the following:

```
%out1 = %daem1:run(%cmds1, %inobj1, %outobj1)
%out2 = %daem2:run(%cmds2, %inobj2, %outobj2)
```

it can be easily split up into:

```
%daem1:runAsynchronously(%cmds1, %inobj1)
%daem2:runAsynchronously(%cmds2, %inobj2)
%out1 = %daem1:waitAsynchronous(%outobj1)
%out2 = %daem2:waitAsynchronous(%outobj2)
```

and the processing on the two daemons would be performed in parallel with each other. Note that this will buy nothing if the requests are CPU-intensive and the request is not running with AMPSUBS>0 (requires MP/204). Note also that if the daemons hold or require record locks that might conflict with each other, such a split will not work. Finally, since RunAsynchronously is not allowed for a transactional daemon, such a split is not feasible for transactional daemons.

5.3.2 **RunIndependently subroutine**

This method runs on the daemon thread the command or set of commands specified by its first argument. Unlike the Run method, this method returns immediately and the thread issuing the RunIndependently method can run in parallel with the daemon thread. Unlike the RunAsynchronously method, this method makes the daemon thread completely independent of the parent thread so that the output from the commands can never be retrieved.

`%daem:RunIndependently(command, [%inputObj])`

RunIndependently syntax

Syntax Terms

- %daem* A previously defined Daemon object.
- command* A string or Stringlist that is the required command or the set of commands executed by the daemon.
- %inputObj* The object passed to the daemon method object. This optional argument is passed by deep copy and not by reference, so *%inputObj* must be deep copyable.

Usage Notes

- Issuing RunIndependently against a transactional daemon results in request cancellation.
- If the daemon thread and its daemons hold record locks that conflict with the parent thread's family (excluding the daemon thread and its daemons), RunIndependently results in request cancellation.
- This is an example RunIndependently call:

```
%strlist = %daem:runIndependently(%list2, %x)
```
- The RunIndependently method is the rough Daemon class equivalent of the \$commbg function.
- After a RunIndependently method, the daemon object is set to null. This is because the daemon thread runs completely independently of the parent thread once a RunIndependently method is invoked so there is nothing useful the parent thread can do with such a daemon object, anyway.
- Even if the parent thread of an independently running daemon logs off, the daemon thread can continue to run.

- No daemon class mechanism is provided for a parent thread to retrieve the output from an independently running daemon.
- RunIndependently's Stringlist argument can pass multiple commands to the daemon.
- The passing of objects and command to daemons is identical whether the method is a Run or RunIndependently.

5.3.3 WaitAsynchronous function

This callable method waits for the completion of the commands issued on a daemon thread by the RunAsynchronously method (“RunAsynchronously subroutine” on page 18) and retrieves various outputs from those commands.

```
[%strL =] %daem:WaitAsynchronous([%outputObj] -
                                [, Info=%infoObj])
```

WaitAsynchronous syntax

Syntax Terms

- %strL* If specified, a Stringlist object that contains the terminal output from the command or commands run on the daemon thread.
- %daem* A previously defined Daemon object.
- %outputObj* The object returned from the daemon method object by the ReturnObject method invoked on the daemon thread. This optional argument is passed by deep copy and not by reference, so *%outputObj* must be deep copyable.
- Because *%outputObj* is an output variable, it cannot itself be contained inside an object: that is, it must be a local or a common %variable.
- Info=* An optional, name required parameter that indicates a second output object returned from the daemon method object by the ReturnInfoObject method invoked on the daemon thread. This optional argument is passed by deep copy and not by reference, so *%infoObj* must be deep copyable.
- Because *%infoObj* is an output variable, it cannot itself be contained inside an object: that is, it must be a local or a common %variable.

Usage Notes

- If WaitAsynchronous is issued against a daemon object that is not currently running asynchronously (RunAsynchronously was issued against it), the request is cancelled. Note that this does not mean that the daemon must actually still be running — if the daemon thread has run all the commands in the RunAsynchronously call, not only is WaitAsynchronous allowed, it is required before anything else can be done with the daemon. And, in any case, it's the only way of retrieving the outputs from the asynchronous request.
- This is an example WaitAsynchronous call:

```
%daem:runAsynchronously(%commands)
...
%strlist = %daem:waitAsynchronous(%list2)
```

- The output Stringlist and parameters from WaitAsynchronous are identical to the output Stringlist and parameters for the Run method.
- The inputs and outputs from the combination of RunAsynchronously and WaitAsynchronous are identical to the inputs and outputs from the Run method. As such, it should be a relatively simple task to split a Run into a RunAsynchronously and WaitAsynchronous, allowing the daemon processing to be performed in parallel with parent thread processing or the processing on another daemon thread. For example, if a request has the following:

```
%out1 = %daem1:run(%cmds1, %inobj1, %outobj1)
%out2 = %daem2:run(%cmds2, %inobj2, %outobj2)
```

it can be easily split up into:

```
%daem1:runAsynchronously(%cmds1, %inobj1)
%daem2:runAsynchronously(%cmds2, %inobj2)
%out1 = %daem1:waitAsynchronous(%outobj1)
%out2 = %daem2:waitAsynchronous(%outobj2)
```

and the processing on the two daemons would be performed in parallel with each other. Note that this will buy nothing if the requests are CPU-intensive and the request is not running with AMPSUBS>0 (requires MP/204). Note also that if the daemons hold or require record locks that might conflict with each other, such a split will not work. Finally, since RunAsynchronously is not allowed for a transactional daemon, such a split is not feasible for transactional daemons.

5.4 Enhancements to File classes

The following sections either list changes specific to the various File classes, or changes which apply to all of the four File classes.

5.4.1 Request cancelled if group field reference nonsensical for file

One of the benefits of the *Janus SOAP ULI* File classes is easier migration between file-context and group-context User Language. In most cases, this is provided by allowing the same User Language code to operate whether in file or group context.

However, there is an opportunity for undetected errors when migrating an application from file to group context. One flexibility of *Model 204* groups is that field definitions of the individual files can differ. In particular, a field can be missing in file “A” yet defined in at least one other group member, or it can be defined as INVISIBLE in file “A” and yet defined and not INVISIBLE in at least one other group member. When “A” is the current file, both of these situations can result in nonsensical references to the value of that field, or nonsensical updates to that field.

With version 7.0 of *Janus SOAP ULI*, your applications are protected against nonsense field references when done in the context of *Janus SOAP ULI* File objects. The **general** (but not exact - see [“Exact rules of group field cancellation” on page 25](#)) guideline is that, when both these conditions exist:

- a record is referenced in a group context bound to an object of one of the File classes

and

- a field in the record is referenced which would not be allowed if it were made in single file context using the current file

then the request will be cancelled.

For example, consider the following field definitions and group definition:

```
IN FILE F1 DEFINE FIELD ONLY1
IN FILE F1 DEFINE FIELD VIS1
IN FILE F2 DEFINE FIELD VIS1 WITH INVISIBLE ORDERED
CREATE TEMP GROUP G FROM F1, F2
PARAMETER UPDTFILE F1
END GROUP
```

Each of the User Language statements within the following `For Each Record` will result in compilation errors:

```
In File F2 For Each Record
  Print ONLY1
  Add ONLY1 = 'Field only in file F1'
  Print VIS1
  Delete VIS1
End For
```

These are illegal because:

- You cannot refer to a field which is not defined (`Print ONLY1` and `Add ONLY1`).
- You cannot refer to the value of a field which is INVISIBLE (`Print VIS1`).
- You cannot delete, by occurrence, a field which is INVISIBLE (`Delete VIS1`).

