



Automation Scripting Language

Reference Guide

for

Sequencer Class NMR Spectrometers



**Delta & Control software
version 6.0**

Delta™ and Control software and the Automation scripting language are
copyright © 1990-2019 by JEOL Resonance, Inc. All rights reserved.

Copyright © 2005-2019
Author: Matthew W. Borchers
borchers@jeol.com

JEOL Resonance, Inc.
3-1-2 Musashino, Akishima, Tokyo
Japan 196-8558

Table of Contents

Introduction.....	1
Automation Jobs	1
Automation Script Files	1
Comments	1
Definition of Terms	2
STRINGS	2
IDENTIFIERS	2
KEYWORDS	2
Location of Support Files	4
FILE PATH URLS	4
DIRECTORY SEARCH ORDER	4
Value Types	5
BOOLEAN.....	5
Boolean Expressions	5
NUMBER	6
Number Bases	7
TEXT	7
LIST	8
DATA	8
SUB-TYPES	8
Value Type Casting.....	9
Duration Syntax	10
Basic Script Structure.....	11
Statements.....	12
CALL.....	13
CONST.....	15
DELAY	18
EMAIL	19
ENUM.....	21
EXIT.....	25

EXPERIMENT	27
FINISH	33
GROUP.....	34
IF.....	35
INCLUDE.....	38
INFORM	40
INVOKE.....	42
LIMIT.....	45
LIST.....	46
MACRO.....	48
METHOD.....	51
NUMBER.....	60
ON ERROR.....	64
PERCIVAL	68
PRESENTATION	70
PRINT	72
PRINT CONTEXT	75
PROCESS.....	77
PROMOTE	79
PROMPT.....	80
RAISE	83
REMARK.....	84
REPEAT	85
RETAIN.....	90
RETRY	91
SET.....	93
TERMINATE	100
TRANSLATE.....	102
TUNE.....	104
VAR.....	105
VISUALIZE.....	110
Parser Instructions	115
Table of Units	117
Table of Substitution Identifiers	119
Writing a Duration Statement Expression.....	127
Example Script.....	129
Automation Script Grammar	131

Introduction

Automation is the ability to perform a task with little to no human control or interaction. For NMR users, this is the ability to acquire data from an NMR spectrometer and manipulate that data in a predetermined way without having to attend to the spectrometer or sit at a computer workstation. Automated tasks are coded into text files called *Automation Scripts* in which are written the ordered series of steps that the spectrometer will perform to accomplish a task and produce a desired result.

Automation Scripts are the way the automated process is facilitated on the JEOL NMR spectrometer. With the JEOL Delta and Control software, Automation can acquire data, process data, print data, and inform the user of its results in various ways. In addition to these basic tasks, the Automation system can prepare the spectrometer for acquisition with automatic tuning and shimming and can conditionally execute and repeat parts of the Script with branching and looping constructs based on user specified pre-conditions and/or execution-time parameter comparisons.

Automation Jobs

Automation *Jobs* are created and submitted with the *Spectrometer Control* window using the *Sample* and *Job Creation* tabs along with the *Experiment Setup* area. The *Job Creation* tab is the only way that multiple Methods can be placed into a Job. Methods can be added to a Job by loading an Automation Script file and selecting any of the Methods that are contained within that file to be in the Job. Any number of Methods from any number of Script files may be added to the Job and Methods may even be added more than once.

The *Experiment Setup* area allows the user to characterize and execute a single experiment as an Automation Job by creating a Method on the fly that contains the experiment.

When Automation executes a Job, it runs each Method within the Job in the order in which the Methods are listed. When no Samples are specified, the Method(s) are executed a single time. When Samples are provided with the Job, the Job is run once for each specified Sample.

Automation Script Files

An Automation Script file can be written using any text-editing program, however a specialized graphical tool for developing Methods is planned. The syntax is nearly free form, meaning that blank space (spaces, tabs, line separators, etc.) is ignored with the one exception of the ELSE clause of the IF statement. Refer to the section describing the IF statement for details. The language is also not case sensitive, meaning for example, that AUTOMATION, Automation, and automation are all equivalent keywords. However, throughout this document UPPERCASE will be used to indicate keywords of the Automation Script grammar.

Comments

Comments are allowed anywhere within the Automation Script text. Comments have meaning to the author and potential users of the script only and so they are ignored by Automation. Single line commented text begins with the two dashes, --, and continues to the end of the same

physical line on which they began. Multi-line comments begin with a forward slash followed by a dash, `/-`, and continue to the next occurrence of a dash followed by a forward slash, `-/`.

Refer to the subsequent section describing the `REMARK` statement for an important caveat regarding comments.

The following are examples of commented text:

```
-- A single line comment

-----
-- Comment Header Block --
-----

INFORM /-This is an inline comment-/ "This will print";

/- This comment
   spans multiple lines
   and ends here. -/
```

Definition of Terms

STRINGS

A String is a series of zero or more characters (including letters, numbers, symbols, and/or white space) surrounded by single or double quotation characters. Strings are used in the Automation Script to represent textual elements such as words or phrases. To include a quote character in the String, precede the embedded quote with a back-slash character. Example: `"say \"hello\""` or use single outer-quotes: `'say "hello"'`. A String can be assigned to a variable of the `TEXT` Type.

IDENTIFIERS

An identifier is a special word used in the Automation syntax that either has a special inherent meaning or is given a meaning. An example of an identifier that has a special inherent meaning is a reserved word. Some examples of identifiers that are given a meaning are the title of an Experiment, the name of a variable, and the title of a Method.

Identifiers have two restrictions pertaining to how they are defined. First, the only characters allowed to be part of an identifier are the alphabetic characters, the numeric characters, and the underscore, `'_'`. Second, an identifier must begin with an alphabetic character. In English, the alphabetic characters are the lowercase letters `'a'` through `'z'` and the uppercase letters `'A'` through `'Z'`. The numeric characters are the numbers `'0'` through `'9'`.

KEYWORDS

Keywords are the special identifiers of the Automation Script language that provide the structure of the syntax. These keywords are reserved by Automation, which means that they cannot be used to name objects (variables, Methods, and experiments, etc.) in an Automation Script. The following table lists the words that are reserved by Automation.

abort	active	after	alert	all	and	as
assert	association	attach	automation			
boolean	buttons	by				
call	capitalize	cast	category	coil	collect	complete
conceal	console	const	constrain	context	continue	
data	date	day	decrease	default	delay	depends
dialog	div	divisible	do	domain	dual	duration
else	email	enable	end	enum	error	evaluate
exclude	exit	expand	experiment	expired	expose	
false	fatal	file	finish	for	force	from
group						
help	hour					
icon	if	ignore	in	include	increase	info
inform	inout	integer	interactive	interim	invoke	is
job	keys					
limit	list	log	lowercase			
machine	macro	message	method	minute	mod	modulo
multiple						
namespace	no	not	null	number		
of	offset	on	optimize	options	or	out
parameter	passive	percival	precision	prepare	presentation	print
printer	probe	process	processed	project	promote	prompt
purpose						
quiet						
raise	raw	ref	remark	remove	repeat	retain
retry						
sample	save	scout	second	service	set	shims
show	status	step	subject			
template	terminate	text	then	time	to	translate
true	tune	types				
unit	until	uppercase	user			
var	version	visualize				
warning	when	while	with			
xor	yes					

Location of Support Files

There are many statements in the Automation Script grammar in which the author of the Script must provide the location of another file for Automation to carry out its purpose. The kinds of support files that may be required to execute an Automation Script include pulse programs, processing lists, and presentation layout files. The places in the Automation Script syntax where these files are specified are:

- The COLLECT statement within an EXPERIMENT block specifies a pulse program to run to acquire data.
- The default file for a VAR and CONST definition of the DATA Type.
- The default file for a DATA Type parameter of a METHOD.
- The TEMPLATE clause in a PRESENTATION statement specifies a Presentation layout file to use to print.
- The WITH clause in a PROCESS statement specifies a processing list file to process data.
- The INCLUDE statement access a separate Automation Script file.

FILE PATH URLS

The named location of a file is often called a URL (Universal Resource Locator). The URL form in an Automation Script is “*server:path*”. *Server* is an IP address or fully qualified hostname like *host.example.com* where *host* is the name of an NMR instrument on the *example.com* network. All that is required, however, is the *path* part of the URL that specifies a relative directory location of a file on disk. Files must be within directory structure rooted at one of the established locations for them to be available to use by Automation.

DIRECTORY SEARCH ORDER

Each Automation Script that is submitted to the spectrometer will likely have a list of support files that are required for the Automation to properly execute. The file list contains the locations of any required file that cannot be found in the standard locations at the time when the Script is submitted to the spectrometer. When Automation requires a support file, the system searches the following locations in the following order:

1. Check the list of required files for a filename match. This list of filenames was generated at the time when the Automation Script was submitted and will most likely contain the names of the files that existed on the user’s workstation. Files not located on the spectrometer need to be uploaded to the spectrometer so that they will be accessible at the time when the job is ready to run. Note that the path parts of the filenames on this list are not used to match a file. This means that files that are put on this list should contain unique names disregarding any directory path they might have.
2. Check the default directory if one is available.
3. If the Automation is not currently running on a spectrometer, check the standard local directory of the kind of file that is being located. Many parts of an Automation Script could be executed on a computer other than the NMR spectrometer computer and, in these situations, Automation will look for files in this location.
4. Check the current owner's private directory on the spectrometer. This location is skipped in the case where there is not a private directory for the current owner. A private directory is created for each registered user where they can upload and store personal files.

5. Check the standard spectrometer directories. These are the directories that contain the files supplied by JEOL and are installed with the JEOL NMR spectrometer control software.

If a file cannot be found after searching the above locations, Automation will stop and an error message will be displayed. The user will have to correct the problem and then resubmit the Automation Script.

Value Types

A Type is a special word that informs Automation of what kind of values a variable, constant, and Method parameter may contain. All variables, constants, and Method parameters must have a Type. Types help the author of the Automation Script avoid the mistake of using a variable in a manner for which it was not intended. Types also help other users gain a better understanding of how they are to use the variables and Method parameters provided for them.

There are five basic Types that are predefined in the Automation Script grammar: **BOOLEAN**, **NUMBER**, **TEXT**, **LIST**, and **DATA** each of which will now be described individually.

BOOLEAN

A variable of the **BOOLEAN** Type can hold either a True or False value. The keywords **TRUE** and **YES** commonly represent the True value and the keywords **FALSE** and **NO** commonly represent the False value. It is the **BOOLEAN** Type that enables the branching statements, such as the **IF** statement, to make decisions about what statements to execute.

A True value is any of the following possibilities:

- either of the literal **TRUE** or **YES** Boolean values
- all positive and negative Numbers
- Strings containing at least one character
- Lists that contain at least one value of any Type
- Data variables that point to a valid data file

False values are defined to be values that are not True. There are only eight constant values that represent the False value: **FALSE**, **NO**, 0, "", "false", "no", {}, and Null. Casing of the letters in "false" and "no" is ignored. The following table shows examples for each basic Type.

Type	True values	False values
BOOLEAN	TRUE, YES	FALSE, NO, Null
NUMBER	..., -3, -2, -1, 1, 2, 3, ...	0, Null
TEXT	"hello", "16", "0"	"", "false", "no", Null
LIST	{1} {0} { "hello" } { "" } { FALSE } { 1, "Delta" }	{}, Null
DATA	<i>Any valid data file reference.</i>	Null

Boolean Expressions

Boolean expressions are phrases that result in a single Boolean value after they are evaluated. Boolean expressions consist of up to three types of elements: Boolean values, comparison operators,

and logical operators. A Boolean value within an expression can only be the four keywords TRUE, YES, FALSE, or NO and, of course, variables of the BOOLEAN Type.

The comparison operators (`=`, `/=`, `<`, `<=`, `>`, `>=`, `in`, `not in`) result in a Boolean value by comparing two values of any Type to each other. Examples of simple Boolean expressions follow. In these examples, `b` is a BOOLEAN Type variable and `n` is a NUMBER Type variable.

```
TRUE
b
n = 5
n < 10
n in {2,4,6}
```

There are four logical operators (`not` `and` `or` `xor`) that are used to build *compound* Boolean expressions. The truth table below shows the Boolean values that result from these operators.

Input		Operators			
A	B	not A	A and B	A or B	A xor B
TRUE	TRUE	FALSE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	FALSE	TRUE	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	TRUE	FALSE	FALSE	FALSE

Compound Boolean expressions can be written using any combination of the above logical operators. `not` is called a unary operator because it uses a single value to compute the opposite truth value. `and`, `or`, and `xor` are called binary operators because each use two values to determine the logical result. Parentheses may be added for readability, but they are required around sub-expressions when using more than one of the binary logical operators. Examples of compound Boolean expressions are:

```
n = 5 or n >= 10
(n > 0 and n < 10) or not b
```

Boolean expressions are used in the IF statement, in the WHILE and UNTIL forms of the REPEAT statement, and in the WHEN clause which may be specified on the RAISE, EXIT, and TERMINATION statements.

NUMBER

A variable of the NUMBER Type can hold a rational number like the integer 5 or a number with a decimal part like 1.6. Numbers are implicitly positive, but may be preceded with a plus sign, `+`. A minus sign, `-`, can precede a number value to make the number negative. In some instances, a unit is permissible with the number. In these cases, the unit is written within brackets following the number using the standard SI abbreviated form. See the ‘Table of Units’ after the ‘Statements’ section for a list of available units and the proper prefixes. INFINITY and -INFINITY are valid numbers and may also have units. Examples of numbers with units are:

```
7 [ s ]      (7 seconds)
```

12.5 [kHz] (12.5 kilohertz)

Powers are acceptable in a unit by putting an integer immediately following the unit name.

For example:

20 [m²] (20 square meters)
 16 [s⁻¹] (16 per second = 16 hertz)

Multiple units can be specified with the multiplication (*) and division (/) operators within the unit. The following two examples are equivalent:

16 [m / s] (16 meters per second)
 16 [m * s⁻¹] (16 meters per second)

Number Bases

A number may be specified in one of four bases: binary, octal, decimal, or hexadecimal. A base is specified with a pound sign, #, preceding the number with a letter code: 'b' for binary, 'o' for octal, 'd' for decimal, or 'h' for hexadecimal. Decimal is the default base if a base indicator is not provided. For binary numbers, 0 and 1 are the only permissible digits. Octal numbers use only the digits 0 through 7. Decimal numbers use the digits 0 through 9. Hexadecimal numbers use the digits 0 through 9 and the letters A through F to represent the values of 10 up to 15. See the following table for number representations in each base up to 15.

Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Binary	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110	1111
Octal	0	1	2	3	4	5	6	7	10	11	12	13	14	15	16	17
Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Examples of the decimal number 13 written in each base for Automation are:

```
#b1101
#o15
#d13 or 13
#hD
```

TEXT

A variable of the TEXT Type contains a series of zero or more characters called a String. Values of type TEXT are written with double quotes (") or single quotes (') surrounding the characters. Examples of TEXT values are:

```
" "
"gauss"
'base correct'
"12"
```

A numeric value and the representation of that numeric value as a textual value are not related and not equivalent. For example, the textual value "12" and the numeric value 12 are not the same. The equality operator will return FALSE when comparing "12" = 12.

A String may contain a substitution identifier that will be replaced by the current value of that identifier at the point in the String where it is used. See the Table of Substitution Identifiers for details on this. For example, the String “Today is: \$(DAY)” will be translated to “Today is Monday” when the String is processed on a Monday.

LIST

A variable of the LIST Type contains a series of zero or more values of any Type surrounded by braces, {}. Lists can best be explained by examples. The following examples are values of the LIST Type:

```
{ }
{ 2 }
{ "water", 12.5 }
{ { 1, 2[s] }, { 3, 4[s] }, TRUE, 9.11 }
{ { "first", 1 }, { "second", 2 }, { "third", 3 } }
```

The first of these examples is a list that does not contain a value. This is typically called an empty list. The reader can see from the third example, a list does not have to be homogeneous – that is to say, a list may contain a mix of values of any Type, even Lists themselves as the fourth example demonstrates. Homogeneous Lists and key-value paired Lists which can be called “Associations”, are created using the LIST declaration statement. Homogeneous Lists are lists that consist of data that all have the same Type. Association Lists are lists that contain only sub-Lists that contain two parts: a *key* and a *value*. The *key* must be a String and the *value* may be any Type. Refer to the LIST entry in the following ‘Statements’ section for more detail.

Placing an integer or a NUMBER Type variable containing an integer within parentheses after a LIST Type variable can access a specific element in a list. LIST Type values are indexed by integers beginning with 1 for the first element, 2 for the second element, etc. For example, suppose that *L* is a LIST Type variable containing at least three elements. To access the third element of *L*, the author would use the expression: *L*(3).

DATA

A variable of the DATA Type contains a reference to a JEOL NMR data file. DATA Type variables can be assigned values using textual URLs that reference a file on disk. The EXPERIMENT block usually creates a DATA Type constant with the value of the file that results from the acquisition. Refer to the EXPERIMENT entry in the following ‘Statements’ section for details of when the EXPERIMENT block will generate in a LIST Type value.

SUB-TYPES

Beyond the five predefined basic Types in the Automation grammar, the author of an Automation Script may define their own sub-Types. Sub-Types are derived from the basic Types with added restrictions. These are created using the NUMBER, ENUM, and LIST declaration statements. A ENUM sub-Type puts constraints on the TEXT Type. A NUMBER sub-Type puts constraints on the NUMBER Type. A LIST sub-Type puts a limitation on the Types that a LIST Type variable may contain. Refer to the ENUM, NUMBER, and LIST entries in the following ‘Statements’ section for full descriptions of the way these Types can be constrained.

Sub-Types must be defined before they can be used. When a sub-Type is declared prior to a METHOD block, that sub-Type may be used in the parameter section of any Method block that follows the Type definition statement.

Value Type Casting

Intentionally transforming a value from one Type to a different Type is known as *casting*. Casting is necessary when we want to use a value in a different form and therefore need to change the value before it is stored. Additionally, there are situations where we might not know the Type of a value being read into Automation (for example, through the Job, Method, or Sample attributes). In these cases, we need to make sure that the value we are storing is of the proper Type for the variable that we are using to hold the value.

The author can request a cast operation be performed when providing an initial value for a variable definition and when assigning a new value to an existing variable. In both cases, the destination of the casted value is a variable and the source value is another variable or a value obtained from an external source.

A value can neither be cast to the DATA Type nor to the LIST Type. Values can be cast to the BOOLEAN, NUMBER, and TEXT Types and to any sub-Type of these three basic Types. The resulting Type is determined by the Type of the variable to which the result is being stored. For example, if the author wants to store a NUMBER Type value to a TEXT Type variable then the cast will attempt to convert the number to a String. If the cast cannot successfully transform the data, then a “type-error” is raised at runtime.

From Type	To Type	From Value	Result
BOOLEAN	NUMBER	TRUE	1
		FALSE	0
	TEXT	TRUE	“TRUE”
		FALSE	“FALSE”
NUMBER	BOOLEAN	0	FALSE
		<i>Non-zero</i>	TRUE
	TEXT	<i>All values</i>	<i>The String representation of the number</i>
TEXT	BOOLEAN	“”	FALSE
		<i>Non-zero length String</i>	TRUE
	NUMBER	<i>Well-formed numerical representation of a number with optional unit</i>	<i>The value of the number representation</i>
LIST	BOOLEAN	{ }	FALSE
		<i>Non-zero length LIST</i>	TRUE
DATA	BOOLEAN	<i>Any valid data file reference</i>	TRUE

A Null value will be transformed into a False, 0 (zero), or empty String when cast depending on the Type the destination variable.

Duration Syntax

There are two ways to specify a duration (a period of time) in an Automation Script. In the following examples, an n will be used to indicate a non-negative number – a number that is greater than or equal to zero. The number does not have to be an integer value. Here are the four basic constructs of the first form:

```
 $n$  SECONDS
 $n$  MINUTES
 $n$  HOURS
 $n$  DAYS
```

Note that the ‘S’ on the end of each duration keyword (shown shaded) is optional and may be added for readability. The above four basic duration constructs may be combined. The longer unit durations must appear prior to the shorter durations when written this way. Here are some examples:

```
10 MINUTES
1.5 MINUTES
1 MINUTE 30 SECONDS
6 HOURS 1 SECOND
1 DAY 12 HOURS 30 MINUTES 30 SECONDS
```

The reader should also note from the example above that 1.5 MINUTES and 1 MINUTE 30 SECONDS produce equivalent results. Specifying a zero for one of the durations has the same effect as if that duration was not specified. The following two duration expressions are equivalent:

```
5 MINUTES 0 SECONDS
5 MINUTES
```

The second encoding form of a duration is written by separating the non-negative numbers with a colon. The number of colons in the duration value will indicate the duration’s units. This form written like this:

```
day-number:hour-number:minute-number:second-number
```

Using this form, *minute-number* and *second-number* are required, but *day-number* and *hour-number* (shown shaded) may be omitted. Note that *hour-number* can be provided without a *day-number*, but because of the positional nature of this form, if *day-number* is specified then *hour-number* must also be specified. Double digits are not required and white space is allowed on either side of the colons as shown in the following examples. These examples have the same durations as the examples shown above using the first form respectively:

```
10:00          -- 10 minutes
1.5:0          -- 1 minute, 30 seconds
1:30          -- 1 minute, 30 seconds
6 : 00 : 01    -- 6 hours, 1 second
1: 12: 30: 30 -- 1 day, 12 hours, 30 minutes, 30 seconds
```

Basic Script Structure

The first keywords in a contemporary Automation Script file (that are not part of a comment) must be:

```
AUTOMATION TYPES VERSION number PURPOSE informational-text;
```

where the *number* represents a positive integer greater than or equal to 2. These keywords and the following *number* identify the file to be an Automation Script usable by version 5.0 and greater of JEOL's NMR software products. This line does not necessarily have to be the first line of the file – blank space and lines containing only comments, as described above, may precede these keywords.

If the keyword **TYPES** is included, then the Automation Script can neither contain Methods nor include other Scripts. Only purpose statements, Type definition statements, language translation statements, remarks, and comments are allowed.

A **PURPOSE** clause may follow the version *number*. The *informational-text* is one or more Strings separated by commas. The text should include a short description of the purpose of the Automation Script. The intent is to provide the user and reader of the Script with a short and helpful description about the Method(s) in the Script and their relationship to each other. Note that if the **PURPOSE** clause is part of this line, the text will not and cannot be translated.

If the **PURPOSE** clause is omitted from the initial **AUTOMATION** line, then a single **PURPOSE** statement may then follow the **AUTOMATION** line and any **REMARK** and/or **TRANSLATE** statements. In this case, the *informational-text* can be made up of one or more mixed Strings or translation identifiers separated by commas. However, translation identifiers can only be included in the *informational-text* if the **PURPOSE** statement follows one or more **TRANSLATE** statements that define the translation identifier(s).

```
PURPOSE informational-text;
```

The author of the Automation Script may choose to categorize the Methods within this file by following the **AUTOMATION VERSION** statement with a **CATEGORY** statement of the form:

```
CATEGORY category-list;
```

where *category-list* is a single quoted string or a comma separated list of strings. The categories in this list should aid the users in locating the Automation Scripts of interest to them.

Following the identifying lines should be one or more of the basic statements: a **METHOD** block, an **INLCUDE**, **TRANSLATE**, and/or a **REMARK** statement, a sub-Type declaration statement (**ENUM**, **NUMBER**, **LIST**), and/or a **MACRO** statement. Each of these statements is described in detail in the following section.

NOTE: In the above syntax line and in the Automation statement descriptions of the next section, any part of a syntax that is displayed with a shaded background is an optional part of the syntax and may be omitted. Some parts may be more darkly shaded indicating that portions of the optional part may themselves be omitted. Any part of a syntax that is underlined indicates that those elements may be repeated.

Statements

Each of the following sub-sections describe the statements that Automation makes available to use in an Automation Script. Near the beginning of each statement description is the syntax of the statement. Some statements are a bit complex so the syntax may be expanded in the paragraphs that follow. Examples of each statement are provided for clarity. The examples are distinguished from the body text by left side vertical lines.

There are a few statements that are permissible at the outermost level of an Automation Script. Not including the initial identification statement beginning with the AUTOMATION keyword, these statements are: METHOD, TRANSLATE, ENUM, NUMBER, LIST, and REMARK. Within a METHOD block, however, all the Automation statements may be used with a few restrictions. The statements that have restrictions on where they can be used are:

- the EXIT statement, which can only be used within a REPEAT block, and
- the RETRY statement, which can only be used as the last statement in an ON ERROR block.

To do work, an Automation Script must contain at least one METHOD block. A METHOD block is the statement used to construct the series of steps that Automation will perform to carry out a task. Any of the following statements may be used within a METHOD block, including other METHOD blocks. The following statements are discussed in alphabetical order, but it may be useful for the reader to peruse the section pertaining to Methods first.

CALL

The CALL statement can execute a Percival operator or a service provided by the spectrometer or an available hardware device attachment. The following syntax is used to call a Percival operator:

```
CALL PERCIVAL percival-operator-name( argument-list );
```

To execute a service on a device via the Service Manager use the following syntax:

```
CALL SERVICE "service-request"( argument-list );
```

The first difference in the syntax is that the PERCIVAL keyword is optional when calling a Percival operator, but the SERVICE keyword is required when invoking a service. The second difference is that quotation marks are required around the *service-request* but quotation marks must not be used around the *percival-operator-name*. In both cases, the argument list, which includes the surrounding parentheses, is optional.

Service Request Syntax

A *service-request* has the following form:

```
"provider.major-version.minor-version::operation"
```

where *provider* is the name of the device that is providing the service, *major-version* and *minor-version* are both integers indicating the version of the service that is being requested, and *operation* is the service function to be executed. Notice that the *minor-version* and the period before it is an optional component of the service name.

Passing Information Through Parameters

An 'argument' (in the computer science sense) is a value that is passed to another routine through its parameter interface. When an *argument-list* is specified, it must be surrounded by parentheses. One or more values, variable names, substitute identifiers (see the 'Table of Substitute Identifiers' after the 'Statements' section for the full list of words that are recognized), or the special keywords (USER, ALL, RAW, or PROCESSED) may be included in the *argument-list*. To specify more than one argument, separate each argument from each other with a comma.

The NULL keyword can be used as an argument for either a Percival operator or for a Service call causing a Null value to be passed for that parameter.

- **Passing Data Files**

The ALL keyword will construct a set of every data file that has been collected or processed up to the point of the CALL statement and use that set as an argument of the operator or service at the position of the ALL keyword in the argument list.

The RAW and PROCESSED keywords behave similarly except that the RAW keyword will construct a set of just the raw data files and the PROCESSED keyword will construct a set of just the processed data files.

Only one instance of either ALL, RAW, or PROCESSED is allowed in the argument list and it may optionally be followed by the FILES keyword for readability.

- **Passing User Authentication**

The USER keyword will cause a Network Context to be passed as the argument at the position of the keyword. The user's Network Context contains the username and password of the operator who submitted the job.

Asynchronous Service Limitation

At this time, asynchronous service calls may be executed, but the resulting value returned from the service call will be lost. This means that Services that make a request and return a result through a callback will **not** function properly in Automation.

Examples

The following are examples of the use of the CALL statement. Note that all of the Percival operators and service names in these examples are fictional.

The simplest for is a basic Percival operator invocation.

```
| CALL my_operator;
```

A CALL statement will return the resulting value to Automation if the Percival operator or service routine that is specified returns a value. This next example of another Percival operator invocation shows this syntax (refer to the SET statement). The 'factorial' function computes the mathematical factorial of 5, 5!, which will result in the value of f being $120 = 5 \times 4 \times 3 \times 2 \times 1$.

```
VAR f : NUMBER;
| SET f = CALL factorial( 5 );
```

The following example passes a constant String of text and all the processed data files (acquired by any prior EXPERIMENT block) to a function called examine_data.

```
| CALL examine_data( "peak", PROCESSED FILES );
```

The following example passes all the raw data files (acquired by any prior EXPERIMENT block) to the check_file operation of the file_service service.

```
| CALL SERVICE "file_service.1.0::check_file"( RAW FILES );
```

The next example passes the JOB_ID and the value of the status variable to a service routine called interface. The result of the service invocation is stored in the variable result.

```
VAR status : TEXT = "ok";
VAR result : NUMBER;
| SET result = CALL SERVICE "device.1.0::interface"( $(JOB_ID), status );
```

The next example passes the user's authentication information to a service with a Boolean value and then to a Percival operator with a Number.

```
| CALL SERVICE "device.1.0::reset"( USER, TRUE );
| CALL reset( 0, USER );
```

CONST

A constant is a named value like a variable except that its value is immutable (it **cannot** be modified during the execution of the Automation Script). Refer to the VAR statement for a description of creating a variable whose value can be modified. A constant must be defined before it can be referenced and it must be assigned a value at the time that it is created. Use the CONST statement to create a named value that cannot be changed during the execution of the Automation Script. The CONST statement has the following syntax:

```
EXPOSE CONST constant-name : value-type = default-value
      WHEN expression , dependency , HELP help-text;
```

The *constant-name* is an identifier that uniquely names the constant. More than one constant, variable, or Method parameter with the same name cannot exist in the same scope level. A constant, variable, or Method parameter with the same name at an outer scope level will be hidden (or eclipsed) by the newly defined constant.

Value-type may be any of the five basic Types defined in the Automation grammar (BOOLEAN, NUMBER, TEXT, LIST, or DATA) or it may be a sub-Type defined prior to the Method using the ENUM, NUMBER, or LIST statement.

The *default-value* can be specified in one of the following ways:

```
constant
JOB job-attribute ELSE constant
SAMPLE sample-attribute ELSE constant
NAMESPACE namespace-path ELSE constant
EVALUATE (expression) ELSE constant
```

Constants of the DATA Type cannot use the JOB, SAMPLE, NAMESPACE, or EVALUATE clauses shown above to set their value. Only a constant String or an expression that results in a TEXT Type value is a legal form for initializing a DATA Type constant. Note that the use of an expression to initialize a constant could potentially raise a Type-Error exception while the Script is running if the expression does not result in a TEXT Type.

Description

Constant is a value that should be of the Type *value-type*, which is the Type of the constant being defined. If constant is not appropriate for *value-type*, the error condition “type-error” will be raised when the statement attempts to set the value of the constant.

Job-attribute is a String or variable of the TEXT Type that specifies the attribute of interest from the running Job Group.

Sample-attribute is a String or variable of the TEXT Type that specifies the attribute of interest from the current sample.

Namespace-path is a String or variable of the TEXT Type representing a stored value in the Namespace parameter database.

Expression is a Percival code expression that will be evaluated.

More than one Job, Sample, Namespace, or Evaluate clause may be specified when they are separated by an ELSE keyword before the optional final ELSE clause.

The optional ELSE clause provides a specific default value for the cases when the *job-attribute*, *sample-attribute*, *namespace-path* cannot be found or when those values or the evaluation

expression cannot be resolved to an appropriate value of the required Type. If the ELSE clause is omitted, then an error condition will be raised if a value is not found or if the *constant* value after the ELSE keyword does not conform to the requirements of the constant declaration.

Help for the User

A short helpful description may be attached to exposed constant definitions (those beginning with the EXPOSE keyword) by adding a comma, the HELP keyword, and one or more Strings (separated by commas) at the end of the constant declaration. Help text is not required, but its inclusion is strongly encouraged so that users may better understand the purpose of the variable.

The help text can be replaced by a previously defined language translation identifier to allow the help text to be determined by the current locale. See the TRANSLATE statement for information about creating and using translatable text.

Exposed Constants

The EXPOSE keyword causes the constant's name and value to appear in the Method attributes area of the Job page on the *Spectrometer Control* window. The description provided in the help text will also be displayed with the constant. Exposing a constant gives the user the ability to override the default initial value specified by the constant declaration. The ultimate value of an exposed constant will be the value that is supplied by the operator on the user interface.

The When Clause

The WHEN clause may only be specified if the EXPOSE keyword is used. The constant will only be enabled on the user interface when the expression evaluates to a True value. The *expression* is evaluated each time a Method parameter or other exposed variable/constant is changed.

