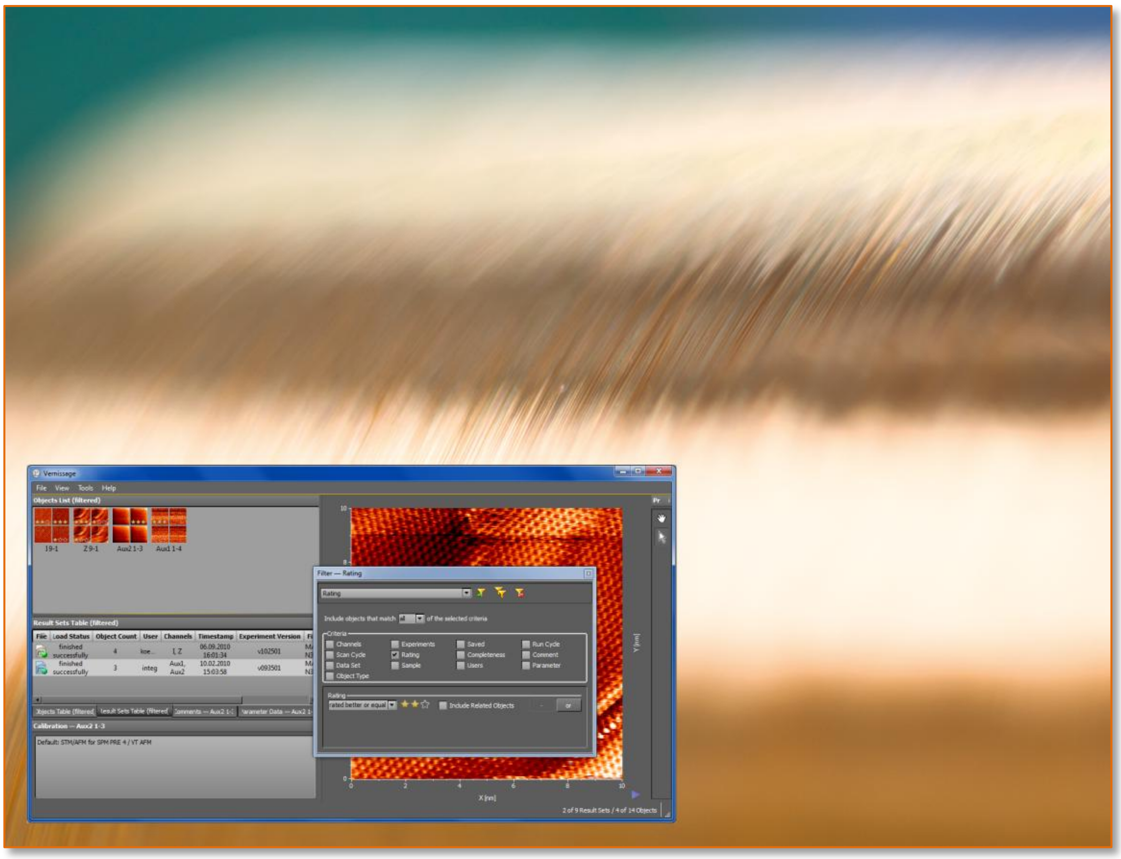
Vernissage Result File Access & Export

Version 2.2 • 31 October 2014



Help Line:

German Headquarters and Service Centre:
Scienta Omicron GmbH
Limburger Str. 75
65232 Taunusstein, Germany
Tel.: +49(0) 61 28 / 987-230
Fax: +49(0) 61 28 / 987-33-230
E-Mail: services@scientaomicron.com

PN05322

scientaomicron

Preface

This document has been compiled with great care and is believed to be correct at the date of print. The information in this document is subject to change without notice and does not represent a commitment on the part of Omicron NanoTechnology GmbH.

Notice

Some components described in this manual may be optional. The delivery volume depends on the ordered configuration.

Notice

This documentation is available in English only.

Caution

Please read the safety information in all related manuals before using the instrument.

Notice

Copyright

Warranty

Omicron acknowledges a warranty period of 12 months from the date of delivery (if not otherwise stated) on parts and labour, excluding consumables such as filaments, sensors, etc.

No liability or warranty claims shall be accepted for any damages resulting from non-observance of operational and safety instructions, natural wear of the components or unauthorised repair attempts.

Contents

List of Figures

List of Tables

1. What is Vernissage?

Vernissage is a tool for accessing any kind of MATRIX measurement data from your data analysis application and for exporting MATRIX result data into other output formats. The Vernissage application supports data preview, browsing, advanced filtering and sorting. Also provided are dedicated plug-ins for data export to different supported output formats such as ASCII, BMP, Omicron Flat File Format, IGOR 5, JPG, PNG, Phi MultiPak, TIFF and VAMAS. However, you can extend the data conversion options by adding Vernissage exporter plug-ins dedicated to other file formats.

If you want to utilise Vernissage for exporting MATRIX result files into a specific output format, you have two options:

1. Visit the MATRIX website on the Internet (www.omicron.de/en/software-downloads/) and check the Plug-Ins section: on this page Omicron offers ready-made Vernissage add-ons and example plug-in code for you to download.

2. If you are somewhat familiar with C++ programming, you may also build and integrate your own Vernissage exporter plug-in.

Notice

Presented code examples are fragments only. They cannot be run as stand-alone program code.

Getting Started

After installation the Vernissage software can be accessed from the Start menu where it provides two entries: a graphical user interface and a command prompt entry.

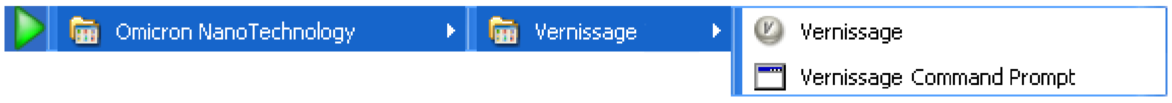


Figure 1. Vernissage entries in your Start menu.

The graphical user interface (GUI) is convenient for interactive data processing where you want to select the relevant files manually. The command prompt method can be handy for batch processing, e.g. if you want to export all measured data at the end of a lab session.

Notice

If you are not familiar with the way the MATRIX software organises result data please have a look on page 33 first.

Vernissage File Path Systematic

The entire matter of paths is complicated business, even more so as it depends on the operating system and the language of the operating system. Vernissage has three important paths that are relevant when manually handling data files:

- The **installation directory**, denoted as **<InstallDir>**, where the executable file resides and where the licence files go.
- The **temporary directory**, denoted as **<TempDir>**, where the log files are saved.
- The **persistence root directory**, denoted as **<RootDir>**, where the Vernissage.ini-file goes.

MATRIX uses the system defined environment variables %TMP%, %APPDATA%, %PROGRAMFILES% and %PROGRAMFILES(X86)% to generate the above paths. The schema works as follows:

Installation Directory

Windows XP:	%PROGRAMFILES%\<Company Name>\<Application Name>\<Software Version>\
Windows 7 (32 bit):	%PROGRAMFILES(X86)%\<Company Name>\<Application Name>\<Software Version>\

Temporary Directory

Windows XP & Windows 7:	%TMP%\Temporary <Application Name> Files\ <Software Version>\

Persistence Root Directory

Windows XP & Windows 7:	%APPDATA%\<Company Name>\<Application Name>\

Place Holders

<Company Name> = "Omicron NanoScience" (used to be "Omicron NanoTechnology" in V2.1 and earlier)
<Application Name> = "Vernissage" (or "MATRIX")
<Software Version>: = "V2.2" (used to be "V2.1" or lower)
<Account Name>: = "Omicron" (default super user), "Matrix" (default user) or your personal user name

Environment Variables

The environment variables can be accessed by typing their name, including both % signs, in the address line of the Windows Explorer. Their contents depends on the operating system, its language and the account user name under which you work.

Examples

The following table gives examples for the different paths for the two common operating systems (old and new) as installed by Omicron.

Windows XP (English)	
<InstallDir>	C:\Program Files\Omicron NanoScience\Vernissage\V2.2\
%TMP% <TempDir>	C:\temp\ C:\temp\Temporary Vernissage Files\V2.2

%APPDATA%
<RootDir>
	C:\Documents and Settings\<Account Name>\Application Data\
	C:\Documents and Settings\<Account Name>\Application Data\Omicron NanoScience\Vernissage\

Windows 7 (32 bit, English)	
<InstallDir>	C:\Program Files (x86)\Omicron NanoScience\Vernissage\V2.2\

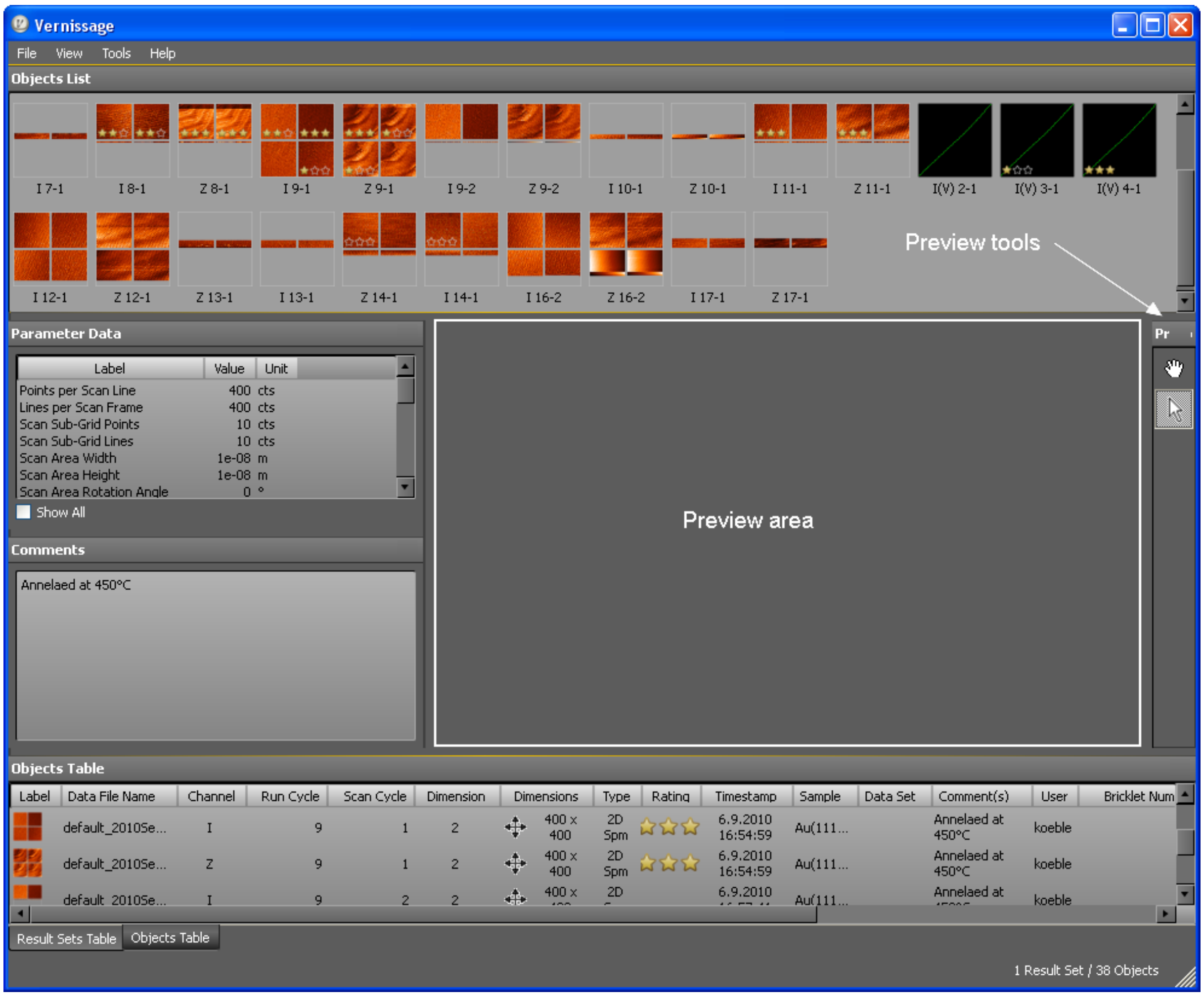
%TMP%
<TempDir>
	C:\Users\<Account Name>\AppData\Local\Temp\
	C:\Users\<Account Name>\AppData\Local\Temp\Temporary Vernissage Files\V2.2

%APPDATA%
<RootDir>
	C:\Users\<Account Name>\AppData\Roaming\
	C:\Users\<Account Name>\AppData\Roaming\Omicron NanoScience\Vernissage\

Table 1. Important paths used by Vernissage.

2. Vernissage Graphical User Interface

Open Vernissage selecting Vernissage from the Microsoft Windows Start menu or via the desktop icon.



Label	Value	Unit
Points per Scan Line	400	cts
Lines per Scan Frame	400	cts
Scan Sub-Grid Points	10	cts
Scan Sub-Grid Lines	10	cts
Scan Area Width	1e-08	m
Scan Area Height	1e-08	m
Scan Area Rotation Angle	0	°

Annelaed at 450°C

Label	Data File Name	Channel	Run Cycle	Scan Cycle	Dimension	Dimensions	Type	Rating	Timestamp	Sample	Data Set	Comment(s)	User	Bricklet Num
🟧	default_2010Se...	I	9	1	2	400 x 400	2D Spm	★★★	6.9.2010 16:54:59	Au(111...		Annelaed at 450°C	koeble	
🟧	default_2010Se...	Z	9	1	2	400 x 400	2D Spm	★★★	6.9.2010 16:54:59	Au(111...		Annelaed at 450°C	koeble	
🟧	default_2010Se...	I	9	2	2	400 x	2D		6.9.2010	Au(111...		Annelaed at	koeble	

Figure 2. Vernissage graphical user interface.

At the top of the window four drop-down menus provide you with a number of options and actions.

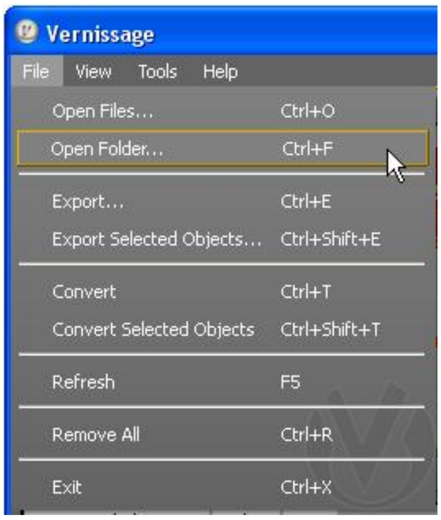
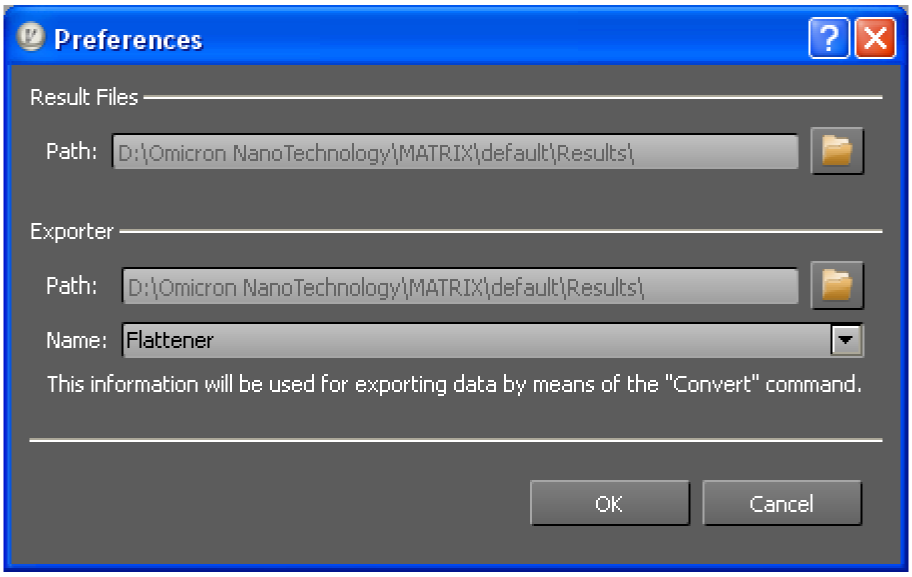


Figure 3. Main window File menu.

- The File menu is for opening and exporting files.
- The View menu allows selecting GUI elements to be visible or invisible.
- The Tools menu allows invoking a Preferences dialogue.
- The Help menu offers information on the current Vernissage version and the available exporters.

Preferences

In the Preferences window, available from the Tools menu, you can preselect import and export paths as well as the export routine you want to use.



Result Files

Path: D:\Omicron NanoTechnology\MATRIX\default\Results\

Exporter

Path: D:\Omicron NanoTechnology\MATRIX\default\Results\

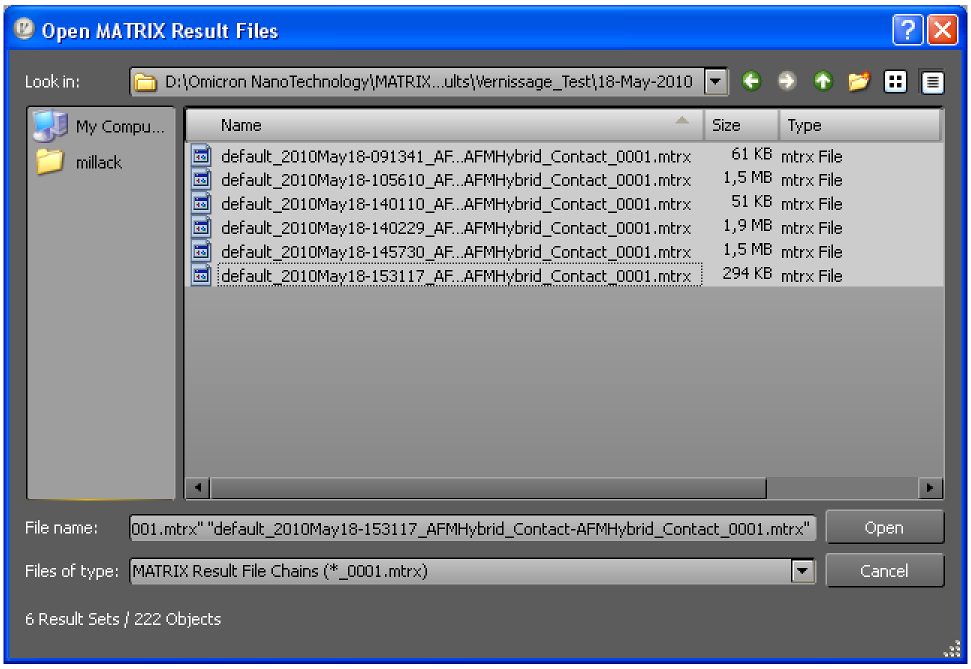
Name: Flattener

This information will be used for exporting data by means of the "Convert" command.

Figure 4. The Preferences window available from the tools menu.

Loading Data into Vernissage

- To open single files or entire folders select the respective entry from the File menu. Navigate to your Results folder, and select the items you want to load and click Open.