If you convert this application (which does not use the *Janus SOAP ULI* File classes) to use a group, it compiles and evaluates without any complaint:

```
In Group G For Each Record
  Print ONLY1
  Add ONLY1 = 'Field only in file F1'
  Print VIS1
  Delete VIS1
End For
```

However, when the current file in the `For Each Record` loop is F2, the following occur, which **may be an application error**:

- The value (`Print ONLY1`) of an undefined field is the null string.
- Updates to undefined fields (`Add ONLY1`) are ignored.
- The value (`Print VIS1`) of an INVISIBLE field is the null string.
- A certain class of updates to undefined fields (`Delete VIS1`) are ignored.

The *Janus SOAP ULI* takes a more conservative approach to handling fields which are undefined or INVISIBLE in some members of a group, when the field references are in records which are bound to an object in one of the File classes.

Using this protection, the following request is cancelled when the first record in file F2 is processed:

```
Begin
%rs Object Recordset In Group G
In Group G Find To %rs
End Find
For Each Record In %rs
  Print ONLY1
End For
End
```

Similarly, the request is cancelled when processing the first record in file F2 if the statement in the `For Each Record` loop is any of the following:

```
Add ONLY1 = 'Field only in file F1'

Print VIS1

Delete VIS1
```

There are two likely errors which the above request cancellations detect; either the field definitions in the group members are incorrect, or the application should have some way to determine when it makes sense to execute statements with references to fields undefined or `INVISIBLE` in some members of the group. One way your application can make this latter determination is shown in [“Using Is Defined/Visible to avoid cancellation” on page 27](#).

The exact request cancellation conditions introduced are shown in the next subsection.

As mentioned in [“Request cancellation due to nonsense group field usage” on page 60](#), this feature can create an incompatibility with existing use of *Janus SOAP ULI* File classes.

5.4.1.1 Exact rules of group field cancellation

When the following conditions exist:

- a record is referenced in group context;

and

- a record is referenced in a group context bound to an object of one of the File classes;

and

- using that record reference, a field referenced in a statement (or expression) **other than** one of the following:

```
Count Occurrences (CTO)
For Each Occurrence (FEO)
Delete Each
Is [Not] Present
Is [Not] Defined
Is [Not] Visible
$Field_Image (i.e., an item of the argument 1 Image)
$Field_LitstI(i.e., an item of the argument 2 Image)
Sort Records
```

and

- **one of** the following conditions exist:

- the field is not defined in the current file

or

- the field is INVISIBLE in the current file and the value of the field is referenced

or

- the field is INVISIBLE in the current file and is referenced in a Note statement

or

- the field is INVISIBLE in the current file and the field is updated by a “CHANGE field To newval” (not “CHANGE field = oldval To newval”) statement

or

- the field is INVISIBLE in the current file and the field is updated by a “DELETE field occurrence” (not “DELETE field = vale”) statement

then the request will be cancelled.

Note: In addition to the list of statements (CTO, FEO, etc.) above which do not cause request cancellation, a few updating statements are shown in the next list, and updating statements for INVISIBLE fields in the list after that, which do not cause request cancellation.

Other statements which do not cancel request

One of the conditions that are necessary for request cancellation due to nonsensical field references is that a record referenced in a group context is bound to an object of one of the File classes, and that a field is referenced using that record. There are other

statements which refer to fields, but which are not in a record context, and so are not subject to the *Janus SOAP ULI* nonsensical field reference protection:

- Store Record
- File Records Under
- For Each Value
- Find Records
- Find And Print Count
- “Where” and “Order By” clauses of For Each Record
- Find Values

Also, the following updating statements are allowed for INVISIBLE fields, and so do not cause request cancellation if the field is INVISIBLE in the current file:

- Add
- Change field = oldval To newval
- Delete field = value
- Insert

5.4.1.2 Using Is Defined/Visible to avoid cancellation

Using the same field and group definitions from [“Request cancelled if group field reference nonsensical for file” on page 23](#), here is a User Language request which skips nonsensical field references:

```
Begin
%rs Object Recordset In Group G
In Group G Find To %rs
End Find
For Each Record In %rs
  If ONLY1 Is Defined Then
    Print ONLY1
    Add ONLY1 = 'Field only in file F1'
  End If
  If VIS1 Is Visible Then
    Print VIS1
    Delete VIS1
  End If
End For
End
```

Note that this is not the same result as the second User Language example (“In Group G For Each Record”) in [“Request cancelled if group field reference nonsensical for file” on page 23](#); in that example, two blank lines are printed for each record in file F2 (by the `Print ONLY1` and `Print VIS1` statements).

5.4.2 Clear subroutine

Clear removes all of the records in a Recordset object, without discarding the object. It provides functionality identical to the User Language CLEAR LIST statement, but it does so for Recordset objects.

```
%rs:Clear
```

Clear syntax

Syntax Terms

%rs A non-null Recordset object variable.

Usage Notes

- If an active For loop references a Recordset object, either via a direct reference to the Recordset object or via a RecordsetCursor instantiated against the Recordset, and that Recordset is cleared by the Clear method, its record will be closed.
- Any RecordsetCursor object instantiated against a Recordset object that is cleared has its State property set to the new CursorState enumeration value of `NoRecord` described in [“New CursorState enumeration value: NoRecord”](#).

5.4.3 New CursorState enumeration value: NoRecord

A new value, `NoRecord`, has been added to the CursorState enumeration. This value corresponds to the state where the cursor **had** a valid position identifying a record in a Recordset, but a subsequent operation (Recordset Clear or RemoveRecord method) removed the record identified by the cursor from the underlying Recordset.

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

6.1 Null string SelectionNamespace value now means "no namespace"

Previously, the following assignment statement:

```
%doc:SelectionNamespace('xxpref') = ''
```

caused the string `xxpref` to no longer be usable as a prefix in an XPath expression. This was sensible, since the XPath recommendation specifies that a prefix used in an XPath expression must be associated with the URI of a namespace, and the null string is not a valid URI.

However, experience has shown that this aspect of the XPath recommendation creates some difficulty in the coding of XPath expressions in *Janus SOAP*. If a section of User Language code has the information about the namespace of a node it needs to select, it must use two different forms, depending on whether the node is in a namespace; for example:

```
* %uri has null string or desired node's namespace URI:
If %uri EQ '' Then
  %node = %inpNode:SelectSingleNode('infoChild')
Else
  %doc:SelectionNamespace('inf') = %uri
  %node = %inpNode:SelectSingleNode('inf:infoChild')
End If
```

To make it easier to handle situations like this, we have extended the XPath specification, allowing a prefix in an XPath expression to be associated with the null string. When such a prefix is used in an XPath expression, it matches Element or Attribute nodes that are not in a namespace. Now the above code can simply be written as:

```
%doc:SelectionNamespace('inf') = %uri
%node = %inpNode:SelectSingleNode('inf:infoChild')
```

As mentioned in [“Null string SelectionNamespace doesn't mean "no association"” on page 58](#), this creates a very small incompatibility if you are testing for a null value of a SelectionNamespace prefix to see if the prefix has any association. The following two

subsections ([“IsSelectionPrefix: Check if prefix has XPath selection association”](#) on page 30 and [“DeleteSelectionPrefix: Delete any XPath selection association for prefix”](#)) describe two new methods which can be used if you need to check or delete a namespace association for a prefix.