Dependent Constants

A *dependency* clause may only be specified if the EXPOSE keyword is used and it stipulates how the initial value of this constant should be affected when other values upon which it depends are changed. A *dependency* clause is specified on an exposed constant in the same way that it can be attached to a Method parameter.

A *dependency* clause has the following syntax:

```
DEPENDS ON parameter-or-variable-list EVALUATE (expression)
```

Refer to the *Parameter Options: Dependencies* section in the description of the METHOD statement for more details.

Examples

The following are examples of the use of the CONST statement. This first example defines a value for mathematical π (represented by the variable named `math_pi`).

```
| CONST math_pi : NUMBER = 3.14159265359;
```

The following example obtains the "filename" attribute of the job and sets `fname` to represent its value. `fname` will be set to "" (an empty String) if the "filename" attribute is not found or if the attribute is not of the TEXT Type.

```
| CONST fname : TEXT = JOB "filename" ELSE "";
```

The next example obtains the “notebook” attribute from the current sample and sets it to the constant named nbook. nbook will be set to “unknown” if the Sample attribute cannot be found.

```
| CONST nbook : TEXT = SAMPLE "notebook" ELSE "unknown";
```

The next example creates a constant minimum sweep width value set to 2[ppm] that will be visible on the Automation user interface by using the EXPOSE keyword.

```
| EXPOSE CONST min_sweep : NUMBER = 2[ppm],  
  HELP "The minimum sweep width for",  
  "all experiments in this Method";
```

The next example gets the “field_strength” value of the magnet from the Namespace parameter database and stores it in the field constant.

```
| CONST field : NUMBER = NAMESPACE "parameter.field_strength";
```

The value of the next example is determined by evaluation of the String expression.

```
| CONST two_pi : NUMBER = EVALUATE (2 * pi);
```

The following example creates a constant Boolean value and a dependent value.

```
| CONST debug : BOOLEAN = FALSE;  
| CONST log_msg : BOOLEAN = FALSE WHEN debug = TRUE;
```

The following examples are errors. The first is a syntax error because it does not provide a default value. The second is an error because the Type of the value does not match.

```
| CONST rate : NUMBER;      --This is an error, no value provided  
| CONST name : TEXT = 5;    --5 is a Number, expected a String
```

The following examples show the use of a user defined Type.

```
ENUM Nuclei IS ( "1H", "2H", "13C" );  
| CONST nucleus : Nuclei = "1H";
```

DELAY

The DELAY statement will cause the execution of the Method to pause for a specified amount of time. The two syntax forms of the DELAY statements are:

```
DELAY time AFTER data-variable-name WHEN expression;
```

The *time* is specified using the Duration syntax. Refer to the section titled ‘Duration Syntax’ that describes how to specify times. Here are two identical examples:

```
2 MINUTES
2:00
```

These examples tell Automation that the Method should stop execution until after two minutes have passed from the time the DELAY statement was encountered.

Waiting After a Data Acquisition

If the AFTER keyword is given with a *data-variable-name*, then the time to wait is relative to the time that the specified data file had finished acquiring and was stored to disk. This form has the following syntax:

```
DELAY 2 MINUTES AFTER prior_data;
```

For this example, *prior_data* is the name of a data file variable from a previous experiment. The actual wait time may be shorter, but will not be longer than the specified time (in the above example, two minutes). There will be no delay if the specified data file had been stored more than the specified amount of time before the DELAY statement was encountered. In this example, if the data specified by *prior_data* had been stored more than two minutes ago, the Method would proceed without delay.

The When Clause

If the WHEN clause is specified, the DELAY statement will cause the Method to pause when the specified *expression* evaluates to a True value. Refer to the section titled ‘Boolean Expressions’ for how an expression evaluates to a Boolean value.

The *expression* can also begin with or be replaced by a Job attribute, a Sample attribute, or a Namespace value. If the value obtained is not one of the False values, the Method will pause.

```
DELAY 1 MINUTE WHEN JOB attribute expression;  
DELAY 1:00 WHEN SAMPLE attribute expression;  
DELAY 60 SECONDS WHEN NAMESPACE path expression;
```

Attribute and *path* must be a quoted String literal value or a variable of the TEXT Type. If *expression* is included, it must be written such that the value of the Job, Sample, or Namespace part is the first value of the conditional test. For example, SAMPLE "count" > 0. If an *expression* is not provided, then the value of the *attribute* or Namespace *path* alone will determine the truth state.

EMAIL

The EMAIL statement will send an electronic message from the user who submitted the job to the specified recipient(s). The syntax of the EMAIL statements is:

```
EMAIL ALERT TO addresses CC addresses BCC addresses  
SUBJECT string-or-variable  
MESSAGE message-text  
ATTACH variable-name;
```

Specifying Recipients

Addresses is a list of one or more of the following options separated by the AND keyword: a comma separated list of properly formatted email address within quotation marks, a constant or variable name containing a TEXT Type value of a properly formatted email address, a constant or variable name containing a LIST Type value holding one or more properly formatted email addresses, or the USER keyword. The USER keyword may only appear after the TO keyword and it will include the email address associated with the account of the person who submitted the Automation Script to the Job queue. This email is specified in the Account Management tool of the spectrometer.

The TO keyword is optional and specifies that the following email *addresses* should be set to the “to” line of the email. The CC (carbon-copy) keyword and its following *addresses* specify the email addresses that are to be set to the “cc” line of the email. The BCC (blind carbon-copy) keyword and its following *addresses* specify the email addresses that are to be set to the “bcc” line of the email. Bcc addresses are not visible to any of the recipients of the email message. The email addresses are processed in order and duplicates are ignored.

Email Alerts

The inclusion of the ALERT keyword indicates that this email contains a message that is of a higher priority than usual. When the ALERT keyword is included and the USER keyword is used to specify a recipient, the special “alert” email address of the user will **also** be added to the email recipient list. Alert messages should be short. It is recommended that the combination of the lengths of the subject line and the *message-text* be less than 160 characters to conform to the standard SMS text messaging limitations. This is because users can specify the email address of their mobile service provider’s email gateway such that the email will be forwarded to their mobile device as a text message. Attachments cannot be added when the ALERT keyword is included.

Subject Line

The SUBJECT keyword specifies the line of text that is to be the subject of the email message. If no subject is provided and the ALERT keyword is omitted, the subject will be automatically generated to be “Automation message from *spectrometer* - Job *job-id*” where *spectrometer* and *job-id* will be replaced with the current name of the spectrometer running the Method and the current Job ID respectively. The author may specify the subject of the message with a quoted String, with a constant or variable containing a TEXT Type value, or with a language translation identifier. Refer to the TRANSLATE statement regarding language translations.

Message Body

The *message-text* follows the MESSAGE keyword and can be either a series of Strings separated by commas or a single language translation identifier. This text will become the body of the electronic mail.

Value Substitution

There are many special symbols that may appear within the subject line and the body text that will be replaced when the message is sent. The special symbols are formed by placing a word within parentheses and preceding it with a '\$' character. As is the case for the entire Automation syntax, the casing of the letters of the word within parentheses is not significant. This way, the author of the Script can include the value of a named constant, variable, or parameter previously defined in the Automation Script into the email. See the 'Table of Substitute Identifiers' after the 'Statements' section for the full list of words that can be used in the subject text and message body text as substitutes for various values.

Including Attachments

When the optional ATTACH keyword is included in an EMAIL statement, it must be followed by a single *variable-name* or a list of *variable-names* separated by AND keywords. The Attach phrase may appear at the end of this statement zero, one, or more times. The inclusion of this keyword causes each of the files specified by *variable-name* to be included into the email as an attachment. Specifying ATTACH multiple times with a single *variable-name* has the same effect as specifying ATTACH one time with a list of *variable-names*.



WARNING! Use data file attachments with caution since JEOL NMR data files can often be quite large which can cause some email servers to reject the message.

Examples

The first example of the EMAIL statement will send an alert email the user who submitted the job with the default subject line and a message body of "Job Complete!".

```
EMAIL ALERT USER MESSAGE "Job Complete!";
```

The next example sends an email to people with a simple message containing some substitutable values. The address is put in the BCC line and the data in the variable `exp` is attached to the message.

```
EMAIL BCC "everyone@my_company.com"
  SUBJECT "Check out this data"
  MESSAGE
    "Date: $(NOW)",
    "Collected with sample $(SAMPLE) using $(SITE)"
  ATTACH exp;
```

In the following example, the user who submitted the job and a supervisor will be sent a simple message with three attached files.

```
EMAIL USER CC "supervisor@my_company.com"
  MESSAGE "Here are your data sets"
  ATTACH data1
  ATTACH data2 AND printed_data;
```


ENUM

The ENUM statement creates a sub-Type of the base TEXT Type. This is called an enumerated Type because each permissible text value must be explicitly specified by the definition of the sub-Type. The basic syntax to create an enumerated sub-Type that allows the author to specify the legal values is:

```
ENUM type-name IS ( values-list );
```

An alternate syntax to create an enumerated sub-Type will cause the *values-list* to be obtained from a specified Namespace path. It has the following form:

```
ENUM type-name IS NAMESPACE casing KEYS namespace-path
    EXPOSE key-name-values-list IGNORE key-name-values-list
    EXCLUDE values-list;
```

A third syntax obtains the *values-list* from the result of a call to a Service or Percival operator that returns a Set of Strings.

```
ENUM type-name IS CALL percival-operator-name( arguments );
ENUM type-name IS CALL SERVICE "service-request"( arguments );
```

It is also possible to mix the above syntaxes by separating the values definition portions with the AND keyword. The author may begin with any of the syntax forms and may specify more than one of each form.

```
ENUM type-name IS ( values-list ) AND
    NAMESPACE namespace-path AND
    CALL percival-operator-name;
```

Description

The *type-name* is an identifier that specifies the name of the new sub-Type. The ENUM statement extends the number of available Types that permits statements that come after the ENUM Type definition to reference the new sub-Type. The *type-name* may be used in the place of a basic Type identifier to restrict the set of permissible TEXT Type values. The set of permitted values, in *values-list*, can be explicitly specified between the parentheses. In this case, each text value must be within quotation marks and separated from each other by a comma.

Namespace can be used to provide the set of permitted values by including the NAMESPACE keyword followed by a *namespace-path* that must refer to a LIST Type Namespace value. The *namespace-path* is a String between quotation marks. If the Namespace value, which becomes its own *values-list*, does not exist or is not a LIST Type of Strings then the only legal value for the new sub-Type is "". Substitution identifiers cannot be used within the *namespace-path* for a Type declaration.

A CALL statement can be used to generate a *values-list* by calling a Percival operator or Service that returns a value that is of the Percival SET Type. The syntax of the CALL statement used within the ENUM statement is the same as other uses of the CALL statement as it can appear elsewhere in an Automation script. See the description of the CALL statement described earlier.

The author of an Automation Script should create an enumerated sub-Type if the user will be required to specify a value to a parameter of a Method or to an exported variable where only a limited set of values is permissible for the variable. Making use of an enumerated sub-Type allows the Automation user interface to provide the user with a ‘Select Enumeration’ widget rather than a basic textual ‘Input Box’ aiding the user in assigning appropriate values to Method parameters and variables.

TEXT Type Relationship

Note that a value of an enumerated sub-Type may be assigned to any variable of the TEXT Type. This implies that all the legal values of the enumerated sub-Type are also legal values of the basic TEXT Type. Conversely, a value of the TEXT Type may only be assigned to a variable of an enumerated sub-Type if that value is within the list of permitted values.

Enumeration Indexing

An enumerated sub-Type can be indexed to retrieve the enumerated value at a specific position within the sub-Type. This is accomplished by following the name of the enumerated sub-Type by an integer between square brackets, *Type*[*n*]. This form cannot be used within a Percival expression since Percival is not aware of the Types defined by the Automation script.

When an enumerated sub-Type is defined, the order of the permissible values is set and never changes. A positive integer will obtain the value counting from the beginning of the *values-list* and a negative integer will obtain the value counting from the end of the *values-list*. An index of 0 (zero) is illegal.

Enumeration indexing is not permitted on a *value-list* that is produced using the following:

- 1) a Namespace path that points to an Association of elements using the KEYS keyword, or
- 2) using the CALL statement to get the values programmatically at run-time.

Obtaining the Values From Namespace

The enumerated values can be retrieved from Namespace when the value of the *namespace-path* is a list or an association type value. If the value of the *namespace-path* is an association type, use the KEYS keyword to get the values of the key names. Omit the KEYS keyword if the value of the *namespace-path* is a list type.

The casing of the values can be adjusted to all lowercase, all uppercase, or capital-case using the keywords LOWERCASE, UPPERCASE, or CAPITALIZE respectively. Capital-case will capitalize the first character and any subsequent character whose left-adjacent character is a space, an underscore, or a hyphen. Capital-case will not change any uppercase characters in the values to lowercase.

It is possible to prevent one or more of the values returned from Namespace from becoming a part of the final list of legal values for the enumerated sub-Type. This is useful if the resulting values from Namespace contain extra items that the author wishes to exclude. Add the EXCLUDE keyword at the end of the NAMESPACE clause followed by a list of values to leave out. The list of values is a comma separated list of Strings surrounded by parenthesis.

If the list of values comes from the names of the keys of a Namespace association (by using the KEYS keyword), then it is possible to permit only certain names of the keys to be included in the resulting enumerated sub-Type when the Type of the value of each of those key names are themselves an association Type. To be concise, this option is available when the Type of the value of *namespace-path* is an association (first-level) and each of the values of the association are also associations (second-level). Use the EXPOSE keyword to permit the first-level key name to be included when the specified second-level key name or names are found within the second level

association. Alternatively, use the IGNORE keyword to cause the key to be excluded when the specified second-level key name or names are found within the second level association. Either the EXPOSE keyword or the IGNORE keyword must be followed by a comma separated list of Strings between parenthesis which are compared to the second-level key names. The author cannot specify both EXPOSE and IGNORE in the same NAMESPACE clause.

Examples

The following are examples of the use of the ENUM statement. The bold font has been used to show how the *type-name* is used when declaring variables. Proper and improper uses of the values of enumerated Types are also described here.

```
ENUM RGB_Color IS ( "red", "green", "blue", "grey" );

ENUM Print_Type IS ( "b&w", "grey", "full-color" );

ENUM GS_Shims IS NAMESPACE "gradient_shim.allowed_shims";

ENUM PValues IS CALL Get_Legal_Values( USER );

ENUM Shims IS ( "" ) AND
              NAMESPACE "shim.shim_names(INCLUDE)" AND
              ( "All" );
```

The last ENUM statement above includes all the shim names in the list specified by the Namespace path in quotes and adds an empty string at the front of the list and appends the value "All" to the end of the list. This makes the entire list of system shim names, the empty string, and "All" valid values for the Shims sub-Type.

Examples of using sub-Types follow:

```
VAR C : RGB_Color;
VAR P : Print_Type;
VAR T : TEXT;

SET C = "green";      --okay
SET C = "orange";    --error ("orange" is not allowed)
SET C = "";          --error (" " is not allowed)
SET T = C;           --okay
SET T = "red";
SET C = T;           --okay
SET T = "test";
SET C = T;           --error ("test" is not allowed for RGB_Color)

SET C = "red";
SET P = C;           --error ("red" is not allowed for Print_Type)

SET C = "grey";
SET P = C;           --okay ("grey" is legal in both RGB_Color and Print_Type)
```

```
SET C = RGB_Color[1];      --okay    (sets C to "red")
SET C = RGB_Color[2];      --okay    (sets C to "green")
SET C = RGB_Color[5];      --error   (5 is out of range)
SET C = RGB_Color[-1];     --okay    (sets C to "grey")
SET C = RGB_Color[-2];     --okay    (sets C to "blue")
SET C = RGB_Color[-5];     --error   (-5 is out of range)
```

The following statement defines a list of domain names that comes from the key names of the association of the Namespace path "gamma". Each first-level key name will only be included in the Gamma_Domains sub-Type if a second-level key named "nodraw" is *not* a part of the second-level association values. The value "None" is prepended to the resulting list.

```
ENUM Gamma_Domains IS
( "None" ) AND
NAMESPACE CAPITALIZE KEYS "gamma" IGNORE ("nodraw");
```

EXIT

The EXIT statement is used to terminate the execution of the sequence of statements that are part of a REPEAT block. It may only be used within the part of a REPEAT block that does the repetition (not the THEN block) but it may be used within other statement blocks that are embedded within the REPEAT block, such as the IF statement. The EXIT statement has the following syntax:

```
EXIT WHEN expression;
```

When the EXIT statement is encountered without a WHEN clause, the execution of the statement block terminates and the statement after the end of the REPEAT block is executed next. The THEN block of the REPEAT statement will not be executed if the EXIT statement causes the repetition to end earlier than it normally would have.

The When Clause

If the WHEN clause is specified, the REPEAT block will be exited if the *expression* evaluates to a True value. Refer to the section titled 'Boolean Expressions' for how an expression evaluates to a Boolean value.

The *expression* can also begin with or be replaced by a Job attribute, a Sample attribute, or a Namespace value. If the value obtained is not one of the False values, the block will exit.

```
EXIT WHEN JOB attribute expression;  
EXIT WHEN SAMPLE attribute expression;  
EXIT WHEN NAMESPACE path expression;
```

Attribute and *path* must be a quoted String literal value or a variable of the TEXT Type. If *expression* is included, it must be written such that the value of the Job, Sample, or Namespace part is the first value of the conditional test. For example, SAMPLE "count" > 0. If an *expression* is not provided, then the value of the *attribute* or Namespace *path* alone will determine the truth state.

Examples

The following two examples make use of the EXIT statement produce the same result – printing the numbers 0 through 10 on the console. They both assume that a variable or constant named count was previously defined to be of the NUMBER Type and that was properly initialized.

```
VAR count : NUMBER = 0;
```

The first example makes use of the basic exit statement. An EXIT statement within an IF block can be used to determine when the repetition should end.

```
REPEAT  
  IF count > 10 THEN  
    EXIT;  
  END IF;  
  
  INFORM TO CONSOLE "Counter is $(count)";  
  SET count = count + 1;  
END REPEAT;
```

The following example uses the WHEN clause to determine the exit condition. It is shown that the WHEN clause causes the EXIT statement to behave as if there was an IF statement around it as in the example above.

```
REPEAT
|   EXIT WHEN count > 10;

    INFORM TO CONSOLE "Counter is $(count)";
    SET count = count + 1;
END REPEAT;
```

The following example will end the repetition when the Sample has an attribute named "done" that evaluates to a True value.

```
REPEAT
|   EXIT WHEN SAMPLE "done";

    IF count > 10 THEN
        SET SAMPLE INTERIM "done" = TRUE;
    ELSE
        ...
        SET count = count + 1;
    END IF;
END REPEAT;
```

The above loop can be written more compactly when the expression is combined with the SAMPLE clause. This is a very silly example but is useful for illustrative purposes.

```
REPEAT
|   EXIT WHEN SAMPLE "count" > 10;

    ...
    SET count = count + 1;
    SET SAMPLE "count" = count;
END REPEAT;
```

EXPERIMENT

The EXPERIMENT block acquires data from an NMR spectrometer. The EXPERIMENT block has the following syntax:

```
CONCEAL INTERIM QUIET SCOUT
EXPERIMENT data-name IS
    SAVE AS string-or-variable;
    COLLECT experiment-filename-or-variable;
    OPTIMIZE optimization-parameters;
    experiment-parameter-assignments-and-constraints;
END EXPERIMENT data-name;
```

The data acquired by this statement can later be accessed by the constant that is implicitly defined with the name given by *data-name*. Note that the author of the Automation Script cannot use quoted text for the data variable name as is permitted for the name of a Method. The difference being that the name of a Method does not define a variable as the EXPERIMENT does with *data-name*.

One File or Multiple Files?

A DATA Type or LIST Type constant value will be created with the name that is provided and it will contain the acquired data file(s). It is possible for multiple data files to be produced by the EXPERIMENT block. The script author should expect a DATA Type value to be created unless the ‘interleave_multi’ or the ‘multi_file’ experiment parameter is set to be TRUE in the specified experiment file. Refer to the *Pulse Programming Guide for Control version 5* for information regarding these two and other experimental parameters.

Acquiring Temporary Data

The optional INTERIM keyword informs Automation that the data file(s) collected by this EXPERIMENT block are not to be kept. All of the raw acquired files, as well as any processed files that were derived from these files, will be deleted when the nearest enclosing Method completes.

Optional Acquisition

The optional SCOUT keyword informs Automation that this EXPERIMENT block will not produce data of significance other than to potentially provide the conditions for another EXPERIMENT block to use. At the time when a file would normally be acquired, a search is performed using the parameters of the scout experiment for a file with similar acquisition parameters. If a match is found, then the acquisition of this EXPERIMENT block is skipped and the matched file is used in the place of the acquisition.⁹ If both the INTERIM keyword and the SCOUT keyword are specified, then the file will be deleted only when a match cannot be found – a pre-existing data file will not be removed.

Specifying the Pulse Program

Each of the sub-components of the EXPERIMENT block must appear in the order shown in the syntax above. However, the COLLECT statement is the only required part inside the block.

The COLLECT statement can only be used within an EXPERIMENT block and it specifies the pulse program that will run to acquire the data. The COLLECT keyword is followed by a

⁹ The search criteria for locating a suitable “stand-in” data file has not yet been determined.

filename that can either be a String or a variable of the TEXT Type that contains a filename. Experiment pulse program files usually end with the .jxp extension. Specifying the extension is not necessary and it is recommended that it be omitted from the filename when specified within an Automation Script. If the filename extension is omitted, pulse program files will be located using the known extensions starting with the newest extension – .jxp. The standard locations are searched to find the file as described in the preceding section titled ‘Locating Support Files’.

Customizing the Filename

There may be multiple identical Methods or Experiments within a Job that vary only by their settings and it is not optimal for each of them to produce a file with the same name. However, it may be important to have parts of the filename be identical in order to group related data files together.

The SAVE AS statement gives the author of the Automation Script the ability to adjust the name of the acquired data when it is stored on the Data Server. The filename String may contain various special identifiers that will be replaced by the appropriate current values at the time the file is stored. See the ‘Table of Substitute Identifiers’ following the ‘Statements’ section for the list of recognized identifiers that can be used in building the filename.

The filename may be provided using a String value or a TEXT Type variable or constant.

The SAVE AS statement is optional and if it is not provided or evaluates to an empty String, the name of the file will be determined by a customizable system preference (‘Filename Pattern’ found on the ‘Environment’ tab in the Spectrometer Preferences window). The default value of the ‘Filename Pattern’ preference is:

```
$ ( SAMPLE ) _ $ ( EXP . FILENAME )
```

This default form specifies that the data filename will ultimately be composed of the name of the Sample followed by the ‘filename’ parameter defined in the Header section of the *experiment-filename-or-variable*.

If the ‘Filename Pattern’ preference is undefined or evaluates to an empty String then the filename is set to be the ‘filename’ parameter defined in the Header section of the *experiment-filename-or-variable*. If this also evaluates to an empty String, then the title of the Experiment Block that is also used as the variable name holding the acquired data, *data-name*, will be used.

After all evaluations are complete, any leading and trailing spaces will be removed from the filename and any multiple spaces within the filename will be reduced to a single space to determine the final storage name of the acquired data.

Optimizing the Parameters of the Acquisition

The OPTIMIZE statement will cause the COLLECT statement to execute repeatedly until the optimization *function* is minimized. The optimization *function* makes modifications to the specified acquisition parameters to produce the “optimized” data.

The optimization syntax has the following form:

```
OPTIMIZE (param1, param2, ... paramn)
  LIMIT max-iterations mode
  CALL function
  WITH param1 = val1,1, val1,2, val1,n, val1,n+1;
      param2 = val2,1, val2,2, val2,n, val2,n+1;
      ...
      paramn = valn,1, valn,2, valn,n, valn,n+1;
```


The parameters to the optimization function, which must be one or more names of experimental acquisition parameters, are specified within parenthesis after the OPTIMIZE keyword. These parameters will vary during the optimization stage.

The author can optionally impose a maximum number of collections that can execute. The LIMIT keyword followed by a positive integer (or the keyword YES or NO) will specify the number of collections that the optimizing loop is allowed to run. If the LIMIT statement is not specified or if the YES keyword is used instead of a number, the maximum number of collections is set to be 100. If LIMIT NO is specified, there will not be a maximum number of collections imposed on the optimization.

WARNING! Specifying NO could potentially cause an infinite loop if the optimizing function never converges! One cause of this could be that the values of optimizing parameters do not change.



When a limit is put on the number of collections, the author of the Automation Script may also specify whether the execution of the Method is to continue or terminate if the maximum number of iterations is reached without convergence. Use the keyword CONTINUE or TERMINATE to stipulate the desired behavior. If left unspecified, the default is to continue the Method as if the optimization had converged at the iteration limit.

The optimizing *function* is specified after the CALL keyword. This *function* can be specified as an expression within quotation marks or as a name of a Percival operator. Each optimizing parameter should be included in the optimizing expression. A Percival operator that is used for this purpose must be written such that it takes a single Delta data file Type parameter. Here is a functional specification for an optimizing function:

```
function OPTIMIZER( fl : FILE ) return NUMERIC is...
```

Finally, each optimizing parameter's initial values are specified after the WITH keyword. The syntax for specifying each parameter's initial condition is by providing its name followed by an equal sign and a comma separated list of numbers. There must be one more number in the list than the total number of optimization parameters in parenthesis after the OPTIMIZE keyword. So, if three (3) parameters are required for the optimization, then there must be four (4) initial values specified for each of those parameters.

Adjusting the Acquisition Parameters

Following the COLLECT statement (and the optional OPTIMIZE statement) is the portion of the EXPERIMENT block where experimental parameters can be set and/or constrained. Experiment parameters are given values using the SET statement and these same parameters may be constrained using the CONSTRAIN statement. The syntax for a SET statement in an EXPERIMENT block is the same as it would appear in a Method and so it will be described in its own section. Refer to the section describing the SET statement. The main difference between the function of a SET statement within an EXPERIMENT block as opposed to outside of an EXPERIMENT block is that the values are assigned to **Experiment** parameters when within the EXPERIMENT block rather than to local Method variables.

Here is an example:

```
SET scans      = 20;
   x_sweep    = prior_data("x_sweep_clipped");
   x_offset   = method_variable_name;
```

The second line in the example above shows that values can be retrieved from parameters of data files by following a data file variable with a parameter name within double quotation marks and parentheses. In this example, `prior_data` is a variable or constant that contains a data file, and `"x_sweep_clipped"` is the parameter name of the value that is desired.

The SET statement in an EXPERIMENT block can make use of the same forms as the SET statement of a Method by requesting a value from Namespace, from the Job's attributes, from the current Sample's attributes, from a Percival operator call, or from a Service Manager call. However, it is not possible to use the PROCESS clause here. Again, refer to the following description of the SET statement for more details of the various ways values can be assigned.

Constraining Acquisition Parameters

The CONSTRAIN statement may only be used to constrain NUMBER Type values. It is unique to the EXPERIMENT block (it cannot be used outside of an EXPERIMENT block) and has many optional components that aid in limiting the range of values that an experimental parameter may be assigned. The following is the syntax of the CONSTRAIN statement:

```
CONSTRAIN parameter bounds multiple modulo units;
```

Parameter is the name of the experimental parameter that is being constrained. Everything after *parameter* is optional and if none of the optional items were specified, this statement would be unnecessary. Each of the four optional parts can be independently used, mix-and-matched, to get the desired constraint. The following paragraphs describe each of these options.

Bounds Constraint

Bounds sets a minimum and/or a maximum value for the specified parameter. The author of the Automation Script would write this using the following syntax:

```
comparator literal-or-identifier
```

comparator is one of the comparison operators: `<`, `<=`, `>`, or `>=`. The author of the script can use this as many times as desired, but really only one of the two less-than comparators and/or one of the two greater-than comparators are necessary. So, it could be written:

```
CONSTRAIN x_sweep > 0[ppm] <= sweep_limit;
```

where `sweep_width` could have been defined with the statement:

```
CONST sweep_limit = 10[ppm];
```

This statement would ensure that the parameter `x_sweep` was greater than `0 [ppm]` and less than or equal to `10 [ppm]`. A variable can also be used in place of a literal value as shown in the example with the constant `sweep_limit`.

Multiplicity Constraint

Another option to the CONSTRAIN statement is the *multiple* clause. This ensures that the value is a multiple of or is divisible by a specified number. The *multiple* clause can be written in two different ways:

```
MULTIPLE OF positive-number mode;  
or  
DIVISIBLE BY positive-number mode;
```

Positive-number is not necessarily a whole number but it must be greater than zero and the optional *mode* is either the keyword INCREASE or the keyword DECREASE. The default is DECREASE if the *mode* is not specified. Specifying 1 for *positive-number* would simply cause the result to be an integer. Note that the sign of the value being constrained is not affected by this clause. For example, INCREASE always finds the next value with higher magnitude (away from zero) and DECREASE will find the next value with lower magnitude (towards zero).

Here are some examples using a parameter named *step* that has the value 14:

```
CONSTRAIN step MULTIPLE OF 5 INCREASE      --step results to 15  
CONSTRAIN step MULTIPLE OF 5 DECREASE     --step results to 10  
  
CONSTRAIN step DIVISIBLE BY 3 INCREASE     --step results to 15  
CONSTRAIN step DIVISIBLE BY 3 DECREASE     --step results to 12
```

Modulus Constraint

The *modulo* clause ensures that the absolute value is greater than or equal to 0 (zero) and is less than the specified *positive-number* by taking the modulus of the value. A whole number is not required here. Examples using the same parameter as above with a value of 14 are:

```
CONSTRAIN step MODULO 12      --step results to 2  
CONSTRAIN step MODULO 8       --step results to 6  
CONSTRAIN step MODULO 7       --step results to 0 (zero)
```

Units Constraint

The last clause affecting a parameter is the *units* clause. The *units* clause forces the value to have the proper units with the value or no units at all. To ensure that a value does not have a unit, use the following:

```
CONSTRAIN step WITH NO UNIT
```

The syntax to use in order to ensure that the value has a unit and that it is correct is:

```
CONSTRAIN step WITH UNIT unit-literal FROM data-variable
```

where *unit-literal* is a valid unit from the ‘Table of Units’, following the ‘Statements’ section. For example, to force a value to be in Hertz the author of the Script could write:

```
CONSTRAIN param WITH UNIT [Hz];
```

When the optional FROM keyword is given, the unit of the current value will be converted to the new *unit-literal* using information from the data file specified by *data-variable*. This is required to convert between Hertz and PPM.

Concealing Experiments

Experiments are normally accessible by the user to adjust its parameters. However, it is possible for the author to hide an Experiment from users. If the CONCEAL keyword is specified before the initial EXPERIMENT keyword then the Experiment will not be accessible by the user and thus not available to be directly manipulated in a Job.

Preventing Automatic Download and Display

Data will normally be downloaded to the local workstation and optionally shown in a Processor window when acquisition completes if the options are enabled in the Master Console's Connections menu. The QUIET keyword can be used in front of the EXPERIMENT statement to prevent the data from being automatically downloaded to the user's workstation. This may be desired for data that is collected that is not useful in and of itself other than for determining proper initialization for a subsequent acquisition.

Examples

The simplest example of the use of the EXPERIMENT block is:

```
EXPERIMENT Proton IS
  COLLECT "single_pulse";
END EXPERIMENT;
```

The next example assumes that a scout experiment file is acquired and that two variables named scans_var and min_sweep exist and are both of the NUMBER Type.

```
EXPOSE VAR scans_var : NUMBER = 16, HELP "Number of scans";
EXPOSE VAR min_sweep : NUMBER = 2[ppm], HELP "Minimum sweep width";
...
```

```
INTERIM SCOUT EXPERIMENT PreScout IS
  COLLECT "single_pulse_short";
END EXPERIMENT;
```

```
EXPERIMENT Proton IS
  SAVE AS "s$(SAMPLE.ID)_book$(SAMPLE.NOTEBOOK)";
  COLLECT "single_pulse";
  SET
    scans      = scans_var;
    x_sweep    = PreScout("x_sweep_clipped");
    x_offset    = PreScout("x_offset");
  CONSTRAIN
    scans > 0 MULTIPLE OF 8 INCREASE WITH NO UNIT;
    x_sweep >= min_sweep <= 10[ppm];
END EXPERIMENT Proton;
```

FINISH

The FINISH statement ends the execution of the current Method. Any statements that come after this statement will not be executed. The FINISH statement has the following syntax:

```
FINISH WHEN expression;
```

Any file(s) that had been created (acquired data, processed files, and printed files) by the Method up to this statement will be preserved. The TERMINATE statement (described later) has a similar function but will remove all files generated by the Method and stop the entire job.