Open MATRIX Result Files

Look in: D:\Omicron NanoTechnology\MATRIX...ults\Vernissage_Test\18-May-2010

Name	Size	Type
default_2010May18-091341_AF...AFMHybrid_Contact_0001.mtrx	61 KB	mtrx File
default_2010May18-105610_AF...AFMHybrid_Contact_0001.mtrx	1,5 MB	mtrx File
default_2010May18-140110_AF...AFMHybrid_Contact_0001.mtrx	51 KB	mtrx File
default_2010May18-140229_AF...AFMHybrid_Contact_0001.mtrx	1,9 MB	mtrx File
default_2010May18-145730_AF...AFMHybrid_Contact_0001.mtrx	1,5 MB	mtrx File
default_2010May18-153117_AF...AFMHybrid_Contact_0001.mtrx	294 KB	mtrx File

File name: 001.mtrx" "default_2010May18-153117_AFMHybrid_Contact-AFMHybrid_Contact_0001.mtrx"

Files of type: MATRIX Result File Chains (*_0001.mtrx)

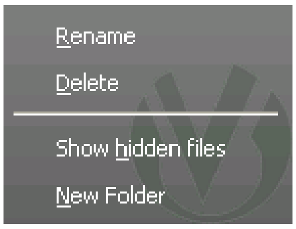
6 Result Sets / 222 Objects

Figure 5. File selection for Vernissage.

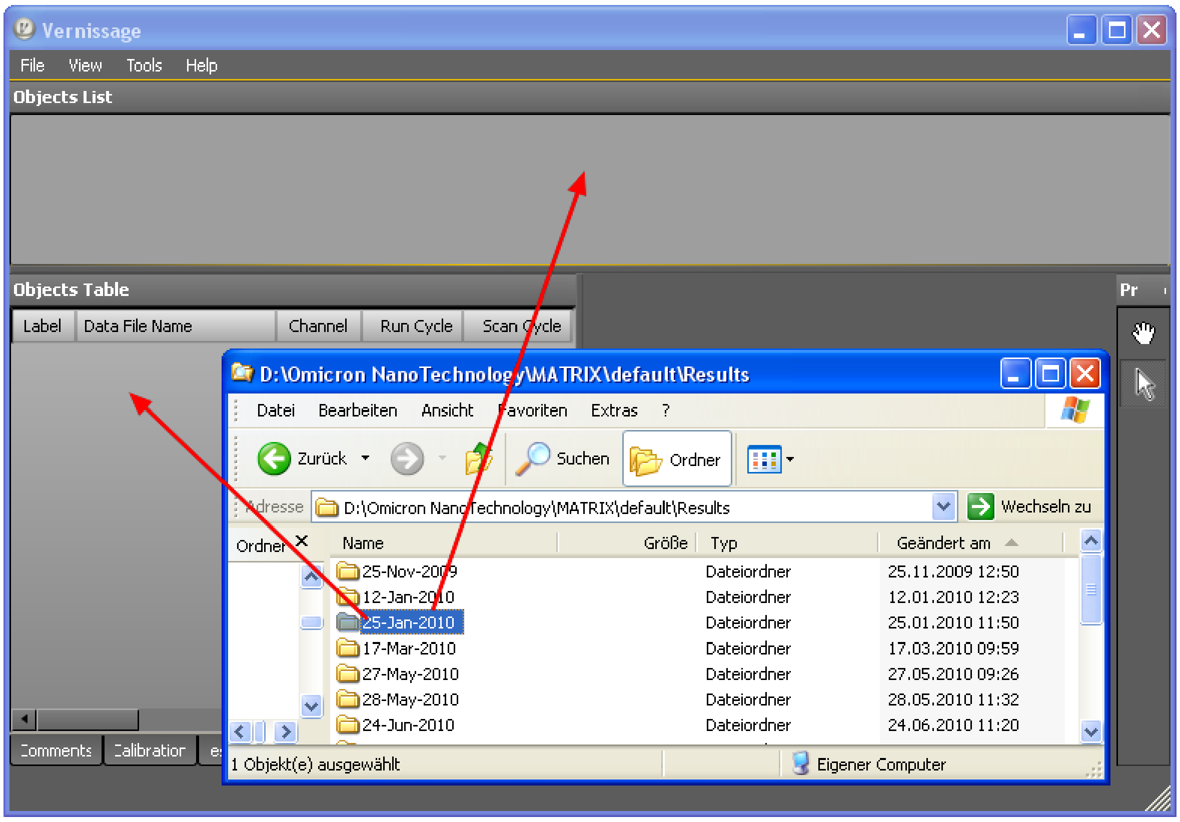
- Click *Cancel* to abort loading if desired. Note however, that data sets already loaded at the time of requesting *Cancel* will stay loaded.

Notice

Hidden files are not shown by default in the open dialogue. If you want to open hidden files select "Show hidden files" from the context menu.



- Alternatively drop files or folders into the Objects List or any other window except the preview area inside Vernissage directly from the Windows Explorer.


Vernissage
File View Tools Help
Objects List
Objects Table
Label Data File Name Channel Run Cycle Scan Cycle
Comments Calibration
Pr
D:\Omicron NanoTechnology\MATRIX\default\Results
Datei Bearbeiten Ansicht Favoriten Extras ?
Zurück Suchen Ordner
Adresse D:\Omicron NanoTechnology\MATRIX\default\Results
Wechseln zu
Ordner
1 Objekt(e) ausgewählt
Eigener Computer

Name	Größe	Typ	Geändert am
25-Nov-2009		Dateiordner	25.11.2009 12:50
12-Jan-2010		Dateiordner	12.01.2010 12:23
25-Jan-2010		Dateiordner	25.01.2010 11:50
17-Mar-2010		Dateiordner	17.03.2010 09:59
27-May-2010		Dateiordner	27.05.2010 09:26
28-May-2010		Dateiordner	28.05.2010 11:32
24-Jun-2010		Dateiordner	24.06.2010 11:20

Figure 6. Loading files per drag and drop.

The loaded data sets will be shown in the main window, see figure 2 on page 11.

Exporting Data from Vernissage

Vernissage comes with a number of ready-made plug-in modules for exporting into different file formats. These can then be used as input files for your own software application. The Omicron Flat File Format has been described in detail, see page 68.

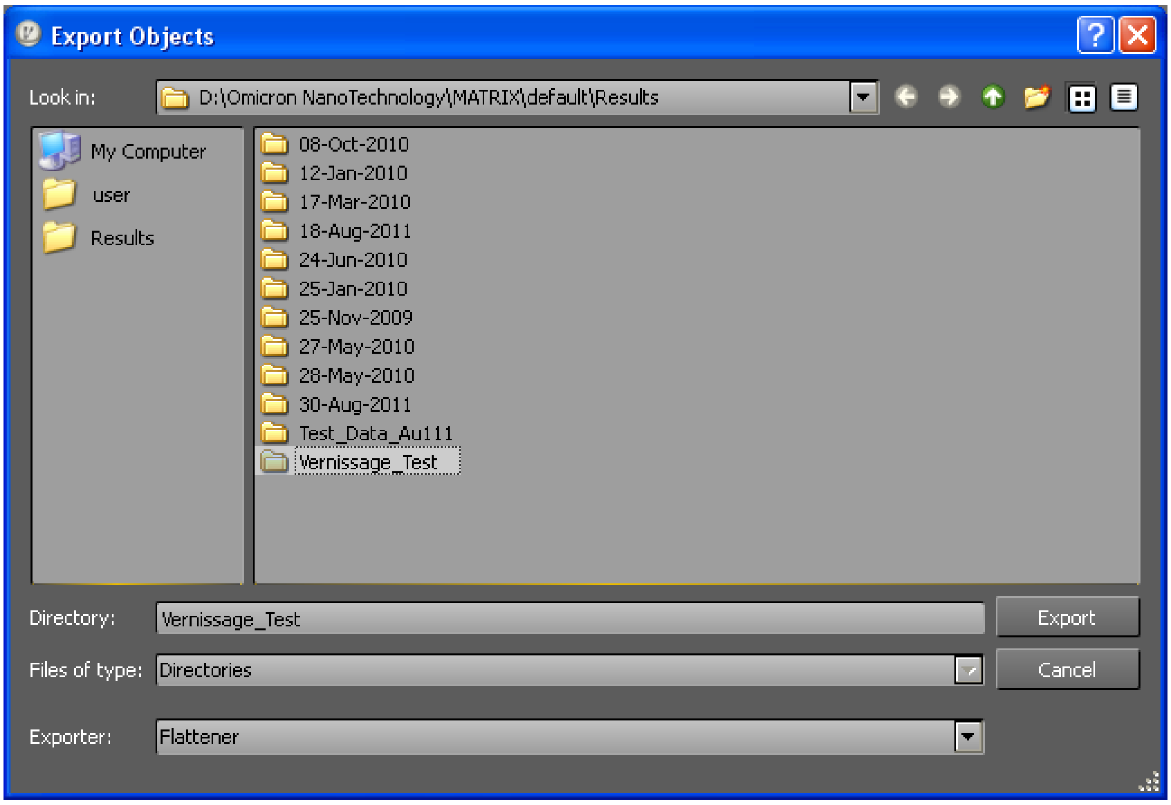
To export all loaded or the selected data sets

- Click *Export…* or *Export Selected Objects…*

- Choose an export routine and a path to create data files in the selected format.

Alternatively

- Click *Convert…* or *Convert Selected Objects…* to perform an export using the previously selected export routine and path.



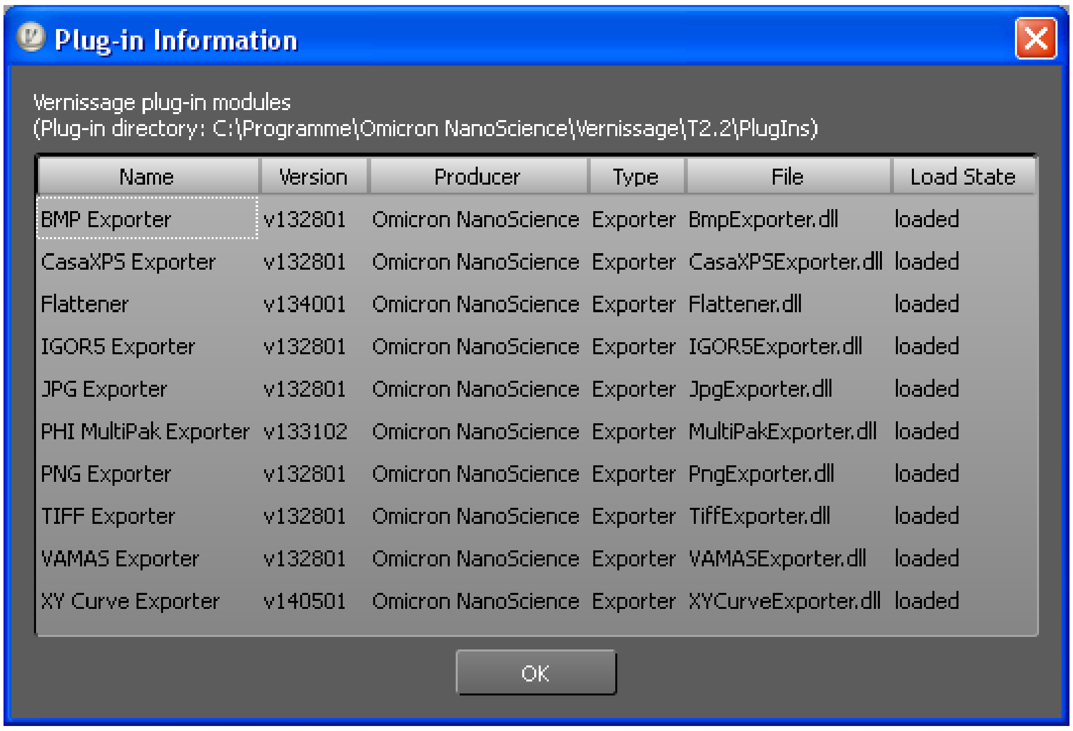
Look in: D:\Omicron NanoTechnology\MATRIX\default\Results

08-Oct-2010
12-Jan-2010
17-Mar-2010
18-Aug-2011
24-Jun-2010
25-Jan-2010
25-Nov-2009
27-May-2010
28-May-2010
30-Aug-2011
Test_Data_Au111
Vernissage_Test

Directory: Vernissage_Test
Files of type: Directories
Exporter: Flattener

Figure 7. Export dialogue window.

The number and type of plug-ins currently linked to Vernissage can be accessed from the *Help* menu:



Vernissage plug-in modules
(Plug-in directory: C:\Programme\Omicron NanoScience\Vernissage\T2.2\PlugIns)

Name	Version	Producer	Type	File	Load State
BMP Exporter	v132801	Omicron NanoScience	Exporter	BmpExporter.dll	loaded
CasaXPS Exporter	v132801	Omicron NanoScience	Exporter	CasaXPSExporter.dll	loaded
Flattener	v134001	Omicron NanoScience	Exporter	Flattener.dll	loaded
IGOR5 Exporter	v132801	Omicron NanoScience	Exporter	IGOR5Exporter.dll	loaded
JPG Exporter	v132801	Omicron NanoScience	Exporter	JpgExporter.dll	loaded
PHI MultiPak Exporter	v133102	Omicron NanoScience	Exporter	MultiPakExporter.dll	loaded
PNG Exporter	v132801	Omicron NanoScience	Exporter	PngExporter.dll	loaded
TIFF Exporter	v132801	Omicron NanoScience	Exporter	TiffExporter.dll	loaded
VAMAS Exporter	v132801	Omicron NanoScience	Exporter	VAMASExporter.dll	loaded
XY Curve Exporter	v140501	Omicron NanoScience	Exporter	XYCurveExporter.dll	loaded

Figure 8. Vernissage plug-in information.

BMP Exporter	The BMP Exporter plug-in module converts curve and image data from any MATRIX experiment into the "Microsoft Windows Bitmap" format.
CasaXPS Exporter	The CasaXPS Exporter is the same as the VAMAS Exporter but it additionally forwards the converted data to the CasaXPS data analysis software if installed on your PC. If the CasaXPS software is not running, Vernissage will launch it automatically. Note: The following environment variables must be set: • VERNISSAGE_CASA_XPS_LOCATION containing the path to the CasaXPS installation directory and • VERNISSAGE_CASA_XPS_PORT=7000
Flattener Exporter	The Flat File Format is a binary data format storing raw measurement data as well as additional information required to interpret the raw data correctly. For more details see chapter Flat File Format Reference from page 68.
IGOR 5 Exporter	This Exporter plug-in module is capable of exporting data acquired by MATRIX SPM experiments and electron spectroscopy curves into the "IGOR 5" format supported by the IGOR Pro technical graphing and data analysis software by WaveMetrics, Inc.
JPG Exporter	The JPG Exporter plug-in module converts curve and image data from any MATRIX experiment into the JPEG image file interchange format ISO/IEC 10918-1 (or CCITT Recommendation T. 81, respectively.)
PHI MultiPak Exporter	This Exporter plug-in module converts data from electron spectroscopy experiments into file structures readable by the *MultiPak* data processing software by Physical Electronics, Inc.
PNG Exporter	The PNG Exporter plug-in module converts curve and image data from any MATRIX experiment into the ISO 15948/IETF RFC 2083 "Portable Network Graphics" format.
TIFF Exporter	The TIFF Exporter plug-in module converts curve and image data from any MATRIX experiment into the IETF RFC 2302 "Tagged Image File Format".
VAMAS Exporter	The VAMAS Exporter plug-in converts data from electron spectroscopy experiments (both spectra and image maps) into the ISO 14976: 1998 "Surface chemical analysis — Data transfer format", often also referred to as "VAMAS format" as it was defined by various members of the *Versailles Project on Advanced Materials and Standards* (VAMAS) community. File contents generated by the VAMAS Exporter can be interpreted by various 3rd party software products, such as the CasaXPS data analysis software by Casa Software, Ltd.
XY Curve Exporter	This Exporter is capable of exporting one-dimensional result data such as electron spectroscopy curves, data from SPM Single Point Spectroscopy operations, Force/Distance curves, and similar. The plug-in will write plain text files using a simple two-column format; important parameter settings will also be included in the output generated.

Please note that the image format modules (BMP Exporter, JPG Exporter, PNG Exporter and TIFF Exporter) are not capable of exporting result data from SPM volume CITS experiments.

If you want to transfer MATRIX data into a special format you can also write your own plug-ins and link them to Vernissage. The greater part of the manual will explain how to do this.

	"Spm Spectroscopy" ~ Session::btc_SPMSpectroscopy	"Atom Manipulation" ~ Session::btc_AtomManipulation	"Force Curve" ~ Session::btc_ForceCurve	"Phase/Amplitude Curve" ~ Session::btc_PhaseAmplitudeCurve	"Signal Over Time" ~ Session::btc_SignalOverTime	"Interferometer Curve" ~ Session::btc_InterferometerCurve	"1D Curve" ~ Session::btc_1DCurve	"2D Spm" ~ Session::btc_SPMImage	"Path Spectroscopy" ~ Session::btc_PathSpectroscopy	"ESp Region" ~ Session::btc_ESpRegion	"Volume CITS" ~ Session::btc_VolumeCITS	"Raw Path Spectroscopy" ~ Session::btc_RawPathSpectroscopy	"ESp SemSam" ~ Session::btc_DiscreteEnergyMap	"ESp Snapshot Sequence" ~ Session::btc_ESpSnapshotSequence	"2D ESp" ~ Session::btc_ESpImage	"ESp Image Map" ~ Session::btc_ESpImageMap
"ASCII Exporter"	+	+	+	+	+	+	+	+	-	-	+	-	-	-	-	-
"Flattener"	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+	+
"IGOR5 Exporter"	+	+	+	+	+	+	+	+	+	+	+	+	-	+	-	-
"PHI MultiPak Exporter"	-	-	-	-	-	-	-	-	+	+	-	+	+	+	+	+
"BMP Exporter", "JPG Exporter", "PNG Exporter", "TIFF Exporter"	+	+	+	+	+	+	+	+	+	+	-	+	+	+	+	+
"SCALA PRO Formatter"	+	-	+	-	-	-	-	+	-	-	+	-	-	-	-	-
"CasaXPS Exporter", "VAMAS Exporter"	-	-	-	-	-	-	-	-	+	+	-	+	+	+	+	+
"XY Curve Raw Exporter", "XY Curve Exporter"	+	+	+	+	+	+	+	-	+	+	-	+	-	+	-	-

Table 2. Exporters versus Object Types.

Converting Data

The Convert menu commands do the same as the corresponding Export commands but without the dialogue, i.e. they export all or the selected files to the previously used destination employing the previously used export routine. If no Export was performed prior to starting Convert, Vernissage uses the path and routine specified in the Preferences window, see figure 4 on page 12.

View Options

The view menu allows selecting the elements to be shown, such as Preview Tools, Result Sets Table, Objects Table, Objects List, Parameter Data, Calibration Data, Comments and Filter. All elements can be selected or de-selected independently. All windows can be fixed to the main window frame or moved around freely.

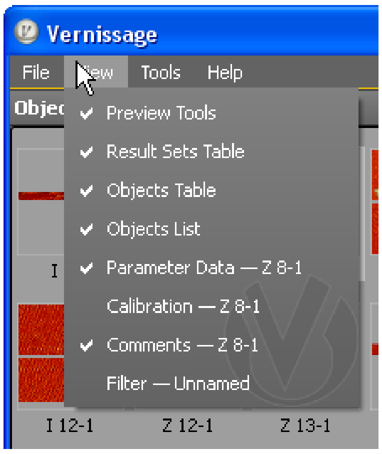


Figure 9. The View menu.

Preview Tools

The preview tools are normally attached to the right hand side window frame.