6.1.1 IsSelectionPrefix: Check if prefix has XPath selection association

This function of the XmlDocument class returns the Boolean value `True` if the string argument is currently associated (either to the null string or to a namespace URI) as a prefix which can be used in an XPath expression. Otherwise it returns `False`.

This method has no cancellation conditions other than the usual requirement that the method object not be Null.

6.1.2 DeleteSelectionPrefix: Delete any XPath selection association for prefix

This subroutine of the XmlDocument class removes any XPath selection association for its string argument.

This method has no cancellation conditions other than the usual requirement that the method object not be Null.

6.2 Next and Previous methods

The Next and Previous methods, both in the XmlNode class, return the XmlNode which is:

- for a non-Attribute node, the next and previous, respectively, sibling (in document order) of the method object. This is exactly the same as the node returned by `SelectSingleNode` with an argument of `'following-sibling::node()'` or `'preceding-sibling::node()'`[1], respectively.
- for an Attribute node, the next and previous, respectively, Attribute node that would be produced when the Element containing the method object is serialized in “normal” (i.e., not `ExclCanonical`) order.

As mentioned in [“Fixed XPath “following” and “xx-sibling” axes”](#) on page 52, the XPath recommendation specifies that the following-sibling and preceding-sibling axes are the empty nodeSet when the context node is an Attribute node. The Next and Previous methods are available if you want to traverse the Attribute nodes of an Element, without creating an XmlNodeList.

6.3 XPathNodeID method

The XPathNodeID method was designed to provide information when sending an error message to an XML client application so it can identify which node in the request XML doc is invalid.

```
strLis = nr:XPathNodeID([selection_XP])
```

This returns, as the first item of the result Stringlist, an absolute XPath expression which identifies the node selected by the argument. If this returned XPath expression contains no prefixes, the Stringlist contains only that one item and no further explanation is required.

However, if the returned XPath expression refers to elements and/or an attribute which have non-null namespace URIs, prefixes are employed in the usual way, but they must be associated with URIs. And so, after the first item in the Stringlist is a series of pairs of items, the first in each pair being a prefix used in the expression, and the second being the URI associated with it.

In most cases, the prefixes used will be taken from the document, which should facilitate spotting the error. However, if a namespace URI is needed and the generated XPath doesn't have a prefix from the document for that URI, or if a single prefix in the document is used for two different URIs, then XPathNodeID can generate an "invented" prefix. Assuming these are somewhat unusual cases, the examples below won't cover them. For example, given the following document contained in an XmlDocument used as the method object for XPathNodeID:

```
<a>
  <?p1?>
  <b>
    <c n="1" />
    <c n="2" />
  </b>
  <?p2?>
  <x:a xmlns:x="u:a">
    <x:b/>
  </x:a>
</a>
```

The following table shows the result's first item for various XPathNodeID arguments, and, where there is more than one item, the number of additional items and their values:

Argument:	First item of result, and any prefix/URI items:
-----	-----
/	/a/b
*/**	/a/b/c[1]
*/**[2]	/a/b/c[2]
*/processing-instruction()	/a/processing-instruction()[1]
*/processing-instruction()[2]	/a/processing-instruction()[2]
*/**/@*	/a/b/c[1]/@n
/	/
/[2]	/a/x:a 2 prefix/URI items: x u:a
/[2]/*	/a/x:a/x:b 2 prefix/URI items: x u:a

6.4 ExclCanonical and WithComments options on Serial method

These options are added to the Serial method so that it can be used to create a Longstring “extract” from an XML document, in support of digital signatures. In order to work with digital signatures, the information being signed must be represented in a unique format.

Since there are many equivalent ways that a string serialization can represent the information in a portion of an XML document, the W3C has produced a specification, Exclusive XML Canonicalization (<http://www.w3.org/TR/xml-exc-c14n/>). This, in turn, is based on another specification, XML Canonicalization (<http://www.w3.org/TR/xml-c14n/>).

Exclusive XML Canonicalization deals specifically with the placement of namespace declarations in a document; for example, the following two documents contain the same information (assuming that namespace declarations' only purpose is to support referencing attributes and elements):

```
<a xmlns:p="urn:ppp">  
  <p:b></p:b>  
</a>  
  
<a>  
  <p:b xmlns:p="urn:ppp"></p:b>  
</a>
```

The latter of these two documents is in Exclusive Canonical form; the declaration of prefix “p” is only placed where it is utilized (here, by element “p:b”).

With ExclCanonical, a namespace declaration is produced only if it is **utilized** by an element or attribute in the subtree; it will be produced in the start-tag of an element which uses it (or has an attribute using it) unless the parent of the element is in the subtree and the declaration is in scope at the parent.

An element **utilizes** an in-scope namespace declaration in either of these cases:

- the element is prefixed and the declaration is of that prefix;
- the element is unprefixed and it is a default namespace declaration.

An attribute **utilizes** an in-scope namespace declaration if the attribute is prefixed and the declaration is of that prefix.

There are two scenarios addressed by these rules; the first, exemplified above, is when a namespace declaration is not needed on the element at which it occurs (it may either be moved “down” to a descendant, or removed entirely if it is not utilized in the subtree).

The second scenario is when a namespace declaration is needed, but it is not present in the serialized subtree (because it is only present in an ancestor of the subtree). This case is exemplified by serializing the subtree with element “w” at the top from the following document:

```
<a xmlns:p="urn:ppp">
  <w>
    <p:b/></p:b>
  </w>
</a>
```

The exclusive canonical serialization of that subtree is:

```
<w>
  <p:b xmlns:p="urn:ppp"></p:b>
</w>
```

Other than control on the placement of namespace declarations, the Exclusive XML Canonicalization specification adopts various other features of the XML Canonicalization specification, to ensure that the information in a portion of an XML document is serialized uniquely. For example, it specifies that “Empty Element” tags are not to be used, so that the element “p:b” above is **not** serialized as:

```
</p:b xmlns:p="urn:ppp">
```

Exclusive XML Canonicalization can be obtained with the ExclCanonical option of the Serial method, by itself or in combination with the WithComments option. In addition to the namespace declaration placement discussed above, the result of using these options is as follows:

- Using the ExclCanonical option implies the results specified by the SortCanonical and NoEmptyElt options.
- Specifying WithComments without specifying ExclCanonical has no effect.
- Specifying ExclCanonical without specifying WithComments causes all Comment nodes to be suppressed from the result.
- Specifying WithComments together with ExclCanonical causes all Comment nodes in the subtree to be serialized.

In addition to the above, there are other rules in the Exclusive XML Canonicalization specification (all of which are followed by the ExclCanonical option) to control:

- how certain special characters are serialized (some of these are enforced by the Serial method regardless of options)
- linefeed insertion between children of the Root node

The changes to the *Janus SOAP Reference Manual* will provide details about these rules, as well as more examples.

Other notes:

- Since use of both the SortCanonical and NoEmptyElt options were baby steps toward Exclusive Canonicalization, these options will be deprecated, but still supported.
- XmlDecl is allowed with ExclCanonical, but it doesn't really make sense, since the spec says that the "XML declaration" is not part of canonical output. For whatever reason, if you use it to serialize the Root node of an XmlDocument which has a non-Null Version property, the XML declaration is followed by a linefeed. Note that Serial does not produce an XML declaration if the XmlDocument is empty.
- The canonical specs are based on the serialization of a **subset** of a document; the *Janus SOAP* Serial method with the ExclCanonical option is based on a **subtree**.
- The specs also support an argument to canonicalization which is a list of namespace declarations to be "forced" into the serialization; Sirius does not provide that.
- Other features which might be of benefit in the overall process of digital signature processing are described in ["AddTrailingDelimiter argument of Serial method"](#) on page 35 and ["LinefeedNoTrailingTabs option of deserialization methods"](#) on page 35.