The When Clause

If the WHEN clause is specified, the statement will only be executed if the *expression* evaluates to a True value. Refer to the section titled 'Boolean Expressions' for how an expression evaluates to a Boolean value.

The *expression* can also be written to obtain a Job attribute, a Sample attribute, or a Namespace value. If the value obtained is not one of the False values, the Method will end.

```
FINISH WHEN JOB attribute expression;  
FINISH WHEN SAMPLE attribute expression;  
FINISH WHEN NAMESPACE path expression;
```

Attribute and *path* must be a quoted String literal value or a variable of the TEXT Type. If *expression* is included, it must be written such that the value of the Job, Sample, or Namespace part is the first value of the conditional test. For example, SAMPLE "count" > 0. If an *expression* is not provided, then the value of the *attribute* or Namespace *path* alone will determine the truth state.

Examples

The following two examples of the FINISH statement produce the same result and both assume that a variable named `quit_early` has been previously defined and set appropriately.

```
VAR quit_early : BOOLEAN = FALSE;  
...  
  
IF quit_early THEN  
|   FINISH;  
END IF;
```

The following line has the same effect as the IF block in the example above.

```
| FINISH WHEN quit_early;
```

The following examples will end the Method when a Sample contains an attribute named "complete" that evaluates to a True value or when the Sample "count" attribute is greater than 10.

```
| FINSIH WHEN SAMPLE "complete";  
| FINISH WHEN SAMPLE "count" > 10;
```

GROUP

The GROUP block prevents the statements contained within it from being interrupted by a user or by the system. Like the way that a graphical design program can group multiple elements together to be treated as a single entity, the statements within the GROUP block are treated as a single uninterruptible step in Automation.

The GROUP block has the following syntax:

```
GROUP
    statement-block
END GROUP;
```

NOTE: It is recommended that the author of the Automation Script use the GROUP block sparingly. It should be used only in cases where it is *essential* that the operations within the group run completely to the end.

Example

This example will print all the factorials from 1 to 100 preventing the user from interrupting the calculations. The use of the GROUP block here is not essential – it is for illustrative purpose only.

```
GROUP
  VAR f : NUMBER;

  REPEAT x TO 100 DO
    SET f = CALL factorial( x );
    INFORM "Factorial of $(x) is $(f)";
  END REPEAT;
END GROUP;
```

IF

The IF statement provides the capability to conditionally execute of a sequence of statements. An IF statement decides which block of statements to execute (if any) at the time it is encountered. Its decision can be based on the state of the Automation system, local Script variables (which include Method parameters), parameters within data files, Namespace values, Job or Sample attributes, and/or the results of a call to an external operator written in JEOL's proprietary Percival programming language.

An IF statement must contain at least one sequence of statements known as a statement block. An expression is evaluated to determine whether a statement block will be executed. If the evaluation of the expression results in a True value, the statement block immediately following the expression is executed. Otherwise, the statement block is skipped and the next test (if any) is evaluated. Refer to the section titled 'Boolean Expressions' for how an expression evaluates to a Boolean value.

An IF statement starts its decision-making process by evaluating the first expression. If the result of the first expression is a False value, it then continues, in order, through each of the expressions in the IF statement until it encounters an expression that evaluates to a True value. All remaining expressions that come after the statement-block that executes will not be evaluated.

In its simplest form, the IF statement has the following syntax:

```
IF expression THEN
    statement-block
END IF;
```

The *expression* can also begin with or be replaced by a Job attribute, a Sample attribute, or a Namespace value.

```
IF JOB attribute expression THEN...
IF SAMPLE attribute expression THEN...
IF NAMESPACE path expression THEN...
```

Attribute and *path* must be a quoted String literal value or a variable of the TEXT Type. If *expression* is included, it must be written such that the value of the Job, Sample, or Namespace part is the first value of the conditional test. For example, SAMPLE "count" > 0. If an *expression* is not provided, then the value of the *attribute* or Namespace *path* alone will determine the truth state.

Compound IF Blocks

It is possible to provide additional expressions and statement blocks using the ELSE IF construct as shown below.

```
IF expression-1 THEN
    statement-block-1
ELSE IF expression-2 THEN
    statement-block-2
END IF;
```

The ELSE IF block may be repeated as many times as necessary for each condition that a solution requires.

What if no Expressions Match?

The last statement block of an IF statement can be executed if none of the prior expressions result in a True value. This is accomplished by using the ELSE keyword **without** the IF keyword immediately following **on the same line**. Basically, this informs Automation that if none of the prior statement blocks are executed, then execute the last one. The following syntax shows this:

```
IF expression THEN
    statement-block-1
ELSE
    statement-block-2
END IF;
```

In the previous syntax example, *statement-block-1* will execute when the *expression* evaluates to a True value, otherwise *statement-block-2* will execute.

Any number of expressions may be provided with statement blocks in an IF statement. The ELSE block must always be last – just before the END IF.

Nesting IF Statements

It is possible for the author of the Automation Script to embed an IF statement within other IF statements. This is called statement nesting. Here we encounter the one place where white space is important to the semantics of the statement. As noted previously, ELSE IF together on the same line of the Script provides an additional expression and statement block for the IF statement. However, when the IF keyword following an ELSE is **not** written on the same line of the Script, it is assumed to begin a new IF statement. Here is an example:

```
IF expression-1 THEN
    statement-block-1
ELSE
    IF expression-2 THEN
        statement-block-2
    END IF;
END IF;
```

This form is rather silly since it can be written more simply using the second syntax example above (which is functionally equivalent). It does, however, illustrate the point that when the IF keyword follows the ELSE keyword on different line of the text, a new IF statement is encountered. This means that additional END IF keywords, to close the each inner IF statement, are required.

IMPORTANT: Always follow the keyword ELSE with the keyword IF on the **same line** of text if you wish to add a test to the IF statement. Write the keyword IF on a new line if you wish to begin a new IF statement.

Examples

These examples of the IF statement refer to the RGB_Color enumeration example from the previous section describing the ENUM statement. Suppose that our script has the following variables defined:


```
VAR ready : BOOLEAN = FALSE;
VAR c      : RGB_Color = "red";
```

The following example will print the text “I’m Ready” if the `ready` variable is TRUE. If the `ready` variable is not TRUE then the text “I’m Waiting” will be printed.

```
IF ready THEN
    INFORM "I'm Ready";
ELSE
    INFORM "I'm Waiting";
END IF;
```

The next example will change the value of the variable `c` to the next color in the `RGB_Color` enumeration. Before the IF statement, the value of the variable `c` represents the current color. If the current color is “red”, then set the current color to “green”, but if the current color is “green”, set the current color to “blue”. If the current color is neither “red” nor “green”, then it must be “blue” (because that is the only other legal value for the variable `c`) so set the current color back to “red”.

```
IF c = "red" THEN
    SET c = "green";
ELSE IF c = "green" THEN
    SET c = "blue";
ELSE
    SET c = "red";
END IF;
```

The following IF statement will set the value of the variable `ready` to FALSE when the current value of `ready` is TRUE and the value of the variable `c` is “red”. If the current value of `ready` is not TRUE, the first nested IF statement will not be executed and testing whether the value of `c` equals “red” will not occur. This IF statement will also set `ready` to TRUE when the current value of `ready` is not TRUE and the value of `c` is not “red”.

```
IF ready THEN
    IF c = "red" THEN
        SET ready = FALSE;
    END IF;
ELSE
    IF c /= "red" THEN
        SET ready = TRUE;
    END IF;
END IF;
```

The last example resets a Sample attribute if it evaluates to a True value.

```
IF SAMPLE "count" > 0 THEN
    SET SAMPLE SAVE "count" = 0;
END IF;
```

INCLUDE

The INCLUDE statement provides access to the Methods and Types that are in a separate Automation Script file.

Using External Automation Script Files

The INCLUDE statement has the following syntax to include an external script:

```
INCLUDE "script-filename" TO DOMAIN domain-name;
```

Script-filename is the relative path or file URL that identifies the external Script file. The *script-filename* must be between double quotation marks. Automation Script filenames typically end with the .jaf extension, but it is neither necessary nor recommended to include the file extension in the *script-filename*.

If the *script-filename* is a valid identifier (that is, the name of the file begins with a letter and contains only letters, digits, and/or underscore characters) then the TO DOMAIN clause may be omitted and the *domain-name* of the included script becomes the *script-filename* itself (without the quotation marks, of course).

Identifying Methods and Types

Domain-name is an identifier that is used to distinguish a Method that is within the included file from another Method with the same name. Domain names are necessary to distinguish Methods if there exists more than one Method with the same title from separate files. An example is:

my_automation_file.jaf contains:

```
ENUM Color is ( "red", "green", "blue" );

METHOD Proton ( IN a : Color = "red" ) IS
...
END METHOD;
```

Executing the Proton Method in the above file would be accomplished with the following:

```
| INCLUDE "my_automation_file" TO DOMAIN otherScript;
...
VAR c : otherScript.Color = "blue";

INVOKE otherScript.Proton( c );
```

Notice the added *domain-name*, "otherScript.", to the front of the Color sub-Type and to the Method title that we invoke from the included Script. If we had omitted the domain in the INVOKE statement, it would have attempted to invoke a Method named Proton within the same file and **not** the Proton Method that had been included from the 'my_automation_file' script.

Inclusion Functionality

An Automation Script that is included into another Script is added as if the separate file was in the main Script at the point of the INCLUDE statement. This means that an INVOKE statement of an included Script could invoke a Method from the main Script. An example follows:

main.jaf contains:

```
METHOD Operate IS
...
END METHOD Operate;

INCLUDE "my_separate_script" TO DOMAIN example;

METHOD Main IS
    INVOKE example.Separate();
END METHOD Main;
```

my_separate_script.jaf contains:

```
METHOD Separate IS
...
    INVOKE Operate();
...
END METHOD Separate;
```

Method Assertions

It is important to realize that the file `my_separate_script.jaf` could not be parsed properly on its own since there is no Method with the title "Operate" that exists in that Script. To avoid this problem, the author must include a Method assertion statement prior to the point of invocation. This is accomplished by following the Method title with a semicolon and optionally including the ASSERT keyword before the METHOD keyword. This causes the otherwise problematic Script to load correctly on its own by claiming that a Method with the specified title will exist at a later point, and in this case, will exist in a different file. Refer to the description of the METHOD statement for more information about Method assertions.

Therefore, after inserting the Method assertion, the file `my_separate_script.jaf` should contain:

my_separate_script.jaf contains:

```
ASSERT METHOD Operate;

METHOD Separate IS
...
    INVOKE Operate();
...
END METHOD Separate;
```

Using this technique, the author of the Script can create a Method, like 'Main' in the above example that invokes another Method from a separately included Script file. The included Method, called 'Separate' in the example, then invokes the Method 'Operate' from the main Script as part of the statements that it performs. Thus, Automation Scripts can include other Scripts that have Methods defined that allow part of their job to be defined by the including Script.

INFORM

The INFORM statement can be used to send a text message to Delta's console, display a message in a dialog window, or to log a message in the Job Queue log file. The syntax of the INFORM statement is:

```
INFORM warning-level WITH DATE AND TIME
      TO destination AND destination message;
```

The only item required after the INFORM keyword is the *message* text. The *message* can be a single String or a series of Strings that are separated by commas. The author can specify a translated message by using a single language translation identifier. A single variable name can also be used in the place of *message* that will show the value of that variable.

How important is the message?

The *warning-level* is an optional indicator to provide a level of concern about the message. The six allowable warning levels (in the order of seriousness) are: INFO, STATUS, WARNING, ALERT, ERROR, and FATAL. It is up to the receiver of the message to decide how to handle the warning level.

When was the message displayed?

If the WITH DATE clause is specified, the current date is added to the message. If the AND TIME clause is also provided, the time is added to the message along with the date. The time can only be included if the date is included.

Where will the message be displayed?

Destination is where the *message* from the INFORM statement will appear. There are three places to potentially send a message: Delta's console, a dialog window, or the Job queue log file. These three places are specified using CONSOLE, DIALOG, and LOG respectively. For *destination*, use the keyword TO and follow it with one of the three destination keywords. A message can be sent to more than one destination simultaneously by separating the destination keywords with the AND keyword. Specifying the *destination* is optional. When it is omitted, the default is to display the message on Delta's console.

Using Special symbols

There are many special symbols that may appear within the *message* text that will be replaced before it is displayed. The special symbols are formed by placing a word within parentheses and preceding it with a '\$' character. The casing of the letters of the word within parentheses is insignificant. The author may also include a constant, variable, or parameter defined in the Automation Script into the message. See the 'Table of Substitute Identifiers' after the 'Statements' section for the list of recognized words that can be used in the *message* text.

Examples

Below are some examples of how the INFORM statement can be used. The first is a basic message that will appear on Delta's console.

```
| INFORM "Beginning job";
```

The following statement displays an alert level two-line message on Delta's console.

```
| INFORM ALERT "No value provided", "Using default";
```

Below, an identifier named `count` is used within the message. The `$(count)` notation will be replaced by the value of the variable named `count` when it is displayed. The date will be added to the beginning of the message as well.

```
| INFORM INFO WITH DATE "Job counter = $(count)";
```

The next statement displays a fatal message with both the date and time to Delta's console.

```
| INFORM FATAL WITH DATE AND TIME "Experiment Error!";
```

This example will display a simple message in a dialog window.

```
| INFORM TO DIALOG "Hello Dave.";
```

This example will write a status message with the current date to both Delta's console and the Job queue log file.

```
| INFORM STATUS WITH DATE TO CONSOLE AND LOG "Job Complete";
```

The two INFORM lines in the next example will produce the identical results of printing out the contents of a variable named `count`.

```
VAR count : NUMBER = 9;
```

```
| INFORM count;  
| INFORM "$(count)";
```

The following example shows how a language translation can be used.

```
TRANSLATE check_data ES "Compruebe por favor sus datos";  
"Please check your data";
```

```
| INFORM ALERT check_data;
```

INVOKE

The INVOKE statement suspends the execution of the current Method (which for the following explanation will be called Method A) and executes another Method (which will be called Method B). Information may be passed from Method A to Method B through the parameter interface defined by Method B. This is known as passing arguments to the invoked Method. When Method B has completed, Method A resumes executing the statement that follows the invocation of Method B.

The syntax for invoking a Method is:

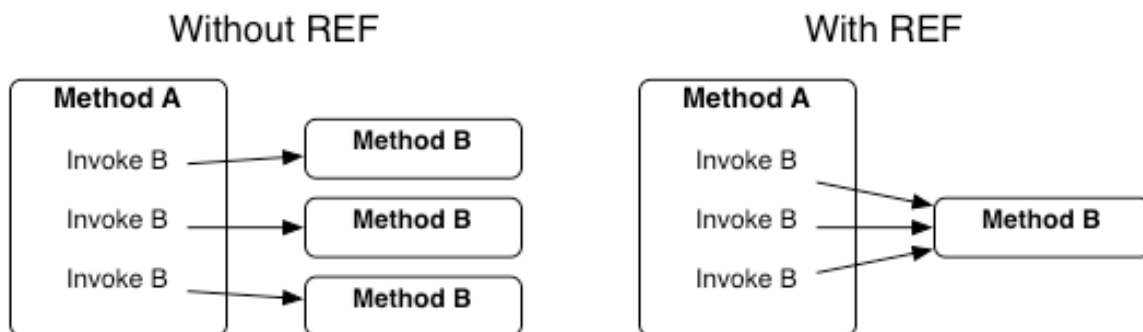
```
INVOKE REF domain-name.method-title( argument-list );
```

Method-title is the title of the Method being invoked. Remember that the title can be specified with or without quotation marks. Refer to the description of the METHOD statement for the rules pertaining to how the title of a Method can be specified. The *domain-name* before the *method-title* is required if the Method was made available from a separate file with the INCLUDE statement. See the prior description of the INCLUDE statement.

Invoking a Method by Reference

Normally when a Method is invoked, it is as if the Method was coded in the script at the point of the invocation. All statements in the Method are duplicated. This provides the user the ability to modify experimental parameters for distinct EXPERIMENT blocks that are within the invoked Method. If the invoked Method contained a single EXPERIMENT block and was invoked three times without the REF keyword, the user would see three distinct experiments – one for each invocation of the Method.

The REF keyword allows the author of the Script to treat the invoked Method as a single instance. The invoked Method reference still runs like a duplicated Method, but the difference is that any EXPERIMENT blocks within a referenced Method invocation are treated as the same experiment. If the invoked Method contained a single EXPERIMENT block and was invoked three times with the REF keyword, the user would see only one experiment – it is the same experiment invoked three times through the Method. It is recommended that the author always use the REF keyword when invoking a Method that does not contain an EXPERIMENT block multiple times.



Passing Data Through Parameters

The optional *argument-list* allows information to be sent to the Method. The values must be placed between parentheses and separated by commas. The number of the arguments and the Types of the values in the argument list are stipulated by the Method that is being invoked. There is no limit to the number of parameters that a Method may have. Some examples of how Methods (with silly names) are invoked:

```
INVOKE method_with_no_params();
INVOKE external.method_with_one_param( 1 );
INVOKE "method with three params"( 1, x, TRUE );
```

Notice that the parentheses are required even if there are no arguments. Also notice that the third invocation in the above examples specified the Method title with quotation marks and is passed a value to that Method using a variable named `x` in the second position. Values can be passed as literal constants or by using a variable name.

Arguments may be given using the names of the parameters of the Method being invoked. When arguments are specified with names, then the order in which they are specified is not important and any prior arguments may be left unspecified without using `NULL` as a placeholder. The name of the argument determines the ultimate position of that argument when the Method is invoked. To name an argument, specify the name of the Method parameter to receive the value followed by the `=>` (map to) symbol and then the value as usual.

```
METHOD meth( IN first : NUMBER = 0; IN second : NUMBER = 0 ) IS..
```

```
INVOKE meth( 1, 2 );
INVOKE meth( first => 1, second => 2 ); --same behavior as first line
INVOKE meth( second => 2, first => 1 ); --same behavior as first two lines
INVOKE meth( second => 2 );
INVOKE meth( NULL, 2 ); --same behavior as third line
```

The order and position of unnamed arguments is important. The value of each unnamed argument will be assigned to the corresponding Method parameter in the same position. If there are unnamed arguments, then those must be provided in the proper order before any named arguments. Any named arguments may follow in arbitrary order.

NOTE: It is important to remember that if the mode of a Method parameter is specified as `OUT` or as `INOUT` then a variable must be used to transmit information through that parameter because variables are the only means of returning values.

When `NULL` is used as an argument or when parameters are not specified, the default value provided for that Method parameter is used. Each parameter's default value is given where the Method is defined.

Method Scoping and Visibility

A Method may only invoke another Method if it has been defined prior to the point of its invocation. A Method that can be invoked at a specific location is said to be “in scope” at that location. There are four places where a Method could be defined so that it will be “in scope”.

- Outside of and prior to the current Method block:

```
METHOD B IS
...
END METHOD B;
...
```

```

METHOD A IS
...
  INVOKE b();
...
END METHOD A;

```

- Inside the current Method block and prior to the invocation:

```

METHOD A IS
...
  METHOD B IS
    ...
  END METHOD B;
...
  INVOKE b();
...
END METHOD A;

```

- Within an Automation Script file accessed via an INCLUDE statement prior to the invocation.

```

METHOD A IS
...
  REMARK contains a Method B
  INCLUDE "script_file" TO DOMAIN local;
...
  INVOKE local.b();
...
END METHOD A;

```

- Invoking the current Method itself. This is known as “recursion”. Be careful to always provide a mechanism by which the recursion can end!

```

METHOD Factorial( IN      n      : NUMBER = 1,
                  INOUT result : NUMBER ) IS

  REMARK the 'if' provides the terminal case

  IF n >= 2 then
    SET result = result * n;
    SET n      = n - 1;

    INVOKE REF factorial( n, result );
  END IF;

END METHOD Factorial;

...
VAR f : NUMBER = 1;

INVOKE REF factorial( 5, f );

INFORM TO CONSOLE "Factorial of 5 is $(f)";

```


LIMIT

The LIMIT block restricts the statements contained within it to run within a specified amount of time. The syntax of the LIMIT block is:

```
LIMIT TIME maximum-duration DO
    statement-block
EXPIRED
    expired-statement-block
END LIMIT;
```

The statements within *statement-block* will begin executing immediately. If all of the statements in the *statement-block* complete within the time limit specified by *maximum-duration* the statement after the LIMIT block will be executed next. If, however, the time limit expires before every statement in the block can successfully complete, the remaining statements will not be executed.

Specifying the Time Limit

Refer to the section titled ‘Duration Syntax’ which describes how to provide a time limit for the *maximum-duration*.

What Happens When the Time Limit Expires

When the time limit expires, the currently executing statement will complete but the remaining statements will not be executed. Then, if an optional EXPIRED block is provided, the statements in the *expired-statement-block* will execute before continuing with the statement following the LIMIT block.

Examples

A silly example to discover how many times it is possible to increment a number within a short amount of time.

```
REMARK How high can we count in 10 seconds?
```

```
VAR count : NUMBER = 0;
```

```
LIMIT TIME 10 SECONDS DO
    REPEAT
        SET count = count + 1;
    END REPEAT;
EXPIRED
    INFORM "Count reached $(count) in one minute";
END LIMIT;
```

LIST

A variable of the LIST Type may contain zero or more of values of any Type. The LIST statement creates a sub-Type of the basic LIST Type that will enforce that the values within a variable of the sub-Type conform to the specified property. The syntax to create a Homogeneous LIST sub-Type (all values in the List must be of the same Type) is:

```
LIST type-name IS OF base-type;
```

The author may also create a “key-value” pair (known as an “association”) LIST sub-Type using the following syntax:

```
LIST type-name IS ASSOCIATION;
```

This sub-Type enforces the rules that:

1. Each element of a value of this sub-Type must be of the LIST Type and must also contain two pieces of information.
2. The first item in each element, known as the “key”, must be a String.
3. The Type and the value of the second item in each element, known as the “value”, is not restricted.

Type-name is an identifier that will be the name of this sub-Type. Any statement after this Type definition can use the *type-name* in any place where one of the basic Types can be used.

LIST Type Relationship

Note that a value of a LIST sub-Type may be assigned to any variable of the basic LIST Type. However, a value of a basic LIST Type may only be assigned to a variable of another LIST sub-Type if *all* of the values of the list are of the same Type (it is homogeneous) and the Type of the values match the sub-Type.

Examples

An example of a LIST statement that defines a sub-Type named `Numbers` is:

```
LIST Numbers IS OF NUMBER;

LIST Assoc IS ASSOCIATION;

VAR L : LIST;
VAR N : Numbers;
VAR A : Assoc;

SET N = {};                --okay
SET N = {1, 2};            --okay
SET N = {"red"};           --error   (String not allowed)
SET N = {1, "green"};     --error   (String not allowed)

SET L = N;                 --okay
```

```
SET L = {2, 4, 8};
SET N = L;                                --okay

SET L = {2, TRUE, "blue"};
SET N = L;                                --error   (Boolean and String not allowed)

SET A = {{ "one", 1}, {"color", "red"}, {"hot", TRUE}};

SET L = A;                                --okay
SET A = N;                                --error   (List required for each element)
SET N = A;                                --error   (Number required for each element)
```

MACRO

A macro definition defines a sequence of Automation statements that can be inserted into the Automation script with the macro expansion syntax. The text of the macro will be parsed as if it was written directly in the script at the point of the macro expansion.

The syntax to define a macro is:

```
MACRO macro-name ( macro-parameters ) IS
    statement-block
END MACRO macro-name;
```

and the syntax to expand a macro is:

```
EXPAND MACRO macro-name ( argument-list );
```

Description

The *macro-name* is a title that must be distinct within its scope level of the Automation Script and it identifies the macro. A macro expansion statement references the *macro-name* to specify the sequence of statements that should be expanded. The body of the macro definition, the *statement-block*, is treated as a single String and is not parsed until it is expanded.

Macros can be defined anywhere in the script file and within any scope level except within other macros. Macros can be expanded within a Method scope level and deeper. Macros cannot be expanded within other macros.

Parameters & Arguments

Parameters are optional and can be used to infuse changes into the *statement-block*. *Macro-parameters* is one or more identifiers separated by commas where each identifier may be optionally followed by an equal sign and a default constant value. Unlike Method parameters, these parameters do not require Type definitions because they are only used to identify areas where textual replacements are to occur. For example, if there were a parameter named 'id' then anywhere within the *statement-block* where \$(id) is found would be replaced by the value of that parameter.

Parameters are assigned values by the *argument-list* in the macro expansion. The number of arguments must match the number of parameters of the macro definition if defaults are not specified and each parameter is assigned the value of the argument that is in the same ordered position in the list. Each argument can be a literal String or a constant value that will be converted to a TEXT Type. A NULL can also be used as an argument. When a NULL is used and a default value is provided for the parameter in the same position, the default value will be substituted for the parameter at that position. NULLs do not need to be provided if they are the last arguments of the Macro.

Textual replacements in the *statement-block* are performed anywhere that one of the parameter names is detected surrounded by parentheses and preceded by a dollar sign. Any replacement indicator that does not contain the name of one of the parameters will not be removed.

Examples

The simplest example of the MACRO statement is without parameters. In this example, the entire line of `EXPAND MACRO mac;` will be replaced with the macro body which is the one line `INFORM "Hello";`. Running the Method named `Macro_Test1` will send the message "Hello!" to the user.

```
MACRO mac IS
    INFORM "Hello!";
END MACRO mac;

METHOD Macro_Test1 IS
    EXPAND MACRO mac;
END METHOD Macro_Test1;
```

The following example adds an exposed variable for the user to provide a value in the Method parameters area. Since there are no macro parameters, the text \$(your_name) is not altered and the two lines of the macro will replace the single macro expansion line in the Method named Macro_Test2 as-is. If the user supplies “Bill” as the value for your_name, then running the Method named Macro_Test2 will send the message “Hello, Bill!” to the user.

```
MACRO mac_var IS
    EXPOSE VAR your_name : TEXT = "", HELP "Enter your name";
    INFORM "Hello, $(your_name)!";
END MACRO mac_var;

METHOD Macro_Test2 IS
    EXPAND MACRO mac_var;
END METHOD Macro_Test2;
```

The next example includes a single parameter named uname. The String “Bill” is passed as an argument of the first macro expansion which will become the value of the parameter uname. Thus, the text \$(uname) will be replaced by the name “Bill”. The third expansion shows how the default value of the parameter is selected by specifying NULL for the argument. The result of calling Macro_Test3 is the three messages “Hello, Bill!”, “Hello, Steve!”, and “Hello, Fred!” being sent to the user.

```
MACRO mac_param( uname = "Fred" ) IS
    INFORM "Hello, $(uname)!";
END MACRO mac_param;

METHOD Macro_Test3 IS
    EXPAND MACRO mac_param( "Bill" );
    EXPAND MACRO mac_param( "Steve" );
    EXPAND MACRO mac_param( NULL );
END METHOD Macro_Test3;
```

Macro_Test4 below is very similar to Macro_Test3. You will notice that the macro definition is identical except that the macro title was changed to help describe its purpose. In this case, we have provided an exposed variable within the Method for the user to provide a value. For the argument, we provide the text that will become new textual replacement indicator that will then be used when the INFORM statement executes. Thus, \$(uname) in the macro becomes \$(your_name) when the macro is expanded and placed within the Method. When the INFORM statement finally executes, it will use the expanded macro content “Hello, \$(your_name)!”. The system knows the value of your_name and replaces it with the value that the user provided for the exposed variable. If the

user provides “Bill” as the value of `your_name`, then running the Method named `Macro_Test4` will send the message “Hello, Bill!” to the user.

```
MACRO mac_indirect_param( uname ) IS
    INFORM "Hello, $(uname)!";
END MACRO mac_indirect_param;

METHOD Macro_Test4 IS
    EXPOSE VAR your_name : TEXT = "", HELP "Enter your name";
    EXPAND MACRO mac_indirect_param( "$(your_name)" );
END METHOD Macro_Test4;
```

The next example shows that macros can also be placed within Methods. This example is very similar to `Macro_Test4` above. It also adds a second parameter, `uage`, and provides an exposed variable named `age`. If the user provides “Bill” and 42 for the values of `your_name` and `age` respectively, then running the Method named `Macro_Test5` will send the message “Hello, Bill! Are you really 42?” to the user.

```
METHOD Macro_Test5 IS
    MACRO mac_inner( uname, uage ) IS
        INFORM "Hello, $(uname)! Are you really $(uage)?";
    END MACRO mac_inner;

    EXPOSE VAR your_name : TEXT = "", HELP "Enter your name";
    EXPOSE VAR age : NUMBER = 21, HELP "Enter your age";
    EXPAND MACRO mac_inner( "$(your_name)", "$(age)" );
END METHOD Macro_Test5;
```

The following example builds upon the prior example by adding a Method parameter named `probeId` along with a constant definition whose value is provided as the first argument to the macro. During the macro expansion, the value of the variable `notice` (which is “Hello”) replaces `$(pre)` in the macro body and the text “\$(your_name)” replaces `$(uname)` in the macro body. `$(probeId)` is not replaced because there is no macro parameter with that name. Again, if the user supplies “Bill” for the value of `your_name` and leaves the value of `probeId` as “ABC”, then running the Method named `Macro_Test6` will send the message “Hello, Bill! Probe = ABC.”

```
METHOD Macro_Test6( IN probeId : TEXT = "ABC" ) IS
    MACRO mac_inner( pre = "Hi", uname ) IS
        INFORM "$(pre), $(uname)! Probe = $(probeId).";
    END MACRO mac_inner;

    EXPOSE VAR your_name : TEXT = "", HELP "Enter your name";
    CONST notice : TEXT = "Hello";
    EXPAND MACRO mac_inner( notice, "$(your_name)" );
END METHOD Macro_Test6;
```

METHOD

The METHOD block is the basis for performing any task with Automation. The syntax of the METHOD block is:

```

CONCEAL INTERACTIVE METHOD method-name ( method-parameters )
  WHEN event IS
    CATEGORY categories;
    HELP help-text;
    PURPOSE help-text;
    PARAMETER param-name param-options;
    DURATION time-expression;
    statement-block
  END METHOD method-name;

```

Description

The *method-name* is a title that must be distinct within its scope level of the Automation Script so that it identifies a single Method. A *Method-name* may be either an identifier or a String that contains at least one character. If the *method-name* is specified as a String (that is, it is surrounded by quotation marks), then there are no restrictions to the title other than the minimum length. If the *method-name* is specified as an identifier, then it must conform to the rules of an identifier. See the description of identifiers in the previous section titled 'Definition of Terms'.

The *method-name* is required after the initial METHOD keyword but it may be omitted after the END METHOD. If it is specified at the end, then it must match the name and form (String or identifier) that is used at the beginning of the Method block.

NOTE: Although legal, it is recommended that the author abstain from using a period character, '.', in a Method title to avoid confusion. A period character is the domain separator. Refer to the descriptions of the INCLUDE and INVOKE statements for more information about Method domains.

Parameters

Parameters are one of the ways by which external information is transmitted to a Method and are the only way that Methods can emit information and thus they are how a Method is able to propagate information to subsequent Methods. See the INVOKE statement for details about executing other Methods from within a Method. If a Method does not need to convey information, then the *method-parameters* and parentheses surrounding them may be omitted. There is no limit to the number of parameters that a Method may have.

If more than one Method parameter is required, each parameter must be separated by a semicolon. Method parameters are provided using one of the following syntax options:

```

CONCEAL IN   param-name : value-type = default-value param-options
CONCEAL INOUT param-name : value-type = default-value param-options
CONCEAL OUT  param-name : value-type help

```

A parameter definition may optionally begin with the keyword CONCEAL to hide the parameter from the user. A parameter marked with the keyword CONCEAL will not be visible in the Method attributes area on the Job tab of the *Spectrometer Control* window.