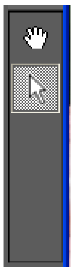
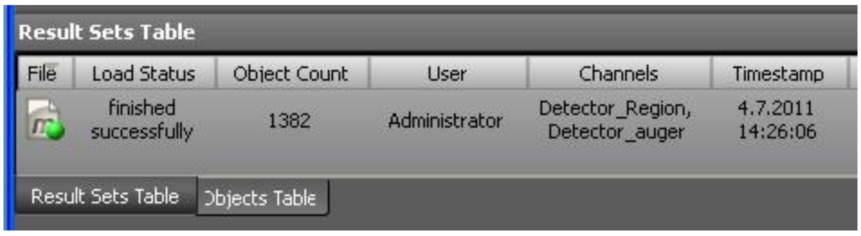


Figure 10. Preview tools, detached window.

With the Hand symbol you can move objects around in the preview area, while the arrow is for pointing and selecting. With the arrow active you can also temporarily switch to the hand pointer using the Shift key.

Result Set Table

The Result Sets Table is normally located at the bottom of the window; it shares the space with the Objects Table. A right-click on the table header lets you chose which columns to display in which order.

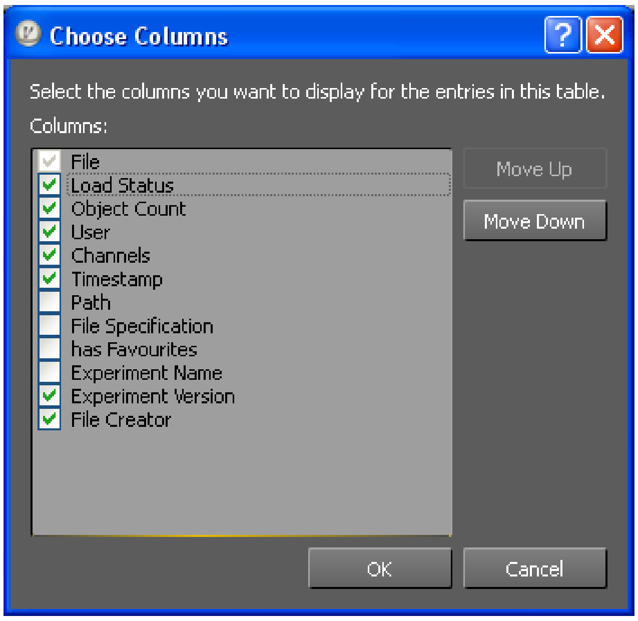


Result Sets Table

File	Load Status	Object Count	User	Channels	Timestamp
	finished successfully	1382	Administrator	Detector_Region, Detector_auger	4.7.2011 14:26:06

Figure 11. Result Set Table.

In the Choose Columns window select the information to be displayed and the positions of the columns. Click OK when ready.



Choose Columns

Select the columns you want to display for the entries in this table.

Columns:

- [✓] File
- [✓] Load Status
- [✓] Object Count
- [✓] User
- [✓] Channels
- [✓] Timestamp
- [ ] Path
- [ ] File Specification
- [ ] has Favourites
- [ ] Experiment Name
- [✓] Experiment Version
- [✓] File Creator

Figure 12. Choose Columns for Result Set Table.

The Result Sets table context menu allows displaying messages (when you right click on the Load Status column) and removing one or all result set(s) from the list (all columns).

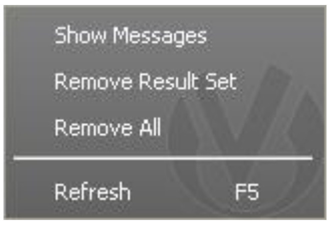
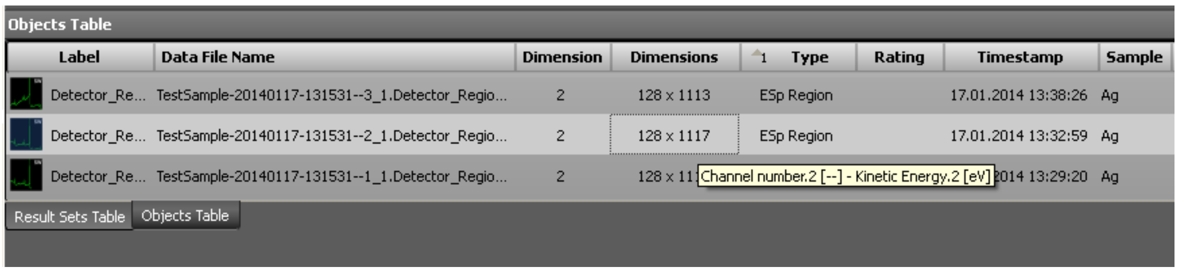


Figure 13. Result Set Table context menu.

Objects Table

Instead of the selected information on the entire result sets you can also list all objects separately by selecting the "Objects Table" tab at the bottom. As above, you can select the columns of interest by right-clicking the table header. Hovering the curser over the dimensions column will display a tool tip stating the cannel number, axes and measured signal/units.



Objects Table

Label	Data File Name	Dimension	Dimensions	1 Type	Rating	Timestamp	Sample
Detector_Re...	TestSample-20140117-131531--3_1.Detector_Regio...	2	128 x 1113	ESp Region		17.01.2014 13:38:26	Ag
Detector_Re...	TestSample-20140117-131531--2_1.Detector_Regio...	2	128 x 1117	ESp Region		17.01.2014 13:32:59	Ag
Detector_Re...	TestSample-20140117-131531--1_1.Detector_Regio...	2	128 x 11			2014 13:29:20	Ag

Channel number.2 [--] - Kinetic Energy.2 [eV]

Figure 14. Objects Table, example shown.

Objects List

The Objects List is the most prominent feature in Vernissage after loading some data. It shows graphical representations of the objects loaded, together with their names and rating. Some of the objects shown in the figure below originate from an SPM experiment with data acquisition activated for forward and backward as well as up and down scans, resulting in four images per object. For phase/amplitude curve pairs the related curve is shown in the background.

A number of image overlays denote additional information

- Filled or empty golden stars denote the user rating of the image, if present.
- Little arrows denoting subsequent objects (repeated SPS, Signal over Time, ESp Snapshot sequence).
- A dog ear indicates more than one Peak Background Set.
- Short descriptors in the top right corner of the thumbnails indicate different ESpec acquisition types: SN (Snapshot) , SW (Sweep), and TH (Threshold).

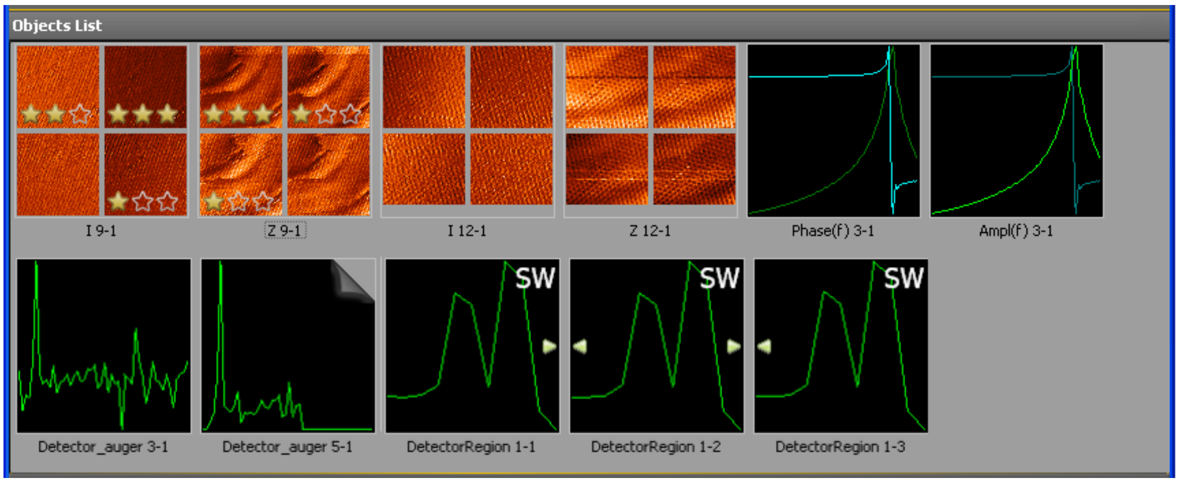


Figure 15. Objects List, example shown. Note that this image has been edited to show SPM (upper row) and ESpec data (lower row) at the same time.

You can double-click the images in the Objects list to show a bigger image in the preview region of Vernissage. Alternatively you can drag objects into the preview area or select the respective option from the context menu shown below.

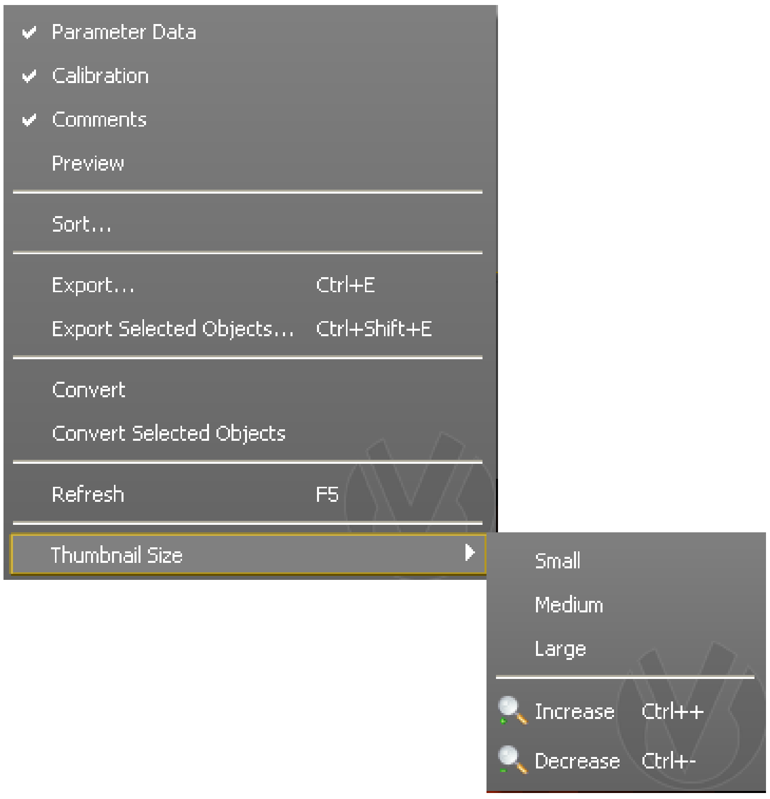


Figure 16. Objects List context menu.

The Objects List context menu offers some additional options:

- Switch Parameter Data and Comments windows/tabs on/off
- Open object in preview area
- Sort objects (various columns available)
- Export (this or all selected objects)
- Convert (this or all selected objects)
- Adjust thumbnail size (three fix values or continuous). The original thumbnail size is "medium"

Sorting

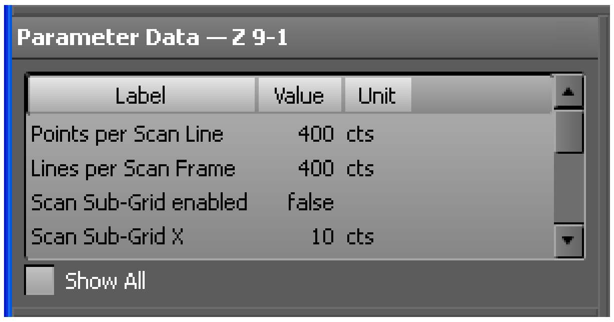
Sorting of the displayed objects can be achieved by

- Specifying sort orders via a dialogue available from the context menu
- Clicking the columns headers. Select subsequent columns with the CRTL key pressed.

Note, however, that not all columns can be used for sorting.

Parameter Data

The Parameter Data field shows the most relevant data for a selected object. Tick the Show All box to display all parameter data available for this data object.



Parameter Data — Z 9-1

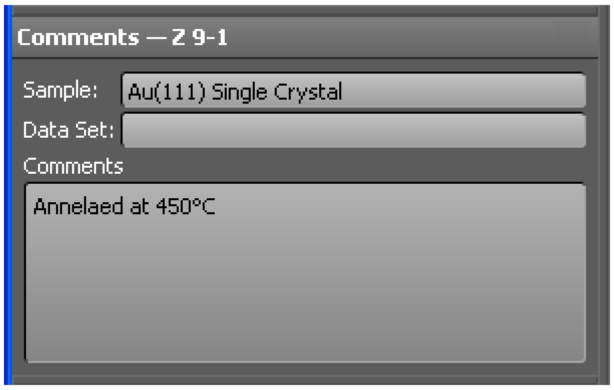
Label	Value	Unit
Points per Scan Line	400	cts
Lines per Scan Frame	400	cts
Scan Sub-Grid enabled	false	
Scan Sub-Grid X	10	cts

Show All

Figure 17. Parameter Data field, example shown.

Comments

Display the comments that were saved with the object. Comments can be saved in the field that pops up upon starting an experiment and also later on in MATRIX (see MATRIX Application Manual for details). Note that this field stays empty if no comment was saved upon data acquisition.



Comments — Z 9-1

Sample: Au(111) Single Crystal

Data Set:

Comments

Annelaed at 450°C

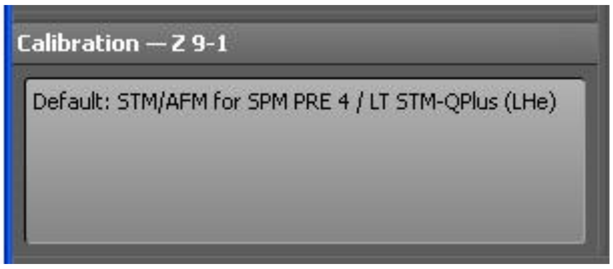
Figure 18. Comments field.

Notice

Note that the Objects Table only shows the first of the available comments, i.e. the global comment if there is one or the first data comment otherwise. If more than one comment is available this will be indicated with an ellipsis "…"

Calibration

This field shows the calibration and parameter sets selected in MATRIX (Tools → Calibration Selection...).



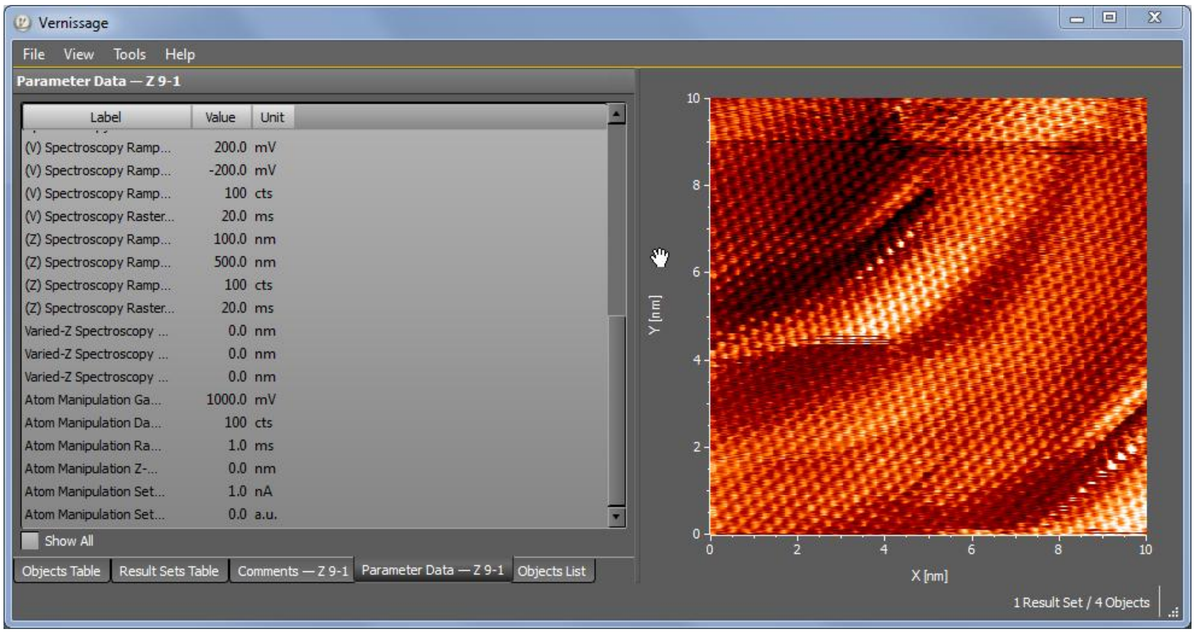
Calibration — Z 9-1

Default: STM/AFM for SPM PRE 4 / LT STM-QPlus (LHe)

Figure 19. Calibration field.

Merge Windows

All inner windows that are attached to the outer window frame can be merged and displayed in tabs as shown in figure 20 below. To do so move one window on top of another until a light blue frame shows that the window is in catch distance and drop it there. When you move a window around you will probably find other catch frames also – just select the desired location.

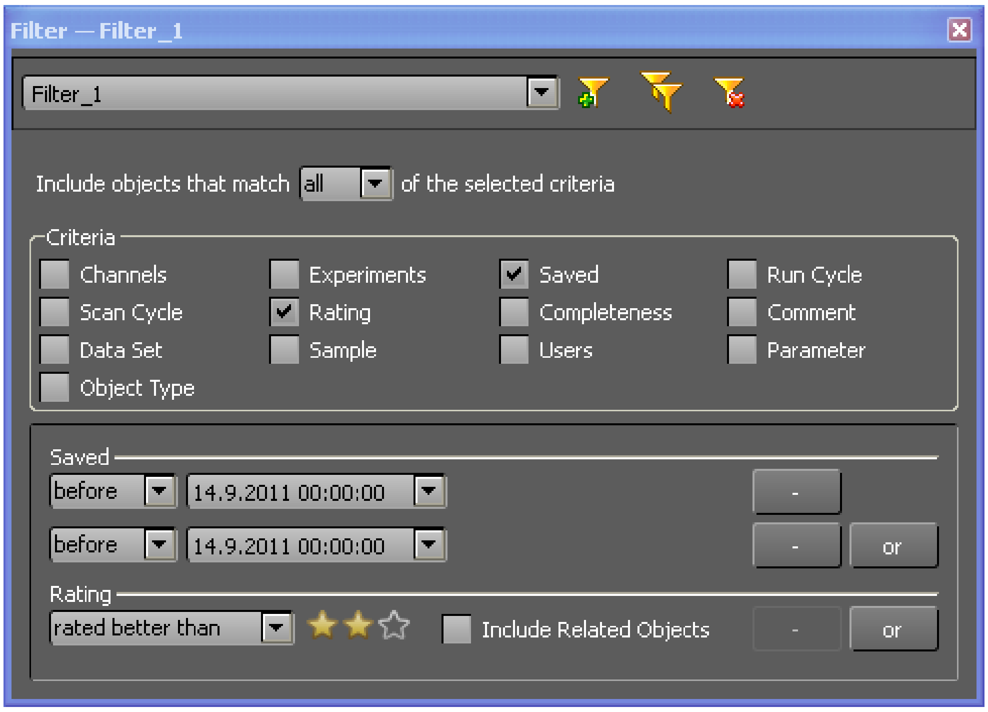


Label	Value	Unit
(V) Spectroscopy Ramp...	200.0	mV
(V) Spectroscopy Ramp...	-200.0	mV
(V) Spectroscopy Ramp...	100	cts
(V) Spectroscopy Raster...	20.0	ms
(Z) Spectroscopy Ramp...	100.0	nm
(Z) Spectroscopy Ramp...	500.0	nm
(Z) Spectroscopy Ramp...	100	cts
(Z) Spectroscopy Raster...	20.0	ms
Varied-Z Spectroscopy ...	0.0	nm
Varied-Z Spectroscopy ...	0.0	nm
Varied-Z Spectroscopy ...	0.0	nm
Atom Manipulation Ga...	1000.0	mV
Atom Manipulation Da...	100	cts
Atom Manipulation Ra...	1.0	ms
Atom Manipulation Z-...	0.0	nm
Atom Manipulation Set...	1.0	nA
Atom Manipulation Set...	0.0	a.u.