6.5 AddTrailingDelimiter argument of Serial method

This optional named Boolean argument determines whether a final newline character is added to the result of Serial when one of the “newline” options (LF, CR, or CRLF) is used.

The default value of AddTrailingDelimiter is True; if it is specified as False, then the final newline character is not added. This is the same purpose as the argument of the same name of the Stringlist CreateLines method.

So, for example, given the following User Language fragment:

```
%doc:LoadXml('<a><b/></a>')
%s = %doc:Serial(, 'LF', AddTrailingDelimiter = False)
Print $Substr(%s, 1, 3) And 'X"' With -
      $C2X($Substr(%s, 4, 1)) With '"' And $Substr(%s, 5, 4) -
      And 'X"' With $C2X($Substr(%s, 9, 1)) With '"' -
      And $Substr(%s, 10)
Print 'Length:' And $Len(%s)
```

The result is:

```
<a> X"25" <b/> X"25" </a>
Length: 13
```

This demonstrates that there is no trailing linefeed character; if the AddTrailingDelimiter argument were omitted, a linefeed would be added after the final end tag, and the total length would be 14.

Note that specifying the AddTrailingDelimiter argument without one of the newline options has no effect on the result.

This feature may be useful if a digital signature must be created which includes newline characters between XML tags, but the XmlDocument does not contain those newline Text nodes. See [“ExclCanonical and WithComments options on Serial method” on page 32](#) for more information about serialization expressly designed for digital signatures.

6.6 LinefeedNoTrailingTabs option of deserialization methods

The following option may now be used in the XML deserialization methods:

```
LinefeedNoTrailingTabs
```

This option can be used with any of the 3 “Wsp*” options, including the default (WspNewline). The result is that if any Text node contains an initial linefeed followed by any number of tabs and nothing else, “before” any of the “Wsp*” handling, then the value of that Text node will be a single linefeed character.

As one example of its use, this option may be useful if an input XML document is deserialized and a digital signature is needed of a subtree in it, when

- the input subtree contains a linefeed and one or more tabs separating markup, and the linefeed must be kept but the tabs discarded for the signature.

By itself, none of the “Wsp*” deserialization options obtain this result. See [“ExclCanonical and WithComments options on Serial method” on page 32](#) for information about serialization expressly designed for digital signatures.

Note that the initial linefeed character can also be a carriage return followed by a linefeed, or a carriage return by itself, since (within Text nodes) all of these are normalized by the XML spec into a single linefeed character. So, in terms of the “raw” data fed to the deserialization routine, when this option is used, then any of the following “regex patterns” of a complete Text node:

```
\n\t*
\r\t*
\r\n\t*
```

will result in a Text node that contains exactly one linefeed character.

Given the following example:

```
%lf String Len 1 Initial($X2C('25'))
%tb String Len 1 Initial($X2C('05'))
%d:LoadXml('<a>' With %lf With %tb With '</a>', -
'LinefeedNoTrailingTabs')
%s = %d:Serial(, 'EBCDIC')
Print $Substr(%s, 1, 3) With ' X"' With -
$C2X($Substr(%s, 4, 1)) With '" ' With $Substr(%s, 5)
```

The result is:

```
<a> X"25" </a>
```

Continuing with the subtree digital signature discussion, this option may enable you to use a “Wsp*” deserialization option which is appropriate for the nodes of your XmlDocument outside the security subtree. It will not be able to perform the desired result if you have Text nodes outside the subtree which must retain a linefeed followed by one or more tabs and nothing else.

If LinefeedNoTrailingTabs is successful except that it leaves a “residue” of linefeed Text nodes which interfere with the rest of the document, you can remove these fairly easily and efficiently:

```
%doc:LoadXml(%input, 'LinefeedNoTrailingTabs ...')
* Remove outer LF Text nodes:
%xpath = '//text()[.=" ' With $X2C('25') With '"']'
%totLF = %doc:SelectNodes(%xpath)
%doc:SelectionNamespace('sec', 'http://...')
%secNode = %doc:SelectSingleNode('/*/*/sec:SignedInfo')
%outer = %totLf:Difference(%secNode:SelectNodes(%xpath))
For %i From 1 To %outer:Count
    %DeleteSubtree(%outer(%i))
End For
```

6.7 Support for numeric comparisons in XPath predicates

Previously, the only form of comparison permitted in *Janus SOAP* XPath predicates was of a locationPath expression to a quoted string; for example:

```
%nlis = %nod1:SelectNodes('order[@date > "2007-01-01"]')
```

The support of comparisons has been extended in version 7.0, allowing comparisons to numeric literals (which are unquoted); for example:

```
%nlis = %nod:SelectNodes('order[item@price > 99.99]')
```

This section is divided into two subsections: one that describes the noteworthy aspects of *Janus SOAP* support for XPath comparisons, and one that provides examples of many of those aspects.

6.7.1 Highlights

This section discusses a few important observations concerning the support of XPath comparisons.

Note: In discussing XPath functions, the name of the function followed by an empty pair of parentheses (for example, `number()`) is sometimes used to name the function, whether or not the particular function being discussed takes arguments.

Direct numeric comparison

If an XPath expression contains a locationPath subexpression compared to a literal numeric value, the result is true if any node in the subexpression result, converted to a numeric value, has the specified relationship (“=”, “<”, etc.) to the literal value.

In the following example, an `order` child will be in the result if it has an `item` child whose `price` Attribute node is greater than 99.99:

```
%nlis = %nod:SelectNodes('order[item@price > 99.99]')
```

See [“Examples: Direct numeric comparison” on page 41](#) for more examples.

Support of numeric comparisons is new in 7.0 *Janus SOAP*.

Direct numeric comparison request cancellation

When a predicate indicates a numeric comparison, as above, the request is cancelled if any node value used in the numeric comparison is non-numeric.

See [“Examples: Direct numeric comparison request cancellation” on page 42](#) for examples.

number([locPath]) function for non-numeric data

The `number()` function takes an optional `locationPath` argument, and, if the result of the argument is a single node, converts the value of the node, after stripping leading and trailing whitespace, to a number, or to the special value **NaN** (“Not a Number”) if the stripped value of the node is not numeric.

If the `locationPath` argument is the empty `nodeSet`, the result of the `number()` function is also NaN.

Comparing the numeric result of `number()` to a literal gives a result according to their relative values and the comparison operator; comparing NaN to a literal is true if the operator is “!=” and it is false otherwise.

Thus `number()` can be used to avoid request cancellation for a numeric comparison if the nodes evaluated by a predicate may be non-numeric. However, see the next two items for special considerations when using `number()`.

See [“Examples: number\(\[locPath\]\) function for non-numeric data” on page 42](#) for examples.

Support of the `number()` function is new in 7.0 *Janus SOAP*.

number(locPath) != n, locPath result is empty node-set

If a predicate contains the `number()` function followed by the “!=” comparison, and the `nodeSet` result is empty, the result of the comparison is true. This is a consequence of the rule for comparing NaN.

If you do not want this behavior (for example, you are using the “!=” comparison and the purpose of `number()` is to avoid the request cancellation caused by non-numeric data), then you can filter out the nodes included by empty `nodeSet` comparisons by expanding the `locationPath` expression from

`number(locPath)` to `locPath` and `number(locPath) =` — adding `locPath` and to the `number()` function causes this part of the predicate to be false. Remember, expanding like this is only called for when “`number()`” is followed by “`!=`”.