Each parameter must have a mode (IN, INOUT, or OUT), a name specified by *param-name*, and a Type specified by *value-type*. The name is used to reference the value of the parameter inside the Method block and can be any word except a reserved word or the name of an instrument parameter. The mode specifies the direction the information flows through the parameter.

- IN mode parameters allow data to be transferred into the Method. The value of an IN mode parameter may change during the execution of the Method but the modified value will not be preserved when the Method ends.
- OUT mode parameters allow data to be transferred out of the Method. This mode does not initialize the parameter with a value when the Method begins execution. Instead, a calculated result should be assigned to this parameter that is then provided to the calling Method when the current Method successfully completes.
- INOUT parameters allow data to be transferred both in and out of the Method. It accepts the value given to the parameter as its initial value as IN mode parameters do, and also passes the potentially modified value back to the caller upon successful completion of the Method as OUT parameters do.

Parameters are like variables that can be defined within a Method using the VAR statement. Refer to the description of the VAR statement. A parameter's name can be used in the same places where variables are used. The subtle difference between a parameter and a variable is that the *caller* provides the initial values for parameters whereas the *Method* predetermines the initial values for variables. The values of variables are lost when the Method ends unlike INOUT and OUT parameters. The caller is defined to be the Method containing the INVOKE statement which caused the execution of the current Method. Values passed to a Method via the INVOKE statement provide the initial value for the parameters.

Value-type may be any of the five basic Types defined in the Automation grammar (BOOLEAN, NUMBER, TEXT, LIST, or DATA) or it may be a sub-Type defined prior to the Method using the ENUM, NUMBER, or LIST statements.

Passive Parameter Mode

Regular (not concealed) IN and INOUT parameters have an additional option. When the value of a parameter is changed, it may have an impact on the run-time duration of the Method and so the duration time will be re-computed. If it is known that a parameter *will not* have an impact on the duration then the parameter can be defined as a 'passive' mode parameter by inserting the PASSIVE keyword before the IN or INOUT keyword. This will cause the time calculation to be skipped for this parameter and it will also make the interface more responsive. By default, without the PASSIVE keyword specified, parameters are defined with a mode set to 'active'. However, the author of the script may choose to add the ACTIVE keyword before the IN or INOUT keyword even though it is neither required nor necessary.

Default Parameter Values

IN and INOUT parameter modes require a default value to be provided except when the specified *value-type* is DATA. The *default-value* can be given in one of the following ways:

```
constant
JOB job-attribute ELSE constant
SAMPLE sample-attribute ELSE constant
NAMESPACE namespace-path ELSE constant
EVALUATE (expression) ELSE constant
```


Parameters of the DATA Type cannot be initialized with the JOB, SAMPLE, NAMESPACE , or EVALUATE clauses. Only a constant String or an expression that results in a TEXT Type value can initialize a DATA Type parameter.

Constant is a value that must be of the Type *value-type* -- the same Type as the parameter being defined. *Job-attribute* is a String or variable representing the attribute of interest of the currently running Job. *Sample-attribute* is a String or variable representing the attribute of interest of the current Sample. *Namespace-path* is a String or variable representing the location of a value in the Namespace Parameter Database. *Expression* is a Percival code expression that will be evaluated to produce the initial value. More than one Job, Sample, Namespace, or Evaluate clause may be specified when they are separated by an ELSE keyword before the optional final ELSE clause.

The optional ELSE clause provides a specific default value for the cases when the *job-attribute*, *sample-attribute*, *namespace-path* cannot be found or when those values or the evaluation *expression* cannot be resolved to an appropriate value of the required Type. If the ELSE clause is omitted, an error condition could be raised prior to executing the Method. If an error is raised, an attempt will be made to find an error handler Method since a Method cannot define its own error handlers prior to its execution. If an appropriate error handler cannot be found that handles the error, the Job will terminate. If an error handler is found, execution resumes in the Method in which the error was handled.

Parameter Options: Dependencies

The initial value of one parameter may depend on the specified initial value(s) of one or more other parameters. In these circumstances, the author of the Automation Script can designate the relationship between the parameters using the DEPENDS clause. The DEPENDS clause follows the *default-value* with a comma and then using one of the following forms:

```
DEPENDS ON parameter-list EVALUATE (expression) ELSE constant
```

```
DEPENDS ON parameter-list ENABLE WHEN (Boolean-expression)
```

The *parameter-list* is one or more identifiers separated by commas. If a parameter is listed in the *parameter-list* it should naturally be part of the *expression* or *Boolean-expression*.

A parameter that includes a DEPENDS clause of the first form will have its initial value recalculated on the user interface whenever a parameter on which it depends is modified. The *expression* is a Percival expression that will be evaluated to determine the new initial value of the parameter. The *constant* in the optional ELSE clause will be the initial value when the *expression* does not result in a legal value for the parameter.

A simple example of a parameter dependency is:

```
( IN str : TEXT    = "" ;
  IN len : NUMBER = 0, DEPENDS ON str EVALUATE (size( str )) ;
  IN big : NUMBER = 0, DEPENDS ON len EVALUATE (10 * len) )
```

See *Parameter Options: Relevancy* below for a description of the ENABLE clause. Include the ENABLE clause after a dependency clause when the relevancy of the parameter depends on the value of one or more other parameters. In this case, the *Boolean-expression* would contain the names of the other parameter(s) on which it depends.

The two forms can be combined to make a third form which specifies a value dependency as well as a rule for when the parameter is relevant to the Method.

```

DEPENDS ON parameter-list
          EVALUATE (expression) ELSE constant,
          ENABLE WHEN (Boolean-expression)

```

NOTE: The DEPENDS clause cannot be included when the parameter or Method is concealed by using the CONCEAL keyword or when the parameter's mode is set to be OUT.

Parameter Options: Relevancy

A Method parameter is normally always relevant to the Method, but there may be a time when that is not true. A parameter that includes an ENABLE clause is said to be irrelevant to the Method when the result of the evaluation of the *Boolean-expression* is False. An irrelevant parameter will appear disabled so that the operator will be unable set or change its value in the Method attributes area on the Job tab of the *Spectrometer Control* window.

An example of when a parameter could become irrelevant is when it is only referenced in certain cases. Such a case would be a Method that optionally prints a message and allows the contents of the message to also be specified. The author could provide two parameters on the Method: one to control the printing of the message and the other to specify the message itself. The Method might be something like this:

```

METHOD Message( IN acknowledge : BOOLEAN = TRUE;
                IN content      : TEXT    = "Ok" ) IS
  IF acknowledge THEN
    INFORM content;
  END IF;
END METHOD Message;

```

Both parameters would be visible, active, and seem to have equally importance in the Method attributes area. It would be better to cause the parameter named *content* to be enabled only when the value of the parameter named *acknowledge* is set to be True. The parameter named *content* becomes irrelevant to the Method when *acknowledge* is False. The Method definition describing this relationship between its two parameters becomes:

```

( IN acknowledge : BOOLEAN = TRUE;
  IN content      : TEXT    = "Ok",
  DEPENDS ON acknowledge
  ENABLE WHEN (acknowledge = TRUE) )...

```

With the above parameter definition, when the operator sets the parameter *acknowledge* to be False in the Method attributes area, the *content* parameter will be disabled.

The DEPENDS clause at the beginning is only required when the relevancy of a parameter depends on the value of another parameter. If the relevancy of a parameter depended on the time of day, for example, the DEPENDS clause can be omitted. Assuming there exists a function, *time_of_day*, that returned the number of hours since the previous midnight, the author could create a Method with a parameter that is enabled after noon.

```
METHOD Acquire( IN fast : BOOLEAN,
                  ENABLE WHEN (time_of_day > 12) ) IS...
```

NOTE: The ENABLE clause cannot be included when the parameter or Method is concealed by using the CONCEAL keyword or when the parameter's mode is set to be OUT. The ENABLE clause also cannot be specified on its own if it had been previously specified at the end of the DEPENDS clause.

Parameter Options: Help

A description of a parameter may be provided to the operator by following the parameter syntax (including any other parameter options) with a comma, the HELP keyword, and one or more mixed Strings or translation identifiers separated by commas. Parameter HELP is not required¹ but its inclusion is strongly encouraged so that other users who might use this Method may better understand the purpose of each parameter and therefore be more apt to use the Method successfully. The text of the help will be displayed on the Automation user interface where the Method parameter input area is located. To provide help per locale, use the TRANSLATE statement (described later) and place the translation identifier(s) after the HELP keyword.

An example of providing help with a parameter is:

```
IN print_data : BOOLEAN = FALSE, HELP "Print the data?"
```

Parameter Options: Alternative Syntax

Another way to specify the options on a Method parameter is with a separate statement following any CATEGORY, HELP, and PURPOSE statements. A Method parameter may only have one of each of the option clauses so each can only be specified in the parameter definition or with this syntax but not both:

```
PARAMETER param-name param-options;
```

param-name must be the name of a Method parameter that is not concealed and whose mode is either IN or INOUT. The *dependency* clause is the same syntax as for specifying the dependency in the parameter definition beginning with the keyword DEPENDS as shown above.

The following example specifies the same dependency rules as the example above.

```
( IN str : TEXT = "";
  IN len : NUMBER = 0;
  IN big : NUMBER = 0 ) IS

PARAMETER len DEPENDS ON str EVALUATE (size( str ));
PARAMETER big DEPENDS ON len ENABLE (len > 0), HELP "long";
```

Notice that the parameter definitions have had the parameter options removed and the same clauses have been added to separate PARAMETER statements.

¹ The HELP clause is required when the parser instruction #REQUIRE help = True is specified previously in the Automation Script file.

Events

As you may recall, Job execution progresses sequentially through every Method of the Job. That is to say that each Method is executed once in the order in which they exist in the Job for each sample of the Job. However, there is one exception to this rule: Methods that have been designed to handle events will only execute when the designated event occurs. These Method may still be invoked deliberately by calling them with the INVOKE statement. See the prior description of the INVOKE statement. Methods that are written to handle events require the WHEN keyword with an *event* name following the *method-name*. Method parameters are not permitted when a Method is written to handle an event.

```
METHOD Gracefully_End WHEN ABORT IS...
```

There are three events: PREPARE, COMPLETE, and ABORT.

- The PREPARE event occurs when a Job begins to execute. These Methods will run automatically once prior to any other Methods in the Job.
- The COMPLETE event occurs when a Job successfully ends without error. The end of a Job occurs when all repetitions for each sample in the Job complete. These Methods will also automatically run once per Job. The FINISH and TERMINATE statements could cause the Job to end early, and this is still considered a successful completion.
- The ABORT event occurs when a Job is cancelled or aborted or when it prematurely terminates because of an error condition. A Job may be cancelled or aborted by a user or externally by another device.

A nested Method (that is, a Method that is declared within a Method) cannot be designated as an event handler. It is thus only valid to include the WHEN clause on Methods that are declared at the outer-most scope level of the Automation Script.

Special Method Statements

The optional CATEGORY statement must be the first statement of a Method block. Categories classify a Method providing a way to more easily find and choose the appropriate Method for a task. One or more Strings, separated by commas, following the CATEGORY keyword, specify the categories to which a Method will belong. The Method categories can be selected on the Automation user interface to restrict the list of available Methods such that only the Methods that match the selected categories will be visible. The category list can be a mixed series of Strings and/or language translation identifiers. Refer to the TRANSLATE statement regarding language translations.

The HELP statement is an optional single String or translation identifier that will succinctly convey the purpose of a Method to the user. If a translation identifier is provided and it refers to a multi-line translation, the first line only will be used for the purpose text.

The PURPOSE statement of a Method is also optional. When the PURPOSE statement is specified, it should provide a short description of what the Method will accomplish when it is successfully executed. It should also provide more detail than the HELP text and so it may contain more than one line of text. A single translation identifier or one or more comma-separated Strings must follow the PURPOSE keyword. If the PURPOSE statement is omitted, the text provided by the HELP statement will be used.

The DURATION statement is also an optional part of the Method block and its expression must result in a number with a unit of seconds when it is evaluated. Including this line lets the Method quickly supply an indication of how much time it is expected to take when it runs

successfully. Method parameters, declared constants, exposed variables, and names of EXPERIMENT blocks declared in the body of the Method may be referenced in the duration expression. Only variables set directly in the parameter assignment section of the EXPERIMENT block will affect the calculated time of the experiment – wildcard assignments will not work. The expression is an estimate of time but it should be designed and written to be as accurate as possible. See the section “Tips for Writing a Duration Statement Expression”.

Method Body

The body of a Method is the *statement-block*. It is a series of ordered operations that will produce a desired result when executed. Any of the statements described in the ‘Statements’ section (except for the EXIT and RETRY statements) may be utilized within the *statement-block* of a Method.

The Automation Script in the following Method expresses the simple classic “Hello World” program example:

```
METHOD Hello_World IS
    INFORM "Hello, world";
END METHOD;
```

An example of a slightly more complicated Method that is parameterized and provides some helpful description is:

```
METHOD Say_Hi( IN color : TEXT = "" ) IS
    HELP "Say hello to the user";
    INFORM "Hello, $(user)!";
    IF color /= "" THEN
        INFORM "Your favorite color is $(color).";
    END IF;
END METHOD Say_Hi;
```

The following Method is a simple example of data passing in to and out from a parameter. This Method also categorizes itself into a “math” category and provides detailed descriptions about what it does.

```
METHOD Power2( INOUT n : NUMBER = 0, HELP "Result is 2^n" ) IS
    CATEGORY "math";
    HELP "Returns 2^n";
    PURPOSE "Calculates the power of 2", "of the parameter 'n'.";

    SET n = 2**n;

END METHOD Power2;
```

Concealing Methods

Methods in an Automation Script at the outermost scope level are normally accessible by the user to select and include in a Job. However, it is possible for the author to hide a Method from users. If the CONCEAL keyword is specified before the initial METHOD keyword then the Method will not be accessible by the user and thus not available to be directly included into a Job.

One reason for possibly concealing a Method is that the Method is intended to be a support routine for the other Methods in the Script file. The other Methods might invoke the concealed Method, but users should not be allowed direct access to the Method.

The CONCEAL keyword has no effect on Methods that are defined within other Methods since these Methods are inherently concealed from the user. Nevertheless, the CONCEAL keyword is permitted on all Methods regardless of where they are defined.

By their nature, concealed Methods cannot be interactive and so the INTERACTIVE keyword cannot be used when the CONCEAL keyword is present.

Methods that Interact with the Operator

Methods are normally run by the Spectrometer without requiring human input once they are submitted. There may be instances when input is required from the operator while the Automation is running. In these instances, you can inform the system that a Method could make requests from the operator while it runs by declaring the method to be interactive. To do this, precede the METHOD keyword with the INTERACTIVE keyword.

Declaring a Method to be interactive permits the use of the PROMPT statement within that Method and any Method that is invoked by that Method. The PROMPT statement is the mechanism by which the Automation Script can request information from the operator while it is running. See the PROMPT statement for more information.

Only Methods at the outermost scope level need to be declared to be interactive. The INTERACTIVE keyword has no effect on Methods that are defined within other Methods. Nevertheless, the INTERACTIVE keyword is permitted on all Methods regardless of where they are defined.

Interactive Methods cannot also be concealed since concealed Methods cannot be submitted directly by the operator. Therefore, the INTERACTIVE keyword cannot be used when the CONCEAL keyword has been specified.

If a Method is declared to be interactive then it can invoke other interactive, concealed or regular non-interactive Methods. However, a Method that was *not* declared to be interactive cannot invoke a Method that is interactive.

Method Assertions

The author of an Automation Script can assert that a Method will exist before it is defined with a full METHOD block by ending the statement with a semicolon immediately after the name of the Method and optionally adding the ASSERT keyword before the METHOD keyword. The syntax for this is:

```
ASSERT CONCEAL METHOD method-name ;
```

This statement informs Automation that a Method titled *method-name* will exist at some point later in the Script. The *method-name* may be an identifier or a String. This assertion of existence provides Methods with the ability to invoke each other (known as a circular reference). Refer to the prior descriptions of the INCLUDE and INVOKE statements.

An example of a circular Method reference is:

```
ASSERT METHOD Second;

METHOD First IS
  ...
```

```

    INVOKE Second();
    ...
END METHOD First;

METHOD Second IS
    ...
    INVOKE First();
    ...
END METHOD Second;

```

WARNING! Circular references can cause infinite loops. Always remember to use some form of conditional statement to break the cycle.



If a Method assertion specifies that the Method be concealed using the CONCEAL keyword then the final definition of the Method must also specify that it is concealed. Similarly, if the INTERACTIVE keyword is specified on the Method assertion then the INTERACTIVE keyword must also be specified on the final definition. That is, the Method assertion and its full definition must match in its use of the CONCEAL or INTERACTIVE keyword.

Method Genericity

The technique of using Method assertions can be used to generalize a Method by allowing all or part of the operations of a Method to be defined by another Method in a separate Script file. This is explained with an example in the prior section describing the INCLUDE statement, but here is a quick example.

main.jaf contains:

```

METHOD Operate IS
    ...
END METHOD;

INCLUDE "my_separate_script" TO DOMAIN example;

METHOD Main IS
    ...
    INVOKE example.Separate();
    ...
END Main;

```

my_separate_script.jaf contains:

```

ASSERT METHOD Operate;

METHOD Separate IS
    ...
    INVOKE Operate();
    ...
END METHOD;

```

NUMBER

The NUMBER statement creates a sub-Type of the basic NUMBER Type that constrains the range of numeric values that can be assigned to variables of the sub-Type. The syntax to create a ranged NUMBER sub-Type is:

```
NUMBER type-name IS precision
      FROM number TO number STEP number
      unit-clause;
```

Another way to create a ranged NUMBER sub-Type is by obtaining the number constraints (bounds, precision, and unit) from Namespace. The syntax has the form:

```
NUMBER type-name IS NAMESPACE namespace-path;
```

A third syntax allows the author to create a listed NUMBER sub-Type. With this syntax, each of the legal values are specified in *number-list* separated by commas. The order of the numbers will be the order in which the operator will step through the available values. This syntax has the form:

```
NUMBER type-name IS precision
      LIST sort-order ( number-list )
      unit-clause;
```

Description

Type-name is an identifier that will be the name of this sub-Type. Any statement after this Type definition can use the *type-name* in any place where one of the basic Types can be used. The range of permitted number values is specified by the two *numbers* on the left and right of the TO keyword. The first number should be smaller than the second number to properly specify a range.

Allowing Only Integer Numbers

When the INTEGER keyword is specified for the optional *precision* clause the sub-Type is constrained to hold only numbers that have no fractional component. Assigning a number with a fractional component to a variable of a ranged NUMBER sub-Type defined with the INTEGER keyword will have its value rounded to the nearest integral value (± 0.5 will be rounded away from zero). Assigning a value to a listed NUMBER sub-Type must be exact – no rounding will occur.

Limiting the Precision of Numbers

When the PRECISION keyword is specified with a following non-negative integer for the optional *precision* clause the sub-Type is constrained to round the numbers to the specified precision. For example, if a precision of 2 is specified, then numbers would be accurate only to the hundredths decimal place. For example, $1/3 = 0.33$ and $1/8 = 0.13$. Specifying PRECISION 0 (zero) for the *precision* clause has the same effect as specifying the *precision* using the INTEGER keyword.

Additionally, when values of this sub-Type are placed on the user interface, the precision informs the widget of the step increment between adjacent values. If no precision is specified, the default step size is 3 – that is, three digits of precision (an effective step size of 0.001).

Specifying Values to Skip with a Step Size

Without a specified step size, any number is a legal value for the sub-Type between the lower and upper limits inclusive. However, a step size may be specified which restricts the allowable values to be multiples of the step size from the lower limit. The step size must be a number greater than zero. Therefore, legal values of the sub-Type can be expressed with the following formula:

$$\text{lower_limit} + n \times \text{step_size}$$

where the value n is a non-negative integer such that the result is less than or equal to the upper limit.

Constraining the Unit

If a *unit-clause* is provided, then numbers assigned to a variable of the NUMBER sub-Type must also conform to the proper unit. Omitting the *unit-clause* will not put any unit constraint on the sub-Type allowing values with and without units to be assigned to variables of the sub-Type.

The *unit-clause* can be specified as WITH NO UNIT that will allow only number values without a unit to be assigned to variables of the sub-Type. When the *unit-clause* is specified as WITH UNIT *unit-literal* (where *unit-literal* is a valid Percival unit) then only values with the properly specified unit can be assigned to variables of the sub-Type. Refer to the ‘Table of Units’ section for a list of units that can be used for the *unit-literal*.

Note that the following two unit clauses are equivalent (the second form uses the empty unit).

```
WITH NO UNIT
WITH UNIT [ ]
```

Enforcing Listed Sort Order

The author can be clear that the listed numbers should be in ascending or descending order by including the keywords INCREASE or DECREASE accordingly for *sort-order*. This will make it clear to other readers who are editing the script that the list must be sorted. The system will alert the operator if numbers are out of the order when *sort-order* is specified. No alert is given if *sort-order* is omitted.

NUMBER Type Relationship

A value of a NUMBER sub-Type may be assigned to any variable of the basic NUMBER Type. However, a value of a basic NUMBER Type may only be assigned to a variable of a particular NUMBER sub-Type if its value is within the range of permitted values and, if specified, has the appropriate unit.

Required Attributes for Namespace Path

A NUMBER sub-Type defined by Namespace is specified using the NAMESPACE keyword and providing a *namespace-path* between quotation marks. The path in Namespace must contain a LOWER and an UPPER attribute. The LOWER attribute naturally specifies the lower limit for the sub-Type and the UPPER attribute specifies the upper limit.

The Namespace path may optionally contain a PRECISION attribute that specifies the number of digits of precision in the decimal for the sub-Type. If this attribute is not specified, the STEP attribute is used; otherwise the value’s precision is not constrained. The PRECISION attribute specifies how many digits of precision are requested for the value. The STEP attribute specifies the distance between the legal values that lie within the lower and upper bounds.

The Namespace path may also optionally specify a unit for the sub-Type as a String on the UNIT attribute. If not specified, the sub-Type will not have a unit constraint. If specified as an empty String, then units will not be allowed on values of the sub-Type like specifying WITH NO UNIT. Otherwise, if the String represents a valid unit, that unit will be required on all values of the sub-Type.

Namespace paths for constraining a NUMBER Type are specified within a Namespace file using the following form:

```
namespace-path(attribute) = value
```

For example, to define a constraint named “ns_num_type” to restrict numeric values to be between 10 and 50 with three digits of decimal precision requiring a unit of Hertz the following lines would have to be defined.

```
ns_num_type(LOWER)      = 10
ns_num_type(UPPER)      = 50
ns_num_type(PRECISION) = 3
ns_num_type(UNIT)       = “Hz”
```

Examples

The following are examples of NUMBER statements and of the behavior of assigning values to those sub-Types:

```
NUMBER Small IS PRECISION 2 FROM 1 TO 9;

NUMBER Sweep IS 2 TO 20 WITH UNIT [ppm];

NUMBER Int IS INTEGER 0 TO 1000 WITH NO UNIT;

NUMBER NS_Hertz IS NAMESPACE “ns_num_type”; -- from above

NUMBER Primes IS INTEGER LIST (2,3,5,7,11,13,17,19);

VAR N : NUMBER;
VAR S : Small;
VAR I : Int;
VAR W : Sweep;
VAR H : NS_Hertz;
VAR P : Primes;

SET S = 1.3;           --okay
SET S = 1.1234;       --okay   (rounds down to 1.12)
SET S = 1.5678;       --okay   (rounds up to 1.57)
SET S = 10;           --error   (out of range)
SET S = 4.80[ppm];   --okay
SET S = 4[s];        --okay

SET N = S;            --okay
```

```
SET N = 9;
SET S = N;          --okay
SET N = 0;
SET S = N;          --error   (out of range)

SET I = 25;         --okay
SET I = 25.2;       --okay   (round down to 25)
SET I = 25.6;       --okay   (round up to 26)

SET I = 500[s];    --error   (unit not allowed)

SET N = I;         --okay

SET N = 2000;
SET I = N;         --error   (out of range)

SET N = 500[m];
SET I = N;         --error   (unit not allowed)

SET W = 2[ppm];    --okay
SET W = 1[ppm];    --error   (out of range)
SET W = 10[Hz];    --error   (incorrect unit)

SET N = W;         --okay

SET N = 5[ppm];
SET W = N;         --okay

SET N = 50[ppm];
SET W = N;         --error   (out of range)

SET N = 5[s];
SET W = N;         --error   (incorrect unit)

SET N = 10;
SET W = N;         --error   (unit required)

SET P = 2;
SET P = 23;        --error   (illegal value – 23 is prime but not in list)
SET P = 4;         --error   (illegal value)
SET P = 2.01;      --error   (illegal value)

SET N = 7;
SET P = N;         --okay
SET N = 6;
SET P = N;         --error   (illegal value)
```

ON ERROR

When an error occurs during the execution of a Method, the normal behavior is that the Job will immediately terminate. The ON ERROR statement block gives Automation a chance to perform some action if an error occurs. The author of the Automation Script may choose to ignore the error and continue, stop executing the Method or current block, or execute a sequence of statements (to perhaps correct the error). The ON ERROR statement has two syntactical forms:

```
ON ERROR error-code action-to-take;
```

and

```
ON ERROR error-code DO
    statement-block
    RETRY WHEN expression;
END ERROR;
```

Error Codes

The optional *error-code* is a String that specifies that the error handler should be triggered only when that named error occurs. If *error-code* is omitted or specified with the keyword ALL, then the handler will run when any error occurs regardless of the actual error. The following named errors can occur while an Automation Script is executing:

Error String	Raised by Automation when...
acq-error	The EXPERIMENT block fails to collect data.
constraint-error	The new value being assigned to a variable of a sub-Type is not within the valid range specified by the sub-Type.
eval-error	The evaluation of an expression could not be completed or it resulted in an illegal value.
exec-error	The execution of a service or Percival program causes an error or returns an error condition.
mail-error	Email could not be delivered. This could be because the addressees were not in the right form or the attached files could not be found.
print-error	The PRINT or PRESENTATION statement could not complete because of an error with the print queue. The Presentation Layout file is not found or is not properly formatted when executing a PRESENTATION statement.
process-error	The PROCESS statement could not complete because there was an error processing the data or the Processing List file is not found or not properly formatted.

Error String	Raised by Automation when...
storage-error	A file is unable to be stored to or retrieved from a data server or if a variable's value is unable to be stored.
translation-error	The TRANSLATE statement is unable to locate a valid translation for the String.
tune-error	The NMR instrument could not be tuned properly as requested by the TUNE statement.
type-error	An assignment is attempted and the Type of the new value does not conform to the Type of the variable to which it is being assigned. This can also occur while evaluating the parameter requirements in a CALL statement or a Method invocation.

Actions

For the first syntax, there are four possible keywords that can be used in place of *action-to-take*. The following table describes the effect each has on the execution of the Script when it is used to handle the error.

Action Keyword	What it does...
CONTINUE	Ignore the error and continue running the Automation Script with the statement following the one that caused the error.
FINISH	Stop executing the current Method and save all the data that has been acquired up to the point of the error. If the error occurred in a Method that was invoked from another Method, then the calling Method will continue to run with the statement following the Method invocation.
TERMINATE	Stop executing the Automation Script and save all the data that has been acquired up to the point of the error. If the Method was invoked from another Method, the calling Method will also terminate.
EXIT	Stop executing the current REPEAT block and continue running the Automation Script with the statement following the end of the REPEAT block. The use of the EXIT action can only appear within a REPEAT block.

Error Correction and Cleanup

The second syntax of the ON ERROR statement gives the author of the Automation Script the ability to alter the normal execution flow (possibly to correct an error) by running additional statements before continuing normal operation. When an error is handled with this syntax, the statements of *statement-block* are executed when the error is detected.

Attempting the Problem Statement Again

The RETRY statement causes Automation to re-execute the statement that caused the error. If a RETRY statement is not executed, then the execution of the Script resumes with the statement following the one that caused the error. Refer to the RETRY statement section for more details.

The WHEN clause can be optionally added to the RETRY statement. If the WHEN clause is specified, this statement is only executed if the *expression* evaluates to a True value. Refer to the section titled ‘Boolean Expressions’ for how an expression evaluates to a Boolean value.

Stopping the Execution

If the author of the Script wishes to stop the execution while handling the error, the FINISH, TERMINATE, or EXIT statements can be used within the *statement-block*. The descriptions of these three statements are described elsewhere in the ‘Statements’ section.

Modifying the Error Condition

The author of the Automation Script can also change the error condition to any other error by using the RAISE statement within the *statement-block*. Refer to the description of RAISE statement.

How Error Handlers are Chosen When an Error Occurs

Error handling statement blocks can be embedded within other statement blocks. This means that the error handling code is “scoped” like any other statement. When an error event occurs, the following steps are taken:

1. A handler in the current scope level with an **exact match** of the error code will handle the error if one is provided. Go to step 4.
2. A handler in the current scope that is designated to handle **any** error code (with the ALL keyword specified for the *error-code*) will handle the error if one is provided. Go to step 4.
3. At this point, the current scope does not have an error handler written to handle this error. If there is a previous scope level, change scope to it and go to step 1. Otherwise, an error handler was not found to handle the error so the Job is immediately terminated.
4. Continue execution of the Method according to the error handler.

IMPORTANT: There is a default error handler that may be overridden. The default error handler is defined to be:

```
ON ERROR ALL TERMINATE;
```

Therefore, if an error occurs and there is not an error handler defined for that specific error code, the Automation Script will terminate because of the default error handler.

Examples

Examples of the use of ON ERROR statements are:

```
METHOD "Get Param"( path : TEXT = "" ) IS

    NUMBER OneToTen IS 1 TO 10;

    VAR num : OneToTen;

    REMARK Create a global handler to end this method smoothly.
    ON ERROR ALL FINISH;

    REMARK Handle a Type error with a message and constant.
    ON ERROR "type-error" DO
        INFORM "Specified path does not result in a number",
            "Using zero";
        SET num = 0;
        RETRY;
    END ERROR;
```

The following statement in this Method will raise a "type-error" if the value of the Namespace path does not return a NUMBER Type. Because the variable num is constrained by the sub-Type definition, it could also raise a "constraint-error" if the value of the Namespace path less than 1 (one) or if it is greater than 10 (ten).

```
SET num = NAMESPACE path;
```

The following if statement will raise "eval-error" which will be handled by the global handler because there is no error handler defined to handle this specific error.

```
IF path + .5 THEN                -- .5 should be written as 0.5
    INFORM "Bad Expression";
END IF;

END METHOD;
```

A variable could be defined within a Method block to determine if the script should attempt to execute the problematic statement again:

```
VAR try_again : BOOLEAN;
...
ON ERROR DO
    INFORM "Script error!";
    RETRY WHEN try_again;
END ERROR;
```

PERCIVAL

The PERCIVAL statement allows the author of the Automation Script to embed Percival program code into a Script. Percival is the proprietary programming language developed by JEOL that is executed by the Delta and Control software. Any embedded Percival code must conform to proper Percival program syntax. It must be placed after three consecutive less-than characters and be followed by three consecutive greater-than characters. These six characters set the Percival code apart from the rest of the Automation Script statements.

The syntax of the PERCIVAL statement is:

```
PERCIVAL <<< Percival-code >>>;
```

Percival code is executed with the CALL statement and referencing the name of an operator defined within the six angle characters. See the prior discussion of the CALL statement for more details.

Syntax Errors Not Detected Unit Run-Time

It is important to realize that errors in the Percival code will not be detected until the PERCIVAL statement is executed. Therefore, it is recommended that any code that is embedded in the PERCIVAL statement be fully tested for correctness before the Script is run. It is beyond the scope of this document to describe the syntax and semantics of the statements and constructs of the Percival programming language.

User Interactive Statements

Do not expect the text output statements in the Percival language (put_line, display, etc.) to behave the same when run under Automation as opposed to how they operate when executed from Delta. The mechanism by which text is displayed on the Master Console does not exist for Automation being executed by the Control software. There is currently no plan to support these statements. Use the INFORM statement in the Automation Script language (described later) to display output text to the user.

NOTE: Presently, the Percival functions that print messages to the screen, pop-up dialog boxes, or do any type of user interaction, will **not** operate properly when called from an Automation Script. These functions should not cause an error, but the user will not see the effects from these statements.