Figure 20. Vernissage GUI with merged windows.

Filtering

Vernissage provides a powerful filtering machine that allows you to select exactly those data objects you want out of the host of data files you might have saved "just in case". The most obvious choices are data with a 3-star rating or data from a certain time span.


Filter — Filter_1

Filter_1

Include objects that match: all — of the selected criteria

Criteria:
- [ ] Channels
- [ ] Scan Cycle
- [ ] Data Set
- [ ] Object Type
- [ ] Experiments
- [✓] Rating
- [ ] Sample
- [✓] Saved
- [ ] Completeness
- [ ] Users
- [ ] Run Cycle
- [ ] Comment
- [ ] Parameter

Saved
before 14.9.2011 00:00:00
before 14.9.2011 00:00:00

Rating
rated better than ★★☆ [ ] Include Related Objects

Figure 21. Filter dialogue window with two options ticked.

The above image shows the filter dialogue. If you select more than one entry you can also select if you want

- Data that match all your given restrictions,
- Data that match some, i.e. at least one but not necessarily all, restrictions or
- Data that match none of the selected conditions.

The latter option is useful for, say, discarding all data from a specific sample and just show the remaining objects. You can also use the MATRIX comment field and write something like "test_only" while you are still fiddling around with the settings so you can later filter these objects out with Vernissage.

Notice

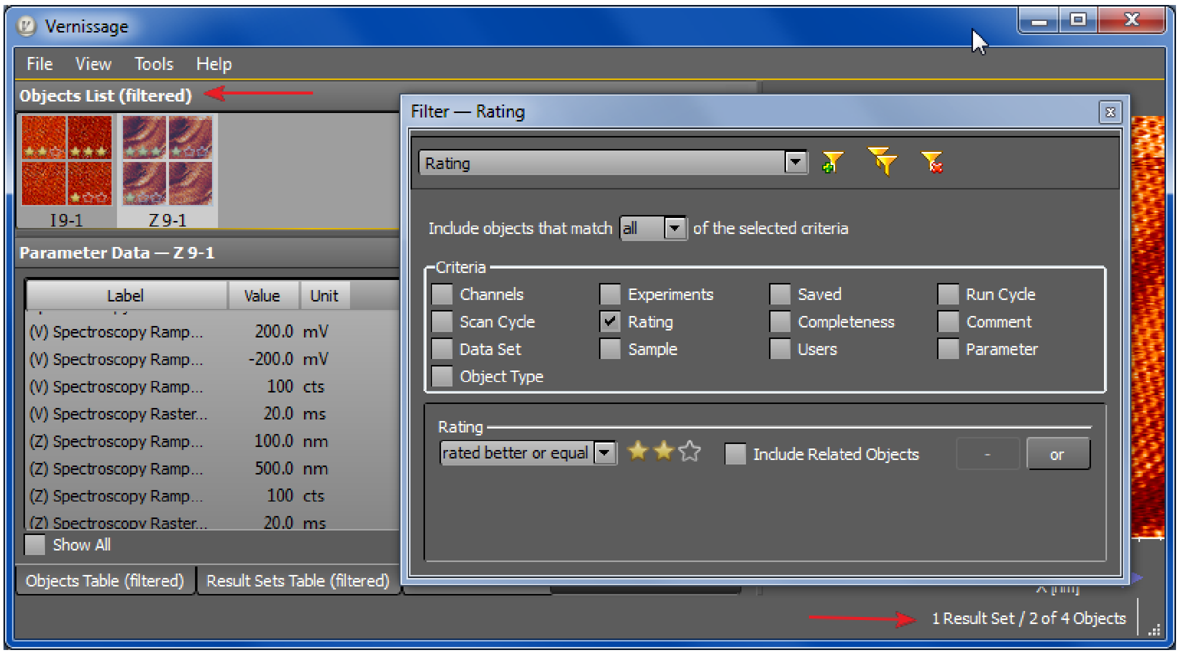
For filtering strings like *Comment, Data Set* or *Sample* the following rule applies:

- The search string is compared **case sensitive** if at least one character is given in upper case.
- The search string is compared **case insensitive** if all characters are given in lower case.

Notice

For rated objects you can explicitly include related objects, e.g. objects from repeated operations, by ticking the respective check box.

The following figure shows Vernissage with filtering active. Note that the Objects List, Objects Table and Result Set Table are updated as soon as you define a restriction, giving you a direct response. The status bar also shows the filtering result. To return to displaying all objects set Criteria to "unset" or un-tick the respective boxes. Note that Vernissage will restore the previous settings when you un-tick and then re-tick a box.



Label	Value	Unit
(V) Spectroscopy Ramp...	200.0	mV
(V) Spectroscopy Ramp...	-200.0	mV
(V) Spectroscopy Ramp...	100	cts
(V) Spectroscopy Raster...	20.0	ms
(Z) Spectroscopy Ramp...	100.0	nm
(Z) Spectroscopy Ramp...	500.0	nm
(Z) Spectroscopy Ramp...	100	cts
(Z) Spectroscopy Raster...	20.0	ms

Filter — Rating

Rating

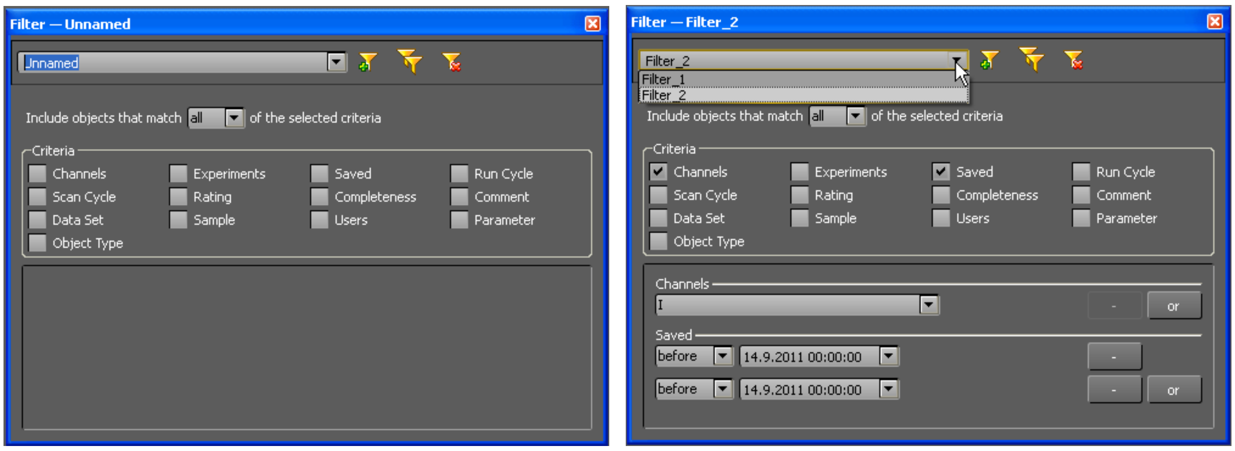
Include objects that match all of the selected criteria

Criteria: Channels, Experiments, Saved, Run Cycle, Scan Cycle, [✓ Rating], Completeness, Comment, Data Set, Sample, Users, Parameter, Object Type

Rating: rated better or equal ★★☆ Include Related Objects - or

Figure 22. Filtering data in Vernissage.

Use the funnel buttons next to the filter name field to generate new filters, either by defining them from scratch or by duplicating an existing filter and changing the settings. The name for a new or copied filter can be set directly in the name field.


Filter — Unnamed

Unnamed

Include objects that match all of the selected criteria

Criteria: Channels, Experiments, Saved, Run Cycle, Scan Cycle, Rating, Completeness, Comment, Data Set, Sample, Users, Parameter, Object Type

Filter — Filter_2

Filter_2 (Filter_1, Filter_2)

Include objects that match all of the selected criteria

Criteria: [✓ Channels], Experiments, [✓ Saved], Run Cycle, Scan Cycle, Rating, Completeness, Comment, Data Set, Sample, Users, Parameter, Object Type

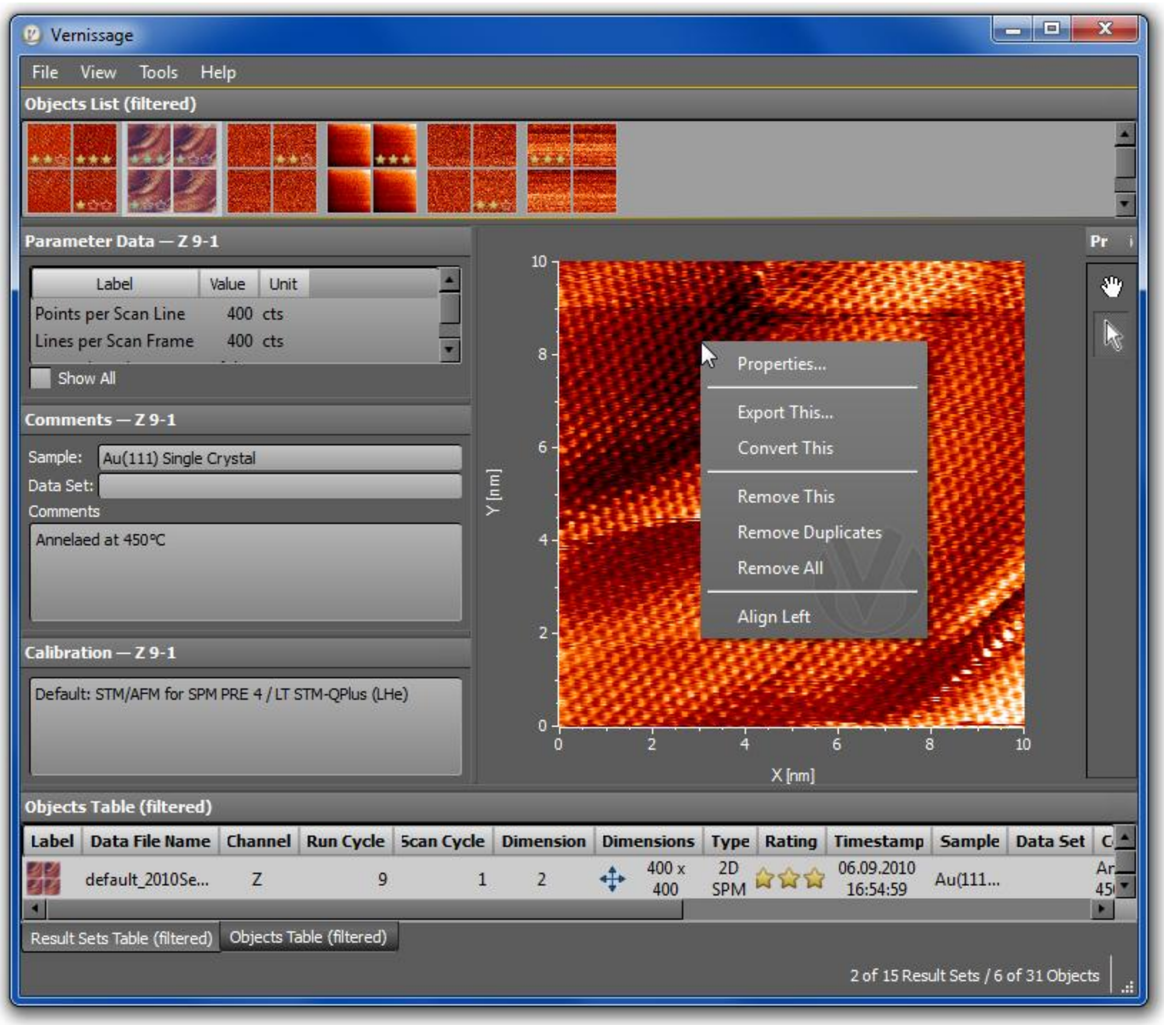
Channels: I - or

Saved: before 14.9.2011 00:00:00 -

before 14.9.2011 00:00:00 - or

Figure 23. Filter set-up and selection.

Preview Area


Vernissage

File View Tools Help

Objects List (filtered)

Parameter Data — Z 9-1

Label	Value	Unit
Points per Scan Line	400	cts
Lines per Scan Frame	400	cts

Show All

Comments — Z 9-1

Sample: Au(111) Single Crystal

Data Set:

Comments

Annelaed at 450°C

Calibration — Z 9-1

Default: STM/AFM for SPM PRE 4 / LT STM-QPlus (LHe)

Objects Table (filtered)

Label	Data File Name	Channel	Run Cycle	Scan Cycle	Dimension	Dimensions	Type	Rating	Timestamp	Sample	Data Set	C
	default_2010Se...	Z	9	1	2	400 x 400	2D SPM	★★★	06.09.2010 16:54:59	Au(111...		Ar 45

Result Sets Table (filtered) Objects Table (filtered)

2 of 15 Result Sets / 6 of 31 Objects

Figure 24. The Preview area.

When you double-click an object in the Objects List a bigger representation of the data will be shown in the preview area. If the object comprises several images, as shown in figure 25 below, tabs appear when you hover the mouse pointer over the object and allow viewing the other images.

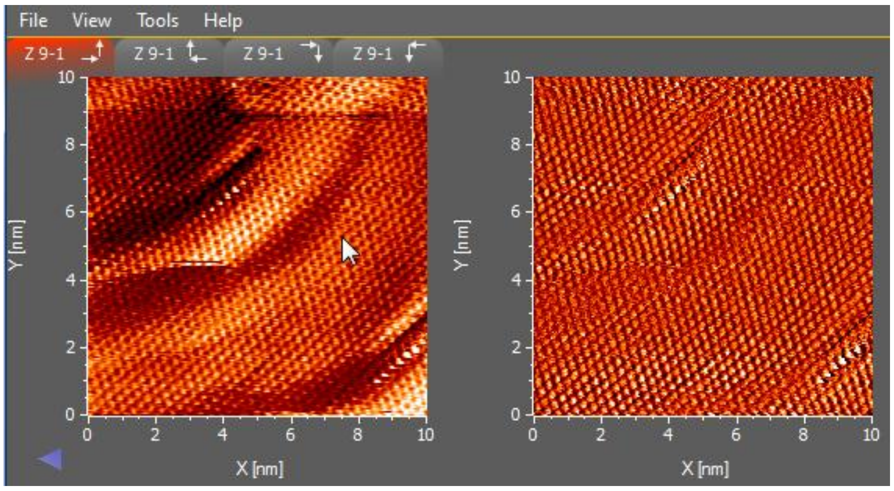
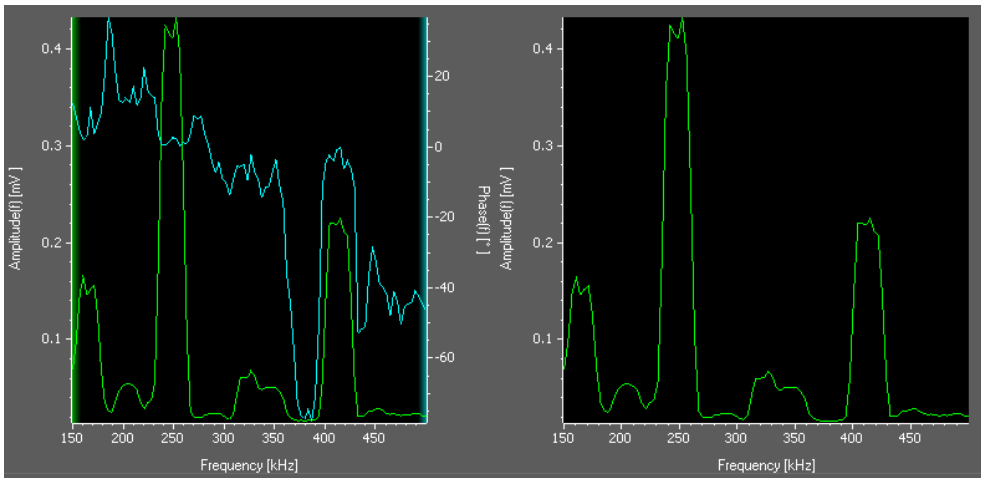


Figure 25. Preview area: tabs for accessing other scan directions.

Notice

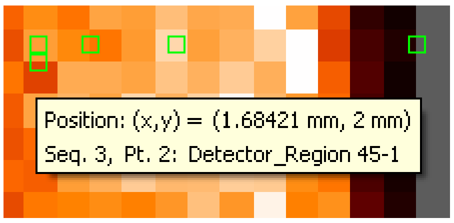
In case of curve data, e.g. ESpCurve, SPSCurve und Phase-Amplitude Curve, a double-click initiates a different action than drag-and-drop:

- Double-click (left): the entire sequence is loaded into the Preview area.
- Drag-and-drop (right): only the dragged frame is loaded to the Preview area.



Notice

In case of SPM Single Point Spectroscopy and ESpec Multipoint Spectroscopy data a tool tip shows the exact positions where the curve was taken.


Position: (x,y) = (1.68421 mm, 2 mm)
Seq. 3, Pt. 2: Detector_Region 45-1

Use the "Hand" Preview Tool or the little blue arrows to "scroll" horizontally in the preview area. A double-click on the blue arrows performs a right or left align. Alignment is also available from the context menu.

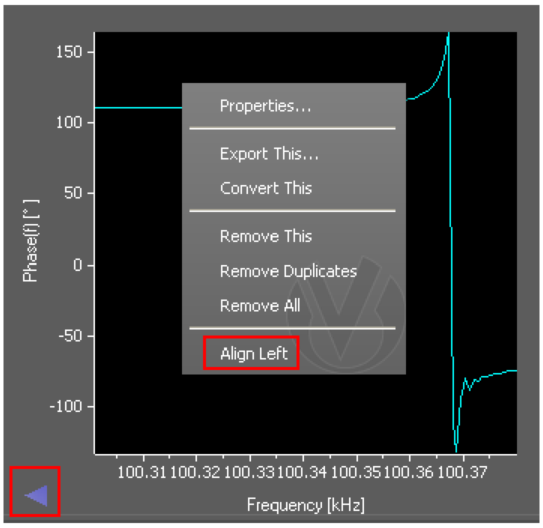


Figure 26. Alignment options.

You can add objects to the preview area by selecting the respective option from the Objects List context menu or by dragging the image there. If you drag a new object on top of another object already in the preview area, you have three options:

- **Merge**. An object can have up to four tabs. If the total number of tabs of the new plus the old object is less than or equal to four, you can merge the two objects by adding the new tabs to the old ones.
- **Replace**. Alternatively you can replace the respective object in the preview area.
- **Insert right**. Or you can simply add the object to the other preview elements, i.e. to the right hand side of the actual image.