See “[Examples: number\(locPath\) != n, locPath result is empty node-set](#)” on [page 43](#) for examples.

number() request cancellation

When a predicate contains the `number()` function, the request is cancelled if the value of the `nodeSet` argument to the `number()` function has more than one node.

See “[Examples: number\(\) request cancellation](#)” on [page 45](#) for examples.

In XPath 1 (which has no exception conditions), if there is more than one node in the `nodeSet` argument, the value of the first node (in document order) is used.

number() default argument is “.”

If you omit the argument to the `number()` function, the default `locationPath` used is “.”; that is, the node being filtered by the predicate is converted to a numeric value.

See “[Example: number\(\) default argument is “.”](#)” on [page 45](#) for examples.

Numeric literals

In *Janus SOAP*, a numeric value (either a literal or a node value) may be of any form available in User Language; in particular, “E-format” literals such as `1.003E-5` (even though they are not very common in XML documents) may be specified. The same form of numbers is available in XPath 2.

XPath 1 only allows decimal numbers; it does not allow E-format literals nor node values.

Ordered string comparison

If an XPath expression contains a `locationPath` subexpression and a quoted string with an ordered comparison (that is, a comparison other than “`=`” and “`!=`”), the result is based on a byte-by-byte ordered comparison between each item of the `nodeSet` result of the subexpression and the literal string value. Repeating the earlier example:

```
%nlis = %nod:SelectNodes('order[@date>"2007-01-01"]')
```

If the value of the `date` Attribute node of an `order` Element child is, for example, “2007-05-17”, that `order` Element node will be included in the result.

This behavior has always been implemented in *Janus SOAP*. However, since most practical ordered comparisons involve numeric values, version 7.0 implements numeric comparisons (using the XPath 2 standard). If a comparison literal is bracketed in double or single quotation marks, a string comparison is performed, whether or not the literal has a numeric format.

In XPath 1, any ordered comparison is done by first converting each operand to a numeric value and then performing the comparison, and the result of any ordered comparison of a non-numeric node value is false. Therefore, the XPath 1 result of the above example would always be empty, because the literal is a non-numeric value.

Other than the request cancellation conditions noted above and the limitations noted in items that follow, the only difference between XPath comparisons in *Janus SOAP* and the XPath 1 standard is that *Janus SOAP* supports ordered string comparisons, while the XPath 1 standard does not.

Comparison syntax limitation

In *Janus SOAP*, the only forms of comparisons are the following:

- position() relOp integer
- locPath relOp "stringLiteral"
- locPath relOp numericLiteral
- number([locPath]) relOp numericLiteral

In the above list,

locPath is an XPath location expression (which, as of version 7.0 of *Janus SOAP*, still has the limitation that it may not contain a predicate)

relOp is one of the comparisons "=", "!=", "<", "<=", ">", or ">="

Comparisons in the XPath standard (XPath 1 and XPath 2) are much more general; either operand may be any form of XPath expression.

number() limitation

In *Janus SOAP*, the number() function must be immediately followed by a comparison operator and a numeric literal. This limitation is not required by either XPath standard (XPath 1 nor XPath 2).

Precision limitation

The precision used in *Janus SOAP* XPath support is that provided by User Language — namely, 15 decimal digits.

Long values/numbers out of range limitations

If the XPath support in *Janus SOAP* attempts to convert a long string (that is, longer than 255 bytes) or a number whose absolute value is beyond the capabilities of User Language (maximum absolute value approximately 7.237E75), then the request is cancelled. Note that this applies to both direct numeric comparisons and comparisons involving the number() function.

Summary: *Janus SOAP* uses XPath 2 comparisons

The XPath 1 standard does not provide for exception conditions and it does not provide for ordered string comparisons. This is also true for Microsoft .Net, which follows the XPath 1 standard.

Janus SOAP follows the XPath 2 standard by providing for exceptions (implemented as request cancellation conditions) and providing for ordered string comparisons.

6.7.2 Examples

The following sub-sections provide a number of examples of numeric comparisons that help to clarify the observations made about comparisons in [“Highlights” on page 37](#). Most of the examples use the following “clothes” document:

```
<clothes>
  <shirt size="32" type="dress" sku="100"/>
  <shirt size="33" type="sport" sku="101"/>
  <shirt size="M" type="sport" sku="102"/>
  <shirt size="34" type="frilly" sku="103"/>
</clothes>
```

6.7.2.1 Examples: Direct numeric comparison

A predicate can contain a comparison of a locationPath to a numeric literal (called a “direct numeric comparison” for the purposes of this discussion). The comparison is true if the numeric value, after stripping leading and trailing whitespace, of any of the nodes in the locationPath result has the specified relationship to the numeric literal. If none of the nodes has the relationship (which includes the case that the locationPath result is empty), the result of the comparison is false.

For example, using the “clothes” document described above in [“Examples”](#), the following statement prints `sku="100"`.

```
%doc:Print('/*/shirt[@size<040]/@sku')
```

Note that the *numeric* value of the node and the *numeric* value of the literal are compared, so the leading zero in `040` here is ignored. An equivalent comparison could be `@size<040.00`, etc. The following statement prints `sku="101"`:

```
%doc:Print('/*/shirt[@size<40 and @type="sport"]/@sku')
```

The following statement prints `sku="103"`; the comparison of the `size` Attribute is processed for only one Element, which has a numeric `size`. As discussed below, this request would fail if the order of the attribute subexpressions were reversed.

```
%doc:Print('/*/shirt[@type="frilly" and @size<40]/@sku')
```

6.7.2.2 Examples: Direct numeric comparison request cancellation

If any of the nodes **used** in a direct numeric comparison has a value that is non-numeric after stripping leading and trailing whitespace, the request is cancelled.

For example, using the “clothes” document described in “[Examples](#)” on page 41, the following statement causes the request to be cancelled when the `size` attribute (“M”) of the second sport shirt is compared to the number 40 (note the difference between this XPath expression and the last one in “[Examples: Direct numeric comparison](#)” on page 41):

```
%doc:Print('/*/shirt[@size<40 and @type="frilly"]/@sku')
```

The following statement causes the request to be cancelled (at the same Element), because `SelectNodes` continues after the first selection, unlike the `Print` example with the same XPath expression in “[Examples: Direct numeric comparison](#)” on page 41:

```
%sh = %doc:SelectNodes( -  
    '/*/shirt[@size<40 and @type="sport"]/@sku')
```

If you want the request cancellation to be avoided in these cases, they are both good candidates for using the `number()` function, as described in the next section, “[Examples: number\(\[locPath\]\) function for non-numeric data](#)”.

6.7.2.3 Examples: number([locPath]) function for non-numeric data

The `number()` function can often be used to avoid request cancellation due to the use of non-numeric data in a direct numeric comparison.

For example, both examples from the previous section (“[Examples: Direct numeric comparison request cancellation](#)”) can avoid request cancellation (when used with the **particular document** shown at the end of “[Support for numeric comparisons in XPath predicates](#)” on page 37) if the locationPath `@size` is “converted” using the `number()` function.