WARNING! Because of the above program note, the author of the script **must** avoid using Percival functions that suspend regular program execution while waiting for the user to respond to the program in any way. An example of an operation to avoid is calling the Delta file dialog box.

Example

The following is an example of how Percival code that defines a functional operation can be incorporated into an Automation Script.

```
PERCIVAL <<<
  function FIND_MEAN( a : NUMBER,
                    b : NUMBER ) return NUMBER is
    return (a + b) / 2;
  end FIND_MEAN;
>>>;
```

```
VAR n : NUMBER;
```

```
SET n = CALL find_mean( 0, 10 );
```

The above example defines a Percival function named “FIND_MEAN” to calculate the mean (the average) of two numbers. At some point, later in the Script, a variable `n` is defined to contain a NUMBER Type and then calls the Percival operator passing it two constant values: 0 and 10. The variable `n` will contain the value 5 after the SET statement is executed.

Relationship of Automation/Percival Types

The following table summarizes the Types that are available in Automation and the equivalent Types provided by Percival. A Percival function should return the Percival Type that is the Automation equivalent of the Type of the variable in the Automation Script that will hold the resulting value.

Variable Type Comparisons	
AUTOMATION	PERCIVAL
BOOLEAN	Boolean
TEXT	String
NUMBER	Numeric Universal Number
LIST	Set
DATA	File

NOTE: It should be noted that there are fewer Types provided by the Automation syntax than there are provided by the Percival language. Although there are similarities (for example the NUMBER type), the Types are distinct and may be incompatible. Please be sure that the Percival functions used by Automation will return values that Automation will recognize. The Percival Types supported by Automation are: Boolean, String, Numeric, Universal, Number, Set, and File.

PRESENTATION

The PRESENTATION statement creates a formatted output by referencing a layout file that was defined using the Page Layout Editor (available within the Delta software). The syntax of the PRESENTATION statement is:

```
PRESENTATION title TEMPLATE layout-filename
  WITH layout-data CONTEXT print-context destination;
```

Page Layout Template File

The *layout-filename* specifies the name of the Page Layout template file to use for generating the output. The layout files also describe how many data files are required and how those data files are put onto a printed page. The *layout-filename* can be either a String of characters within double quotation marks or a variable of the TEXT Type that contains a filename. Page Layout template files usually end with the .pmt extension, but specifying the extension is not necessary and it is recommended that it be omitted from the filename when specified within an Automation Script. If the filename extension is omitted, Page Layout template files will be located using the .pmt extension. The standard template directory locations are searched to find the file as described in the preceding section titled 'Locating Support Files'.

The format of a Page Layout template file is beyond the scope of this document.

Referencing the Printed Data

Like the PRINT statement, the PRESENTATION statement can generate an output file that can be referenced by subsequent statements in the Script. See the sub-section with the same heading in the following description of the PRINT statement for how to specify the *title* and how it will affect the behavior of this statement.

Providing Data Files for the Page Layout

Page Layout templates that require data obtain the data files through parameters. Each of the data files for the layout is specified using the following syntax:

```
DATA data-variable FOR PARAMETER positive-integer
```

A single data file should be provided for each required parameter of the Page Layout template and each data file specification must be separated by the AND keyword. The *data-variable* is the data file that will be passed to the template as a parameter and assigned to a Page Layout box. *Data-variable* must be a reference to a JEOL NMR data file. If it is not, or if a parameter is omitted, the area on the page designated for that file parameter will be empty.

Normally, each specified data file will be associated with the next consecutive Page Layout template parameter (the first parameter being number 1). However, if FOR PARAMETER *positive-integer* is provided, it overrides the parameter number of the Page Layout template for the associated data file. If this phrase is omitted, the next positive integer after the last assigned parameter number (beginning with 1) will be assumed.

Specifying a Print Context

A print context provides information to the printer about how the output should be generated. A print context is optional and can be specified using the CONTEXT keyword.

The value for *print-context* can be a literal String, a TEXT Type identifier, or an identifier previously declared by a PRINT CONTEXT statement. The print context supplied here will be used for all of the output generated unless the context is specifically overridden within a *destination*. Refer to the discussion of the PRINT CONTEXT statement for more details.

How and Where to Print

Refer to the following discussion of the PRINT statement for a description of the syntax and purpose of *destination* part of this syntax. These terms are used in the PRESENTATION statement in the exact same manner as they are used in the PRINT statement.

Examples

Below are examples of how the PRESENTATION statement can be used. Each statement refers to the following file variables `exp` and `d`.

```
EXPERIMENT exp IS
...
END EXPERIMENT ;

VAR d : DATA = "my_data_file";
```

The following line prints a file in a page layout specified by the 'single_layout' template file to the default printer.

```
| PRESENTATION TEMPLATE "single_layout" WITH exp;
```

The next example will print the layout to a printer identified by `local` and store a file on the Data Server named "result.pdf". Two files will be passed to the page layout template named "layout2". Notice that `exp` will be passed to the page layout template for parameter 1, and that `d` will be passed for parameter 3. The area of the page reserved for parameter 2 will be left blank in the output. Access to the generated PDF file is provided to future statements in the Automation Script through the variable named `my_pdf`.

```
| PRESENTATION my_pdf TEMPLATE "layout2"
  WITH DATA exp AND d FOR PARAMETER 3
  TO PRINTER local AND FILE "result.pdf";
```

PRINT

The PRINT statement generates a full-page printed image of a data file that can be sent to a printer or saved to a disk file. The syntax of the PRINT statement is:

```
PRINT title DATA data-variable WITH parameter-requirements
CONTEXT print-context destination;
```

Data-variable is the data file that will be printed and it must be a variable reference to a JEOL NMR data file. The data file can be recently acquired by a preceding EXPERIMENT block or a pre-existing data file on disk assigned to a DATA Type variable.

Referencing the Printed Data

To reference the printed data later in the Automation Script, the printed data must be given a name. *Title* is the name of the generated file and, when specified, will cause the printed data to be saved to a file (perhaps temporarily) so that a subsequent statement in the same Method can access it. In effect, *title* becomes the name of a new constant value that contains a reference to the printed file that will be generated. The form of this file will depend on the Operating System on which the Automation Script is running. The author of the Automation Script is not required to declare a DATA Type variable prior to this statement for *title*, but it is good practice to do so for clarity.

Printing Data Parameters

If the *parameter-requirements* clause is included (beginning with the WITH keyword), additional pages will be printed following the data graph that will show the acquisition parameters and their associated values. There are a few forms for this clause described below.

Clause form	What is printed
WITH PARAMETERS	Prints the set of generally useful parameters. This parameter list is stored in the Namespace path: <code>parameter_filters.Print_Information</code>
WITH ALL PARAMETERS	Prints every acquisition parameter stored with the data.
WITH PARAMETER LIST <i>name</i>	Prints a custom set of parameters by specifying a filename or Namespace path to locate the custom parameter set. The following rules are followed, in order, to locate the parameter set: <ol style="list-style-type: none"> 1. If <i>name</i> begins with “namespace:”, use the specified Namespace path (following the two colons) for the parameter set. 2. Read the disk file with the exact <i>name</i>. 3. Read the disk file with the filename: “<i>name.params</i>”. 4. Read the Namespace path: “<code>parameter_filters.<i>name</i></code>”. 5. Use the set of generally useful parameters for printing.
WITH PARAMETER LIST <i>var</i>	Prints the contents of <i>var</i> , a variable of the LIST Type that represents a set of parameter names to be printed.
WITH PARAMETER LIST (<i>list</i>)	<i>List</i> is one or more comma separated Strings. Each String is the name of a single parameter to be printed.

Specifying a Print Context

A print context provides information to the printer about how the output should be generated. A print context is optional and can be specified using the CONTEXT keyword.

The value for *print-context* can be a literal String, a TEXT Type identifier, or an identifier previously declared by a PRINT CONTEXT statement. The print context supplied here will be used for all the output generated unless the context is specifically overridden within a *destination*. Refer to the discussion of the PRINT CONTEXT statement for more details.

How and Where to Print

The last optional item of the PRINT statement is the *destination* – where the output is to be printed. The *destination* is specified using the following syntax:

```
TO target AND target
```

where the *target* is specified as the printer given in the Job attributes:

```
JOB PRINTER CONTEXT print-context
```

or as a specific printer:

```
PRINTER printer-name CONTEXT print-context
```

or as a file destination:

```
FILE REF output-name CONTEXT print-context
```

Notice that the *printer-name* is optional. When a specific printer is not named, the instrument's default printer will be used.

The optional *print-context* provided on a specific *target* will override the general *print-context* described earlier. It follows the same syntactic rules as the general CONTEXT clause above.

In the case where a *title* is provided to reference the data later in the Script and more than one FILE *target* is given, the REF keyword may be used to specify which of the generated output files is to be used as the file that is to be referenced. The REF keyword can only be specified on one file output destination. If REF is omitted, the **first** file generated will be the file that is referenced by *title*.

Specifying Multiple Destinations

The AND clause can be included to send the output to multiple targets. Using AND repeatedly will cause the output to be sent to as many targets as specified. *Printer-name* is a String or variable of the TEXT Type that names a specific network printer. *Output-name* is the name of the file to which the results of the PRINT statement will be stored on the default Data Server. The filename extension provided on the *output-name* will impact the format of the output file.

Default Behavior

If a *destination* clause is omitted but a *title* is provided, a temporary PDF file will be generated and stored with the name that is given by *title*. However, Automation will attempt to locate a default printer to print the data file when a *destination* clause and the *title* are both omitted. If a default printer is found, a data image will be sent to it for printing.

Examples

The following examples of the PRINT statement use a DATA Type variable, `d`, that could have been created with an EXPERIMENT block, a PROCESS statement, or with a VAR statement that loads a data file like:

```
VAR d : DATA = "my_data_file";
```

The following line prints the data `d` to the default printer.

```
| PRINT DATA d;
```

The following line will generate a temporary PDF file named “ref.pdf” which will contain an image of the data as well as a second page containing a list of the data file’s acquisition parameters. The constant `pref` may be used later in the script to reference the printed data file.

```
| PRINT pref DATA d WITH PARAMETERS;
```

The following line will try to print to the printer specified by the Job attribute “printer”. If the “printer” attribute is found and names an actual printer on the network, the data and its acquisition parameters will be sent to it to print.

```
| PRINT DATA d WITH PARAMETERS TO JOB PRINTER;
```

The following line will try to find the printer named “inkjet” and, if found, will send the data and its acquisition parameters to it to print.

```
| PRINT DATA d WITH PARAMETERS TO PRINTER "inkjet";
```

The following line is a combination of two previous examples with the exception that the name of the generated PDF file will be “result.pdf”. Aside from that difference, both a hard copy and a PDF file will be generated with a statement like the following example.

```
| PRINT pref DATA d WITH PARAMETERS  
    TO JOB PRINTER AND FILE "result.pdf";
```

PRINT CONTEXT

The PRINT CONTEXT statement declares a context for printing that can be used to control the output generated by the PRINT and PRESENTATION statements. Create and use a printing context to ensure that the settings of the printer are correct for the output being generated. The syntax of the PRINT CONTEXT statement is:

```
PRINT CONTEXT printing-context IS
    attribute-assignment;
END PRINT CONTEXT;
```

Printing-context is an identifier that becomes the name of the group of attributes that define the printing context. This identifier can be used in the CONTEXT clause of any subsequent PRINT and PRESENTATION statements.

An *attribute-assignment* is made up of an identifier that is followed by an equal sign and a value for that identifier. The author can (and usually will) supply more than one *attribute-assignment* lines in a PRINT CONTEXT statement and each line must be terminated with a semicolon.

Print Context Attributes

A print context attribute can have any name (including reserved keywords). The following table shows the attributes and their Types that are recognized by the Automation printing system.

Attribute Name	Type Required	Legal Values <i>blank is unrestricted</i>
border	BOOLEAN	
border_width	NUMBER	>= 0
bottom_margin	NUMBER	>= 0
color	BOOLEAN	
contour_thickness	NUMBER	>= 0
copies	NUMBER	>= 1
data_thickness	NUMBER	>= 0
deconvolution_thickness	NUMBER	>= 0
deconvolution_fixed_colors	BOOLEAN	
fda_21cfr11	BOOLEAN	
font_name	STRING	
font_size	NUMBER	>= 1
footer	STRING	
footer_on_page_one	BOOLEAN	
geom_square	BOOLEAN	
grid_grey_scale	NUMBER	0 .. 100
header	STRING	
header_on_page_one	BOOLEAN	
integral_thickness	NUMBER	>= 0
left_margin	NUMBER	>= 0
neg_contour_grey	BOOLEAN	
neg_contour_grey_value	NUMBER	0 .. 100

Attribute Name	Type Required	Legal Values <i>blank is unrestricted</i>
orientation	STRING	LANDSCAPE, PORTRAIT, REVERSE_LANDSCAPE, REVERSE_PORTRAIT
page_scale	NUMBER	>= 0
paper_height	NUMBER	>= 0
paper_name	STRING	
paper_width	NUMBER	>= 0
parameters	BOOLEAN	
parameters_location	STRING	LEFT, RIGHT, TOP, BOTTOM
process_list	BOOLEAN	
right_margin	NUMBER	>= 0
ruler_thickness	NUMBER	>= 0
tessellate	NUMBER	2 .. 64
top_margin	NUMBER	>= 0

Examples

The example below declares a printing context with a few commonly used attributes.

```
PRINT CONTEXT High_Quality IS
  border      = TRUE;
  color       = FALSE;
  tessellate  = 64;
  geom_square = FALSE;
  orientation = "Landscape";
END PRINT CONTEXT;
```

This example uses a variable to set a value to one of the attributes.

```
ENUM Page_Positions IS ( "Left", "Right", "Top", "Bottom" );

EXPOSE VAR param_pos : Page_Positions = "Left";
```

```
PRINT CONTEXT MoveParams IS
  parameters      = TRUE;
  parameters_location = param_pos;
END PRINT CONTEXT;
```


PROCESS

The PROCESS statement is used to process a data file and in so doing create a new instance of the original data modified by the processing. This statement can be used on its own or as part of an SET assignment statement. The full syntax of the PROCESS statement is:

```
PROCESS DATA data-variable WITH process-list-filename  
TO destination-variable SAVE AS destination-name;
```

The data to be processed is specified by *data-variable*. The *data-variable* must be a previously declared variable of the DATA Type or the title of a prior EXPERIMENT block.

Processing List Options

Each recently acquired Delta data file contains an intrinsic processing list that may be used to process the data. If the PROCESS statement does not specify a specific processing list to use, the intrinsic processing list within the data file will be used to process the data. Typically, this processing list is more than adequate. For the instances when the intrinsic processing list is known to be insufficient, the WITH clause can be included to override the intrinsic processing list with an alternate processing list. Including the WITH clause will process the data using the commands within the processing list file specified by *process-list-filename*. The *process-list-filename* may be a literal filename within double quotation marks or a variable of the TEXT Type that contains the filename. Automation will try to locate this file by looking in the standard locations as described in the section titled 'Location of Support Files'.

Alternatively, the ELSE keyword can be used in place of the WITH keyword. This informs the PROCESS statement to use the intrinsic processing list if it exists within the data file. If an intrinsic processing list is not found, the alternate processing list will be used. This may be necessary if a data file requires processing but it is not known if an intrinsic list is present within the data file.

Process-list-filename should be specified with a relative path. Processing list filenames usually end with the .list extension, but the .list extension is not required and it is recommended that the extension be left off. Names like, "1dh.list", "1dh", "plot/1dh_plot" are valid processing file references.

Referencing the Processed Data

The TO *destination-variable* clause is optional and when it is **not** specified, the newly processed data will replace the data in *data-variable* and the old data reference will be lost. The actual data file will **not** be deleted – only access to the data from the Automation Script will be lost. When the TO clause is included and it is specified to be a variable other than *data-variable*, the newly processed data will be stored in the variable specified by *destination-variable*. The author of the Automation Script may but is not required to have previously declared a DATA Type variable for *destination-variable*. A variable will be created automatically if one with the specified name was not previously declared.

Where to Store the Processed Data

Normally the processed data will not persist after the Method ends, but if you want to keep the data, then you must specify a storage filename with the SAVE AS clause. The data will then be stored on the Data Server with the name specified by *destination-name* that is either a literal name within double quotation marks or a variable of the TEXT Type containing the name.

Examples

The following are all valid PROCESS statements that assume two DATA Type variables named d and f exist. d and f could have been acquired by an EXPERIMENT block or simply be references to pre-existing files opened with a VAR or CONST statement.

```
PROCESS DATA f;  
  
PROCESS f ELSE "ldh";  
  
PROCESS f WITH "plot/ldh_plot" TO d;  
  
PROCESS f SAVE AS "my_processed_data";  
  
PROCESS f TO d SAVE AS "my_processed_data";
```

Processing With Assignment

As mentioned earlier, the PROCESS statement can be used in the new value part of an SET assignment statement. The only difference with using this form is that the TO clause is not permitted since the destination variable is provided by the assignment variable. Examples of this form are:

```
SET d = PROCESS f;  
  
SET d = PROCESS f WITH "ldh";  
  
SET d = PROCESS f SAVE AS "my_processed_data";
```

The last of the three examples above has the same effect and is equivalent to the last line of the previous example group.

PROMOTE

The PROMOTE statement preserves the values of one or more Namespace paths beyond the current operating scope level. The syntax of the PROMOTE statement is simply:

```
PROMOTE path AND path TO instrument-scope;
```

A *path* may be specified with the keyword SHIMS so that all of the current shim parameter values will be preserved to the specified *scope-level*. A *path* can also be specified with a literal String value that represents a Namespace path or a variable that contains a String. More than one *path* may be specified by separating each *path* with the AND keyword.

Instrument Scope Levels

The *instrument-scope* specifies the extent to which the Namespace values will be preserved. It is specified using one of the keywords from the following table:

Inst. Scope	Description
USER	This is the outermost instrument scope level and is in effect while an operator has ownership of the spectrometer. This will occur when an operator intentionally becomes an owner (by depressing the “ownership” button on the <i>Spectrometer Control</i> window) or when a queued Job begins to run.
PROJECT	This instrument scope level is currently not supported and will be treated as if the USER scope was specified. In future versions, this instrument scope level will become effective when the Project is loaded prior to a queued Job being run and will end when after the Job is finished and the Project session is over.
JOB	This instrument scope level becomes effective immediately when a queued Job begins to run and it will remain in effect while the Job is running which could possibly be over multiple Samples.
SAMPLE	This instrument scope level becomes effective when a Sample is loaded into the spectrometer and ends when the Sample is ejected or when the Job is finished, whichever occurs first.
METHOD	This instrument scope level is the current operating scope level of a running Job. Specifying the <i>instrument-scope</i> with METHOD will have no effect. It is allowed just in case a lower instrument scope level becomes necessary.

Examples

The following statements preserve the shim values to the USER instrument scope level and the Receiver Gain to the JOB instrument scope level.

```
PROMOTE SHIMS TO USER;
PROMOTE "temp_state" TO SAMPLE;
```

The following preserves both the shims and the Receiver Gain in a single statement.

```
PROMOTE SHIMS and "temp_state" TO SAMPLE;
```

PROMPT

The PROMPT statement allows the Automation runtime system to obtain information from the operator while a Job is running. Only Methods that have been declared interactive (by using the INTERACTIVE keyword when defining the Method block) can utilize the PROMPT statement. The operator must have some knowledge that the Job they will be submitting requires them to remain by the console to supply the responses to the prompts. These types of interactive Methods will be identified by a special icon next to its name.

The PROMPT statement will ask a question by displaying the query over the *Message Area* of the Spectrometer Control tool and expect the operator to respond by providing an answer to the question. Depending on the type of answer the system is expecting in response, the operator could be required to type a value into a text box, choose a value from an enumerated list, or press a button.

The syntax of the PROMPT statement is:

```
PROMPT question TO variable
       style option-list DEFAULT value ICON icon-name;
```

The PROMPT statement can be used in an assignment statement as well in which case the TO clause can be omitted. Assuming a variable named *r* exists, this form has the syntax:

```
VAR r ...;
SET r = PROMPT question
           style option-list DEFAULT value ICON icon-name;
```

The author could have declared the variable *r* and initialized it with the result of a PROMPT statement, like:

```
VAR r : type = PROMPT question
           style option-list DEFAULT value ICON icon-name;
```

but the variable *r* cannot be exposed. The following statement is illegal because of the use of the EXPOSE keyword before the VAR declaration.

```
EXPOSE VAR r : NUMBER = PROMPT "How many scans?";
```

Ask a Question

The PROMPT statement is for obtaining data from the operator that cannot be known at the time the Job is submitted. The question following the PROMPT keyword should be a literal String or Text variable that is formed as a question with enough information for the user to be able to correctly answer the question using the options available.

Use the INFORM statement to send a message to the operator without needing a response.

NOTE: If the information that the Method needs can be known prior to submitting the Job, then it is much better to use a Job attribute, Method parameter, or exposed variable to get the necessary values into the system. In this way, the operator does not have to remain by the console to answer the prompts that appear.

What Choices for Responding Will the Operator See?

Without an *option-list*, the operator will see a simple input box to type any text value and a button labeled ‘Continue’ which will send the response back to Automation. In this mode, the value entered will be evaluated to determine if it can be converted into a NUMBER or BOOLEAN Type. If an *option-list* is provided, it lists the values from which the operator can choose for the response value. The *style* is a keyword that indicates how to display the values. *Style* can be set to BUTTONS or OPTIONS. If the BUTTONS keyword is used, then each possible response will be displayed as a button and the response will be sent when the user clicks on one of the buttons. Up to four buttons can be displayed. If there are more than four values in the *option-list* then the *style* of BUTTONS will be ignored and will be automatically set to OPTIONS. With a *style* of OPTIONS, an enumeration widget will be displayed to let the user choose one from a drop-down list.

Each response value is specified in the options-list separated by commas. There must be at least two possible responses in the *option-list*. Each response can be specified only with the value itself or optionally with a textual representation. The values can be of any Type or a variable. Specifying the values alone would look like this:

```
OPTIONS 8, 16, 32, 64
```

The person answering the question would see a drop-down list with the numbers in it. Specifying the values with a textual representation might look like:

```
OPTIONS
  SHOW 8 AS "Quick", SHOW 16 AS "Fast",
  SHOW 32 AS "Normal", SHOW 64 AS "Long"
```

In the above case, the operator will see the choices as “Quick”, “Fast”, “Normal”, and “Long” in a drop-down list, and the response will be the integer value associated with the text value chosen. For example, if the operator selects “Fast” then the value of 16 will be returned.




The Probable Response

The value following the DEFAULT keyword is what will be the suggested response value. It will be the actual value returned if the operator does not respond within the time allotted by the system configuration. This default response value must have the same Type as the *variable* to which it will be assigned. In the second syntax example above, the value must match the Type of *r*.

Alert Icon

A small image (icon) can be shown with your question to give the operator a clue as to what kind of question is being asked. The icon desired is specified after the ICON keyword. If the ICON clause is not specified or if the icon file cannot be found, a simple question mark icon will be shown.

The six icon keywords that can be used to produce three different icons are shown in the following table.

Keyword	Icon Name	Icon
INFO, STATUS	help_bubble.rgb	
WARNING, ALERT	alert.rgb	
ERROR, FATAL	stop.rgb	

Examples

This simple example will prompt the operator to name a file with an input box.

```
VAR filename : TEXT;  
PROMPT "Name of the data file?" TO filename;
```

The example below will prompt the user to increase the scans but will return 0 if the operator does not respond before the allotted time allowed.

```
VAR scans : NUMBER;  
PROMPT "Could not reach adequate signal/noise. Do you want to  
increase scans?" TO scans  
OPTIONS 64, 96, 128, 192, 256, 384, 512, 768, 1024 DEFAULT 0;
```

The next example asks the operator if they thought the acquisition time was too long and will show buttons with "Yes" and "No" options.

```
VAR too_long : BOOLEAN;  
SET too_long = PROMPT "Was the acquisition time too long?"  
BUTTONS SHOW True AS "Yes", False AS "No" DEFAULT False;
```

Like the previous example, but we now show three options.

```
VAR feedback : TEXT;  
SET feedback = PROMPT "How was the acquisition time?"  
BUTTONS  
SHOW "short" AS "Too fast",  
SHOW "long" AS "Too slow",  
SHOW "fine" AS "Okay"  
DEFAULT "fine"  
ICON "help_bubble";
```

RAISE

The RAISE statement allows the author of the Automation Script to cause a named error condition to occur which can be handled by an ON ERROR statement. The occurrence of an error condition, either intentionally with the use of this statement or because of a programmatic error, is known as an error being “raised”. See the prior description of the ON ERROR statement.

The syntax of the RAISE statement is:

```
RAISE error-code WHEN expression;
```

Error Codes

The *error-code* is a String naming the error to be raised. It can be one of the standard error codes described in the ON ERROR section or a String of the author’s whim. For the sake of other user’s understanding the script, it is recommended that an *error-code* String be defined to describe the error condition as best as possible.

Although the *error-code* is a String literal, the instrument operator will not see the value of the *error-code*. If you want to inform the operator of the problem, the author should use the INFORM statement prior to raising the error condition.

The When Clause

If a WHEN clause is specified after the *error-code*, the error condition is raised only when the *expression* evaluates to a True value. Refer to the section titled ‘Boolean Expressions’ for how an expression evaluates to a Boolean value.

The *expression* can also be written to obtain a Job attribute, a Sample attribute, or a Namespace value. If the value obtained is not one of the False values, the *error-code* will be raised.

```
RAISE WHEN JOB attribute expression;  
RAISE WHEN SAMPLE attribute expression;  
RAISE WHEN NAMESPACE path expression;
```

Attribute and *path* must be a quoted String literal value or a variable of the TEXT Type. If *expression* is included, it must be written such that the value of the Job, Sample, or Namespace part is the first value of the conditional test. For example, SAMPLE “count” > 0. If an *expression* is not provided, then the value of the *attribute* or Namespace *path* alone will determine the truth state.

Examples

Examples of the use of the RAISE statement which assume a NUMBER Type variable has been defined are:

```
RAISE “type-error”;  
  
RAISE “problem” WHEN x = 0;  
  
RAISE “range-error” WHEN x < 1 or x > 10;  
  
RAISE “sample-error” WHEN SAMPLE “error”;  
RAISE “counter zero” WHEN JOB “count” <= 0;
```

REMARK

The REMARK statement is a comment. There is no action performed by this statement. It is entirely ignored by Automation. All characters that follow the REMARK keyword on the same line (up to the carriage return or new-line character at the end of the physical line of text) are considered part of the comment. Because of this, the REMARK statement does not have to be terminated with a semicolon like the other statements and blocks in the Automation Script.

Use the REMARK statement to document the Script for other users who may wish to read the Script. Script authors use the REMARK statement to describe parts of the Script that may be unclear. Provide as much detail as necessary so that other people can understand the intent of the Script.

Example

The following line is an example of the REMARK statement.

```
|REMARK This is just a comment line.
```



WARNING! It is also possible to “comment out” text in an Automation Script by preceding the comment text with two consecutive dashes, --. The default Script header text is commented using this form. The text after and including the two dashes is discarded by Automation and thus is not preserved when the Automation Script is re-written to a file. That is, if a user loads an Automation Script containing text following two dashes, that text will not be written back to the file when the script is saved again.

REPEAT

The various forms of the REPEAT block execute a sequence of statements zero or more times. The number of repetitions is determined by the conditions stipulated by the block and the current state of Automation at the point of execution. The syntax of the REPEAT block is:

```
REPEAT loop-criteria DO
    statement-block
THEN
    then-statement-block
END REPEAT;
```

The REPEAT block syntax provides multiple ways that the statements in the *statement-block* can be repeated. The *loop-criteria* (or its absence) specify how the REPEAT block will repeat the statements of the *statement-block*. The following list provides an overview of ways that repetition can occur. The statements in the *statement-block* can be repeated:

- until an EXIT statement is encountered or until one of the statements causes a programmatic error (when the *loop-criteria* is omitted).
- a specific number of times.
- once for each item in a list (or variable of the LIST Type).
- until a stop condition that is specified in the *loop-criteria* is met.

The Then Block

Immediately after the successful completion of the *statement-block*, the statements in the *then-statement-block* will execute one time. The *then-statement-block* will only execute if all of the following conditions are satisfied: each of the statements of the *statement-block* must be executed at least once, the loop did not stop early by encountering an EXIT statement, and no error condition was raised during the execution of the *statement-block* that would cause it to be interrupted.

Indefinite Repetition

To execute a series of statements a unspecified number of times, use the simplest syntax:

```
REPEAT
    statement-block
END REPEAT;
```

The *statement-block* in the above syntax will continue to repeat until either an EXIT, FINISH, or TERMINATE statement is executed, or a programmatic error occurs.

WARNING! It is possible to create a loop that never ends (known as an “infinite loop”) using this statement. An infinite loop is a REPEAT block that does not stop running because the conditions for termination are never met. It is very important to always write a provision for ending the REPEAT block and thus preventing the statements in the *statement-block* from repeating forever.



The remaining descriptions will detail the syntax of the first line of the REPEAT block – specifically how to specify the *loop-criteria*.

Repeating a Specific Number of Times

To execute a series of statements a specific number of times, use the syntax:

```
REPEAT number TIMES DO...
```

The *number* specified must be a non-negative integer or a variable containing a NUMBER Type. The fractional component of the numeric value of a variable will be ignored. In the following example, the *statement-block* would execute only 5 times.

```
VAR x : NUMBER = 5.9;
```

```
REPEAT x TIMES DO...
```

Repeating a Specific Number of Times with Options

This form of the REPEAT block is called the “REPEAT TO” form because it repeats the statements from a specific value **to** another specific value. The following syntax is used to repeat a sequence of statements a specific number of times:

```
REPEAT loop-counter
      FROM from-expression
      TO   to-expression
      STEP step-expression DO...
```

The number of repetitions could be determined at the time the REPEAT block is executed if variables are used in any of the expressions following the FROM, TO, and/or STEP keywords. The actual number of repetitions could potentially be less than the specified number if an EXIT statement is encountered that would terminate the loop before all of the repetitions take place.

A special variable called a “loop counter” can be specified at the beginning of the loop after the REPEAT keyword, if necessary. The author of the Automation Script provides the name for this variable in the place of *loop-counter*. The loop counter variable is defined to be of the NUMBER Type and will contain the current iteration value for each repetition of the *statement-block*.

The *from-expression*, *to-expression*, and *step-expression* are either constant numeric values, expressions that result in numeric values, or variables of the NUMBER Type.

From-expression sets the initial value of the *loop-counter* for the first iteration.

To-expression specifies the stopping condition for the REPEAT block. When the value of *loop-counter* surpasses the value of *to-expression* the loop will terminate successfully. Note that the *loop-counter* may be counting down if *step-expression* is negative.

Step-expression specifies the amount of change of the *loop-counter* value after each repetition of the *statement-block*. If *from-expression* is specified to be 1 (one) and *step-expression* is specified to be 2 (two) then the *loop-counter* variable will hold the values 1, 3, 5, 7, etc. for each repetition.

The *from-expression* and *step-expression* both default to the value of 1 (one) when they are omitted. The TO keyword and *to-expression* are the only required elements of this syntax. If the value of *from-expression* is **less than** the value of *to-expression* and the value of *step-expression* is negative then the statements in the *statement-block* will not execute. The converse is similar – the *statement-block* will not execute if the value of *from-expression* is **greater than** the value of *to-expression* and the value of *step-expression* is positive.

NOTE: Setting the value of *step-expression* to be 0 (zero) would create a loop that would never end (known as an “infinite loop”). Therefore, the REPEAT block will execute exactly once when the value of *step-expression* is detected to be 0.