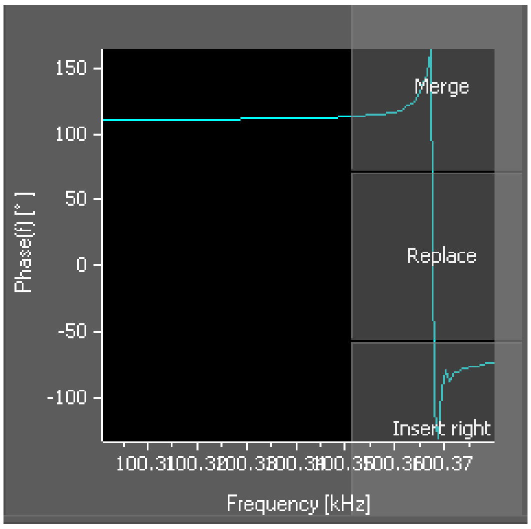
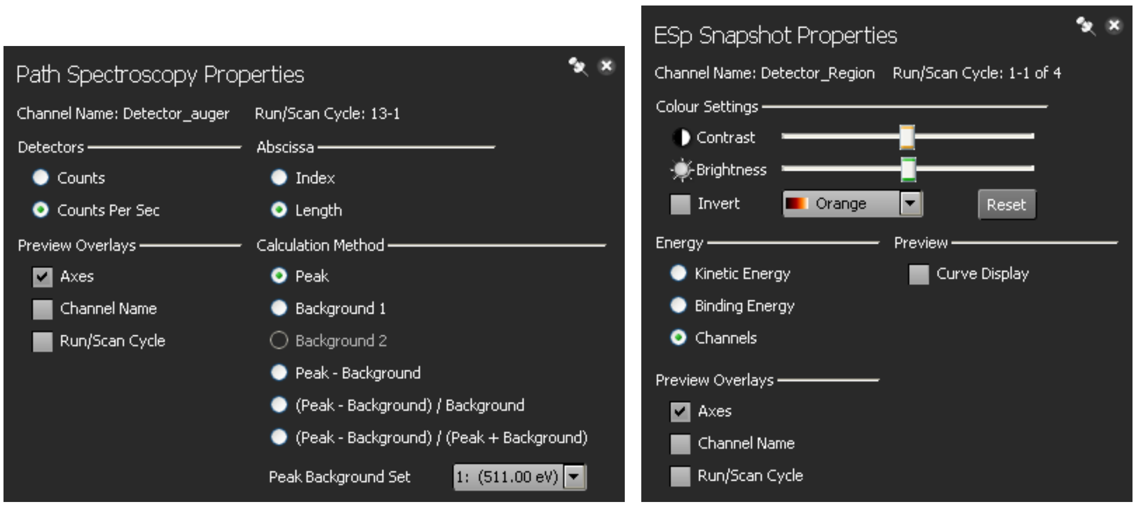


Figure 27. Preview area tools.

Conversely, you can pull (click and drag) unwanted tabs away from the set, either those you have combined or those that come as a set originally. This allows you to "undo" a merge operation but may also help to discard unfinished or otherwise spoilt scans from an otherwise perfect measurement.

A context menu allows further actions, such as aligning and removing images, removing duplicates and invoking a properties page. Two examples of properties pages are shown in figure 28 below.



Path Spectroscopy Properties

Channel Name: Detector_auger Run/Scan Cycle: 13-1

Detectors: Counts [✓ Counts Per Sec]

Abscissa: Index [✓ Length]

Preview Overlays: [✓ Axes], Channel Name, Run/Scan Cycle

Calculation Method: [✓ Peak], Background 1, Background 2, Peak - Background, (Peak - Background) / Background, (Peak - Background) / (Peak + Background)

Peak Background Set: 1: (511.00 eV)

ESp Snapshot Properties

Channel Name: Detector_Region Run/Scan Cycle: 1-1 of 4

Colour Settings: Contrast, Brightness, Invert, Orange, Reset

Energy: Kinetic Energy, Binding Energy, [✓ Channels]

Preview: Curve Display

Preview Overlays: [✓ Axes], Channel Name, Run/Scan Cycle

Figure 28. Properties pages from the preview context menu, examples shown.

On the Properties pages you can make a number of adjustments to the image/curve, the options depending on the data type involved. For all object types you can enable overlays showing Run/Scan Cycle, Channel Name, or Axes. The operation positions of SPM Single Point Spectroscopy, Atom Manipulation, and ESpec Multipoint Spectroscopy can be enabled where applicable.

In case of images you can select the colour scheme and adjust contrast and brightness. For topographic images the property sheet offers various plane subtraction modes. The advanced plane subtraction mode applies a least square method, the other modes are the same as in MATRIX.

For ESpec measurement data, which include one or more Peak Background Sets (e.g. SAM, Imaging XPS), you can select the energy level and the calculation method to be applied to the data.

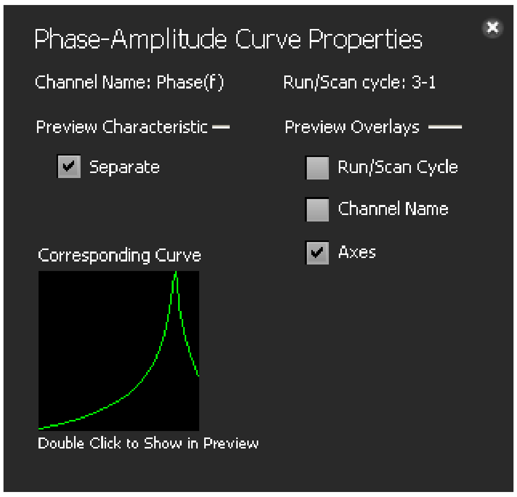
Further display options allow switching between Kinetic and Binding Energy, Counts and Counts per Second, Index and Length, Enable Averaging (e.g. repeated SPM Single Point Spectroscopy) or the display of Logarithmic Values.

Note that the changes to the image are for display purposes only. They are not included in the export process.

Previews of Sequence Objects

For sequence data the preview object represents the entire sequence. Selecting a preview selects all objects of this sequence in the Objects Table and Objects List. Consequently choosing *Export This* or *Convert This* exports the entire sequence of objects.

In case of phase/amplitude or VolumeCITS data you can separate the curves in the Preview Area via the Properties page from the context menu.


Phase-Amplitude Curve Properties

Channel Name: Phase(f)

Run/Scan cycle: 3-1

Preview Characteristic

✓ Separate

Preview Overlays

Run/Scan Cycle

Channel Name

✓ Axes

Corresponding Curve

Double Click to Show in Preview

Figure 29. Properties page of phase/amplitude data.

In *Signal over Time* und *ESP Snapshot Sequence* objects you can scroll by moving the mouse with the middle button pressed

3. Using the Batch Mode

Besides the graphical user interface, Vernissage also supports a dedicated *batch mode*. In batch mode, you specify the operations to be executed by means of command line parameters; Vernissage will run the specified operations unattended then.

The batch mode is useful for automating MATRIX result file conversion processes; examples for such automated processes include (but are of course not limited to):

- Scripted pre-processing of MATRIX result files — By means of a dedicated shell script, you could initially run Vernissage in batch mode in order to convert one or more result sets into some other format, and then pass the resulting files to a third party application for further processing.

- Repeated export operations — By combining operating system services and shell scripts, the Vernissage batch mode could be used for triggering automated conversion operations e.g. at a specific time or at programmed intervals.

- Running Vernissage from within an application — By utilising operating system services, you could launch a Vernissage conversion process in batch mode from within your own application code.

The batch mode interface of Vernissage is provided through a dedicated executable binary called `VernissageCmd.exe`, you can find this program in the "Bin" directory of a Vernissage installation (`<InstallDir>\Bin`). `VernissageCmd.exe` must be run from within a shell process, a convenient way of opening a command shell is to select *Programs* → *Omicron NanoScience* → *Vernissage Vx.y* →*Vernissage Vx.y Command Prompt* from the Microsoft Windows "Start" menu.

The command line synopsis of `VernissageCmd.exe` takes the following form:

```
VernissageCmd {-path path-spec | -file file-spec}[...]
              [-exporter plug-in]
              [-outdir dirSpec]
```

Qualifier	**Description**
–path *path-spec*	Specifies an absolute or relative path to a directory containing MATRIX result files; Vernissage will load all result sets found in the specified directory.
–file *file-spec*	Specifies a result file or result data file to be loaded.
–exporter *plug-in*	Specifies the name of the exporter plug-in to be used for conversion processes. If this qualifier is omitted, the exporter plug-in "Flattener" will be used.
–outdir *path-spec*	Specifies an absolute or relative path to a directory in which Vernissage will store the results of the conversion processes. If this qualifier is omitted, the current working directory will be used as output directory.

By utilising the qualifiers *–path* and *–file* you may specify an arbitrary number of directories and result files to be processed. After reading all result sets, Vernissage will export the resulting data into an output format determined by the qualifier *–exporter*. The output directory to which the selected exporter plug-in will write its results can be determined by the *–outdir* qualifier.

Consider the following example:

```
> VernissageCmd -path "D:\Results\11-Jun-2008" -outdir "D:\Flat Files"
-exporter Flattener
%VERN-I, Loading started
.....................
%VERN-I, Loading stopped
%VERN-I, Export started
.............
%VERN-I, Export finished
>
```

In the above example, all result sets stored in the directory "D:\Results\11-Jun-2008" will be exported into the Flat File Format (FFF); the Flattener plug-in will store its results in the directory "D:\Flat Files". During the reading and conversion processes, the Vernissage software will issue status messages informing you about the progress of the operations.

The next example shows a more complex batch mode operation: Vernissage is directed to process different result files from a variety of experiments:

- The complete result set represented by the file "D:\Results\22-Sep-2008\default_STM-STM_Basic_0001.mtrx"
- All result sets found in the directory "D:\Results\11-Jun-2008"
- A single *I(V)* spectroscopy curve stored in the file "D:\Results\13-Aug-2008\Gold--2_1.I(V)_mtrx"

Again, the result file contents will be exported into the Flat File Format (FFF), and the Flattener plug-in will store its results in the directory "D:\Flat Files".

The example also shows that qualifier names can be abbreviated, hence you can use "–p" or "–f" instead of "–path" and "–file".

```
> VernissageCmd -f "D:\Results\22-Sep-2008\default_STM-
STM_Basic_0001.mtrx" -p "D:\Results\11-Jun-2008" -f "D:\Results\13-Aug-
2008\Gold--2_1.I(V)_mtrx" -o "D:\Flat Files" -e Flattener
%VERN-I, Loading started
.....................
%VERN-I, Loading stopped
%VERN-I, Loading started
.....
%VERN-I, Loading stopped
%VERN-I, Loading started
..
%VERN-I, Loading stopped
%VERN-I, Export started
.................
%VERN-I, Export finished
>
```

If you run Vernissage in batch mode without specifying any parameters (type `VernissageCmd` at the shell command prompt and press the "Return" key), the software will print information about the command line syntax, as shown in the example below:

```
> VernissageCmd

Vernissage Version T2.0-2 (v101901)
Copyright (c)2008-2010 by Omicron NanoScience GmbH
Batch processor interface

Synopsis: {-path path-spec | -file file-spec}[...]
          [-exporter plug-in]
          [-outdir path-spec]

>
```

4. Understanding the Result File System

If you intend to develop Vernissage plug-in modules, want to link your software against the Vernissage services, or program a file reader for the Flat File Format, it is essential to be familiar with the way the MATRIX software organises result data. In this chapter, you will learn about the most important concepts and terminology related to the MATRIX result data system.

Experiment Result Terminology

To use the Vernissage API functions effectively (or to understand the Flat File Format structure thoroughly), you need to understand the way MATRIX organises experiment results. In particular, you should be familiar with the following concepts:

- **Bricklet:** A Bricklet is a container for all sorts of measurement data. Since a Bricklet comprises the data of a whole scan cycle, the number of stored curves, images etc. can be greater than one.
- **Result data file:** A result data file saves exactly one Bricklet.
- **Result file:** A result file saves the experiment logbook as well as the links to the related result data files. It is the master file which must not be lost.
- **Result file chain:** When a result file is getting too big it will be split into a number of interlinked result files: the result file chain.
- **Result set:** A result set comprises all data that have been generated during an experiment, i.e. all measurement data and the logbook.

This section explains the above terms and discusses the underlying concepts.

The MATRIX software stores all information related to a particular experiment in a *result set*. A result set is created when you open an experiment and initialise it by clicking the *Initialise experiment* button of the experiment state control element. The result set gets closed when you shutdown an experiment by clicking the same button again.

While active, a result set will store static and dynamic information about an experiment, including:

- The experiment name, version and structure (i.e. the Experiment Element instances it consists of) and the initial experiment parameters
- The experiment initialisation date and time and the name of the user account from which the MATRIX software is run
- Device calibration data, i.e. information on how raw data can be transformed into physical values
- Experiment state changes, i.e. information about experiment starts, stops, pausing, etc.
- Parameter changes
- Contents of the Favourites Gallery

- Acquired data
- Comments, sample names etc.

A result set usually comprises several files, one or more *result files* (forming a *result file chain*) and (most often many) *result data files*.

While a *result data file* stores the data acquired through a single data source during a single scan cycle, a *result file* contains all other information items listed above. Because this information details the flow of incidents during experiment execution (e.g. when was an experiment started or stopped, at which time was a parameter changed to a new value, when and at which sample location was a particular single point spectroscopy operation initiated, etc.) as well as information on the experiment itself, a result file is also often referred to as the experiment *logbook*.

Result files tend to become quite large if an experiment is used for a longer period of time, hence the MATRIX software will automatically split a large result file into smaller chunks. (By default, a new chunk will be created when a result file grows larger than 512 Megabytes, however, you may change the size by means of the *Result File Preferences* dialogue of the MATRIX software.) Each chunk can be considered as a link in a chain, thus a result file consisting of several parts is referred to as *result file chain* while the files which make up such a chain are called *result file chain links* (or *chain links*, for short.).

Result files can easily be identified by their running chain link number, which will be appended to their file name. For example, the file

```
default_2008Apr29-104314_STM-STM_Spectroscopy_0001.mtrx
```

is the first part ("0001") of a result file chain that may consist of several parts. (If there is another link in the chain, its name would be

```
default_2008Apr29-104314_STM-STM_Spectroscopy_0002.mtrx
```

and so forth.)

As already stated above, *result data files* store the actual raw data acquired through a source. To interpret the contents of a result data file correctly, additional information is required; this is stored in the associated result file. Note that it is actually impossible to exploit acquired data in case the result file of the respective experiment is lost (e.g. has been deleted.)

Result data files are referenced by their associated result file, so the overall file relationships can be depicted as shown in figure 30 below.

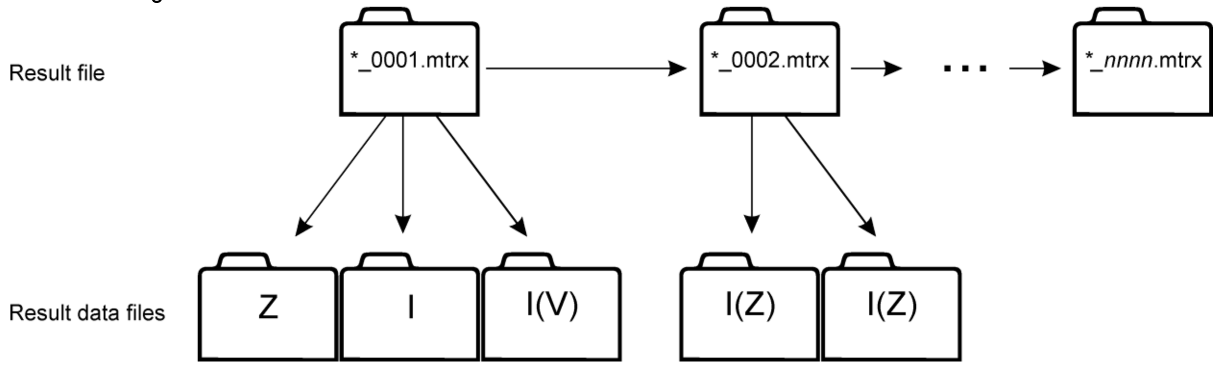


Figure 30. MATRIX result file relationships.

Due to this relationship of the files one must **never** rename result data files or result files once they have been created.

The structure of the measurement data stored in a result data file always depends on the axis hierarchy associated with the data channel through which the data were actually acquired. As a result, a single result data file may store several data *entities*. For example, a spatial data channel (such as the SPM topography channel *Z*) is associated with the two scan axes *Y* and *X* and thus delivers raw data values that can be expressed as *Z(y, x)*. As each result data file will store data resulting from a complete acquisition cycle, the amount and structure of data stored in a file actually depends on what *complete acquisition cycle* means for a particular axis configuration. For instance, if the *X*-axis has been configured to acquire data on the forward part of a scan sweep only and the *Y*-axis has been set to *scan upwards only* mode, then the acquisition cycle is complete as soon as the *Y*-axis has reached its maximum position and the *X*-axis has completed its forward sweep. Thus, the result data file will store raw *Z(y, x)* data representing a single image. If, however, the *X*-axis has been configured to acquire data on the forward and backward part of a scan sweep and the *Y*-axis has been set to *scan upwards/downwards* mode, a *Z(y, x)* result data file will actually contain *four* images (forward/up, backward/up, forward/down and backward/down), as the acquisition cycle is complete when both *Y*– and *X*-axis have returned to the scan origin.

The block of data stored in a result data file is referred to as *Bricklet*. A Bricklet can consist of any number of *n*-dimensional data entities such as one curve (e.g. from a single point spectroscopy operation), several curves (e.g. from a volume CITS operation), an image, several images, etc.

When accessing result data by means of the Vernissage API, you will always get data Bricklet by Bricklet. For each Bricklet retrieved, you can inquire additional information such as the experiment and axes parameters effective when the Bricklet was stored, the structure and dimensions of the Bricklet data, the raw value-to-physical value transformation rule, the physical unit of the data and much more. In addition, you can query the Bricklet itself in order to retrieve the comprised data entities one after another.

Summary

- Each *result data file* stores an *n*-dimensional data structure called a *Bricklet*.
- This data structure has been delivered by a particular data source during exactly one data acquisition cycle.
- Result data files are associated with a *result file chain* consisting of one or more *result files*.
- Result files store information about the experiment and the experiment execution and are thus also often referred to as the *experiment logbook*.
- The files of the result file chain and all associated result data files are subsumed under the term *result set*.

How Vernissage Processes Result Sets

The Vernissage approach to managing result sets generated by the MATRIX system is based on two subsequent steps. Initially, the software will process one or more results sets by analysing the respective result file chains and result data files. During this phase, Vernissage will create an internal database storing all information a plug-in module (or a third party application) requires for using the result set contents. Note however, that the raw data acquired during the various experiment runs will **not** be loaded, which allows Vernissage to handle a large number of result sets simultaneously without exhausting the computer memory.

When using the graphical user interface of Vernissage, you initiate this first phase of result set processing by selecting result files or result data files. When utilising the Vernissage API, you will call dedicated functions for analysing result sets and creating the information database.

During the second step, a plug-in module (or a third party application) will access the Vernissage result set database and use the information from the database for processing Bricklets. At some stage in this process, the respective software must also load the raw data contained by a Bricklet, but it can do this "on demand". ("On demand" basically means that a software module will load the raw data contents of a Bricklet just when it needs to process it and unload the data again once it does not need to access it any longer.)

For loading and unloading Bricklet contents, the Vernissage API offers dedicated functions that allow handling raw data efficiently. See section **Accessing Raw Data** for more information on these functions.

Understanding MATRIX Experiments

To become familiar with the Vernissage services, you need some basic understanding of how the MATRIX system organises experiments.

The MATRIX system does not provide any hard-coded experiment control functions but implements a framework for executing experiment *descriptions*. Each experiment description is actually a blueprint of actions and operations a particular experiment should offer, including all parameters and even the graphical user interface.