The following statement prints `sku="103"`, even though the `size` Attribute equal to "M" is processed before the selected Element:

```
%doc:Print('/*/shirt[number(@size)<40 and -
           @type="frilly"]/@sku')
```

Similarly, the following statement succeeds even though the `size` Attribute equal to "M" is processed (and not selected):

```
%sh = %doc:SelectNodes( -
      '/*/shirt[number(@size)<40 and @type="sport"]/@sku')
```

Note that comparisons with the `number()` function are always false for a non-numeric node value, unless the comparison is "!=". The following statements both print `None` found:

```
Print %doc:ValueDefault( -
      '/*/shirt[number(@size) < 40 and @sku="102"]/@size', -
      'None found')
```

```
Print %doc:ValueDefault( -
      '/*/shirt[number(@size) >= 40 and @sku="102"]/@size', -
      'None found')
```

The following, however, prints `M` — the size of shirt with this SKU; its size is not less than 40 nor greater than or equal to 40, but it is not equal to 40 (nor any other number):

```
Print %doc:ValueDefault( -
      '/*/shirt[number(@size) != 40 and @sku="102"]/@size', -
      'None found')
```

Note: The `number()` function should not be substituted into a direct numeric comparison without first understanding the node-set differences described in the following sections ([“Examples: number\(locPath\) != n, locPath result is empty node-set”](#) and [“Examples: number\(\) request cancellation”](#) on page 45).

6.7.2.4 Examples: `number(locPath) != n`, `locPath` result is empty node-set

When a predicate contains the `number()` function followed by the “!=" comparison, if the `nodeSet` result is empty, the result of the comparison is true; if any other comparison is used, the result is false.

For example, consider this document:

```
<t>
  <w a="1" />
  <x a="PI" />
  <y a="e" b="1" />
  <z a="e" b="2" />
</t>
```

If you are using a numeric comparison to search for Attribute `a`, you should use the `number()` function to avoid request cancellation, because `a` has non-numeric values. The following statement sets the result nodelist to the Element `w`:

```
%nlis = %doc:SelectNodes('/t/*[number(@a) = 1]')
```

The following statement sets the result nodelist to the Elements `x`, `y`, and `z`:

```
%nlis = %doc:SelectNodes('/t/*[number(@a) != 1]')
```

The `b` Attribute, however, does not have any non-numeric values, so it can be used without `number()`. Each of the following two statements sets the result nodelist to the Element `y`:

```
%nlis = %doc:SelectNodes('/t/*[@b = 1]')
%nlis = %doc:SelectNodes('/t/*[number(@b) = 1]')
```

However, the following two statements differ in their result:

```
%nlis = %doc:SelectNodes('/t/*[@b != 1]')
%nlis = %doc:SelectNodes('/t/*[number(@b) != 1]')
```

The first sets the result nodelist to the Element `z`, while the second includes Elements `w` and `x` as well as the Element `z`. Since they do not contain the `b` Attribute, the result of `number(@b) != 1` at elements `w` and `x` is true.

If you want to make `number()` similar to a direct comparison in this respect, you “and” the `locationPath` argument with the `number()` factor in the predicate. So, for example, the following sets the result nodelist to the Element `z`, just like the direct comparison approach:

```
%nlis = %doc:SelectNodes('/t/*[@b and number(@b) != 1]')
```

Note: The other way in which `number()` differs from direct comparison is described in “[Examples: number\(\) request cancellation](#)” on page 45.

6.7.2.5 Examples: number() request cancellation

When a predicate contains the number() function, the request is cancelled if the value of the nodeSet argument to the number() function has more than one node.

For example, consider this document:

```
<t>
  <x a="1" b="2"/>
  <y b="pi" a="3.14159265"/ >
</t>
```

If you are searching for all Elements that have any Attribute greater than one, you can use the locationPath @* as a wildcard comparison for any Attribute. However, we cannot use direct comparison, because some of the attributes are non-numeric.

So, you might try to use number(@*), as in the following example:

```
%nlis = %doc:SelectNodes('/t/*[number(@*) > 1]')
```

However, this will cause a request cancellation, because the value of @* contains more than one node. In such situations, you must decide which node is to be converted to a number for the comparison. In this case you would probably want to use:

```
%nlis = %doc:SelectNodes( -
  '/t/*[number(@a) > 1 or number(@b) > 1]')
```

This would set the result nodelist to Elements x and y.

6.7.2.6 Example: number() default argument is “.”

If you omit the argument to number(), the nodeSet that is converted is the context node.

So, for example: in the following XPath expression, the number() function converts the value of the size Attribute to a number:

```
/*/shirt/@size[number()] > 10]
```

6.8 Document deserialization relinquishes CPU

Previously, When a user is deserializing (WebReceive, ParseXml, or LoadXml) a very large (for example, over 6MB) XML document, other users may find response time to be extremely slow. This could be especially noticeable in a non-MP *Model 204* environment.

Now the deserialization operation will periodically relinquish the CPU, if there are other users of equal or greater priority ready to run.

6.9 LoadXml now supports Stringlist items longer than 6K

Previously, if the first argument to the LoadXml method were a Stringlist, only the first 6,124 bytes of each of the Stringlist items were processed. Now the entire value of each Stringlist item is processed by LoadXml.

Note: This may result in an incompatibility, as discussed in [“LoadXml support of Stringlist items longer than 6K” on page 58](#).

This feature has also been introduced by maintenance to version 6.9 of *Janus SOAP*, by ZAP6940 and ZAP6949.

6.10 XmlDoc restored after parse errors

Previously, in the XML parsing methods, if the ErrRet option was specified and there was an XML document parsing error, all nodes and properties of an XmlDoc were initialized, losing all information in the XmlDoc.

Now if there is a parse error and ErrRet is specified, all properties are retained; this change applies to the LoadXml and WebReceive functions, and to the ParseXml function of the HttpResponse class. Also, for the LoadXml method, if the method object is XmlNode, the nodes present in the XmlDoc prior to the LoadXml are preserved.

As mentioned in [“XmlDoc not reinitialized after parse errors” on page 59](#), this can represent a very slight incompatibility.

This feature has also been introduced by maintenance to version 6.9 of *Janus SOAP*, by ZAP6913.

6.11 XML parsing cancels request if CCATEMP full

Previously, if a CCATEMP full condition occurred during parsing of an XML document, the request was not cancelled if the ErrRet option was present.

Now, when this condition occurs, the request is always cancelled. This change applies to the LoadXml and WebReceive functions, and to the ParseXml function of the HttpResponse class.

6.12 **InvalidCharacterPosition function (actually part of 6.9)**

InvalidCharacterPosition is a shared function in the XmlDocument class:

```
%pos = xmlDocQual:InvalidCharacterPosition(string)
```

The function verifies whether the characters in the *string* argument are valid as characters to be used as the value of an Element or Attribute node. If all the string characters are valid, the function returns 0; otherwise, it returns the position of the first invalid character.

Notes:

- The invalid characters of an Element or Attribute node are the “control characters.”
- Like any shared method, the method object can be an object of the class of the method (XmlDoc), null or not, or a class name qualifier:

```
* Example of first type, %myDoc may be null or non-null:  
%pos = %myDoc:InvalidCharacterPosition(%str)
```

```
* Example of second type:  
%pos = %(XmlDoc):InvalidCharacterPosition(%str)
```

- With the introduction of this method, the IsValidString function is being deprecated.
- This feature was actually introduced into version 6.9 of *Janus SOAP*, but the documentation was missing for so long that it is documented here to give customers an opportunity to learn of it.

CHAPTER 7 *SirTune*

7.1 SYSPARM data

SirTune now issues an internal `VIEW SYSTEM CWAIT` command and saves the output to the sample dataset. This data can then be used to produce the SYSPARM report by the *SirTune* Report Writer.