Examples

The following loop will repeat 4 times and *level* will initially contain the value 1 during the first iteration, then 2 during the second, 3 during the third, and finally 4 during the fourth and last repetition.

```
REPEAT level TO 4 DO
    ...
END REPEAT;
```

In the next example, the loop will also repeat 4 times but *level* will initially contain the value 5 during the first iteration, then 10 during the second, 15 during the third, and finally 20 for the fourth and last repetition.

```
REPEAT level FROM 5 TO 20 STEP 5 DO
    ...
END REPEAT;
```

The next example will repeat the statements 3 times and *level* will initially contain the value 5 during the first iteration, then 3.5 during the second, and 2 during the third and final iteration.

```
REPEAT level FROM 5 TO 1 STEP -1.5 DO
    ...
END REPEAT;
```

Note that in the example above, the *statement-block* never executed with the *level* variable equal to the actual specified TO value of 1 (one). This is because the fourth iteration would have subtracted 1.5 from 2 leaving 0.5. Since 0.5 is less than our criteria for stopping, the loop exits.

Repeating Over a List of Values

Another way to repeat a sequence of statements is with the “REPEAT IN” form. This syntax is given this name because it repeats the *statement-block* once for each value that is **in** a list. Here is the syntax:

```
REPEAT item IN value-list DO...
```

This syntax instructs Automation to repeat the *statement-block* one time for each value contained in the *value-list*. For example, if the *value-list* had three values in it, the loop would repeat three times. *Value-list* can be a constant LIST Type value or a variable that holds a LIST value.

Like the “REPEAT TO” syntax discussed earlier which has a *loop-counter* variable that is available to access during each repetition of the loop, this syntax has an *item* variable that will hold the value of each item in the *value-list* during each iteration. The *item* variable will contain each consecutive value in the *value-list* in sequence. It is important to realize that because it is possible for the basic LIST Type to contain many distinct values of various Types simultaneously, the Type of

the *item* variable could potentially have a different Type from one iteration of the loop to the next. The *statement-block* will execute the same number of times as there are elements in the *value-list*.

Examples

An example of this form of the REPEAT block is:

```
REPEAT n IN {1, 2, 5, 10, 20} DO
  ...
END REPEAT;

VAR power2 : LIST = {2, 4, 8, 16};
REPEAT n IN power2 DO
  ...
END REPEAT;
```



WARNING! The Type of the *item* variable may change from one iteration to the next if the list variable *value-list* is not homogeneous, meaning that the variable contains values of varying Types.

Repeating Conditionally

The “REPEAT WHILE” form of the REPEAT block repeats a sequence of statements **while** a condition is True. It has the following syntax:

```
REPEAT WHILE expression DO...
```

The *expression* of the “REPEAT WHILE” form is evaluated **prior** to each iteration of the loop. If the result is True, the loop continues. If the result is False, the loop is finished and if a *then-block* is provided, the statements in the *then-block* will be executed. It is possible for the *statement-block* to be skipped entirely if the *expression* evaluates to False prior to the first iteration. The *then-block* will **not** execute if the loop terminates early by encountering an EXIT statement in the *statement-block*.

The “REPEAT UNTIL” form repeats a sequence of statements **until** a condition becomes True. It has the following syntax:

```
REPEAT UNTIL expression DO...
```

The *expression* of the “REPEAT UNTIL” form is evaluated **after** each repetition of the loop. If the result is False, the loop continues. If the result is True, the loop is finished and if a *then-block* is provided, the statements in the *then-block* will be executed. Since the *expression* is evaluated **after** each iteration, the statements in *statement-block* will execute at least once. Like the “REPEAT WHILE” form, the *then-block* will **not** execute if the loop terminates early by encountering an EXIT statement in the *statement-block*.

The *expression* in each of these two forms of the REPEAT statement result in a truth value. Refer to the section titled ‘Boolean Expressions’ for how an expression will evaluate to a Boolean value. Optionally, the *expression* can begin with or be replaced by a Job attribute, a Sample attribute, or a Namespace value.

```
REPEAT while-or-until JOB attribute expression DO...
REPEAT while-or-until SAMPLE attribute expression DO...
REPEAT while-or-until NAMESPACE path expression DO...
```

Attribute and *path* must be a quoted String literal value or a variable of the TEXT Type. If *expression* is included, it must be written such that the value of the Job, Sample, or Namespace part is the first value of the conditional test. For example, SAMPLE "count" > 0. If an *expression* is not provided, then the value of the *attribute* or Namespace *path* alone will determine the truth state.

Please read the warning about how to avoid writing infinite loops in the prior section titled, "Indefinite Repetition".

Examples

Examples of the conditional form of the REPEAT block follow. The examples are based on the initial condition that the value of the variable x is set to 1 (one) before the loops execute.

```
VAR x : NUMBER;
```

The value of the variable x will equal 5 after executing the following example.

```
SET x = 1;
REPEAT WHILE x <= 5 DO
  -- x will be 1, then 2, 3, 4, 5
  ...
  SET x = x + 1;
END REPEAT;
```

The value of the variable x will not be less than 5 after executing the next example. The value of x (from the Job attribute "reps") will not be changed if it is already greater than 5.

```
SET x = JOB "reps";
REPEAT WHILE JOB "reps" <= 5 DO
  ...
  SET x = x + 1;
  SET JOB "reps" = x;
END REPEAT;
```

The value of the variable x will equal 0 after executing this last loop.

```
SET x = 5;
REPEAT UNTIL x = 0 DO
  -- x will be 5, then 4, 3, 2, 1
  ...
  SET x = x - 1;
END REPEAT;
```

RETAIN

The RETAIN statement informs Automation that a file should not be deleted when the variable holding the file goes out of scope. The syntax of the RETAIN statement is simply:

```
RETAIN file-variable SAVE AS string-or-variable;
```

The *file-variable* is an identifier of the DATA Type (or LIST Type if the variable is indexed) that indicates which file is to be preserved. This statement becomes necessary when data files are generated and are the result of executing Percival code using the CALL statement. Other potential uses include preserving generated plot files (from the PRINT and PRESENTATION statements that omit their own SAVE AS clause).

The SAVE AS clause is optional and, when provided, specifies the name with which the file is to be preserved. When the SAVE AS clause is omitted, the name of the *file-variable* is used for the name of the file.

Examples

The following are examples of the use of the RETAIN statement.

```
| RETAIN myFile;
```

```
| RETAIN myFile SAVE AS "option1";
```

RETRY

The RETRY statement informs Automation that the next statement to be run after the RETRY statement completes will be the same statement that raised the error condition. The RETRY statement can only be used within an error handler. See the prior description of the ON ERROR statement. The syntax of the RETRY statement is simply:

```
RETRY WHEN expression;
```

After an error handler completes without the RETRY statement, execution of the Automation Script normally resumes with the statement that follows the one that raised the error condition. The RETRY statement changes this behavior by causing the statement that raised the error to be executed again unless the statement that raised the error is a RAISE statement.

The When Clause

If the WHEN clause is specified, this statement only influences the execution path if the *expression* evaluates to a True value. Refer to the section titled 'Boolean Expressions' for how an expression evaluates to a Boolean value.

The *expression* can also be written to obtain a Job attribute, a Sample attribute, or a Namespace value. If the value obtained is not one of the False values, the block will exit.

```
RETRY WHEN JOB attribute expression;  
RETRY WHEN SAMPLE attribute expression;  
RETRY WHEN NAMESPACE path expression;
```

Attribute and *path* must be a quoted String literal value or a variable of the TEXT Type. If *expression* is included, it must be written such that the value of the Job, Sample, or Namespace part is the first value of the conditional test. For example, SAMPLE "count" > 0. If an *expression* is not provided, then the value of the *attribute* or Namespace *path* alone will determine the truth state.

IMPORTANT: It is crucial to remember that if the RETRY statement is included in an error handler, it is *essential* that the other statements in the same error handler change the programmatic conditions sufficiently such that the same statement that originally caused the error will not raise the same error condition again. Care must be taken to avoid infinite retry attempts.

Examples

The following are examples of the use of the RETRY statement.

```
NUMBER Small IS 1 TO 99;
```

```
VAR n      : Small = 10;
```

```
VAR tries : NUMBER = 5;
```

```
ON ERROR ALL DO
  INFORM "Oops!";
  SET tries = tries - 1;
  RETRY WHEN tries > 0;
END ERROR;
```

```
ON ERROR "range-error" DO
  SET n = 1;
  RETRY;
END ERROR;
```

```
...
```

```
SET n = n * 10;
```


SET

The SET statement most commonly assigns a value to a variable. It can also be used to set a value to a machine or experiment parameter, to a Namespace entity, or to an attribute of the Job or current Sample. A SET statement begins with the keyword SET and is immediately followed by an optional keyword that specifies where the value is to be stored. This is then followed one or more assignments which identify the entity or entities for which the value(s) will be associated and provides the new value(s). The syntax of the SET statement is as follows:

```
SET mode assignment-expression;
```

When assigning a value to a variable, the variable must have been previously defined with a VAR statement. Refer to the description of the VAR statement. Assignments can also be made to a Method parameter that is designated with one of the OUT or INOUT keywords. Refer to the description of Method parameters in the prior discussion of the METHOD block.

Setting the Destination of the Assignment

When SET is used outside of an EXPERIMENT block, *mode* can be one of the keywords VAR, MACHINE, NAMESPACE, JOB, or SAMPLE. If a *mode* is not specified, the statement assumes that you are setting the value of a variable and uses the VAR *mode*.

When SET is used within an EXPERIMENT block, *mode* can be optionally specified with the PARAMETER keyword. There is no other choice except to set Experimental parameters when using the SET statement within an EXPERIMENT block.

Either of the keywords PARAMETER or the plural PARAMETERS may follow the MACHINE keyword for readability.

Assignment Expressions

There may be one or more *assignment-expression* in the SET statement and each must be terminated by a semicolon. More than one *assignment-expression* in a SET statement is referred to as an “assignment block” (see below). An *assignment-expression* has the following form:

```
name = new-value;
```

Depending on *mode*, *name* specifies where the *new-value* will be set. The following table describes the possibilities.

When <i>mode</i> is...	<i>name</i> is...
VAR (or omitted <u>outside</u> an EXPERIMENT block)	An identifier that names a previously defined variable or Method parameter specified to be OUT or INOUT.
PARAMETERS (or omitted <u>within</u> an EXPERIMENT block)	An identifier that names a parameter of an experiment that will be set in the pulse program prior data acquisition.
MACHINE PARAMETERS	A String or variable that identifies an established NMR instrument Namespace parameter that exists within the parameter.jnd file.

When <i>mode</i> is...	<i>name</i> is...
NAMESPACE	A String or variable name that identifies a path into the Namespace parameter database. Setting the default value to NULL will remove the Namespace parameter.
JOB	A String or variable name that identifies an attribute of the Job Group.
SAMPLE	<p>A String or variable name that identifies an attribute of the current Sample. There are four keywords (SAVE, INTERIM, CONCEAL, and CONST) that affect how the Sample attributes are stored when using the SAMPLE <i>mode</i>. Either SAVE or INTERIM can be specified since those two keywords are mutually exclusive.</p> <ul style="list-style-type: none"> • Use the SAVE keyword to cause the attribute value to be stored permanently in the Sample Database. It is recommended that this keyword be used to store information that is learned about the sample from experimentation and processing. • Use the INTERIM keyword to cause the attribute value to be stored temporarily in the Sample Database. This attribute will be removed when the Sample is ejected from the instrument. It is recommended that this keyword be used to set attributes that affect behaviors that need to persist from Job to Job, but must be reset when the operator is finished with the Sample. • Use the CONCEAL keyword to prevent this Sample attribute from appearing on the User Interface. Specifying CONCEAL implicitly includes the SAVE behavior. • Use the CONST keyword if the attribute is to be immutable – that is, the user must not be able to modify the attribute value. Specifying CONST implicitly includes the SAVE behavior.

Assignment Blocks

Although it is possible for the author of the Script to write each variable assignment with its own SET statement, it is often preferable (especially when *mode* is set to JOB or SAMPLE) to assign as many values as possible with a single assignment block. The reason for using an assignment block is that all the values that will affect the instrument state will be applied together at the end of the assignment block. This allows the parameters to obey the semantic order for applying their new values to the instrument. The semantic rules for parameter order cannot be enforced when individual SET statements are used to assign each variable.

Providing the New-Value

The *new-value* in the *assignment-expression* can be specified in one of the following ways:

```

constant
JOB job-attribute ELSE constant
SAMPLE sample-attribute ELSE constant
NAMESPACE namespace-path ELSE constant
EVALUATE (expression) ELSE constant
PROCESS data-variable WITH processing-list

```

Variables of the DATA Type cannot use the JOB, SAMPLE, NAMESPACE, or EVALUATE clauses shown above to set their value. Only a constant String or an expression that results in a TEXT Type is a legal new value for a DATA Type variable.

Constant is a value that has a Type that is appropriate for the variable that is being assigned.

Job-attribute is a String or variable of the TEXT Type that represents the attribute of interest on the running job.

Sample-attribute is a String or variable of the TEXT Type that represents the attribute of interest on the current sample.

Namespace-path is a String or variable of the TEXT Type representing a Namespace path to a stored value in the Namespace parameter database.

Expression is a Percival code expression that will be evaluated.

More than one Job, Sample, Namespace, or Evaluate clause may be specified when they are separated by an ELSE keyword before the optional final ELSE clause.

The optional ELSE clause specifies a specific default value for the case when the *job-attribute*, *sample-attribute*, or *namespace-path* cannot be found or when those values or the evaluation *expression* cannot be resolved to an appropriate value of the required Type. If the ELSE clause is omitted, an error condition could be raised if the new value does not conform to the requirements of the variable being assigned.

Casting a Value to the Proper Type

When assigning values to variables, the author must make sure that the Type of the variable used to store the new value is compatible with the actual type of the new value. In some situations, we may not know the Type of the value being assigned because it comes from an external source. Sample attributes are a good example of this because Type rules are not enforced on them. If a Sample attribute is intended to be a TEXT Type but the value looks like a number, it will be stored as a NUMBER Type in the Sample. This means that we should *cast* the value from the Sample to the Type that we want it to be. We do this explicitly by adding the keyword CAST immediately after the equal sign of the assignment. The CAST keyword will try to convert the new value to the Type of the variable that is being used to hold the new value.

Casting a value to a Type is only permitted when assigning from a variable to another variable because it is the Type of the variable that determines what the casting process will do. Casting a constant value is not permitted because the value is known and so it can be written in the proper form.

See the section titled “Value Type Casting” to learn how values can be interpreted.

Obtaining a Value by Calling a Percival Program or Service

In addition to the previously described ways to provide a *new-value* for a variable, the author of the Script can also call a Percival program or call a service:

```
CALL percival-program( parameters )
CALL SERVICE "provider::operation"( parameters )
```

The result of the CALL operation should return a value. As usual, the Type of the variable to which the result of the CALL will be assigned must be the appropriate Type. If a value is not explicitly returned by the Percival program or service, the Boolean value TRUE will be assigned to the variable.

Some Percival programs and some services return Status Codes as a resulting value. Status Codes are translated into Boolean values for Automation. The Success and Complete Status Code is translated into a TRUE value and all other Status Codes are translated into a FALSE value.

Refer to the prior explanations of the CALL statement for more details.

Wildcards (*) – Setting more than one value simultaneously

The author may use a name-value pair LIST Type variable (called an association) to assign values to multiple variables and/or parameters. An association has the following form:

```
{ {name1, value1}, {name2, value2}, ... {namen, valuen} }
```

The *new-value* can also be set from Namespace if the *namespace-path* provided results in a Namespace association value. In this form of assignment, the *name* must be specified with an asterisk (*). When *mode* is set to NAMESPACE, no special action is taken.

When using wildcard assignment within an Experiment block, an acquisition parameter will be created if the *name_n* does not exist. Outside of an Experiment block, any *name_n* that is not recognized to be a pre-declared Automation variable will be ignored.

Assigning values into LIST Type variables

The author can assign a value into a LIST Type variable by specifying the position of the List element as an integer number after the variable within parenthesis.

```
lst(position) = new-value;
```

To modify the value of an item in the list, specify *position* to be a number between 1 and the length of the list. To add a number to the front of the list, specify *position* to be 0 (zero). To add a number to the end of the list, specify *position* to be a number greater than the length of the list.

A special LIST sub-Type called an Association is a bit different. Numerical positions are not used to specify the element to modify like a typical LIST Type, but rather, the element to change is specified by name. To assign a value to an element of an association List, specify the name of the element as a String after the variable within parenthesis.

```
lst(name) = new-value;
```

To modify an existing value in the Association, specify *name* to be the name of the item to change. To add a value to the Association, specify *name* to be a new name that is not currently in the Association. To remove a value from the Association specify *name* to be the name of the item to remove and also specify *new-value* to be NULL.

Removing a Sample Attribute

Sample attributes can be removed by assigning them the NULL value. A special syntax is also provided for this by specifying the keyword REMOVE after SAMPLE then followed by one or more Sample attribute Strings separated by the keyword AND or commas.

```
SET SAMPLE REMOVE "my_attribute";  
SET SAMPLE REMOVE "attr1" AND "attr2";
```

Examples

Examples of the use of the SET statement follow. These examples assume that all the referenced variables are defined. The following statement will set the value 5 to the variable x if x is of the NUMBER Type. Otherwise, it will raise an error condition if 5 is not a valid value for the Type of x.

```
| SET VAR x = 5;
```

The following statement assigns values to multiple variables.

```
| SET  
s = SAMPLE "id";  
b = TRUE;
```

The following statement sets the variable x to be the value of the Namespace path. If the Namespace value is not an appropriate value for x, then the value 25 will be assigned to x.

```
| SET x = NAMESPACE "my_preference" ELSE 25;
```

The following statement sets the variable x to the value that is twice the value of n or it will raise an error condition if n is not of the NUMBER Type or if the product is not a valid value for the variable x.

```
| SET x = n * 2;
```

The following statement sets the value of an instrument parameter.

```
| SET MACHINE "spin_state" = "SPIN ON";
```

Assuming a variable named param exists and it is defined to hold a TEXT Type value and that value is the name of a machine parameter and further that another variable named value exists that contains the new value for the parameter, the author can write:

```
| SET MACHINE PARAMETER param = value;
```

The following statements set values to the Job scope and Sample scope respectively.

```
| SET JOB "discovered" = TRUE;  
| SET SAMPLE "peak_at" = 28.1449[Hz];
```

The following statements set or create attributes on the current sample. However, specifying one or more of the special keywords after SAMPLE indicates that this action will store the attribute into the Sample database.

```
SET SAMPLE INTERIM "tint" = "blue";
SET SAMPLE SAVE CONST "mode" = 43.5;    -- SAVE is not required here but allowed
SET SAMPLE CONCEAL "secret" = TRUE;
```

The following statement sets (or creates) a Namespace path to hold a parameter of a data file. This example also illustrates the ability to specify the Namespace path using a substitution identifier as specified within the String surrounded with parenthesis and preceded with the dollar symbol, \$. See the Table of Substitution Identifiers for a list of words that can be used. In this example, the username of the person who is running this script will be inserted into the path. If the name of the user is "NMRguy", the Namespace path will be "AUTOMATION.NMRguy.x_sweep".

```
SET NAMESPACE "AUTOMATION.$(user).x_sweep" = dat("x_sweep_clipped");
```

The following statement assigns the result of an expression to the variable area.

```
SET area = EVALUATE (pi * radius**2);
```

The following statement assigns d the data that results from processing e with the processing list named "ldh". d and e are previously defined variables of the DATA Type.

```
SET d = PROCESS e WITH "ldh";
```

The following examples assign the result of the Percival program my_op and the service call to the variables p and s respectively. p and s must be defined to be the same Type that the Percival program and the service returns.

```
SET p = CALL my_op( 5 );
SET s = CALL SERVICE "provider::service"( TRUE );
```

The following EXPERIMENT block sets the values of a few experimental parameters prior to acquiring data.

```
EXPERIMENT
...
SET PARAMETERS
    scans = 16;
    x_points = 2**n;
END EXPERIMENT;
```

The following statements show assigning to Lists and Associations. Each statement example operates on the value as set by the following VAR statements.

```
LIST Assoc IS ASSOCIATION;
```

```

VAR c : LIST = {"red", "green", "blue"};
VAR a : Assoc = [{"one",1}, {"two",2}];

SET c(1) = 1;  --c changed to {1, "green", "blue"}
SET c(0) = 1;  --c changed to {1, "red", "green", "blue"}
SET c(4) = 1;  --c changed to {"red", "green", "blue", 1}

SET a("one") = "first";  --a changed to [{"one","first"}, {"two",2}]
SET a("new") = 3;        --a changed to [{"one",1}, {"two",2}, {"new",3}]
SET a("two") = NULL;     --a changed to [{"one",1}]

```

The following statements show how it is possible to set multiple values based on an association List. An association List is effectively a List of zero or more sub-Lists where each sub-List contains two elements the first of which must be of the TEXT Type.

```

LIST Assoc IS ASSOCIATION;

VAR my_values : Assoc = NAMESPACE "samples.initialize" ELSE {};

SET * = my_values;

SET SAMPLE * = NAMESPACE "samples.initialize";

SET NAMESPACE * = my_values;

```

Each value of the keys of the association will be assigned to the variable of the same name using the first syntax, a Sample attribute with the same name using the second syntax, or a Namespace path using the third syntax. In the case of Sample attributes and Namespace paths, if the attribute or path does not exist, a new one will be created.

The three examples above may be used outside of an EXPERIMENT block, but only the first of these examples may be used within an EXPERIMENT block.

It is therefore possible to obtain the new value for a variable by using a variety of techniques and there are many possibilities for where the new value can be set.

WARNING! An attempt to assign a new value to a variable that is not of a compatible Type will raise the error condition “type-error”. Similarly, an attempt to assign a value that is out of range of a constrained sub-Type will raise the error condition “constraint-error”. If there are no error-handling blocks to handle these errors when they occur, the Method will terminate immediately with the error status that was raised. Refer to the prior discussion of the ON ERROR statement.



TERMINATE

The TERMINATE statement immediately stops the execution of the Method. Any statements following this statement will not be executed. The TERMINATE statement has the following syntax:

```
TERMINATE WITH STATUS message WHEN expression;
```

Any file(s) that had been created (acquired data, processed files, and printed files) by the Method up to this statement will be deleted. The previously described FINISH statement has a similar function but will preserve the files generated by the Method.

The Status Message

An optional reason can be provided when a Script is going to be terminated. The specified *message* should be a short explanation for why the script is being terminated prematurely. When the Script completes, the *message* can be displayed on the user interface and it will be logged. *Message* can be a quoted String literal, a variable of the TEXT Type, or translation identifier. The WITH keyword is optional and can be provided to enhance readability.

The When Clause

If the WHEN clause is specified, this statement only has an effect if the *expression* evaluates to a True value. Refer to the section titled 'Boolean Expressions' for how an expression evaluates to a Boolean value.

The *expression* can also be written to obtain a Job attribute, a Sample attribute, or a Namespace value. If the value obtained is not one of the False values, the Method will terminate.

```
TERMINATE WHEN JOB attribute expression;
TERMINATE WHEN SAMPLE attribute expression;
TERMINATE WHEN NAMESPACE path expression;
```

Attribute and *path* must be a quoted String literal value or a variable of the TEXT Type. If *expression* is included, it must be written such that the value of the Job, Sample, or Namespace part is the first value of the conditional test. For example, SAMPLE "count" > 0. If an *expression* is not provided, then the value of the *attribute* or Namespace *path* alone will determine the truth state.

Examples

The following two examples of the TERMINATE statement produce the same result and both assume that a variable named `err` has been previously defined and set appropriately.

```
VAR err : BOOLEAN = FALSE;
```

The simplest form contains the TERMINATE keyword itself.

```
IF err THEN
|   TERMINATE;
END IF;
```


The following does the same as the example above but also provides a reason.

```
IF err THEN  
|   TERMINATE WITH STATUS "No data acquired!";  
END IF;
```

The following line has the same effect as the IF block in the previous example.

```
| TERMINATE WITH STATUS "No data acquired!" WHEN err;
```

The following line terminates the Method if the Sample has an "error" attribute that evaluates to a True value.

```
| TERMINATE WHEN SAMPLE "error";
```

TRANSLATE

The purpose of the TRANSLATE statement is to provide language translations for the text elements of Automation Script statements that will be readable by users on the user interface. The syntax of the TRANSLATE statement is:

```
TRANSLATE name language-code string-list;
```

The *name* following the TRANSLATE keyword is a unique identifier within the Script. It will be used in the place of String elements of the Automation Script syntax to indicate the necessary translation based on the locale of the user.

The *string-list* is a series of comma-separated Strings. This text should be written in the language designated by the *language-code*.

Language Codes

The *language-code* is the standardized code consisting of two or three letters that represents the language in which the following text translation is written. The standard language codes can be found on the Internet at the Library of Congress (the definitive resource):

http://www.loc.gov/standards/iso639-2/php/English_list.php

Default Language Translations

If *language-code* is omitted, or if it is specified as ALL, the text becomes the default translation. The default translation will be used when a specific translation (as specified by the user's locale preference) is not available. If a default translation is not specified and an English translation is provided, English will become the default translation. Otherwise, the first translation provided will be used as the default if there is no English translation and no designated default.

Examples

The author of the Script can use separate TRANSLATE statements for each language or the different languages may be combined for the same *name* identifier. The examples below show both of these forms. *The author apologizes for any bad translations – blame "Babel Fish".*

```
TRANSLATE Proton_Help EN "Acquire Proton data";
TRANSLATE Proton_Help ES "Adquiera los datos del protón";
TRANSLATE Proton_Help ALL "Proton acquisition";
                        ES "Adquiera los datos del protón";
                        ITA "Acquisti i dati del protone";
                        FR "Acquérez les données de proton";
TRANSLATE paper_print EN "Print result on paper?",
                        "Output will go to default when TRUE.";
                        ES "¿Resultado de la impresión en el papel?",
                        "La salida irá a omitir cuando es VERDAD.";
```

Common Language Codes

Language	Code	Language	Code
Chinese	ZH, CHI, ZHO	Italian	IT, ITA
English	EN, ENG	Japanese	JA, JPN
French	FR, FRE, FRA	Russian	RU, RUS
German	DE, GER, DEU	Spanish	ES, SPA
Greek	EL, GRE, ELL	Swedish	SV, SWE

TUNE

The TUNE statement attempts to tune a specific aspect of the spectrometer. At this time, only the Probe may be tuned. The syntax of the TUNE statement is:

```
TUNE PROBE FORCE Boolean-or-variable DUAL Boolean-or-variable
      COIL string-or-variable
      DOMAIN string-or-variable
      OFFSET number-or-variable;
```

The *string-or-variable* following the COIL keyword specifies the coil to be tuned. Possible values for the COIL are: "LOCK", "HF1", "HF2", "LF1", "LF2", "FGX". There may be more or fewer of available COIL values depending on the configuration of the NMR instrument.

The *string-or-variable* following the DOMAIN keyword specifies the nucleus to be tuned. Examples of values for DOMAIN are: "Proton", "Deuterium", "Fluorine19". The value of the COIL that is set determines the set of valid options for the DOMAIN.

The *Probe Tool* can assist in discovering the available COIL and DOMAIN values for your instrument. Choosing "Probe Tool" from the "Config" menu in the *Spectrometer Control* window will open it.

The *number-or-variable* following the OFFSET is a number with a unit specifying the offset at which the tune should be accomplished.

The FORCE keyword will ensure that the probe is tuned again even when the system thinks that it is already tuned. If the keyword is omitted, then the TUNE statement will be skipped if the probe has already been tuned. If the optional *Boolean-or-variable* following FORCE is omitted then the force state is True. If not specified, then the force state will be set to the value provided.

The DUAL keyword will allow dual-tune probes to be tuned to multiple domains. If the keyword is omitted, then the TUNE statement will only TUNE to a single domain. Similarly, the *Boolean-or-variable* can be provided to set the state of the dual mode.

The author can specify any number of COIL / DOMAIN / OFFSET triplets (up to the physical limit of the instrument) so long as all three keywords are used the same number of times.

Examples

Examples of the use of the TUNE statement follow:

```
| TUNE PROBE COIL "LF1" DOMAIN "Carbon13" OFFSET 2.5[ppm];

VAR c : TEXT;
VAR d : TEXT;
EXPOSE VAR toffset : NUMBER = 0[Hz], HELP "Tuning offset";
EXPOSE VAR always : BOOLEAN = FALSE, HELP "Force tune?";
...
REMARK - Set the values of c and d here.

| TUNE PROBE FORCE always COIL c DOMAIN d OFFSET toffset;

| TUNE PROBE
  COIL "LF1" DOMAIN "Carbon13" OFFSET 2.5[ppm]
  COIL c      DOMAIN d      OFFSET 100[Hz];
```

VAR

A variable is a named value like a constant except that its value is mutable (it **can** be modified during the execution of the Automation Script). Refer to the prior descriptions of the CONST and SET statements. A variable must be defined before it can be referenced and it can optionally have its value set to a specific initial value at the time it is created. The VAR statement may have one of the following syntax forms:

```
VAR var-name : value-type = default-value;
```

```
EXPOSE mode VAR var-name : value-type = default-value
      WHEN (Boolean-expression) , var-options , HELP help-text;
```

The *var-name* is an identifier that uniquely names the variable. More than one variable, constant, or Method parameter with the same name cannot exist in the same scope level. A variable, constant or Method parameter with the same name at an outer scope level will be hidden (or eclipsed) by the new variable.

Value-type may be any of the five basic Types defined in the Automation grammar (BOOLEAN, NUMBER, TEXT, LIST, or DATA) or it may be a sub-Type defined prior to the VAR statement using the ENUM, NUMBER, or LIST statements.

The *default-value* can be specified in one of the following ways:

```
constant
JOB job-attribute ELSE constant
SAMPLE sample-attribute ELSE constant
NAMESPACE namespace-path ELSE constant
EVALUATE (expression) ELSE constant
```

Variables of the DATA Type cannot use the JOB, SAMPLE, NAMESPACE, or EVALUATE clauses shown above to set the initial value. Only a constant String or an expression that results in a TEXT Type value is a legal form for initializing a DATA Type variable. More than one Job, Sample, Namespace, or Evaluate clause may be specified when they are separated by an ELSE keyword before the optional final ELSE clause.

WARNING! The use of an expression to initialize a variable could potentially raise a Type-Error exception at run-time if the evaluation of the expression does not result in a value of the correct Type.



Description

Constant is a value that is of the Type *value-type*. It must be of the same Type as the variable being defined. If *constant* is not appropriate for *value-type*, the error condition “type-error” will be raised when the statement attempts to set the initial value of the variable.

Job-attribute is a String or variable representing the attribute of interest of the currently running Job. *Sample-attribute* is a String or variable representing the attribute of interest of the current Sample. *Namespace-path* is a String or variable representing the location of a value in the Namespace Parameter Database. *Expression* is a Percival code expression that will be evaluated.

The optional ELSE clause provides a specific default value for the cases when the *job-attribute*, *sample-attribute*, or *namespace-path* cannot be found or when those values or the evaluation *expression* cannot be resolved to an appropriate value of the required Type. If the ELSE clause is omitted or if the *constant* value is not valid for the Type of the variable, an error condition (normally “type-error”) will be raised.

Changing the Value of a Variable

The SET statement is used to modify a variable’s value. Refer to the prior description of the SET statement to learn how to change the value of a variable that is already defined.

Casting the Initial Value to the Proper Type

The author must make sure that if a default value is provided, it is of the proper Type of the variable being defined. In some situations, we may not know the Type of the value being used as the initial value because it comes from an external source. Sample attributes are a good example of this because Type rules are not enforced on them. If a Sample attribute is intended to be a TEXT Type but the value looks like a number, it will be stored as a NUMBER Type in the Sample. This means that we should *cast* the value from the Sample to the Type that we want it to be. We do this explicitly by adding the keyword CAST immediately after the equal sign of the assignment. The CAST keyword will try to convert the new value to the Type of the variable that is being used to hold the new value.

Casting a value to a Type is only permitted when the initial value comes from another variable or from a JOB, SAMPLE, or NAMESPACE clause. Casting a constant value is not permitted because the value is known and so it can be written in the proper form.

See the section titled “Value Type Casting” to learn how values can be interpreted.

Exposed Variables

The EXPOSE keyword causes the variable’s name and value to appear in the Method attributes area of the Job page on the *Spectrometer Control* window. The description provided by the help clause will also be displayed with the variable. Exposing a variable gives the operator the ability to override the default initial value specified by the variable declaration.