Experiment is a term you will encounter frequently while using the MATRIX software (during a MATRIX session, you will *open*, *upload*, *start*, *stop*, etc. experiments). Another important concept of the MATRIX system is usually less visible: Experiments are actually constructed from distinct building blocks called *Experiment Elements*.

An Experiment Element is a small software module dedicated to a particular purpose such as generating scan movements, controlling the tip/sample distance, acquiring data and other. Examples for Experiment Elements are:

- *XYScanner* — An Experiment Element dedicated to controlling and generating scan sweeps.
- *Spectroscopy* — Experiment Elements of this type support spectroscopy operations by generating and controlling spectroscopy ramps.
- *Channel* — Experiment Elements of type *Channel* are able to acquire data by means of a sensor device, i.e. an Analogue-to-Digital Converter (ADC) component.
- *GapVoltage* — An Experiment Element for configuring the gap voltage applied between probe and surface.

Experiments are constructed by *deploying instances* of Experiment Elements; deployment takes place by specifying the Experiment Element type of a particular instance, assigning a unique name to the instance and configuring a set of parameters (called *deployment parameters*) for the instance. (See the experiment structure description files — such as `STM_Spectroscopy.exps` — for examples on experiment construction by deploying Experiment Element instances.)

Each experiment can use multiple instances of the same Experiment Element type and this happens quite frequently. For example, each of the MATRIX standard SPM experiments offer several data acquisition channels, each of which is actually an Experiment Element instance. Consider the well-known MATRIX standard experiment *STM_Spectroscopy*: this experiment consists of more than 25 Experiment Element instances of several different types. Not surprising, the element type occurring most often is *Channel* (14 instances).

Experiments can consist of virtually any number of Experiment Element instances, however, only very complex experiments consist of more than three dozen Experiment Element instances.

Any experiment parameter you can modify while using an experiment actually belongs to a particular Experiment Element instance. SPM scan parameters such as scan area dimensions, scan mode, raster time and others belong to element instances of type *XYScanner*, the oversampling control parameters are part of the parameter set of Experiment Element instances of type *Channel*, etc. When developing Vernissage plug-in modules you may sometimes want to retrieve the value of an Experiment Element parameter, although most information required to interpret result data can also be inquired by other means. For retrieving the value of a parameter at a particular time you must know the instance name of the Experiment Element hosting the respective parameter and the parameter name itself. A good starting point for exploring the parameters supported by a particular Experiment Element is the onscreen help facility of the MATE script manager available through the "Tools" menu of the MATRIX experiment window. The "Catalogues" topic of the onscreen help provides information on available Experiment Elements, their purpose, characteristics, properties, and other information.

Data Types and Formats

Data delivered by the Vernissage services takes one of two forms:

- A *raw data* item is an unprocessed 32-bit value as delivered by the MATRIX hardware.
- A *physical value* is a data item representing a physical quantity. In contrast to raw data, physical data are associated with a *unit* such as volt or newton.

Physical values can be transformed into raw values and vice versa and this is often a task Vernissage plug-in modules have to accomplish.

Technically, any raw value is represented as a signed long integer data item, while physical values take the form of a 64-bit double-precision floating point figure. Please note, however, that Vernissage will pass Experiment Element parameter values (which often represent physical quantities) as character strings with associated unit information (see below also).

Within the MATRIX system, all physical quantities have SI compatible units; consequently, any parameter value is associated with a unit such as ampere, newton, metre, second, etc. For example, a scan area width of 500 nm will be represented as $5 \cdot 10^{-7}$ m.

The following table summarises the most important physical quantities and their units as used by the MATRIX system.

Physical Quantity	**Unit (long form)**	**Unit (short form)**
length	Meter	m
velocity	Meter/Second	m/s
duration	Second	s
frequency	Hertz	Hz
current	Ampere	A
voltage	Volt	V
electron volt	ElectronVolt	eV
force	Newton	N
angle	Degree	deg
Counts	Counts	cts
Decibel	Decibel	dB
Percentage	Percent	%
Temperature	Kelvin	K

Table 3. Important physical quantities and their unit representations in MATRIX.

Please note that the Vernissage API will return the character string "--" when the physical unit of a data item is not known. (If this happens, the respective data item is most often not a physical quantity but represents some other piece of information, such as a Boolean value.)

The MATRIX software — and thus also the Vernissage modules — utilises the *Unicode* system for representing character sequences (also referred to as *strings*) internally. Each character of a string is represented by the UTF-16 encoding and will usually be stored in a single 16-bit word. (For Unicode experts: strings requiring surrogate pairs are currently not found in MATRIX, hence you can also consider all characters as being UCS-2 encoded.)

For C++ software, the use of Unicode throughout the MATRIX system has some consequences:

- The string class for storing character sequences returned by Vernissage is `std::wstring` (instead of `std::string`)
- The array representation of a character sequence is `wchar_t[]` or `wchar_t*` (compared to `char[]` and `char*` for ANSI/ASCII-encoded character sequences)
- For writing Unicode character sequences to files (or an output device), one must utilise library functions such as `fwprintf` (or the wide-character stream class `std::wostream`) instead of `fprintf` (or `std::ostream`, respectively)

The Vernissage API will pass parameters by means of a value type/value/unit sequence, with the value expressed as a (Unicode) character string. For example, inquiring the value of parameter *Width* of the Experiment Element *XYScanner* (describing the width of the current scan area) could result in the following information:

```
value type = vt_Double
```

```
unit = "Meter"

value = "5e-7"
```

The above information describes the value of parameter *Width* by means of three separate attributes:

- A type code (`vt_Double`) specifying the data type of the parameter value (in this example: double-precision floating point figure)
- A character string denoting the SI unit used
- A character string describing the (numerical) parameter value

Parameters representing non-numerical data will be treated identically; for example, the current value of parameter *X_Retrace* (Experiment Element *XYScanner*) representing the scan mode in *X*-direction could be represented as shown below:

```
value type = vt_Boolean

unit = "--"

value = "true"
```

The above information indicates that the value of parameter *X_Retrace* is of type Boolean (i.e. a flag) and has been set to *true* — The experiment has thus been configured to acquire data on the *retrace* (or backward) part of the scan sweep.

The parameter value types and type codes used by Vernissage are listed in the table below.

Value type	**Type code ID**	**Type code value**	**Example value string**
Boolean	`vt_Boolean`	3	`true`
Enumeration	`vt_Enum`	4	`1`*
Double-precision floating point	`vt_Double`	2	`5e-7`
32-Bit integer	`vt_Integer`	1	`-42`
Unicode character string	`vt_String`	5	`Hello`

Table 4. Parameter value types and type codes used by Vernissage. *) An enumeration value is always expressed by its associated numerical constant (although such a value has also a character string representation.)

Understanding Data Views

Data Views (or short *Views*) play an important role in the MATRIX system, as the type and configuration of a View determines how acquired data gets visualised when running experiments.

Actually, every display that is used for visualisation of acquired data at run-time of the MATRIX software must be associated with a specific View that processes the raw data in a way that enables the display to render the data correctly.

However, Views can also be of interest when analysing Bricklets, as the type of a View associated with a Bricklet (more precisely, with the data channel a particular Bricklet originated from) can be helpful for determining the interpretation of the data stored by a Bricklet from an application perspective. For example, if a Bricklet stores one-dimensional data, it actually stores a curve. However, it is not really clear if the curve represents (for example) the results of a single point spectroscopy operation, or has been generated during a force/distance curve acquisition experiment, etc. In this case, knowing which Views were associated with the Bricklet at experiment run-time can be essential: If there were Views of type *ForceCurve* connected to the channel that has produced a particular Bricklet, the one-dimensional data represent a force/distance curve beyond doubt. However, Views of type *Spectroscopy* clearly hint at single point spectroscopy data.

View types commonly found are:

View Type Code	Bricklet Dimensions	Usage
vtc_DiscreteEnergyMap	2/3	Electron spectroscopy curves from a series of acquisition operations using discrete energy maps.
vtc_ESpImageMap	3/4	Electron spectroscopy image data from a series of acquisition operations using discrete energy maps.
vtc_CurveSet	2	Curve data used for electron spectroscopy curves acquired by operations on energy regions.
vtc_ParameterisedCurveSet	3	Curve data used for electron spectroscopy curves acquired by “snapshot” operations.
vtc_ContinuousCurve	1	Curve data continuously acquired over time.
vtc_PhaseAmplitudeCurve	1	Associated phase and amplitude curves.
vtc_1DProfile	1	Curve data acquired over time while moving probe along some path. (Used e.g. for curve data acquired during by atom manipulation operations.)
vtc_ForwardBackward2D	2	Topography images, current images and similar 2D data.
vtc_2Dof3D	3	Planes from a volume CITS data cube.
vtc_Spectroscopy	1/3	Spectroscopy curves (SPS, or volume CITS).
vtc_ForceCurve	1	Force/distance curves.
vtc_Interferometer	1	Interferometer adjustment curves (Omicron Cryogenic SFM instrument only.)
vtc_Downward2D	2	ESp - SEM images by generic scanner

Axes and Axis Hierarchies

The data acquisition model of MATRIX is based on the idea of *triggered channels*: A channel acquires data by means of its associated sensor device or acquisition hardware every time it is triggered. The entity that can trigger a channel is referred to as *axis*; an axis has a start value, an end value and a number of equidistantly distributed *clocks* between the start and end values (the start and end values are also considered to be clocks). When an axis is started, it proceeds from start to end value, generating a trigger at each clock. (The time it takes the axis to reach the next clock is known as the *raster time*.)

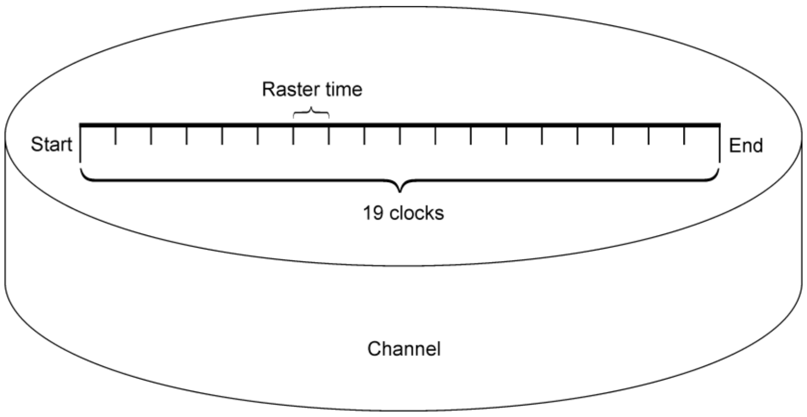


Figure 31. Axis and triggered channel

For a data acquisition operation producing one-dimensional data (i.e. a curve) a single axis is all that is actually required. (Consider a single point spectroscopy operation proceeding from the ramp start value to the ramp end value at a certain raster time.) For generating two-dimensional images, two "nested" axes are required, as shown in the following figure.

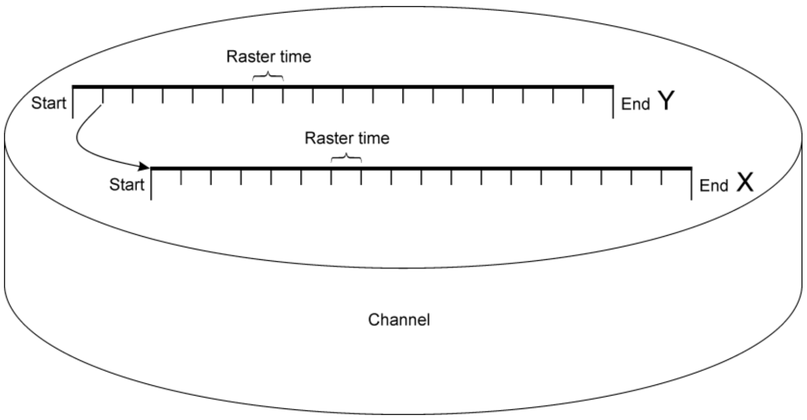


Figure 32. Axis hierarchy.

The two axes *Y* and *X* form an *axis hierarchy* because axis *Y* does not trigger a channel directly but another axis (*X*), which will finally trigger the data acquisition. When started, the *X*-axis in the axis system shown above will proceed from its start value to its end value each time the *Y*-axis triggers it at one of its clock positions. In the above example, axis *X* is the *trigger axis* of the channel (while *Y* is the trigger axis of *X*). Because *Y* is not associated with a trigger axis itself, it is called the *root axis* of the axis hierarchy.

Axis hierarchies can become quite complex, as the MATRIX system does not limit the number of nested axes. However, the standard experiments shipped as part of the MATRIX kit use a maximum of four nested axes.

Each axis of an axis hierarchy has a name that can be either viewed as *plain* or *qualified*. Plain names are simple character strings (such as "X", "V" or "Energy") that are most often sufficient as in the context of a particular experiment a specific axis name is usually unique. Qualified axis names contain additional information about the associated *instrument* and the *Experiment Element instance* and take the following form:

Instrument Name::Element Instance Name::Plain Name

For example:

Default::XYScanner::X
Gemini::GeminiScanner::Y

Internally, all axis names are qualified names; however, the service routines of the Vernissage API can process plain names as well.

An axis is referred to as *mirrored* if it does not only proceed from the configured start value to the end value, but then also back to the start value (which effectively doubles the number of clocks it uses). The *mirrored* characteristic is, for example, assigned to the scan axes *X* and *Y* (supported by the Experiment Element *XYScanner*) if these axes have been configured to use a *forward/backward* or *up/down* mode of operation.

By default, an axis will trigger another axis or a channel at each clock position, but this behaviour can be altered by applying filters called *table sets*. A table set specifies a set of intervals on an axis for which triggers are generated; each interval specification consists of a start value, an end value and an increment. Consider the following examples:

- The interval (start = 1; stop = 300; step = 1) applied to an axis with a "length" of 300 clocks would generate a trigger at each clock position.
- The interval (start = 1; stop = 300; step = 5) applied to an axis with a "length" of 300 clocks will generate a trigger at clock positions 1, 6, 11, etc.
- The interval (start = 151; stop = 300; step = 2) applied to an axis with a "length" of 300 clocks will suppress all trigger for the first 150 clock positions and generate triggers at every second clock position on the second half of the axis.

The most prominent use of table sets is grid spectroscopy (SPM volume CITS): The spectroscopy sub-grid is actually established by applying appropriate table sets to the scan raster (i.e. the *Y*- and *X*-axes). Consider the axis hierarchy shown in figure 32 on page 41: assuming that both axes have been configured to be 300 clocks long, the following table sets would establish a *X* = 10, *Y* = 5 sub-grid:

- *Y*-axis: (start = 1; stop = 296; step = 5)
- *X*-axis: (start = 1; stop = 291; step = 10)

If the *Y*- and *X*-axis are *mirrored* (resulting in 600 clocks effectively), the table sets must be configured as follows:

- *Y*-axis table set: (start = 1; stop = 296; step = 5) and (start = 305; stop = 600; step = 5)
- *X*-axis table set: (start = 1; stop = 291; step = 10) and (start = 310; stop = 600; step = 10)

The Vernissage API offers a number of service routines allowing to obtain information on the axes involved into the acquisition of data contained by a particular Bricklet.

Axis Parameters

Axes can be associated with meta information, i.e. information being logically related to the respective axis although it does not affect the axis configuration—such as start value, end value, number of clocks, etc.—directly. Meta information is provided by means of dedicated name/value tuples known as axis parameters; whether axis parameters are associated with a particular axis (and, if so, what types of parameters) actually depends on the axis type:

- Axes used by SPM experiments (such as *X*, *Y*, *Z*, V_{Gap} etc.) have currently no associated meta information and thus not provide axis parameters.

- Experiments from the electron spectroscopy domain make frequent use of meta information; the respective parameter sets include (but is not limited to):
    - Dwell time
    - Dispersion and normalisation factors
    - CAE/CRR mode and value
    - Transition label
    - Peak and background energy configuration of discrete energy maps
    - Centre energy and inter-acquisition delay of snapshot-mode data acquisition operations
    - Length of the scan vector of a line-oriented spectroscopy operation
    - Divide-by-ten-mode state of the power supply unit

The Vernissage user interface will display axis parameters just like other experiment parameters in the Parameter Data view. Users who want to develop exporter plug-in modules for electron spectroscopy data will call dedicated application programming interface routines (in this case `getAxisParameter()` or `getAxisParameters()`, respectively) in order to obtain information on one or more axis parameters, including the parameter values and physical units.

Related Bricklets

Certain type of Bricklets can be associated with other Bricklets due to some logical interdependency; such Bricklets are referred to as *related Bricklets*. Currently, the Vernissage software recognises the following related Bricklets types.

Depends-on/References Relationship

A particular Bricklet references another Bricklet; the referenced Bricklet is considered to "depend" on the referencing Bricklet. This type of Bricklet relationship affects the following experiment data:

- Single point spectroscopy curves depend on SPM images, as they have been acquired at a certain location.

- Atom manipulation curves depend on SPM images, as the data they consist of have been acquired along a vector between a particular start location and an end location.

- Spectroscopy curves also depend on SPM images if they have been acquired along a vector.

- Electron spectroscopy curves depend on SAM or XPS images if they have been acquired at a defined sample location.

- Discrete energy maps depend on SAM or XPS images if they have been acquired at a defined sample location.

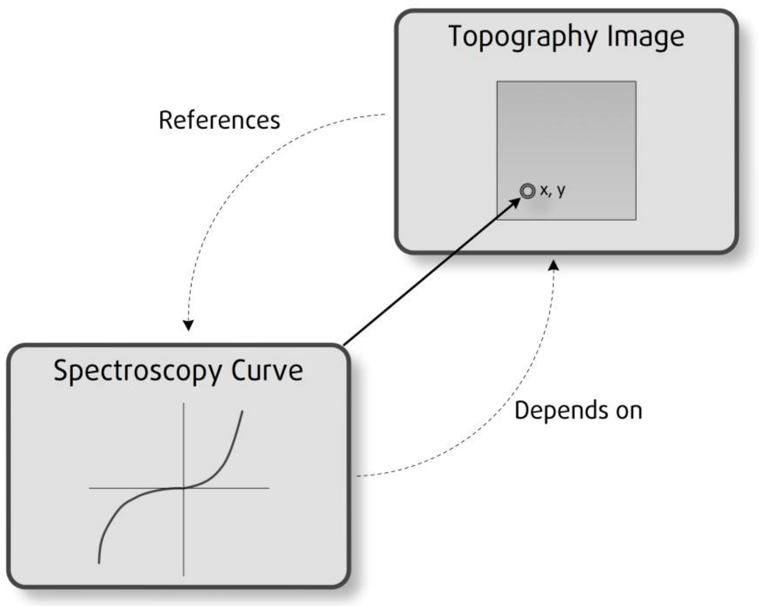


Figure 33. Relationship between a single point spectroscopy curve Bricklet and a topography image Bricklet

Successor/Predecessor **Relationship**

An operation has produced a sequence of consecutive Bricklets, or a series of Bricklets logically connected. This type of Bricklet relationship affects the following experiment data:

- Phase/amplitude curves consist of a Bricklet containing phase data and a second Bricklet storing the associated amplitude data.
- Single point spectroscopy curves generated with the automatic spectroscopy operation repetition option enabled.
- Electron spectroscopy curves generated with the region repetition option enabled.
- Continuously acquired curves consisting of several consecutive Bricklets.
- Electron spectroscopy detector snapshots consisting of several consecutive Bricklets.

Notice

Please note that a single Bricklet can have both a Depends-on/References and a Successor/Predecessor relationship with other Bricklets.