7.2 System method quads

As of this release, the *SirTune* data collector can distinguish system methods from each other. Before this release, all system methods would show up as `$CLASS_METHOD` in the QUAD reports.

7.3 Compatibility

A sample dataset collected by *SirTune* version 7.0 and later requires version 1.6 or later of the *SirTune* Report Writer.

Compatibility/Bug fixes

This chapter lists any compatibility issues with prior 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 prior version (6.9).

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. We 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).

8.1 Backwards incompatibilities

Backwards incompatibilities are described per product in the following sections.

8.1.1 Janus SOAP XML processing

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

8.1.1.1 XPath with “!=” and Element with more than 1 child

The processing of the **not equals** comparison in XPath predicates was, in some cases, incorrect. If the comparison was done to an Element with more than one child, and in fact the value of the Element was different from the literal being compared, the Element was not included in the result, although it should have been.

For example, given the following XML document:

```
<top>
  <emphasis>Crazy</emphasis> bug
</top>
```

The result of the following statement should be `True`:

```
Print %d:Exists('*[. != "OK"]'):ToString
```

However, prior to this bug fix, the result was `False`.

Note: This fix has also been introduced by maintenance to version 6.9 of *Janus SOAP*, by ZAP69D0.

8.1.1.2 Fixed XPath “following” and “xx-sibling” axes

The change described in this section was delivered as maintenance (ZAP69B9) for *Sirius Mods* version 6.9.

These behaviors, which did not conform to the XPath standard, have been fixed:

- Previously, using XPath's `following` axis (for example, `following::*`) resulted in the incorrect selection of extra nodes, in particular, Attribute and descendant nodes of the step's context node(s). Selecting these nodes is explicitly excluded by the XPath standard.
- Previously, using the `following-sibling` and `preceding-sibling` axes with an Attribute context node (for example, `@x/following-sibling::*` or `@x/preceding-sibling::*`), which by the standard should *not* select any nodes, might incorrectly select nodes.

In version 7.0 of the *Sirius Mods*, the ability to navigate from one Attribute node to its sibling (and in turn, to the rest of the sibling Attribute nodes), is provided by the `Next` and `Previous` methods (which can be used from nodes other than Attribute nodes, as well). See “[Next and Previous methods](#)” on page 30.

For information about the performance implications of the axes discussed above, see “[Performance considerations: Document order, certain axes](#)” on page 54.

8.1.1.3 Corrected XPath results with unusual axes

Prior to this correction, errors might occur in some cases using these axes:

```
following
descendant
descendant-or-self (which can be obtained with //)
ancestor
ancestor-or-self
preceding-sibling
```

and in some unusual cases using these axes:

```
parent (which can be obtained with ..)
following-sibling
```

The errors that can occur are:

- Return of an incorrect node, or return of a node other than the first node in document order

Occurs with methods that return the first selected node, such as `Value` and `SelectSingleNode`

- Incorrect, duplicate, and/or out-of-order result nodes

Occurs with methods that return an `XmlNodeList` (`SelectNodes` and `UnionSelected`)

For example, given the following document:

```
<t>
  <a a="aa" n="∅">
    <a a="ab" n="1">
      <b n="1"/>
      <b x="1"/>
    </a>
    <b n="2"/>
    <c n="2"/>
  </a>
</t>
```

The result of `Serial('//a[2]')` before the code fix was:

```
<a a="ab" n="1"><b n="1"/><b x="1"/></a>
```

This result element is the first child of its parent; it is not the second "a" child as specified in the XPath expression. In fact, the request in this example should be cancelled, because the XPath result is empty — there is no second "a" child of any element in the document.

With the same document, the result of `Serial('//self::node()[@a]/b')` before the code fix was:

```
<b n="2"/>
```

However, this is not the first element satisfying the XPath expression; the correct result is the element `<b n="1">`.

The result of `SelectNodes('t/a/a/@*/following:*')` with the same document before the code fix was:

```
<b n="1" />
<b x="1" />
<b n="2" />
<c n="2" />
<b n="1" />
<b x="1" />
<b n="2" />
<c n="2" />
```

Notice that the last four items in the `XmlNodeList` (duplicates of the first four) should not be present.

For information about the performance implications of using the axes discussed above, see [“Performance considerations: Document order, certain axes”](#).

This change was also delivered as maintenance (ZAP69B9).

8.1.1.4 Performance considerations: Document order, certain axes

The “A.5. Order of nodes: performance,” section in the *Janus SOAP Reference Manual* prior to November 9, 2006 discussed the performance implications of using different axes in the XPath expressions in *Janus SOAP XML* method arguments. That section has been revised and expanded to incorporate the two changes listed below, and the rewritten version of the section is displayed here, below. The section's title and number are changed to “A.4. Performance considerations: Document order, certain axes” as of December 28, 2006) in the *Janus SOAP Reference Manual*.

The main changes incorporated in the revised section of the manual are:

1. In addition to the axes and axis combinations specified in the preceding appendix subsections, extra processing can occur for these axes and circumstances:

<i>Axis</i>	<i>Circumstances</i>
ancestor, ancestor-or-self	Regardless of anything else in the XPath expression
attribute	If it is subsequent to a <code>parent</code> axis that is not the first step in the expression
following	If it is not the first step in the expression

2. There is a special case that is an exception to the general rule for the `descendant-or-self` axis followed by `child`. An example is `//chapter`.

Revised version of “A.5 Order of nodes: performance”

This section discusses the performance implications of evaluating certain XPath expressions. The expressions of concern have a common characteristic — they are not **simple XPath expressions**.

Simple XPath expressions, which have **no special performance considerations**, are any of these:

- One or more steps containing only `child`, `attribute`, or `self` axes.
- A `parent` axis used in the first step, after which may be one or more steps containing only `child`, `attribute`, or `self` axes.
- A `following-sibling` axis *used alone*.

The rest of this section considers XPath expressions that are not simple and therefore might have negative performance implications. If your use of XPath is confined to the simple expressions defined above, the following discussion is not your concern.

The XPath expression arguments of methods like `SelectNodes` and `UnionSelected` in the `XmlNodeList` class designate a set of nodes. In addition to these “set-valued” methods, XPath expressions can be used in many *Janus SOAP* XML document methods (`SelectSingleNode`, `Value`, `DeleteSubtree`, `QName`, and more) to operate on a single node that satisfies the expression. For the simple XPath expressions (described above), the “single node” methods scan fewer than or as many nodes before determining the desired node, thus give better performance than the set-valued methods.

The single-node XPath selection by *Janus SOAP* returns the first node in document order, but with non-simple XPath expressions, this is not the same as the first node found by the XPath **internal selection algorithm**, which may visit nodes in a different order. In those cases, *Janus SOAP* examines an entire subtree to determine the first node, in document order, that the XPath expression selects.

In other words, given an XPath expression *expr* that uses any of the axis cases described below in “[The extra-processing expressions](#)” on page 56, and given any single-node selection method *XMeth*, this expression:

```
%obj:XMeth(expr)
```

scans as many nodes as:

```
%obj:SelectNodes(expr):Item(1):XMeth
```

For all other XPath expressions, the number of nodes scanned by the first of these two approaches may be significantly lower, because the first node internally selected will also be the first node in document order.

For example, consider the following document:

```
<top>
  <a>
    <b x="1" />
  </a>
  <b x="2" />
</top>
```

When, say, `Value('/*/b/@x')` is evaluated, the document search ends when the first match is found (and the `Value` method returns 1).