When the value of an exposed variable is changed, it may have an impact on the run-time duration of the Method and so the duration time will be re-computed. If it is known that an exposed variable *will not* have an impact on the duration then the variable can be defined as a ‘passive’ mode variable by inserting the PASSIVE keyword before the VAR keyword. This will cause the time calculation to be skipped for this variable and it will also make the interface more responsive. By default, without the PASSIVE keyword specified, exposed variables are defined with a mode set to ‘active’. However, the author of the script may choose to add the ACTIVE keyword before the VAR keyword even though it is neither required nor necessary.

The When Clause

The WHEN clause may only be specified if the EXPOSE keyword is used. The variable will only be visible in the Method attributes area when the expression evaluates to a True value. The expression is evaluated each time a Method parameter or other exposed variable/constant is modified.

Variable Options: Dependencies

A dependency clause may only be specified if the EXPOSE keyword is used and it stipulates how the initial value of a variable should be affected when other values upon which it depends are changed. In these circumstances, the author of the Automation Script can designate the relationship between the variable and the other variables and/or parameters using the DEPENDS clause.

This form of the dependency clause is:

```
DEPENDS ON parameter-or-variable-list  
EVALUATE (expression) ELSE constant
```

The *parameter-or-variable-list* is one or more identifiers separated by commas. If a variable or parameter is listed in the *parameter-or-variable-list* it should naturally be part of the *expression* or *Boolean-expression*.

A variable that includes a DEPENDS clause of this form will have its initial value recalculated on the user interface whenever a variable or parameter on which it depends is modified. The *expression* is a Percival expression that will be evaluated to determine the new initial value of the variable. The *constant* in the optional ELSE clause will be the initial value if the *expression* does not result in a legal value for the variable.

Parameter Options: Relevancy

A variable is normally always relevant to the Method in which it is declared, but there may be times when that is not True – for example, a variable only referenced within parts of an IF statement. A variable definition that includes an ENABLE clause is said to be irrelevant to the Method when the result of the evaluation of its associated *Boolean-expression* is False. Include the ENABLE clause after a dependency clause when the relevancy of the variable depends on the value of one or more other variables or parameters. Therefore, the *Boolean-expression* should likely contain the names of the other variable(s) and/or parameter(s) on which it depends. An irrelevant variable will appear disabled so that the operator will be unable set or change its value in the Method attributes area on the Job tab of the *Spectrometer Control* window.

```
DEPENDS ON parameter-or-variable-list  
ENABLE WHEN (Boolean-expression)
```

The DEPENDS clause at the beginning is only required when the relevancy of a parameter depends on the value of another parameter. If the relevancy of a parameter depended on the time of day, for example, the DEPENDS clause can be omitted. Assuming there exists a function, `time_of_day`, that returned the fractional number of hours since the previous midnight, the author could simply write the following to create a parameter that is enabled after noon.

```
ENABLE WHEN (time_of_day > 12)
```

The examples provided in the *Parameter Options: Relevancy* section in the description of the METHOD statement can be modified to use exposed variables rather than Method parameters and are similar examples for exposed variables.

NOTE: The ENABLE clause cannot be specified on its own if it had been previously specified at the end of the DEPENDS clause.

The two forms of the dependency clause described above can be combined together to make a third form which specifies a value dependency as well as a rule for when the variable is relevant to the Method.

```
DEPENDS ON parameter-or-variable-list
      EVALUATE (expression) ELSE constant,
      ENABLE WHEN (Boolean-expression)
```

Help for the User

A description may be attached to an exposed variable (those definitions beginning with the EXPOSE keyword) by inserting a comma, the HELP keyword, and one or more Strings (separated by commas) at the end of the variable declaration. The HELP clause is not required², but its inclusion is strongly encouraged so that users may better understand the purpose of the variable. The text of the help will be displayed with the variable Method attributes area of the Job page on the *Spectrometer Control* window.

The help text can be replaced by a previously defined language translation identifier to allow the help text to be determined by the current locale. See the TRANSLATE statement for information about creating and using translatable text.

Examples

Below are some examples of the use of the VAR statement. The following statements create basic variables. The colons following the variable names of adjacent VAR statements are often aligned to improve readability.

```
VAR x          : NUMBER;
VAR running   : BOOLEAN;
```

Create a variable named `min_sweep` for the minimum sweep width with an initial value of 2[ppm]. This variable will be visible and changeable on the Automation user interface.

```
EXPOSE VAR min_sweep : NUMBER = 2[ppm],
      HELP "The minimum sweep width for",
           "all experiments in this Method";
```

The next statement gets the “mode” attribute from the running Job and stores it in the variable `job_mode`. `job_mode` will be set to “None” if the Job attribute is not found or is not a String value.

```
VAR job_mode : TEXT = JOB "mode" ELSE "None";
```

The next statement gets the “notebook” attribute from the current sample and stores it in the variable `nbook`. This statement will raise an error condition if the value of the specified sample attribute is not a String value.

```
VAR nbook : TEXT = SAMPLE "notebook";
```

² The HELP clause is required when the parser instruction `#REQUIRE help = True` is specified previously in the Automation Script file.

The next statement gets the user value “x_sweep” from Namespace and stores it in the variable `s`. `s` will be set to `0 [Hz]` if the Namespace path is not found or if the value is not a NUMBER Type.

```
| VAR s : NUMBER = NAMESPACE "AUTOMATION.$(user).x_sweep" ELSE 0[Hz];
```

The next statement creates a variable with an initial evaluated value.

```
| VAR y : NUMBER = EVALUATE (x**2 - 4*x + 3) ELSE 0;
```

The next two statements create a Boolean variable, `debug`, with an initial value of `FALSE` and a dependent variable, `log_msg`, to log debugging messages when the `debug` variable is set to be `True`.

```
| VAR debug      : BOOLEAN = FALSE;  
| VAR log_msg    : BOOLEAN = FALSE WHEN debug = TRUE;
```

The following examples show the use of a user defined Type and a dependency on the nucleus.

```
ENUM Nuclei IS ( "1H", "2H", "13C" );
```

```
| EXPOSE VAR nucleus : Nuclei = "1H";  
| EXPOSE VAR sweep   : NUMBER,  
|     DEPENDS ON nucleus  
|     EVALUATE ((nucleus = "13C") ? 4.5[ppm] : 2[ppm]);
```

VISUALIZE

The VISUALIZE statement is used to return one or more data files to the workstation and the screen of the user executing the Method. It has the following syntax:

```
VISUALIZE data-file-variable-list IN "Percival-tool-name";
```

Description

The *data-file-variable-list* specifies the content that will be sent to the user's workstation. Each variable name used in this statement must have been declared to be of the DATA Type. File variables can be created in any of the following ways: by an EXPERIMENT block, by declaring them with the VAR or CONST statement to be of the DATA Type, or by declaring them as DATA Type parameters of the enclosing Method.

More than one file variable can be specified and this can allow for some helpful arrangements of the data. When two or more variables are used, each variable must be separated by the keyword AND or the keyword WITH depending on the desired arrangement which is discussed later.

Each data file will appear on the screen of the user who requested the Method to run. The only caveat is that the same user must have an active connection to the spectrometer at the time that the VISUALIZE statement is executed by the running job for the data to be successfully sent to the user. This provides for the user to be able to submit a Method from *machine A* and see his data on the screen of *machine B* when his job runs at a later time. If the user cannot be found with a connection to the NMR instrument, this statement is skipped.

Data Persistence

Only DATA Type variables that hold files that will persist beyond the scope of the Method can be used with the VISUALIZE statement. These variables are considered to be non-transitory. Typically, non-transitory DATA Type variables can be created in the following ways:

- A variable created with the title of an EXPERIMENT block holding acquired data,
- A variable created by a PROCESS statement that includes the SAVE AS clause,
- A variable preserved using the RETAIN statement, and
- All forms of Method parameters.

Note that a non-transitory variable can become transitory (thus unable to be visualized) if the variable is used as the destination of a SET assignment statement using another transitory source.

This statement may be used any number of times to conditionally send data to the user's workstation and to additionally place the data on the user's screen.

NOTE: This statement is skipped when the data cannot be displayed on the user's screen. To locate the data, use the File Browser to navigate to the Data Server of the instrument that acquired the data.

Configuration Options

There are two configuration options that affect the behavior of this statement. Firstly, the "visualize" Job parameter must not be set to FALSE. This parameter is the controlling gate of whether the VISUALIZE statement executes – regardless of whether the user is connected to the instrument. Secondly, after the data is uploaded to the workstation, the "Spectrometer Control: Open Delivered Data" preference will control if the data is to be put on the user's screen.

Destination Tool Window

By default, data will be shown in a 1D or nD Processor window ready for the user to work and analyze the spectrum. PDF files will be displayed by the default PDF viewer program as set in the operating system settings of the workstation. Although PDF files will always be handled by the operating system, the Delta software can be instructed how to handle the incoming NMR data files. In a custom workflow, another tool could be used (or created).

The name of the destination Percival tool to display the data can be specified within quotes after the IN keyword. The only requirement being that the first parameter of the Percival program must be a type that can accept a FILE (to display singular data), a SET of files (to display more than one data, typically side-by-side), a SET of sub-sets of files (to display multiple data, typically overlaid), or permit any of the formats. The format required is agreed upon between the interface of the Percival program and the construction of the VISUALIZE statement described in the next section.

Multiple Data: Side-by-Side & Overlays

A single file variable used in the *data-file-variable-list* will always require the receiving Percival program to accept a FILE as the first parameter. There are a number of Percival programs that can be used in this case including the data processors and data viewers.

Multiple file variables joined together using only the AND keyword will require the receiving Percival program to accept a SET of files as the first parameter. The AND keyword produces a list of files for side-by-side display.

Multiple file variables joined together using the WITH keyword will require the receiving Percival program to accept a SET of sub-sets of files. The WITH keyword in combination with the AND keyword can produce a list of lists of files. This is typically for displays requiring data overlays.

The Data Slate viewer tool, “data_slate”, is a non-trivial but great example of a program that will accept any of the above as inputs and will display multiple data sets in various configurations. The comment line below the following lines show the input format that the Percival program would expect to receive for its first argument.

```
VISUALIZE a IN "data_slate";
  -- File: a
VISUALIZE a AND b AND c IN "data_slate";
  -- Set: { a, b, c }
VISUALIZE a WITH b AND c WITH d IN "data_slate";
  -- Set: { { a, b }, { c, d } }
```

Examples

The following example acquires data from the spectrometer, stores it into a variable with the name `trial`, and will attempt to send the data to the user’s workstation. And if the “on screen” preference is set, the file is put into a data processing tool on the screen of the operator.

```
EXPERIMENT trial IS
...
END EXPERIMENT;

| VISUALIZE trial;
```

The next example processes the acquired data two different ways and sends each processed file to the user's workstation to be displayed in a separate data processor window.

```
PROCESS trial TO spectrum SAVE AS "a";  
PROCESS trial WITH "alternate_process" TO alternate SAVE AS "b";
```

```
| VISUALIZE spectrum AND alternate;
```

Appendix

Parser Instructions

An Automation parser instruction is a command that will cause the parser to perform some action or to change its behavior when reading an Automation script file. Each instruction begins with a # (pound sign) at the left-most position (column one) and any text to the end of the line are options for the instruction or are ignored. Automation parser instructions may only be used between statements at the outermost scope level of the script text unless noted otherwise. The Automation parser recognizes the following commands:

#DEFINE *ident* [=] *value*

The #DEFINE instruction adds a symbolic name for a constant Boolean, String, Number (including the keywords INFINITY and -INFINITY), or Namespace value. The Namespace path is defined by putting the marker “NAMESPACE::*path*” (without the quotes) in all caps at the beginning of the *value*. The actual *path* portion may have quotation marks around it, but they are not required.

The author is then able to specify “\$*ident*” or “\$(*ident*)” (without the quotes) in the place of a literal constant value anywhere within their script where a constant value would be legal. This will insert the defined constant value in place of the identifier and has the same effect as if the actual constant were used. The benefit is the elimination of proliferated duplicate constant values and having named identifiers which can be easily modified at a single location in the script file.

#ECHO *text*

All characters following the #ECHO instruction up to the end of the line will be printed on Delta’s main console window when the script is parsed. This may be used anywhere in the script.

#REQUIRE *option* [=] *state*

The #REQUIRE instruction provides the script author some control of the requirements that the syntax of the script must follow. The following options are available:

#REQUIRE help = *Boolean*

When set to TRUE, help text will be required on all exposed variables and on every Method parameter that is 1) not concealed, 2) has a mode of IN or INOUT, 3) is one of the basic Types other than DATA, and 4) is defined on an exposed Method at the highest scope-level. The author may want to turn this feature on to ensure that help text is not accidentally skipped and will be displayed on the user interface of the Spectrometer Control tool.

#REQUIRE case *what* = *mode*

Sets the case style requirement for portions of the input script. <What> can be one of “keyword”, “identifier”, “method”, “type”, or “subtype” and <mode> can be one of “uppercase”, “lowercase”, “capitalize”, or “none”. Do not enter the quotation marks. Keyword refers to the reserved words in the language, Identifier refers to the named values which are variables and parameters, Method refers to the non-quoted Method names, Type and SubType refer to the names of the kinds values a variable can hold. Specifying “subtype” will override any setting specified by “type”. If a “subtype” case requirement is not specified, then subtype names will follow the rule for base Types. The default for any unspecified case requirement is “none” indicating that case will not be checked for that portion of text.

#SORT *entity* [=] *Boolean*

The #SORT instruction informs the system to order the specified entity. The only entity supported at this time are Methods. Use the word 'methods' for the *entity* to specify whether the Methods should be returned sorted alphabetically (TRUE) or kept in the order of the automation script file (FALSE). Methods are sorted alphabetically by default when a sort order is not specified.

#STOP

The #STOP instruction causes the parser to ignore all text following it up to the end of the file or the point at which it encounters a #START instruction.

#START

If the parser has been instructed to ignore text with the #STOP instruction, parsing will resume on the line following the #START instruction. #START has no effect unless parsing has been interrupted with #STOP. Because block comments cannot be nested, this is a good way to eliminate large portions of text easily without using single line comments.

Future ideas:

#IF [NOT] *condition*
#ELSE
#END

This idea is to allow for conditional parsing. The condition can be Namespace based or user based to potentially disallow access to parts or features of the script based on clearance level.

Table of Units

The following tables show the available units that can be used in an Automation Script. Every unit must be enclosed in square brackets, []. Each unit has a full and an abbreviated form. Either form can be used to specify a unit. Similarly, unit prefixes have a full and an abbreviated form. If the abbreviated form of the unit is used, then the prefix must also be abbreviated. For example: [kHz], [khertz], and [kilohertz] are all valid units, however [kiloHz] is not valid.

UNITS	
Name	Abbreviation
abundance	abn
ampere	A
byte	B
candela	cd
celsius	dC
coulomb	C
dalton	Da
decibel	Db
degree	deg
electronvolt	eV
farad	F
gram	g
gray	Gy
henry	H
hertz	Hz
joule	J
kelvin	K
liter	l
lumen	lm
lux	lx
meter	m
mole	mol
newton	N
ohm	O
pascal	Pa
percent	%
point	pnt
ppm	ppm
radian	rad
second	s
siemens	S
sievert	Sv
steradian	sr
tesla	T
thompson	Tn
volt	V
watt	W
weber	Wb

UNIT PREFIXES			
Name	Abbrev.	Power	
yotta	Y	10 ²⁴	septillions
zetta	Z	10 ²¹	sextillions
exa	E	10 ¹⁸	quintillions
pecta	P	10 ¹⁵	quadrillions
tera	T	10 ¹²	trillions
giga	G	10 ⁹	billions
mega	M	10 ⁶	millions
kilo	k	10 ³	thousands
milli	m	10 ⁻³	thousandths
micro	u	10 ⁻⁶	millionths
nano	n	10 ⁻⁹	billionths
pico	p	10 ⁻¹²	trillionths
femto	f	10 ⁻¹⁵	quadrillionths
atto	a	10 ⁻¹⁸	quintillionths
zepto	z	10 ⁻²¹	sextillionths

Units and unit prefixes (the full names and the abbreviations) are case sensitive. The full name of each unit must be specified with all lowercase letters. However, the unit abbreviations often contain mixed case.

Refer to the discussion of the NUMBER Type to see how units can be written. In some statements, units can be specified without a number, but it is more common for a unit to be attached to the end of a number without any space separation.

Table of Substitution Identifiers

The following table enumerates the symbols that have special meanings when they are used within certain Strings in an Automation Script. These symbols are set apart from the text of the String by surrounding their name with parentheses and preceding them with a dollar sign, \$. For example, the String “Today is \$(DAY)” would become “Today is Monday” if today was, in fact, Monday. The symbols are listed in upper case, but the case of the symbols does not matter.

The special symbols can be used:

- within a TEXT Type argument to a Percival operator or Service Manager request using the CALL statement.
- within a TEXT Type argument to a Method invocation using the INVOKE statement.
- within a Namespace path, a Job attribute name, a Sample attribute name defining the default value for a variable using the VAR or CONST statement.
- within the String to be evaluated following an EVALUATE keyword.
- within the value of an assignment of the SET statement.
- within the subject and message parts of the EMAIL statement.
- within the message part of the INFORM statement.
- within the filename of the SAVE AS statement in an EXPERIMENT block.

Symbol	Includes the...	Type
\$(DATA_SERVER)	Hostname of the data server storing the acquired data.	TEXT
\$(DATE)	Current date formatted per the settings specified by the Control Panel / System Preferences of the Operating System. If the date/time format preference cannot be discovered, the following format will be used instead: \$(DAY_NUM)-\$(MONTH3)-\$(YEAR)	TEXT
\$(DAY)	Name of the current day.	TEXT
\$(DAY3)	Three letter abbreviation of the current day.	TEXT
\$(DAY_NUM)	Number of the current day of the month (1-31).	NUMBER
\$(ERROR)	Name of the current error condition. “None” if there is no error.	TEXT
\$(EVENT)	Name of the current event. “None” if there is no event currently being handled.	TEXT
\$(EXP.FILENAME)	Filename of the Experiment. This is specified by the filename parameter in the Header portion of the Experiment file.	TEXT
\$(FILENAME)	Innermost scoped filename attribute found in this order: Job, Sample, then Method. If this attribute does not exist or is empty, then the Method title will be used as the value.	TEXT
\$(INSTRUMENT)	IP Address of the NMR instrument executing this job.	TEXT
\$(JOB)	Title of the running job.	TEXT
\$(JOB_ID)	ID number of the running job.	NUMBER
\$(JOB.FILENAME)	Filename of the Job specified in the Job’s attributes.	TEXT

Symbol	Includes the...	Type
\$(JOB.attribute)	Value of the specified attribute of the current Job. This value can be overridden by an attribute with the same name in the Sample or Method. If * is used in place of <i>attribute</i> , a table will be inserted which will list all the current Job's attributes. <i>The use of * is only available where multi-line text is permitted.</i>	<i>Indeterminate</i>
\$(METHOD)	Name of the currently executing Method.	TEXT
\$(METHOD.FILENAME)	Filename of the Method specified in the Method's attributes.	TEXT
\$(METHOD.attribute)	Value of the specified attribute of the current Method. If * is used in place of <i>attribute</i> , a table will be inserted which will list all the current Method's attributes. <i>The use of * is only available where multi-line text is permitted.</i>	<i>Indeterminate</i>
\$(MONTH)	Name of the current month.	TEXT
\$(MONTH3)	Three letter abbreviation of the current month.	TEXT
\$(MONTH_NUM)	Number of the current month (1-12).	NUMBER
\$(NOW)	Current date and time formatted per the settings specified by the Control Panel / System Preferences of the Operating System. If the date/time format preference cannot be discovered, the following format will be used instead: \$(DATE) \$(TIME) \$(TIMEZONE)	TEXT
\$(SAMPLE)	Current sample identifier used for this Method. This shorthand provides the same value as \$(sample.sample id).	TEXT
\$(SAMPLE_KEY)	Current sample database id number currently in use.	NUMBER
\$(SAMPLE.FILENAME)	Filename of the Sample specified in the Sample's attributes.	TEXT
\$(SAMPLE.attribute)	Value of the specified attribute of the current Sample. This value can be overridden by an attribute with the same name in the Method. If * is used in place of <i>attribute</i> , a table will be inserted which will list all the attributes of the current Sample. <i>The use of * is only available where multi-line text is permitted.</i>	<i>Indeterminate</i>
\$(SITE)	Site name of the instrument as specified by the Site preference. Same as \$(INSTRUMENT) if the Site preference has not been set.	TEXT
\$(TIME)	Current system time of day formatted as <i>HH:MM:SS</i> . Where <i>HH</i> represents the hour, <i>MM</i> represents the minute, and <i>SS</i> the seconds.	TEXT
\$(TIMEZONE)	Abbreviated time zone offset of the NMR spectrometer.	TEXT
\$(TIMEZONE_LONG)	Full name of time zone offset of the NMR spectrometer.	TEXT
\$(USER)	Name of the user who is running this job.	TEXT
\$(variable-name) \$(VAR.variable-name)	Value of the variable (including constants and Method parameters) specified by <i>variable-name</i> . If <i>variable-name</i> happens to be one of these recognized words, use the second form.	<i>Indeterminate</i>
\$(YEAR) \$(YEAR_NUM)	Current year.	NUMBER

Any Method parameter or variable that shares its name with one of these special symbols is eclipsed and cannot be accessed using the regular \$(*variable-name*) form. To access the eclipsed value, use the VAR keyword and a period before the name: \$(VAR.*variable-name*).

There are a few symbols in the table above that have their Type listed as *Indeterminate*. The value of these symbols is not knowable because these forms can produce the value of any attribute of the Job, Sample, or Method, or the value of any parameter or variable of the Method. Since these values could be of any Type, it is not possible to know the Type of value to which these identifier forms will resolve.

Below are some common attributes that *may* exist for a Job. They can be added to a Job when a Job is selected in the Open Jobs list on the Job tab of the *Spectrometer Control* window. Job attributes can be referenced within an Automation Script String literal using \$(JOB.attribute).

Job Attribute	Description
allow email	<p>This attribute controls how email can be sent during the execution of an Automation script.</p> <ul style="list-style-type: none"> • When this attribute is not specified or if it has a value of TRUE or the value “all”, email with any attachment is permitted with the EMAIL statement. • When this attribute is specified with a value of FALSE or with the value “off”, the EMAIL statement will be ignored and no email will be sent. • If this attribute is specified with the value “none”, then email will be permitted but requested attachments will not be included with the message body. • If this attribute is specified with the value “data”, then email will be permitted but only data file attachments will be permitted with the message body. • If this attribute is specified with the value “pdf”, then email will be permitted but only PDF attachments will be included with the message body.
allow printing	<p>This attribute controls whether paper or PDF output can be generated.</p> <ul style="list-style-type: none"> • When this attribute is not specified, the PRINT and PRESENTATION statements will only execute if the “Allow Printing” spectrometer system preference is checked on the <i>Printer</i> tab of the Spectrometer Preferences panel. • When specified and is checked, the PRINT and PRESENTATION statements will be allowed to execute even if the “Allow Printing” spectrometer system preference is not checked. • When specified and not checked, the PRINT and PRESENTATION statements will not execute even in the “Allow Printing” spectrometer system preference is checked.
comment	<p>This attribute is intended for the operator to document the purpose of the Job. It is unformatted text.</p>
filename	<p>This attribute specifies the Job part of the storage filename that can be a part of the ultimate filename that is used to name and store the raw experimental data files. By default, this is not a part of the storage filename.</p>
folder	<p>This attribute specifies part of the local storage path under the system DATA directory where data files will be stored.</p>

Job Attribute	Description
gradient shim	<p>Gradient Shimming will be <i>allowed</i> to run at the point when a Sample is loaded (prior to running the Job group) if either of the two conditions are met:</p> <ul style="list-style-type: none"> • When this attribute is not specified and the “Allow Gradient Shimming” spectrometer system preference is checked on the <i>Lock & Shim</i> tab of the Spectrometer Preferences panel, or • When this attribute is specified and checked. <p>Setting this Job attribute will not guarantee that Gradient Shimming will run. Each sample must either not contain a “gradient shim” attribute or have its own “gradient shim” attribute checked for Gradient Shimming to occur.</p>
precedence	<p>This attribute controls the order in which Methods and Samples are selected to run when a Job is queued containing multiple Methods and multiple Samples. The available options are:</p> <ul style="list-style-type: none"> • “sample” (default): Run all Methods on each sample before loading the next Sample. • “method”: Run the first Method on every sample then run the second Method on every sample continuing until the final Method is run on every Sample.
printer	<p>This attribute specifies the name of the printer to which the PRINT and PRESENTATION statements will send the data.</p>
priority	<p><i>Reserved for future use.</i></p>
project †	<p>This attribute specifies the Project name to which the data acquired and the output produced by the Job is to be associated. It is also included as part of the local storage path under the system DATA directory and following the optional ‘folder’ attribute. <i>See above.</i></p>
recur †	<p>Normally a Job is run a single time. When this attribute is specified, it allows a Job to run at a specific time at the specified interval. The Start Time of the Job must be specified for this attribute to have an effect. Available intervals are: hourly, daily, weekly, biweekly, monthly, and yearly. It is recommended that the operator avoid using “recur”, “resubmit”, and “sample repeat” together.</p>
resubmit †	<p>Normally a Job is run a single time. When this attribute is specified, it allows a Job to run multiple times. This is specified as a positive integer and is decremented each time the Job is run. When the value of this attribute reaches 0 (zero), the Job is removed from the Automation Job Queue. With Jobs using multiple Samples, the Samples are unloaded after each run of the Job. It is recommended that the operator avoid using “recur”, “resubmit”, and “sample repeat” together.</p>
sample eject	<p>When specified and checked, the last Sample (that is normally left loaded in the spectrometer) will be ejected from the spectrometer when the Job has been completed.</p>

Job Attribute	Description
sample repeat †	Similar to “resubmit”, this can make a Job run multiple times. When used with a single sample, the behavior is the same as “resubmit”. However, with Jobs using multiple samples, it causes the Job to run the specified number of times <i>before</i> the Sample is unloaded. It is recommended that the operator avoid using “recur”, “resubmit”, and “sample repeat” together.
visualize	When specified and not checked, VISUALIZE statements will not be executed so that acquired data will not appear on the operator’s computer screen.
storage_comments	<i>Reserved for internal use.</i>

Below are some common attributes that *may* exist for a Sample. They can be added to a Sample by revealing the attributes area of a Sample in a Sample table row on the Sample tab of the *Spectrometer Control* window. Sample attributes can be referenced within an Automation Script String literal using `$(SAMPLE.attribute)`.

Sample Attribute	Description
abort on GS fail	When specified and checked, the Job will be terminated if Gradient Shimming runs and fails. Otherwise, if Gradient Shimming is unable to finish normally the Job will continue.
analyst	The name of the person who prepared the Sample or the intended person to analyze the data generated from this Sample.
autoshim_mode	When specified and checked, performs an AutoShim when the Sample is loaded prior to any Gradient Shimming.
calibration	<i>Reserved for future use.</i>
comment	This attribute is intended to document the Sample by providing a short description. It is unformatted text.
concentration †	A number representing the volume of actual sample per solvent.
filename	The sample component to the storage filename of acquired data.
folder	This attribute specifies part of the local storage path under the system DATA directory where data files will be stored.
gradient shim	When specified and checked, Gradient Shimming will run when the Sample is loaded if Gradient Shimming is permitted to run by the state of the “gradient shim” Job attribute or the “Gradient Shim Allowed” spectrometer system preference setting.
intent †	A word or short phrase indicating the specific purpose for the Sample. This is hidden from the user.
kind †	An identifier indicating a Sample classification. Other attributes may become more easily accessible based on the value of this attribute.

Sample Attribute	Description
load_shims	Set to TRUE to reset the shims to the stored system shims when the Sample is loaded into the spectrometer.
lock_achieve_point	The minimal lock strength that must be achieved for the Lock to be successfully enabled. The Lock Receiver Gain may be increased to reach this level.
lock_gain	Specifies the value of the Lock Receiver Gain to be used when a Job is running on the Sample.
lock_level	Specifies the power level of the Lock channel to be used when a Job is running on the Sample.
lock_osc_offset	Allows setting of the oscillator offset for the lock channel.
lock_phase	Specifies the value of the Lock Phase to be used when a Job is running on the Sample.
lock_settle_point	The maximum lock strength that is allowed for the Lock to be successfully enabled. The Lock Receiver Gain may be decreased to maintain this level.
lock_state	The Lock channel mode to be used on this Sample.
mas_spin_delay	Additional delay after any timed MAS settling before an experiment may begin.
mas_spin_set	The rate of spin to be used for the MAS controller when a Job is running on the Sample. See “spin_set” for Liquids samples.
mas_spin_state	The solids Sample will spin when this attribute is specified and checked. See “spin_state” for Liquids samples.
molecule	This attribute is intended to hold a textual representation of the Sample molecule – such as the International Chemical Identifier (InChI) or the Simplified Molecular Input Line Entry Specification (SMILES).
notebook id	This attribute can be used at the discretion of the operator who created the Sample. It is intended to be text that will identify the notebook that contains the lab data about the Sample.
owner †	This attribute is set to be the name of the operator who created the Sample. It is not editable.
page number	This attribute can be used at the discretion of the operator who created the Sample. It is intended to be the page number of the notebook (specified by the “notebook id” attribute) that contains the lab data about the Sample.

Sample Attribute	Description
preparation †	This attribute specifies whether the Sample preparation Method(s) should be invoked from the Preamble Method in the Utilities.jaf script. The Preamble Method should be invoked once before any EXPERIMENT block. The preparation Method(s) currently includes only Gradient Shimming.
printer	The default printer to use for this sample. This will override the system printer and any printer specified in the Job attributes.
probe required	The name or number of the required probe. If the value is a number, then the Sample will run only if the value matches the Id of the installed probe. If the value is text, then the Sample will only run if the value matches the Type of the installed probe.
reserved	Specifies the operator user id for whom this Sample is reserved. Other users will not be able to submit Jobs with this Sample.
sample eject	When specified and checked, the last Sample (that is normally left loaded in the spectrometer) will be ejected from the spectrometer when the Job has been completed. If specified, this will override the setting in the Job attributes.
sample height †	The offset height of the liquid sample in the tube. This should be measured in \mp millimeters.
sample id †	The unique textual identification tag for the Sample.
shared †	<i>Reserved for internal use.</i>
slot †	The slot number into which the Sample will be physically placed.
solid_cap	The material of the solids sample tube cap.
solid_tube	The material of the solids sample tube body.
solvent †	The solvent in which the Sample is dissolved.
spin_set	The rate of spin to be used when a Job is running on a liquid Sample. See “mas_spin_set” for solids samples.
spin_state	The liquid Sample will spin when this attribute is specified and checked. See “mas_spin_state” for solids samples.
temp_delay	The number of seconds to wait after the temperature controller has reached the specified temperature set point.
temp_ramp_step	The maximum interval of increase or decrease in temperature that is allowed while the temperature controller is moving toward the target temperature value specified by the “temp_set” attribute.
temp_ramp_wait	The number of seconds to wait at each temperature interval while the temperature is moving toward the target temperature.

Sample Attribute	Description
temp_set	Specifies the system temperature setting in degrees Celsius when a Job is running on this Sample.
temp_state	The temperature controller will be turned on when this attribute is specified and checked.
tube type †	The type of tube in which the Sample resides.
verified †	<i>Reserved for internal use.</i>

There is only one common attribute that *may* exist for a Method. Method attributes can be added when a Method is selected on the Job tab of the *Spectrometer Control* window. Method attributes can be referenced within an Automation Script String literal using `$(METHOD.attribute)`.

Method Attribute	Description
filename	This attribute specifies the Method part of the storage filename that, by default, is a part of the ultimate filename used to name and store the raw experimental data files.

The author of an Automation Script and the user defining a Job may, of course, add an attribute of any name to a Job, Sample, or Method. The lists above are the common attributes defined by the Delta and Control software – some of which are used by the Automation system (like “comment” and “solvent”) and others exist for user convenience (like “concentration” and “notebook id”).

† Attributes marked with this symbol cannot be set or modified in an Automation Script. Their values cannot be changed at run-time. The values for these attributes must be set prior to submitting a Job to the instrument.