Raw Data Organisation

Raw data delivered by Vernissage consists of a series of 32-bit signed integer values stored in the order they were originally acquired by the MATRIX system. Thus, in order to identify and interpret the data items contained by a Bricklet, one must know their acquisition order.

The MATRIX system acquires data with respect to the axis hierarchy associated with a specific data channel. For a channel associated with a single axis, the data organisation is straightforward: the axis proceeds forwards from its start value to the end value, acquiring a data item at each clock position. If the axis has the *mirrored* characteristic, the axis will also return backwards to its start value, again acquiring a data item at each clock position. The data organisation in the resulting Bricklet is shown in figure 34 below.

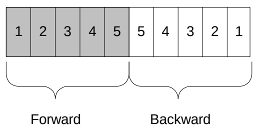


Figure 34. Data order generated by single mirrored axis.

In the scenario depicted in figure 32 on page 41, the *Y*-axis triggers the *X*-axis, so the associated channel will deliver data items line by line, with each line proceeding from the start value of the *X*-axis to its end value. If both the *X*– and the *Y*-axis have been assigned the *mirrored* characteristic, the resulting Bricklet data organisation will look as depicted in figure 35 below.

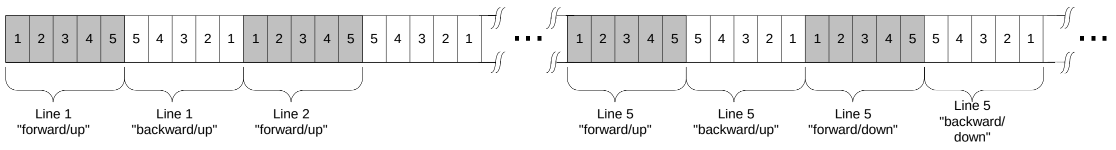


Figure 35. Data order generated by simple mirrored axis hierarchy

Bricklets resulting from an SPM spatial scan process (e.g. Bricklets containing topography images) will be structured exactly as shown above if the scan subsystem was configured to use the forward/backward and up/down scan modes.

More complex axis hierarchies will cause similar data structures, as the channel through which data are acquired gets triggered by the lowest axis within the hierarchy. For example, consider the following data structure:

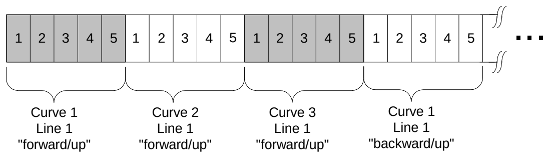

Curve 1 Line 1 "forward/up"
Curve 2 Line 1 "forward/up"
Curve 3 Line 1 "forward/up"
Curve 1 Line 1 "backward/up"

Figure 36. Data order generated by Y-X-V hierarchy (Volume CITS)

The above data structure may result from a three-axis-hierarchy as typically used by volume CITS experiments. As the channel will be triggered by the spectroscopy axis (which is triggered by the *X*-axis), the Bricklet will store the acquired spectroscopy data curve-by-curve and line-by-line.

It is important to notice that Bricklets only store *acquired* data, i.e. any time an axis is *not* triggered (because of some filter applied through a table set) there will be also no change of the Bricklet data structure.

Please see **Appendix: Raw Data Structures** for details on all Bricklet raw data structures supported by Vernissage.

5. The Vernissage Programming Interface

The Vernissage software offers a dedicated Application Programming Interface (API) for integrating with third party software. Basically, you may use this API in two different ways:

- By linking the Vernissage software against your own application code, you gain access to the result file load and data access mechanisms of Vernissage. You may call API functions to load result data files or entire result data sets and for accessing the contents of result files.

- By developing a Vernissage plug-in, you may extend the capabilities of the Vernissage software. In this case, the API functions will help you to traverse result data structures, to obtain information about result data and to convert acquired measurement results into other data formats.

This chapter discusses both aspects of the Vernissage API; you will learn how to access the Vernissage facilities from within your own application code and how to develop a Vernissage exporter plug-in.

Accessing Vernissage Facilities from Your Application Code

If you want to process the contents of MATRIX result files by means of your own application software you can take two different approaches. Basically, you may use Vernissage to convert one or more result files into some other output format and process the resulting files afterwards. (By utilising the Vernissage command line interface this can be even done in an automated way.) The more elegant solution, however, is to utilise the Vernissage facilities for reading and accessing result file contents directly from within your application code.

Omicron grants you the right to link your own software against the Vernissage core libraries (or load these libraries at run-time) so that you can call Vernissage API functions just like a plug-in module would do. (Note however, that you must neither redistribute parts of the Vernissage binaries, nor the complete product kit with your own software.) In this way, you can integrate the Vernissage software facilities for reading result files and accessing their contents with your own application code.

As already stated, using Vernissage facilities from third party application code is very similar to using Vernissage services from a plug-in module, with the following exceptions:

1. Your application must first gain access to the Vernissage core software libraries before you can use the service API.

2. Your application must create an instance of the API interface object (an instance of the Vernissage class `Session`) in order to call API functions.

3. Your application must use the Vernissage API to load (and unload) result sets and data files itself.

The subsequent sections discuss the above topics only; for an overview of the Vernissage result file contents access services, see chapter **Building Exporter Plug-Ins**.

Getting Access to the Vernissage Core Libraries and Session Object

In order to access Vernissage services, your application software must utilise the Vernissage Dynamic Link Library `Foundation.dll`. The DLL is located in the installation directory of the Vernissage system and you may take one of two approaches for integrating it with your application software.

- Link your application against the object library `Foundation.lib`; the Microsoft Windows image loader software will then automatically load the required DLLs at application run-time.

The main disadvantage of this approach is that the Vernissage binaries have to reside in the same directory as your application binaries. Note that the library file `Foundation.lib` ships with the Vernissage Software Development Kit (SDK) which you must install first. The SDK comes as a ZIP archive and can be downloaded from the following website: http://www.omicron.de/en/software-downloads (topic "Vernissage", sub-topic "Development Kit")

- Load `Foundation.dll` and its associated libraries dynamically at run-time of your application. This is the preferred method, as the binaries of your application and the Vernissage system can be separated easily.

The disadvantage of loading `Foundation.dll` dynamically is that you must locate the library files first. You may of course use a hard-coded file-specification in your application code, however, the recommended approach is described below:

1. Look up the Microsoft Windows registry to obtain the following value:

```
HKEY_LOCAL_MACHINE\Software\Omicron NanoScience\Vernissage\Vx.y\Main\InstallDirectory
```

2. Append the subdirectory name "Bin" to the path specification retrieved through the above value.

3. Append the DLL file name to the path specification. The resulting file specification will be similar to the following:

```
<InstallDirectory>\Bin\Foundation.dll
```

4. Load the DLL, e.g. by calling the Microsoft Windows API functions `LoadLibrary()` or `LoadLibraryW()`.

5. Repeat steps 2 to 4 for all required library files.

6. Find the entry points of the two Vernissage API functions `getSession()` and `releaseSession()`, e.g. by calling the Microsoft Windows API function `GetProcAddress()`.

Afterwards, you can call the `getSession()` function to retrieve an interface object of class `Session` which is required for utilising the services of the Vernissage API.

A C++ example routine for gaining access to the Vernissage API is shown below; the routine requires the path specification to the Vernissage "Bin" directory to be passed and returns the Session object representing the Vernissage API.

```
Vernissage::Session* getSessionObject (LPCWSTR pDllDirectory)
{
  // List of DLL files to be loaded
  LPWSTR pDllNames[] = { L"Ace.dll",
                         L"Platform.dll",
                         L"Base.dll",
                         L"Xerces.dll",
                         L"Store_XML.dll",
                         L"Store_ResultWriter.dll",
                         L"Store_Vernissage.dll",
                         L"Foundation.dll",
                         NULL
                        };
  LPWSTR *pT = pDllNames;
  HMODULE module;

  // Load all DLLs
  do
  {
    if (*pT != NULL)
    {
      // Construct the file specification
      size_t length = wcslen(pDllDirectory) + wcslen(*pT) + 2;
      LPWSTR pFileSpec = new WCHAR[length];
      wcscpy(pFileSpec,pDllDirectory);
      wcscat(pFileSpec,L"\\");
      wcscat(pFileSpec,*pT);

      // Load the DLL
      module = LoadLibraryW(pFileSpec);
      delete[] pFileSpec;
      pT++;
    }
  } while ((*pT != NULL) && (module != 0));

  if (module != 0)
  {
    // Last DLL loaded was 'Foundation.dll', so we can get the
    // addresses of the 'getSession()' and 'releaseSession()'
    // routines now.
    typedef Vernissage::Session * (*GetSessionFunc) ();
    typedef void (*ReleaseSessionFunc) ();
    GetSessionFunc pGetSession;
    pGetSession = (GetSessionFunc)GetProcAddress(module,
                                                 "getSession");
    Vernissage::Session *pSession = (pGetSession)();
  }

  // This routine is just an example, it's pretty stupid because it
  // drops the address of 'releaseSession' which is required at
  // application shutdown ...
  return pSession;
}
```

The header file `Vernissage.h` used in the above code fragment contains the declarations of all data items and API functions required for using the Vernissage services; this file is part of the Vernissage SDK. For more information on the Vernissage SDK please refer to section **Building Exporter Plug-Ins**.

Loading and Unloading Result Files

As explained earlier, the Vernissage approach to result set processing comprises two distinct steps:

- Analyse one or more result sets and create an internal database storing information about the contents of these result sets.
- Use the database for processing the Bricklets contained by the result sets; load and unload the raw data content of a Bricklet dynamically during this process.

Both of the above steps are supported by the Vernissage API which offers various functions for managing MATRIX result sets:

- There is a set of functions dedicated to loading one or more result files, or one or more result data files. (There is also a function for loading all files contained by a particular directory.) These functions will analyse the respective files and create the information database.
- A different set of functions supports loading and unloading the raw data of the Bricklets associated with the result files loaded.

The following table lists all API routines dedicated to the management of result files and raw data.

Routine	**Description**
`loadResultSet`	Loads one or more result files (or result data files) into the information database.
`loadAllResultSets`	Loads all result sets from a specific directory into the information database.
`eraseResultSets`	Erases all result sets from the information database.
`loadBrickletContents`	Loads the raw data content of a particular Bricklet.
`unloadBrickletContents`	Marks the raw data content of a particular Bricklet for unloading and unloads the data if no other software module still uses it.

Table 5. Result file and data management routines.

Building Exporter Plug-Ins

For building your own Vernissage plug-ins, you will need a C++ development environment. Omicron recommends using *Microsoft Visual Studio 2005* for this purpose, as the Vernissage tool itself has been developed using this software. If you don't have access to a *Microsoft Visual Studio 2005* installation (or its variant *Microsoft Visual C++ 2005 Express Edition*), you can download the successor version *Microsoft Visual C++ 2008 Express Edition* free of charge from the following website:
http://www.microsoft.com/express/Downloads/#Visual_Studio_2008_Express_Downloads.

Plug-in developers must, however, be aware that this edition of the Microsoft C++ environment produces binary code that is *not* compatible with previous versions; plug-ins compiled with *Microsoft Visual C++ 2008 Express Edition* (or its commercial counterpart *Visual C++ 2008*) will cause Vernissage to terminate abruptly when used for data export operations. To remedy the situation, Omicron offers two different versions of the Vernissage Kit and the Vernissage Software Development Kit (SDK):

Kit Name	Environment	Plug-In Development SDK	Compiler
Vernissage_V2.2.zip	Microsoft Visual C++ 2005	VernissageSDK_V2.2.zip	VC8
Vernissage_V2.2_VC9.zip	Microsoft Visual C++ 2008	VernissageSDK_V2.2_VC9.zip	VC9

If you do not intend to develop your own plug-in modules, the type of Vernissage kit you install does actually not matter. Omicron recommends installing the "Vernissage_V2.2.zip" kit in this case.

If you download the binary code of a plug-in module from the Omicron website, make sure that you select the version that is compatible with the Vernissage kit you have installed. To determine the correct plug-in type, choose "About" from the "Help" menu of Vernissage. The type of development environment that must have been used for compiling plug-in modules is indicated by the "Plug-in compiler" version information text. (For example, "Plug-in compiler: VC8 (VS 2005)" for the VC8 compiler shipped with Microsoft *Visual C++ 2005*.)

Please note that the versions 2010 and 2012 of *Microsoft Visual Studio* (as well as their respective "*Express*" counterparts) can currently not be used for developing Vernissage plug-in modules.

Before you can start to develop a plug-in module, you must install the Vernissage Software Development Kit (SDK) on a computer running Microsoft Windows. The SDK comes as a ZIP archive and can be downloaded from the following website: http://www.omicron.de/en/software-downloads (topic "Vernissage", sub-topic "Development Kit")

After unpacking the archive to an arbitrary location, you have all the files and tools required for developing new plug-in modules, or modifying existing plug-in source code, or integrating the Vernissage core software with your own applications.

The directory tree of the Vernissage SDK has the following structure:

Vernissage SDK			Main directory
	Incl		C++ header files
	Debug		Debug version of *Foundation.lib*
	Release		Release version of *Foundation.lib*
	PlugIns		Plug-in development directory
		ASCIIExporter	Example plug-in: ASCII Exporter
		SCALAPROFormatter	Example plug-in: SCALA PRO Formatter

The `PlugIns` directory contains two example Exporters: The `ASCIIExporter` converts MATRIX result sets into simple ASCII files, while the `SCALAPROFormatter` generates Omicron SCALA PRO-compatible data files from MATRIX result sets. The ASCII exporter is an example for a straightforward plug-in, while the SCALA PRO exporter demonstrates how a very complex export task can be accomplished. Both examples are provided as complete C++ projects: the respective directories contain all required files, including C++ source code, Microsoft Visual Studio "project" and "solution" files, etc.

Notice

You can download binary versions of the ASCII exporter and SCALA PRO formatter plug-ins from the Omicron MATRIX website http://www.omicron.de/en/software-downloads (topic "Vernissage", sub-topic "Plug-In Examples")

After compiling and linking the plug-in DLL, you must manually copy the resulting file to the Vernissage plug-in directory. The plug-in directory is usually located at:

```
<InstallDir>\PlugIns
```

Caution

- You **must** compile all plug-in code as *release build* **not** as *debug build* version.
- **Never** copy a plug-in DLL that has been compiled and linked as *debug build* into the Vernissage plug-ins directory!

Debug build code uses a different heap memory manager than release build code, therefore debug build DLLs are structurally incompatible with Vernissage and will most probably cause the software to crash during its start-up phase.

Anatomy of an Exporter Plug-In

A Vernissage plug-in module must be provided as a multi-threading-capable Microsoft Windows Dynamic Link Library (DLL) compiled and linked as *release build*. (Note that a plug-in DLL **must not** be provided as *debug build* as such DLLs will most probably cause the Vernissage software to crash during its start-up phase.)

As an absolute minimum, the DLL must contain a single C++ class derived from the Vernissage base class `PlugInBase`; this base class specifies a few simple methods that your derived plug-in class must implement.

The below listing shows the contents of a C++ header file declaring the interface of a class that Vernissage would recognise as a valid plug-in:

```
#include "..\Incl\PlugIn.h" ❶
namespace PlugIn ❷
{
  class MyPlugIn : virtual public PlugInBase ❸
  {
    public:
      MyPlugIn ();
      ~MyPlugIn ();

      // Plug-in identification
      void getIdentity (std::wstring& name, std::wstring& version,
                        std::wstring& producer) const; ❹
      PlugInType getType () const; ❺

      // Init and shutdown
      void init (); ❻
      void shutdown ();

      // Plug-in main function point of entry
      bool run (Vernissage::Session *pSession,
                std::wstring outputDirectoryPath, ❼
                void *pFilterSet);

      // Cancel operation
      void stop (); ❽
  };
}
```

The most important elements of the class declaration are discussed below:

❶ The interface class `PlugInBase` is declared in the Vernissage header file `PlugIn.h` which must hence be included into the Plug-In code. You can find the header file in the "Incl" source code directory of the Vernissage SDK.

❷ All Vernissage plug-in code must be placed in the namespace `PlugIn`.

❸ A Vernissage plug-in class must be derived from the base class `PlugInBase`.

❹ The Vernissage core software will call this method when loading the plug-in DLL in order to obtain information about the *identity* of a plug-in module. The method implementation should fill in useful information into the following variables passed as arguments:

- `name` — The name of the plug-in module (e.g. "ASCII Exporter", "SCALA PRO Formatter", etc.) Note that the name must be unique for all plug-in modules, i.e. two plug-in modules returning the same name cannot co-exist in the same Vernissage environment. (If the same name is returned by more than one plug-in module, the Vernissage software will only load one of them and mark all other modules using the same name as invalid. By choosing *About Plug-Ins* on the *Help* menu of the Vernissage main window you can direct the software to display a list of all plug-in modules that were successfully loaded.)

- `version` — An arbitrary character string denoting the software version of the plug-in module, e.g. "V1.0".

- `producer` — A character string denoting the plug-in producer, e.g. "Omicron NanoScience GmbH".

❺ The Vernissage core software will call this method when loading the plug-in DLL in order to determine the type of plug-in. As the only plug-in type currently supported is *Exporter* (a plug-in dedicated to exporting MATRIX result data into some other format), this method must return the type code constant `PlugInBase::PlugInType::pit_Exporter`.

❻ The Vernissage core software will call special methods after it has completed loading a plug-in module (routine `init()`) and when it is about to shut down (routine `shutdown()`). You may use these routines if your plug-in module requires dedicated initialisation or shutdown procedures; if this is not the case, you may provide empty implementations for either of the methods.

❼ The Vernissage core software will call this method each time the plug-in module should execute, i.e. when the user has selected the plug-in and initiated an export operation. The method arguments are:

- `pSession` — A pointer to an interface object that allows your plug-in code to access the Vernissage programming interface.

- `outputDirectoryPath` — A character string specifying the path specification of the output directory the user has chosen as target directory for the export operation.

- `pFilterSet` — An opaque pointer to a filter set object. This object describes the data the user has designated for conversion and can be used to restrict an export operation.

Note that the `run()` method may only return after it has finished its task (i.e. completed an export operation); there are no constraints on the time a `run()` method executes in order to complete its job.

The `run()` method returns a Boolean flag indicating the completion status of the operation: the value *true* indicates that the export operation was completed successfully, while *false* signals an error condition, i.e. the export operation was terminated due to some problem. (See the description of the message management support routines below in order to learn more about the options for providing details on error conditions.)

❽ The Vernissage core software will call this method if the user decides to cancel an ongoing export operation. Your plug-in code must take care that it actually terminates the `run()` method as fast as possible then.

Plug-in modules may utilise a central string buffer mechanism for issuing messages to the user. All messages placed in this buffer will be displayed automatically when the `run()` method of a plug-in returns *false*. Thus, the main purpose of the message buffer facility is to provide detailed information about the cause of an error condition.

There is only one message buffer which is shared among all plug-in modules. As the buffer will not be cleared automatically, you may want to call the `clearMessages()` service routine of the Vernissage API to erase messages issued by any plug-in module previously.

Routine	**Description**
`addMessage`	Adds an arbitrary character string to the message buffer.
`clearMessages`	Clears all messages from the message buffer.

Table 6. Message management routines.