But when `Value('//b/@x')` is evaluated, the document search first finds the match `x=2`, then it continues searching the entire document for all matches, to ensure that the match which is lowest in document order (`x=1`) is the result.

The performance implications of the expressions that involve extra processing apply to the set-valued methods as well. The set methods must produce their results in document order, but the nodes selected during XPath evaluation may be selected in an order (due to the selection algorithm) that differs from document order.

The extra-processing expressions

Extra processing can occur in the following cases:

1. The presence of the `preceding-sibling` axis
2. The presence of the `ancestor` axis
3. The presence of the `ancestor-or-self` axis
4. The presence of the `following` axis, if it is not the first step in the expression
5. The presence of any of these axis-combinations:

One of the following axes:

- `descendant`
- `descendant-or-self`
- `following`
- `parent`, if it is not the first step in the expression

followed, in a subsequent step, by any of these axes:

- `parent`
- `child`
- `following-sibling`
- `descendant`

- `descendant-or-self`

The

- `parent` axis, if it is not the first step in the expression,

followed, in a subsequent step, by the

- `attribute` axis.

In addition to the cost of the actual XPath search performed with the above expressions, they can incur an additional cost for XPath evaluation. If the document has been modified in such a way that the internal order of the nodes cannot be guaranteed to be the same as document order (this will always happen with any of the XML `Insert..Before` methods, and usually will happen with any of the `Add..` methods), then the entire document (not only the subtree being searched) must be scanned so that the order is adjusted. This does not involve any internal movement of the nodes, but does require a full scan.

Note:

1. One important exception to the above rules is the `descendant-or-self::node()` **step** followed immediately by the `child` axis without any predicate. An example of the usual way to specify this is:

```
//chapter
```

In this case, the internal node selection algorithm operates in document order, and no extra processing is incurred.

Even with this special case, it is better to avoid the `descendant-or-self` step (specified explicitly or by using `//`) if your document structure lends itself to explicitly specifying the “intermediate” elements (and, even better, their names) that should be matched.

2. The considerations described in this section only apply to the “outer” XPath expression; they do not apply to any expression within a predicate. Although it is still better, for the sake of efficiency, to prune the search by explicitly specifying “intermediate” elements rather than using `//`, there is no efficiency concern due to the internal order of node selection with an XPath predicate such as the following:

```
Print %d:Value('/book/chapter' With -  
    '[./credit/details/@auth="Dave"]')
```

3. In conclusion, except when you must use the “`//chapter`” exception discussed in Note 1, above, *avoid these extra-processing axes and axis combinations* (especially in outer XPath expressions) if your documents are relatively large and performance is a consideration.

8.1.1.5 LoadXml support of Stringlist items longer than 6K

Previously, if the first argument to the LoadXml method were a Stringlist, only the first 6,124 bytes of each of the Stringlist items were processed. Now the entire value of each Stringlist item is processed by LoadXml.

The previous behavior could result in a combination of the following incorrect behaviors:

1. Apparent successful deserialization of a well-formed XML document, with omission of some of its contents.
2. Apparent successful deserialization of a non-well-formed XML document.
3. Report of a deserialization error (either by request cancellation or by a non-zero result value of LoadXml) of what is in fact a well-formed XML document.

With the enhancement to LoadXml that processes all of the Stringlist items' values, the above cases will now result in different behaviors to applications. The new behavior corresponding to the respective above cases will be:

1. The formerly apparent successful deserialization will now report an error, either by request cancellation or by a non-zero result value of LoadXml.
2. The value of the XmlDocument will change.
3. The formerly failed deserialization will now successfully deserialize.

Note: This feature has also been introduced by maintenance to version 6.9 of *Janus SOAP*, by ZAP6940 and ZAP6949.

8.1.1.6 Indent option limited to 254

For the various serialization methods (for example, Print, or Serial without the ExclCanonical option), the Indent width is limited to 254.

8.1.1.7 Null string SelectionNamespace doesn't mean "no association"

As mentioned in ["Null string SelectionNamespace value now means "no namespace" on page 29](#), the null string may now be associated with a prefix string, which can then be used in XPath expressions to select Element and Attribute nodes that are not in a namespace.

Previously, the null value assigned to or returned by the SelectionNamespace property of a prefix indicated that the prefix does not have an XPath selection association.

In some unusual cases, you may be using this previous behavior to manage the use of XPath prefixes. For example, the following rather strange code could formerly use the old behavior to find and use a “new” prefix, without changing any existing prefix associations:

```
***** Note: This code only works prior to the 7.0 mods *****
For %i From 1 To 100
  If %doc:SelectionNamespace('p' With %i) EQ '' Then
    Loop End
  End If
End For
%doc:SelectionNamespace('p' With %i) = 'urn:localxxx'
%node = %inpNode:SelectSingleNode(%p With %i -
  With ':infoChild')
* Done with prefix:
%doc:SelectionNamespace('p' With %i) = ''
```

Generally speaking, there is no need to discover (as done in the “For” loop above) whether a prefix has an association, nor to remove (as done in the last line above) the association for a prefix; you can always “save and restore” a prefix as in the following code:

```
***** This code works in all versions of the mods *****
%savURI = %doc:SelectionNamespace('tPrefix')
%doc:SelectionNamespace('tPrefix') = 'urn:localxxx'
%node = %inpNode:SelectSingleNode('tPrefix:infoChild')
* Restore tPrefix:
%doc:SelectionNamespace('tPrefix') = %savURI
```

However, if either need arises, it can now be accomplished with new methods described in [“IsSelectionPrefix: Check if prefix has XPath selection association”](#) on page 30 and [“DeleteSelectionPrefix: Delete any XPath selection association for prefix”](#) on page 30.

8.1.1.8 XmlDoc not reinitialized after parse errors

As mentioned in [“XmlDoc restored after parse errors”](#) on page 46, the XML parsing methods no longer re-initialize the entire XmlDoc when a parsing error occurs and the ErrRet option is specified; rather, the XmlDoc is restored to its state prior to the deserialization operation.

In very odd cases, this could affect the behavior of programs which depend on the old initialization behavior of parse errors. In the unlikely event that you want the old behavior, you can use the New or Discard method (as appropriate for your application) when a parsing error occurs.

8.1.2 Janus SOAP ULI File classes

The following backwards compatibility issues have been introduced in the *Janus SOAP ULI* File classes.

8.1.2.1 Request cancellation due to nonsense group field usage

As mentioned in [“Request cancelled if group field reference nonsensical for file”](#) on page 23, your applications are now protected against nonsense field references when done in the context of *Janus SOAP ULI* File objects.

If this causes an existing application to fail, you should decide whether to change field definitions, or change the User Language code to determine when it makes sense to execute statements with references to fields undefined or INVISIBLE in some members of the group. One technique for accomplishing the latter is shown in [“Using Is Defined/Visible to avoid cancellation”](#) on page 27.

8.2 Fixes in Sirius Mods 7.0 but not in 6.9

This section lists fixes to functionality existing in the *Sirius Mods* version 6.9 but which, due to the absence of customer problems, have not, as of the date of the release, been fixed in that version.

- There are no fixes in *Sirius Mods* 7.0 that are not available in previous versions.

8.3 Version corequisites

This section lists any restrictions on usage of various products (including *Sirius Mods* itself) that will be imposed by use of version 7.0 of *Sirius Mods*.

- There are no corequisites associated with *Sirius Mods* 7.0.