Writing a Duration Statement Expression

The DURATION statement expression can be a tricky thing to get correct and accurate. This explanation and the tips below should help the author to write elaborate expressions.

The evaluation of the entire expression must result in a number with a unit of seconds. If this will not be true for any part of the expression, then the expression will be discarded and the Method time will fail to update.

Literal values, named constants, exposed variable names, names of Method parameters, and EXPERIMENT block names can all be referenced in a DURATION statement. Literal values are STRING, NUMBER, and BOOLEAN values like: “atn”, 7 or 7[s], and True respectively. Named constants are replaced in the expression by the values they represent. Exposed variable names and the names of Method parameters are replaced in the expression by the values that the user has supplied for them on the user interface. EXPERIMENT block names are replaced in the expression by the time value that is calculated for the experiment specified by the COLLECT clause after all of the experiment parameter assignments (SET statements within the EXPERIMENT block) have been updated to the experiment.

Besides the basic mathematical operators that can be used in the expression, there is a decision-making operator to allow expression branching. The ternary operator, denoted with the symbols `? :`, has three parts: a test and two result expressions. The operator is written like in the form: *test ? true-expression : false-expression*.

«TIP» Each of these three components must be surrounded by parenthesis if they are compound expressions and if this operator is used as a sub-expression, then it will probably require parenthesis around the entire operator as:

```
(( compound-test )
  ? ( compound-true-expression )
  : ( compound-false-expression ) )
```

The first part of the ternary operator is the test. This part must result in a Boolean value when evaluated. If the result of evaluating the *test* part is True, then the evaluated value of the *true-expression* (after the question mark) will be the result. Otherwise, if the result of evaluating the *test* part is False, then the evaluated value of the *false-expression* (after the colon) is the result.

«TIP» *All* of the sub-expressions need to be proper mathematical expressions regardless of whether the *true-expression* or the *false-expression* of the ternary operator will be the result

«TIP» Oftentimes a variable that holds a unitless numeric value needs to be used in the expression in places where a unit is required. For example, units much match when adding two numbers. To ensure that the value will have the expected unit when the expression is evaluated, the author can provide the proper unit for the number with the ``` (backtick) operator. To force the unit of seconds on a NUMBER variable named ‘foo’ write: `foo `[s]`.

More sophisticated expressions must be written to gain timing accuracy. This often requires substituting a variable value into a named experiment parameter when the assignment does not

happen within the SET statements of the EXPERIMENT block. In these cases, there is a special syntax that can be added to follow an EXPERIMENT block name. This syntax associates a variable name to an experiment parameter one or more times. The syntax is:

```
experiment-name@ ( exp-param <- var-name )
```

There can be no other characters, including spaces, around the @ character – it must follow the EXPERIMENT block name immediately and be followed by the open parenthesis. To associate more pairs to an experiment, add more *exp-param <- variable-name* parts within the parentheses and separate each pair with a comma.

```
exp-name@ ( exp-param1 <- var-name1, ..., exp-paramn <- var-namen )
```

Example Script

```

AUTOMATION VERSION 2;

TRANSLATE printer_help
  ALL "Print result on paper?";
  ES "¿Resultado de la impresión en el papel?";

METHOD Proton( IN print_to : TEXT = "", HELP printer_help ) IS

  CATEGORY "1d", "1h", "standard";

  HELP "Proton Acquisition";
  PURPOSE "Proton with 12[ppm] -> -0.5[ppm] plot",
    "Optionally send the data to a printer.";

  EXPOSE VAR rgain : NUMBER = 50, HELP "Observation receiver gain";
  EXPOSE VAR scans : NUMBER = 8, HELP "Number of scans to perform";

  TRANSLATE acquire ALL "Acquiring data...";
    ES "Adquiriendo datos...";

  INFORM TO CONSOLE acquire;

  SET MACHINE temp_state = "TEMP ON";
  PROMOTE "temp_state" TO SAMPLE;

  EXPERIMENT Proton IS
    SAVE AS "$(EXPERIMENT)_PROTON";
    COLLECT "1d/single_pulse";
    SET
      auto_gain      = TRUE;
      force_tune     = FALSE;
      relaxation_delay = 4[s];
      scans          = scans;      --job_parameter = Method_variable
      x_offset       = 5[ppm];
      x_sweep        = 15[ppm];
    CONSTRAIN scans >= 20;
  END EXPERIMENT;

  REMARK handle any processing errors
  ON ERROR "process-error" DO
    INFORM "There was a problem processing the data!";
    FINISH;
  END ERROR;

  INFORM TO CONSOLE "Processing data...";
  PROCESS Proton ELSE "ldh_noclip.list";

  IF print_to /= "" THEN
    PRESENTATION ProtonPlot TEMPLATE "params_right_runtime_proton.pmt"
      WITH DATA Proton TO PRINTER print_to;

    EMAIL USER SUBJECT "Proton data"
      MESSAGE "Processed Proton data print-out attached"
      ATTACH ProtonPlot;
  ELSE
    INFORM INFO WITH DATA AND TIME TO LOG "Data was not printed";
  END IF;

END METHOD;

```


Automation Script Grammar

action ::= **CALL** ([**PERCIVAL**] *ident*) | (**SERVICE** *string*) [(*argument_list_files*)]

ident is the name of a Percival operator which may be defined in a separate file external to the Automation script or the operator may be constructed within a PERCIVAL statement. *String* is the standard invocation format of a service call. Note that resulting values will not be received by asynchronous service calls.

action_statement ::= *action* ;

addresses ::= *email_destination* {**AND** *email_destination*}

argument ::= *constant* | *arrayed_ident*

argument_files ::= (**RAW** | **PROCESSED** | **ALL** [**FILES**]) | *argument*

RAW specifies a set of all of the raw data files that have been collected up to this point. **PROCESSED** specifies a set of all the processed files that have been created up to this point. **ALL** specified all the raw data files that have been collected and all of the processed data.

argument_list ::= [*ident* =>] *argument* {, [*ident* =>] *argument*}

argument_list_files ::= *argument_files* {, *argument_files*}

Even though the grammar cannot show it, only one instance of either **RAW**, **PROCESSED**, or **ALL** may exist in the entire argument list.

arrayed_ident ::= *ident* [(*positive_integer*)]

assign_block ::= **SET** *assign_variable* | *assign_external* | *sample_attr_rem*

Allows values to be assigned to one or more destination storage locations.

assign_exp_block ::= **SET** [**PARAMETER**[**S**]] <(*ident assign_no_process*) | *wildcard_assign* >

Allows values to be set to one or more experimental parameters.

assign_external ::= **JOB** | **NAMESPACE** | (**SAMPLE** *sample_set_modes*) | (**MACHINE** [**PARAMETER**[**S**]])
<(*string_var assign_no_process*) | *wildcard_assign* >

assign_no_process ::= = *external_value* | ({ *external_value* **ELSE** } *action_statement* | *prompt* | (*expression* ;))

assign_variable ::= [**VAR**] <*assignment* | *wildcard_assign* >

assignment ::= *ident* = [**CAST**] *external_value* |
({ *external_value* **ELSE** } *action_statement* | *processing* | *prompt* | (*expression* ;))

The *ident* is the name of a variable that has been previously declared with a VAR statement. If *processing* is used then the TO clause of the PROCESS command cannot be included.

assignments ::= <*assign_exp_block* > { *constrain_block* }

association ::= { { { *ident* , *constant* } } }

attribute_reference ::= **JOB** | **SAMPLE** | **NAMESPACE** *string_var*

AUTOMATION* ::= *version* [**CATEGORY** *string* {, *string*} ;] <*basic_statements*>

base_type ::= *simple_type* | **DATA**

Variables of the **DATA** Type can hold references to data files.

basic_statements ::= *method* | *method_assertion* | *comment* | *type_declaration* | *translation* | *include* | *macro_definition*

binary_digit ::= **0** | **1**

binary_number ::= **#b** <*binary_digit*>

boolean_expression ::= *sub_expression* {**AND** | **OR** | **XOR** *sub_expression*}

boolean_value ::= **TRUE** | **FALSE** | **YES** | **NO**

casing ::= **LOWERCASE** | **UPPERCASE** | **CAPITALIZE**

code ::= **PERCIVAL** <<< Percival_Code >>> ;

Percival_Code is a block of valid Percival statements. If these statements were to be wrapped in an operator definition, then the operator can be executed with the CALL statement.

collection ::= [**CONCEAL** | **EXPOSE**] [**INTERIM**] [**QUIET**] [**SCOUT**] **EXPERIMENT** *ident* **IS**
exp_block
END EXPERIMENT [*ident*] ;

ident is the title of the experiment to be performed. This title can be used as a constant reference to a data file after the experiment has completed. The optional *ident* at the end of the EXPERIMENT block must match the first *ident* exactly. If the COLLECT statement within the EXPERIMENT block results in multiple data files being collected, then the individual files can be referenced like an array with the syntax: *ident*(*n*) where *n* is a positive integer. Note that *ident* is equivalent to *ident*(1). Specifying INTERIM will cause every data file collected within the block (both raw and processed) to be deleted once the enclosing Method completes. Specifying SCOUT will cause the system to check for a similar and recently collected file to possibly skip the acquisition.

comment ::= **REMARK** {Character} \$

This is a comment line and does not perform any action. The \$ character indicates the end of the line so any text between the REMARK keyword and the end of the line is ignored as a comment.

conditional_test ::= (*attribute_reference* [Expression_Remainder]) | *boolean_expression*

The Expression_Remainder is the rest of a Boolean expression with the assumption that *attribute_reference* is the first value part of the full expression. So, Expression_Remainder could be "> 0" or "* 2 = 10". Of course the quotes should not be included as part of the expression.

const_def ::= **CONST** *nondata_init* | (*ident* : **DATA** = *expression*)

Creates a named reference to a value that is immutable. *Expression* must result in a String value.

constant ::= *simple_constant* | *list_value* | *enum_indexed* | **NULL**

constrain_block ::= **CONSTRAIN** <*constrain_element*>

`constrain_element` ::= *ident* { *constraint constant* | *ident* } [*multiple_constraint*]
 MODULO *positive_number* [*units_constraint*];

Both of the uses of *ident* are references to experiment parameters.

`constraint` ::= <= | < | >= | >

`days` ::= *nonneg_number* **DAY**[**S**]

`decimal_digit` ::= *octal_digit* | **8** | **9**

`decimal_number` ::= [**#d**] <*decimal_digit*>

`delay_statement` ::= **DELAY** (*time* [**AFTER** *ident*]) |
 (**UNTIL** *time_of_day* [[**WITH**] **COUNTDOWN** [**INTERVAL** *time*]]) [*when_clause*];

If specified, *ident* must be data file variable. By including the optional **AFTER** clause, the author informs acquisition to wait the specified amount of time from the completion of the specified data rather than from the current time.

`depends_value` ::= (**EVALUATE** (*expression*)) | (**NAMESPACE** *string*) [**ELSE** *constant*]

`dependency` ::= **DEPENDS** [**ON**] *ident_list* *enable_clause* | (*depends_value* [, *enable_clause*])

`email_destination` ::= **USER** | *string_var*

There are three options for the destination of an email message.

- **USER** will send email to the operator who submitted the Automation script for processing.
- A literal String specifies a particular email address to which the email will be sent.
- An identifier should reference a **TEXT** variable that holds a valid email address to which the email will be sent.

`enable_clause` ::= **ENABLE** [**WHEN**] (*boolean_expression*)

`enum_indexed` ::= *ident* [*signed_integer*]

The *ident* is the name of a previously defined **ENUM** Type.

`enum_list` ::= (*string_list*)

`enum_declaration` ::= **ENUM** *ident* **IS** *enum_element* { **AND** *enum_element* } ;

The **ENUM** statement creates a new Type that could be thought of as a sub-class of **TEXT**. *ident* is the name of the new variable Type.

`enum_element` ::= *enum_list* | *action* | *enum_namespace*

`enum_namespace` ::= **NAMESPACE** [*casing*] [**KEYS**] *string*
 [**EXPOSE** | **IGNORE** *enum_list*] [**EXCLUDE** *enum_list*]

EXPOSE and **IGNORE** can only be used if the **KEYS** keyword is also specified.

`error_block` ::= **DO** *statement_block* [**RETRY** [*when_clause*];] **END ERROR**

The **RETRY** statement does not have to be the last statement at the end of the *statement_block*. It may also be used multiple times and within other statement blocks within the *error_block*.

error_handler ::= **ON ERROR** [**ALL** | *string*] **CONTINUE** | **EXIT** | *termination_mode* | *error_block* ;

event_name ::= **PREPARE** | **COMPLETE** | **ABORT**

exit_statement ::= **EXIT** [*when_clause*] ;

The EXIT statement may only be used inside of a REPEAT block.

exp_block ::= [*save_clause* ;] *exp_file* [*optimization*] {*assignments*}

If a filename is not specified by not including the *save_clause* then the Experiment title, as defined by the EXPERIMENT block, will be used for the filename.

exp_file ::= **COLLECT** *string_var* ;

string_var contains the experiment file used to acquire the data. If *string_var* is an identifier, it must be declared as a constant.

expression ::= *constant* | Percival_Expression

external_value ::= *attribute_reference* | (**EVALUATE** (*expression*))

string_var is either the name of an attribute of the job, an attribute of the current sample, or it is a Namespace path.

factor ::= *parenthesized* | (**NOT** *factor*) | *boolean_value* | *ident*

ident must be a variable of the BOOLEAN Type.

fractional_part ::= . *integer* [**E** *signed_integer*]

group ::= **GROUP**
 statement_block
 END GROUP ;

A GROUP block defines a set of operations that cannot be interrupted.

help_info ::= **HELP** *translatable_list*

hex_digit ::= *decimal_digit* | **A** | **B** | **C** | **D** | **E** | **F**

hex_number ::= **#x** <*hex_digit*>

hours ::= *nonneg_number* **HOUR**[**S**]

ident ::= *letter* {[_] *letter* | *decimal_digit*}

ident_list ::= *ident* {, *ident*}

if_block ::= *if_part* {**ELSE** *if_part*} [**ELSE** *statement_block*] **END IF** ;

An IF keyword that follows an ELSE keyword on the same physical line of text is considered an ELSE case for the same IF block. If a nested IF block is required after the ELSE keyword, then the IF keyword following the ELSE must be on a separate line.

if_part	::= IF <i>conditional_test</i> THEN <i>statement_block</i>
include	::= INCLUDE <i>string</i> [[TO] DOMAIN <i>ident</i>] ; <i>string</i> is the filename or URL of an external Automation script file.
inform	::= INFORM [<i>warnings</i>] [WITH DATE [AND TIME]] [TO <i>print_mode</i> { AND <i>print_mode</i> }] <i>ident</i> <i>string_list</i> ;
initial_value	::= <i>external_value</i> (<i>external_value</i> ELSE) <i>expression</i>)
inout_data_def	::= <i>ident</i> : DATA [= <i>expression</i>]
inout_def	::= ([EXPOSE] [ACTIVE PASSIVE] <i>inout_param</i> [, <i>dependency</i>]) (CONCEAL <i>inout_param</i>) (OUT <i>typed_ident</i>) [, <i>help_info</i>] A parameter of the DATA Type does not require an initialization value. If an initial value is given for DATA Type parameters, the <i>expression</i> must result in a TEXT Type value. All other IN and INOUT parameters must have an initialization value.
inout_param	::= IN INOUT <i>nondata_init</i> <i>inout_data_def</i>
inouts	::= <i>inout_def</i> { ; <i>inout_def</i> }
integer	::= <i>decimal_number</i> <i>hex_number</i> <i>binary_number</i> <i>octal_number</i>
invocation	::= INVOKE [REF] { <i>ident</i> . } <i>string_var</i> ([<i>argument_list</i>]) ; <i>string_var</i> must be the title of a previous Method block or the name of a TEXT type variable. <i>Ident</i> is the domain name that was specified to locate the included Method.
letter	::= A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
limit_block	::= LIMIT <i>limiter</i> DO <i>statement_block</i> [EXPIRED <i>statement_block</i>] END LIMIT ; The LIMIT block will put a constraint on the set of operations in the first <i>statement_block</i> after the DO keyword. Currently, the only constraint is a time limit that will cause the first <i>statement_block</i> to abort when the time limit is exceeded. If the optional EXPIRED section is provided, the <i>statement_block</i> after the EXPIRED keyword will execute if the time limit is exceeded.
limiter	::= TIME <i>time</i>
list_value	::= { [<i>constant</i> { , <i>constant</i> }] }
list_declaration	::= LIST <i>ident</i> IS ASSOCIATION (OF <i>base_type</i>) ; Creates a new Type of LIST named <i>ident</i> whose values must be of the Type specified by <i>base_type</i> .
listed_numbers	::= LIST [INCREASE DECREASE] (<i>number</i> { , <i>number</i> })

loop_block ::= <statement | exit_statement>

The *exit_statement* can also be nested within other statement blocks within the *loop_block*.

loop_condition ::= **WHILE** | **UNTIL** *conditional_test*

loop_expression ::= *loop_condition* | *loop_range* | *loop_over* | *loop_times*

loop_over ::= *ident* [**IN** *ident* | *list_value*]

The first *ident* is an automatically declared loop variable that exists only during the execution of the loop. The second *ident* specified after the **IN** keyword is a variable that must be of the **LIST** Type. If the **IN** part is omitted then infinite loop.

loop_range ::= [*ident*] [**FROM** *expression*] **TO** *expression* [**STEP** *expression*]

The first *ident* is the name of a loop variable that exists only during the execution of the loop.

loop_times ::= *integer* | *ident* **TIME**[**S**]

The *ident* is a variable that holds a **NUMBER** Type value.

macro_definition ::= **MACRO** *ident* [(*macro_param* { , *macro_param* })] **IS** *statement_block* **END MACRO** [*ident*] ;

The first *ident* is the title of the macro. The optional *ident* at the end must match the macro title. **MACRO** statements cannot be used in the *statement_block*.

macro_expansion ::= **EXPAND MACRO** *ident* [(*argument* { , *argument* })] ;

The *ident* is the title of a defined macro. If a variable is used for an argument, then it must be declared as a constant.

macro_param ::= *ident* [= *constant*]

method ::= [**CONCEAL** | **EXPOSE**] **METHOD** *string_var* [(*inouts*)] [**WHEN** *event_name*] **IS**
method_block
END METHOD [*string_var*] ;

string_var is the title for the Method. The title will be visible externally in order to choose and launch the Method. The optional *string_var* at the end must match the first *string_var*.

method_assertion ::= [**ASSERT**] [**CONCEAL** | **EXPOSE**] **METHOD** *string_var* ;

The order of **ASSERT** and **CONCEAL** does not matter when both are specified.

method_block ::= *method_optionals* *statement_block*

method_optionals ::= [**CATEGORY** *translatable_list* ;] [**HELP** *string_var* ;] [**PURPOSE** *translatable_list* ;]
 { **PARAMETER** *ident* (*dependency* [, *help_info*]) | *help_info* ; }
 [**DURATION** *number_unit* | *Percival_Expression* ;]

A variable after the **HELP** or **PURPOSE** keyword must be a translation identifier. The *ident* must be a Method parameter name.

minutes ::= *nonneg_number* **MINUTE**[**S**]

multiple_constraint ::= (**DIVISIBLE** [**BY**]) | (**MULTIPLE** [**OF**]) *positive_number* [**INCREASE** | **DECREASE**]

multiplier ::= * | / | **DIV** | **MOD**

nondata_init ::= *typed_ident_nodata* = *initial_value*

nonneg_number ::= [+]*integer* [*fractional_part*]

number ::= *signed_integer* [*fractional_part*]

number_constraint ::= [**INTEGER** | (**PRECISION** *positive_integer*)]
 ((**FROM**) *number* **TO** *number* [**STEP** *positive_number*]) | *listed_numbers*
 [[**WITH**] (**UNIT** *unit*) | (**NO UNIT**)]

If the optional **INTEGER** keyword is specified, then the **NUMBER** Type will be restricted to integral values and numbers assigned to this Type will be rounded.
 If the optional **UNIT** clause is provided, numbers of this Type are required to have the specified *unit*.

number_declaration ::= **NUMBER** *ident* **IS** *number_constraint* | (**NAMESPACE** *string*) ;

Creates a new Type that could be thought of as a sub-class of **NUMBER**. *ident* is the name of the new variable type. Variables that have the Type specified by *ident* are restricted to a range of values.

number_sign ::= + | -

number_unit ::= *number* [*unit*]

octal_digit ::= *binary_digit* | **2** | **3** | **4** | **5** | **6** | **7**

octal_number ::= **#o** <*octal_digit*>

optimization ::= **OPTIMIZE** (*ident* {, *ident*})
 [**LIMIT** **NO** | (**YES** | *positive_integer* [**CONTINUE** | **TERMINATE**])]
CALL *string_var* **WITH** *optimizer_param* {, *optimizer_param*}

optimizer_param ::= *ident* = *number_unit* {, *number_unit*} ;

parenthesized ::= (*boolean_expression*)

positive_digit ::= **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

positive_integer ::= {**0**} *positive_digit* {*decimal_digit*}

positive_number ::= (*positive_integer* [*fractional_part*]) | (<**0**> . *positive_integer* [**E** *signed_integer*])

presentation ::= **PRESENTATION** [*ident*] **TEMPLATE** *string_var* **WITH** *use_list*
 [*printer_context*] [*print_destination*] ;

The optional *ident* is the title of the print job. This title is required if the result of the print will be used in a subsequent statement. *string* refers to a presentation layout file.

print_attribute ::= *ident* = *ident* | *simple_constant*

print_destination ::= **TO** *printer_or_file* {**AND** *printer_or_file*}

print_mode ::= **CONSOLE** | **DIALOG** | **LOG**

printer_context ::= **CONTEXT** *string_var*

printer_or_file ::= (**JOB PRINTER**) | (**PRINTER** [*string_var*]) | (**FILE** [**REF**] *string_var*) [*printer_context*]

The first *string_var* is either the name of a printer or a variable containing the name of a printer. The second *string_var* is a literal filename or an identifier containing a filename to where the print output will be saved. The extension of this filename will dictate the type of file that will be generated.

printing ::= **PRINT** [*ident*] **DATA** *arrayed_ident*
 [**WITH** ((**ALL**) **PARAMETER**[**S**]) | (**PARAMETER LIST** *string_var* | ((*string_list*)))]
 [*printer_context*] [*print_destination*] ;

The optional *ident* is the title of the print job. This title is required if the result of the print will be attached to an email. The *arrayed_ident* identifies the data file that will be printed by referencing a title of an EXPERIMENT block.

printing_context ::= **PRINT CONTEXT** *ident* **IS** <*print_attribute* ;> **END** [**PRINT**] **CONTEXT** ;

probe_tune ::= **PROBE** [**FORCE** [*boolean_value* | *ident*]] [**DUAL** [*boolean_value* | *ident*]]
 <**COIL** *string_var* **DOMAIN** *string_var* **OFFSET** *number_unit* | *ident*>

processing ::= **PROCESS** [**DATA**] *arrayed_ident*
 [**WITH** | **ELSE** *string_var*] [**TO** *arrayed_ident*] [*save_clause*] ;

The first *arrayed_ident* is the data file to process. The second optional *arrayed_ident* is the variable to store the result. *string_var* is a literal filename or identifier referencing a filename containing a processing list. If *string_var* is an identifier, then it must be of the TEXT Type.

promote_scope ::= **PROMOTE SHIMS** | *string_var* {**AND SHIMS** | *string_var*} **TO** *scope_level* ;

prompt ::= **PROMPT** *string_var* *prompt_options* ;

string_var is a literal String or a Text variable that asks a question of the operator. The query will be displayed on the information area of the Spectrometer Control tool. The result needs to be stored into a properly typed variable.

prompt_answer ::= (*constant* | *ident*) | (**SHOW** *constant* | *ident* **AS** *string*)

prompt_answers ::= *prompt_answer* <,*prompt_answer*>

prompt_statement ::= **PROMPT** *string_var* **TO** *ident* *prompt_options* ;

prompt_options ::= [**OPTIONS** | **BUTTONS** *prompt_answers*] [**DEFAULT** *string_var*] [**ICON** *warnings* | *string*]

raise_error ::= **RAISE** *string* [*when_clause*] ;

string is the name of the error to raise.

relational_op ::= = | /= | < | <= | > | >=

repeat_block ::= **REPEAT** [*loop_expression* **DO**]
 loop_block
 [**THEN**
 statement_block]
 END REPEAT ;

retention ::= **RETAIN** *arrayed_ident* [*save_clause*] ;

sample_attr_rem ::= **SAMPLE REMOVE** *string* {**AND** | , *string*} ;

sample_set_modes ::= [**SAVE** | **INTERIM**] [**CONST**] [**CONCEAL**]

The order of these keywords does not matter when more than one is specified.

save_clause ::= **SAVE** [**AS**] *string_var*

string_var is a filename or a variable that contains a filename.

scope_level ::= **USER** | **PROJECT** | **JOB** | **SAMPLE** | **METHOD**

seconds ::= *nonneg_number* **SECOND**[**S**]

send_email ::= **EMAIL** [**ALERT**] [**TO**] *addresses* {**CC** | **BCC** *addresses*} [**SUBJECT** *string_var*]
MESSAGE *translatable_list*
 {**ATTACH** *arrayed_ident* {**AND** *arrayed_ident*}} ;

It is possible for the author of the Automation script to include variables into the SUBJECT line or the MESSAGE text by including the variable name of the desired value preceded by a back-slash ‘\’ character. The variable name will be replaced by the textual representation of the variable’s current value. The textual representation of a DATA Type variable is the filename of the data. *arrayed_ident* identifies a data file or printed output variable to include in the email as an attachment.

signed_integer ::= [*number_sign*] *integer*

simple_constant ::= *boolean_value* | *string* | *number_unit*

simple_expression ::= [*number_sign*] *term* {*number_sign* | *relational_op term*}

simple_type ::= **BOOLEAN** | **NUMBER** | **TEXT** | **LIST**

Variables of the **BOOLEAN** Type can hold values of TRUE or FALSE.
 Variables of the **NUMBER** Type can hold numeric values and numeric values with units.
 Variables of the **TEXT** Type can hold Strings.
 Variables of the **LIST** Type can hold sets of the above three Types and DATA.

statement ::= *action_statement* | *assign_block* | *basic_statements* | *code* | *collection* | *delay_statement* |
error_handler | *group* | *if_block* | *inform* | *invocation* | *limit_block* | *presentation* | *printing* |
printing_context | *processing* | *promote_scope* | *prompt_statement* | *raise_error* | *repeat_block* |
retention | *send_email* | *termination* | *tuning* | *visualization* | *macro_expansion*

statement_block ::= <*statement* | *variable_def*>

string ::= “ {Character} ”

string_list ::= *string* { , *string* }

string_var ::= *string* | *ident*

sub_expression ::= *simple_expression* { *relational_op* *simple_expression* }

term ::= *factor* { *multiplier factor* }

termination ::= *termination_mode* [*when_clause*] ;

termination_mode ::= **FINISH** | (**TERMINATE** [[**WITH**] **STATUS** *string_var*])

FINISH will end the executing Automation Method and retain any data collected.
TERMINATE will end the executing Automation Script as well as remove all generated files including acquired data and printed files. Obviously, printed files or sent email cannot be undone.

time ::= *time_short* | *time_long* | *ident*

time_long ::= (*days* [*hours*] [*minutes*] [*seconds*]) | (*hours* [*minutes*] [*seconds*]) | (*minutes* [*seconds*]) | *seconds*

time_12hour ::= ([**0**] *decimal_digit*) | (**1 0** | **1 2**) : *time_60minutes* [**AM** | **PM**]

time_24hour ::= ([**0** | **1**] *decimal_digit*) | (**2 0** | **1 2** | **3**) : *time_60minutes*

time_60minutes ::= (**0** | **1** | **2** | **3** | **4** | **5**) *decimal_digit*

time_of_day ::= *time_12hour* | *time_24hour*

time_short ::= [[*nonneg_number* :] *nonneg_number* :] *nonneg_number* : *nonneg_number*

translatable_list ::= *string_var* { , *string_var* }

translation ::= **TRANSLATE** *ident* <[*Language_Code* | **ALL**] *string_list* ;>

The *ident* defines a new translation key to be used in place of Strings in various statements.
Language_Code is one of the standard two or three letter language codes defined by ISO 639. Only one instance of **ALL** or *Language_Code* being omitted is permitted.

tuning ::= **TUNE** *probe_tune* ;

type_declaration ::= *enum_declaration* | *number_declaration* | *list_declaration*

typed_ident ::= *ident* : *base_type* | *user_type*

typed_ident_nodata ::= *ident* : *simple_type* | *user_type*

unit ::= [**Unit**]

units_constraint ::= [**WITH**] (**UNIT** *unit* [**FROM** *arrayed_ident*]) | (**NO UNIT**)

arrayed_ident is the name of the data file to use for the unit conversion.

use_entry ::= [**DATA**] *arrayed_ident* [**FOR** [**PARAMETER**] *integer* | *string_var*]

arrayed_ident is the name of a data file. The value after the **PARAMETER** keyword indicates the runtime parameter name or number for the presentation template.

use_list ::= *use_entry* { **AND** *use_entry* }

user_type ::= { *ident* . } *ident*

The last of all the *idents* is the name of a user defined Type. All preceding *idents* are domain names.

var_def ::= **VAR** *inout_data_def* | (*typed_ident_nodata* [= *initial_value*])

Creates a named reference to a value that can be modified by the SET statement.

var_exposed ::= **EXPOSE** [**ACTIVE** | **PASSIVE**] *const_def* | *var_def* [, *dependency*]

variable_def ::= ([**CONCEAL**] *const_def*) | ([**CONCEAL**] *var_def*) | (*var_exposed* [, *help_info*]) ;

version ::= **AUTOMATION** [**TYPES**] **VERSION** *positive_integer* [**PURPOSE** *string_list*] ;

visualization ::= **VISUALIZE** *visualized_list* { **AND** *visualized_list* } ;

visualized_list ::= *visualized_overlays* { **AND** *visualized_overlays* } [**IN** *string*]

The *string* following the optional keyword IN is the name of a Percival tool that can receive a file or set of files as its first parameter.

visualized_overlay ::= *ident* { **WITH** *ident* }

warnings ::= **INFO** | **STATUS** | **WARNING** | **ALERT** | **ERROR** | **FATAL**

Various status levels for messages from least important to most importance.

when_clause ::= **WHEN** *conditional_test*

wildcard_assign ::= * = *association* | *ident* | (**NAMESPACE** *string_var*)

Future ideas:

wait ::= **WAIT** *wait_rule* [**INTERVAL** *time*] [**EXPIRES** *time*] [**THEN** *statement_block*] ;

Holds up execution by periodically checking the *wait_rule*. The check will happen every *N* seconds or after the *time* specified after the **INTERVAL** keyword has past.
EXPIRES will place a maximum amount of time to pause giving the *wait_rule* a chance to succeed.
 The **THEN** clause is only executed if the *time* after the **EXPIRES** keyword has past. Execution continues with the following statement when either the *wait_rule* succeeds or the *time* expires – whichever occurs first.

wait_rule ::= (**UNTIL** | **WHILE** *conditional_test*) | (**UNTIL SIGNAL** *string*)

UNTIL will cause the execution to hold until the *conditional_test* evaluates to a True value.
WHILE will cause the execution to hold until the *conditional_test* evaluates to a False value.
string is an event received from an external source – possibly Namespace.

delete ::= **DELETE** *arrayed_ident* ;

Removes a file from a Data Server.

report ::= **PRINT** [**PEAKS** | **INTEGRALS**] **DATA** *arrayed_ident* [*printer_context*] [*print_destination*] ;

set_process_list ::= **SET PROCESS** *ident* = *string* ;

Sets the unapplied processing list in a data file. *Ident* must reference data and *string* must name a processing list.

Symbol	Description
[]	Optional - 0 or 1 of items within.
{ }	Optional series - 0 or more of items within.
< >	Series - 1 or more of items within.
	Choice of left or right side. (Use grouping to put multiple items into a choice.)
()	Group of items.
\$	The end of a physical line of text.

* The rule title in CAPITALS is the root (starting point) of the Automation grammar.