The plug-in "skeleton" will probably always look similar to the code shown below.

```
#include "MyPlugIn.h"   ❶
PlugIn::PlugInBase* createInstance ()   ❷
{
  return new PlugIn::MyPlugIn;
}

namespace PlugIn
{
  MyPlugIn::MyPlugIn ()   ❸
  {
  }

  MyPlugIn::~MyPlugIn ()
  {
  }

  void MyPlugIn::getIdentity (std::wstring& name,   ❹
                              std::wstring& version,
                              std::wstring& producer) const
  {
    name = L"MyPlugIn";
    version = L"V1.0";
    producer = L"Dr. John Doe";
  }

  PlugInBase::PlugInType MyPlugIn::getType () const   ❺
  {
    return pit_Exporter;
  }

  void MyPlugIn::init ()   ❻
  {
  }
```

```
void MyPlugIn::shutdown ()
{
}

bool MyPlugIn::run (Vernissage::Session *pSession,   ❼
                    std::wstring outputDirectoryPath,
                    void *pFilterSet)
{
  bool result = true;
  //
  // Process Bricklets here
  //
  return result;
}

void MyPlugIn::stop ()   ❽
{
}
```

❶ We include the private header file of the plug-in first.

❷ The plug-in must provide a *factory routine* the Vernissage core software will locate and call after loading the plug-in module. The sole function of this routine is to create an instance of the plug-in class and to return that instance to the caller (i.e. the Vernissage core).

The declaration of the *factory routine* is as follows: (note that you don't need to declare the routine yourselves, however, the plug-in code must implement it.)

```
extern "C"
{
  __declspec(dllexport) PlugIn::PlugInBase* createInstance ();
}
```

❸ You may use standard C++ constructor and destructor methods, if required for some purpose.

❹ In this example, you can see the most simple implementation of the `getIdentity()` method one can choose. Remember that the name passed by the method must be unique for all plug-in modules that will be loaded by Vernissage at run-time.

❺ The `getType()` method simply returns the plug-in type identification constant. (The only plug-in type currently supported is "Exporter".)

❻ The `init()` and `shutdown()` methods can be used for providing module start-up and rundown code that you cannot (or don't want to) put into the class constructor and destructor functions.

❼ The `run()` method is the "point of entry" utilised by the Vernissage core software when the user initiates an export operation. The filter set and the path specification of the output directory chosen by the user as well as a pointer to the Vernissage API object will be passed as actual arguments.

❽ The `stop()` function must be implemented in a way it stops the operations of the plug-in module as fast as possible. (See below for more information on this topic.)

The `run()` method of a Vernissage plug-in module executes in a dedicated system thread and you must provide a suitable mechanism for terminating the execution of the routine if the user decides to cancel an export operation. The most simple way of implementing such a termination mechanism is to monitor a flag variable indicating whether the `stop()` method has been called by the Vernissage core software. Consider the example below:

```
static bool gRun;
bool MyPlugIn::run (Vernissage::Session *pSession,
                    std::wstring outputDirectoryPath,
                    void *pFilterSet)
{
  bool result = true;
  gRun = true; // Initialise to "run"
  while (gRun)
  {
    //
    // Process Bricklets here
    //
  }
  return result;
}

void MyPlugIn::stop ()
{
  gRun = false; // Signal "cancel run"
}
```

Of course, a more elegant solution would be utilising a class member variable instead of a global static flag.

Please note that whatever implementation you choose for terminating the `run()` method, the plug-in module should react as fast as possible to calls of its `stop()` method. (Failing to do so will almost certainly annoy users who expect that they can continue to work immediately once they have decided to click the *Cancel* button for stopping a progressing export operation.)

Iterating through the Bricklet Collection

Bricklets are at the centre of all software that processes MATRIX result files and most Vernissage services require you to pass a Bricklet reference. Hence, using the mechanisms for traversing the Bricklets contained by one or more result files is essential for both plug-in modules and third party applications.

The Vernissage API provides a mechanism that plug-in modules (but also third party applications) may use to traverse the set of Bricklets the various result sets loaded consist of. This mechanism will pass the Bricklets result set by result set and in the chronological order in which they were created. In addition, the traversal mechanism is capable of applying filters and selections a Vernissage user has set up so that a plug-in module only considers Bricklets that match the current selection or filter criteria.

The below table lists the routines providing access to the Bricklet traversal mechanism of the Vernissage software.

Routine	**Description**
`getNextBricklet`	Returns the next Bricklet from the set of Bricklets loaded.
`releaseBrickletContext`	Terminates a Bricklet traversal process and restarts it.
`getBrickletCount`	Returns the total number of Bricklets loaded

Table 7. Bricklet traversal routines.

The below C++ code fragment shows how to iterate through the set of Bricklets that has been loaded. (It is assumed that the variable `pSession` contains a pointer to an interface object of type `Vernissage::Session`.)

```
void *pContext = 0;
do
{
  void *pBricklet = pSession->getNextBricklet(&pContext);
  if (pBricklet != 0)
  {
    // Process the Bricklet here
  }
} while(pBricklet != 0);
```

The above code fragment will pass all data loaded, regardless which Bricklets had been selected for export by the user.

However, plug-in modules can access the data selection and filter set the user has applied by means of the "pFilterSet" actual parameter of the plug-in's `run()` function. Pass this parameter to the `getNextBricklet()` service for restricting the iteration process to the Bricklets that have been selected for export:

```
  void *pBricklet = pSession->getNextBricklet(&pContext,pFilterSet);
```

The API function `getNextBricklet()` returns a pointer to an opaque object representing a Bricklet; however, plug-in modules and third party applications do not access the Bricklet representation object directly through that pointer. Instead, you may pass the opaque object retrieved by a call to `getNextBricklet()` to the various Bricklet-related Vernissage API functions, as shown in the below example.

```
void *pContext = 0;
void *pBricklet = pSession->getNextBricklet(pContext,pFilterSet);

// Determine which channel has produced the Bricklet we've just retrieved
std::wstring channelName = pSession->getChannelName(pBricklet);

// What physical unit is associated with the data stored by the Bricklet?
std::wstring unitName = pSession->getChannelUnit(pBricklet);

// ...
```

By calling `getNextBricklet()` multiple times, an application or plug-in module can iterate through the set of Bricklets. Internally, the Vernissage software utilises the context argument passed to the `getNextBricklet()` function for storing information about the state of the traversal operation. The operation is complete when the last Bricklet in the set has been returned (in this case, `getNextBricklet()` will return a null pointer); however, you may terminate the traversal operation also by calling the API function `releaseBrickletContext()` as shown in the below example.

```
void *pContext = 0;
void *pBricklet;

// Get the first Bricklet
pBricklet = pSession->getNextBricklet(pContext,pFilterSet);

// Terminate the traversal operation, and restart from the beginning
pSession->releaseBrickletContext(&pContext);

// Get the first Bricklet (again)
pBricklet = pSession->getNextBricklet(pContext,pFilterSet);
```

Retrieving Related Bricklets

Bricklets that are logically associated ("related") as outlined in section **Related Bricklets** can be inquired by a set of dedicated service routines.

Routine	Description
`getSuccessorBricklet`	Returns the next Bricklet from a consecutive series of Bricklets.
`getPredecessorBricklet`	Returns the previous Bricklet from a consecutive series of Bricklets.
`getReferencedBricklets`	Returns a list of Bricklets being referenced by a specific Bricklet.
`getDependingBricklets`	Returns a list of Bricklets that "depend" on a specific Bricklet.

Table 8. Routines for finding related Bricklets.

The API functions `getSuccessorBricklet()` and `getPredecessorBricklet()` require a Bricklet as input and will return the succeeding or preceding Bricklet provided that the Bricklet passed as input is part of a consecutive series of Bricklets (such as a spectroscopy curve generated as part of a repeated single point spectroscopy operation), or has a logical "successor" or "predecessor". (An example for a Bricklet having a logical successor would be a Bricklet storing the phase data of a phase/amplitude curve set.)

A dedicated API function is provided for retrieving Bricklets that are referenced by a specific Bricklet: The service routine `getReferencedBricklets()` will return a list of Bricklets that are being referenced by the Bricklet specified as input to the routine, such as an SPM image referenced by a particular single point spectroscopy curve. Vice versa, the service routine `getDependingBricklets()` will return a list of Bricklets having a "depends-on" relationship with the Bricklet passed as input to the routine, such as a list of single point spectroscopy curves referencing a particular SPM image.

The below code fragment demonstrates how to determine the Bricklets related to a particular Bricklet retrieved by a call to the database iteration routine `getNextBricklet()`.

```
std::vector<void *> relatives;
void *pBricklet = pSession->getNextBricklet(&pContext,pFilterSet);
switch (pSession->getType(pBricklet))
{
  case Vernissage::Session::btc_SPMSpectroscopy:
    // Single point spectroscopy curve, this one should have
    // a "parent" image.
    relatives = pSession->getReferencedBricklets(pBricklet,pFilterSet);
    if (relatives.size() > 0) // "Parent" Bricklet not filtered out?
    {
      // We may have also successors if the SPS Bricklet was acquired
      // as part of a repeated spectroscopy operation.
      void *pSuccessor;
      pSuccessor = pSession-> getSuccessorBricklet(pBricklet,
                                                   pFilterSet);
      if (pSuccessor != 0) // Any successor Bricklet present?
      …
```

Obtaining Bricklet Information

The majority of the Vernissage API functions return a particular piece of information about a Bricklet. In general, these functions fall into one of three categories:

1. *Bricklet information routines* inquire general information about a Bricklet, e.g. the file specification of the result data file storing the Bricklet, the date and time of the Bricklet creation,

the minimum and maximum raw values that a Bricklet stores, the Bricklet's size and dimensionality, etc.

4. *Channel information routines* return information about the data channel that delivered the raw data stored by a Bricklet, such as the name of the channel, the physical unit associated with the data produced by the channel, the instance name of the Experiment Element representing the channel, etc.

5. *Axis information routines* inquire information about the configuration of axes associated with a Bricklet (more specific, with the data channel through which a Bricklet was delivered), such as the physical unit associated with an axis, the number of axis clocks (i.e. the *length* of an axis), the table sets associated with an axis, etc. Please note that axis information routines are usually capable of handle both plain and qualified axis names.

The following table summarises the general information query routines supported by the Vernissage API.

Routine	**Description**
`getCreationComment`	Returns the Bricklet creation comment entered by the MATRIX user.
`getCreationTimestamp`	Returns the date and time of Bricklet creation.
`getDataComment`	Returns the set of data-specific comments entered by the MATRIX user.
`getDataSetName`	Returns the name of the data set as entered by the MATRIX user.
`getDimensionCount`	Returns the dimensionality of a Bricklet (e.g. "1" for curves, "3" for volume CITS data structures, etc.)
`getResultDataFileSpec`	Returns the file specification of the result data file storing a particular Bricklet.
`getParentResultFileSpec`	Returns the file specification of the result file referencing a particular Bricklet.
`getSpatialInfo`	Returns information on the sample location(s) at which the data stored by a particular Bricklet was acquired.
`getRawMin`	Returns the minimum raw value a Bricklet stores.
`getRawMax`	Returns the maximum raw value a Bricklet stores.
`getRunCycleCount`	Returns the run cycle count of a Bricklet (see below).
`getSampleName`	Returns the sample name entered by the MATRIX user.
`getScanCycleCount`	Returns the scan cycle count of a Bricklet (see below).
`getSequenceId`	Returns the sequence identification count of a Bricklet (see below).
`getType`	Returns a code identifying the type of raw data stored by a Bricklet.
`getTriggerAxisName`	Returns the plain name of the axis triggering the channel through which a Bricklet was delivered.
`getTriggerAxisQualifiedName`	Returns the qualified name of the axis triggering the channel through which a Bricklet was acquired.
`getViewTypes`	Returns the type names of the Data Views associated with a Bricklet.
`getBrickletDataItemCount`	Returns the number of raw data values stored by a Bricklet.

Table 9. Bricklet information routines.

Each Bricklet stores information about its position in the stream of Bricklets generated during an experiment. Four different information items can be distinguished:

- Sequence ID
- Run Cycle Count
- Scan Cycle Count
- Timestamp

The *sequence ID* is a running number, starting at "1", that gets incremented for each new Bricklet. However, the sequence ID will be reset after the user has applied some structural change to the data space: Each time users modify parameters such as points per scan line (or points per curve), lines per scan frame, sub-grid state, scan mode and similar, the data space characteristics change and the next Bricklet will thus carry the sequence ID "1".

The *run cycle count* starts at "1" and gets incremented each time you start a new a data acquisition operation; what "operation" is actually counted depends on the configuration of the respective data channel. For example, topography and similar data channels will indicate a new run count each time you restart the scan process while a channel acquiring single point spectroscopy curves increases the run count for each new single point spectroscopy operation you initiate. (Note that if you use the automatic repetition facility for single point spectroscopy operations, all curves acquired during the various repetition cycles will have the same run count assigned but utilise different scan cycle counts.) For electron spectroscopy data, the situation is very similar: Each new spectrum acquisition operation will increase the run cycle count, while operations repeated automatically will use an identical run cycle count and a scan cycle count incremented for each repetition.

Notice

By default, the run cycle count and scan cycle count information will also be rendered as information overlay by the data displays when running MATRIX; you'll find this overlay in the upper left corner of each display. (The displays show the two counters as *run cycle – scan cycle* — for example "4 - 2" for the second scan cycle during the fourth operation.) In addition, the run cycle count and scan cycle count will also be used for generating result data file names (such as "my_data--4_2.I_mtrx".)

Finally, the *timestamp* specifies at which date and time a particular Bricklet has been written. When inquired, the timestamp will be provided with respect to the computer's *locale*, i.e. time zone and daylight saving time settings.

A plug-in module (or a third party application) will most often need to determine the characteristics of the data stored by a particular Bricklet. A suitable way for getting hints about the nature of such data is to analyse the dimensionality of a Bricklet and to check the type of Data Views the respective experiment had associated with the data channel that originally delivered the Bricklet. Consider the below C++ code fragment:

```
void *pBricklet = pSession->getNextBricklet(&pContext,pFilterSet);
int dimensions = pSession->getDimensionCount(pBricklet);
std::vector<Vernissage::Session::ViewTypeCode> views =
                                       pSession->getViewTypes(pBricklet);
// One-dimensional data indicates a curve
if (dimensions == 1)
  if (std::find(views.begin(),views.end(),
                Vernissage::Session::vtc_ForceCurve) != views.end())
  {
    // A force/distance curve
  }
  else if (std::find(views.begin(),views.end(),
                 Vernissage::Session::vtc_Spectroscopy) != views.end())
  {
    // A curve from a single point spectroscopy operation
  }
  ...
```

As shown in the above example, the combined information of Bricklet dimensionality and Data View type associations provide a good means to determine the experimental context in which a Bricklet was produced; however, in most cases it is sufficient to call the `getType()` service for determining the contents of a Bricklet. The `getType()` routine returns a code identifying the raw data contents of a Bricklet, as demonstrated below:

```
Vernissage::Session::BrickletTypeCode code = pSession->getType(pBricklet);
if (code == Vernissage::Session::btc_SPMImage)
  // An SPM image
  ...
```

You may also choose different approaches to getting hints on the nature of a Bricklet; however, simple checks are often inadequate, as shown in the below example.

```
void *pBricklet = pSession->getNextBricklet(&pContext);
std::wstring channelName = pSession->getChannelName(pBricklet);
if (channelName == L"Z")
{
  // We *cannot* be sure that this is a topography image!
}
```

As the name of the channel through which a Bricklet was delivered is actually arbitrary, you should not use it for "guessing" the contents of a Bricklet.

The following table summarises the Vernissage API routines dedicated to the query of channel-related information.

Routine	**Description**
`getChannelName`	Returns the logical name of the data channel associated with a Bricklet.
`getChannelInstanceName`	Returns the name of the Experiment Element instance representing the data channel associated with a Bricklet.
`getChannelUnit`	Returns the physical unit of the data delivered by a data channel.
`getChannelRawMin`	Returns the minimum raw value a particular data channel may deliver.
`getChannelRawMax`	Returns the maximum raw value a particular data channel may deliver.

Table 10. Channel information routines.

For SPM experiments, the difference between the logical channel name and the Experiment Element instance name of a data channel is significant: The instance name of the Experiment Element is a *technical* name, similar to the identifier name of a C++ class or routine, while the logical name of a channel most often represents the signal the particular channel acquires. For example, the Experiment Element instance "*I_V* " used by the standard STM spectroscopy experiment shipped with the MATRIX system represents a channel acquiring the tunnelling current during *V*-spectroscopy operations. Consequently, the logical name of the channel represented by the Experiment Element instance "*I_V* " is "*I(V)* ".

As outlined in section **Axes and Axis Hierarchies**, information about the configuration of the axis hierarchy associated with a data channel is essential for analysing the contents of a Bricklet. Thus, the Vernissage API offers several functions that assist in determining the axis hierarchy characteristics.

Routine	Description
`getRootAxisName`	Returns the plain name of the root axis of the axis hierarchy associated with a particular Bricklet.
`getRootAxisQualifiedName`	Returns the qualified name of the root axis of the axis hierarchy associated with a particular Bricklet.
`getTriggerAxisName`	Returns the plain name of the axis triggering the data channel through which a particular Bricklet was delivered.
`getTriggerAxisQualifiedName`	Returns the qualified name of the axis triggering the data cannel through which a particular Bricklet was delivered.
`getAxisUnit`	Returns the name of the physical unit associated with an axis.
`getAxisClocks`	Returns the number of clocks configured for an axis.
`getAxisDescriptor`	Returns the complete set of configuration data for an axis.
`getAxisTableSets`	Returns the table sets associated with an axis.
`getAxisParameter`	Returns information about an axis parameter associated with a particular axis.
`getAxisParameters`	Returns information about all axis parameter associated with a particular axis

Table 11. Axis information routines.

The axis hierarchy associated with a Bricklet has to be considered for almost any Bricklet in order to process the Bricklet contents correctly. For each axis, you may inquire an *axis descriptor* providing all information characterising a particular axis configuration. In C++, the axis descriptor is defined as follows:

```
struct AxisDescriptor
{
  int rawStart;
  int rawIncrement;
  double physicalStart;
  double physicalIncrement;
  std::wstring physicalUnit;
  bool mirrored;
  int clocks;
  std::wstring triggerAxisName;
};
```

The below example code fragment utilises axis descriptors to determine the configuration of a Bricklet containing two-dimensional data.

```
if (pSession->getDimensionCount(pBricklet) == 2)
{
  // Two dimensions, probably an image
  Vernissage::Session::AxisDescriptor triggerAxis =
    pSession->getAxisDescriptor(pBricklet,
                                pSession->getTriggerAxisName(pBricklet));

  // Determine the length of one "line" of data
  int lineLength = triggerAxis.clocks;
  if (triggerAxis.mirrored)
  {
    // The axis has the "mirrored" characteristic, thus it has a
    // "forward" and a "backward" section. Thus, the length of one line
    // is only half the number of clocks that triggered the channel.
    lineLength = lineLength / 2;
  }

  // Now we can determine the Bricklet "width". If the axis has the unit
  // "meter", then it would be really a width (probably the scan area
  // width.)
  double width = triggerAxis.physicalStart +
                 (lineLength - 1) * triggerAxis.physicalIncrement;

  // There must be another axis, because the Bricklet has two dimensions:
  Vernissage::Session::AxisDescriptor masterAxis =
    pSession->getAxisDescriptor(pBricklet,
                                triggerAxis.triggerAxisName);
  ...
```

Please note that most of the axis-related API functions of Vernissage can handle both plain axis names and qualified axis names. (See section **Axes and Axis Hierarchies** for more information on the different views to axis names.) For example, the information returned by the routine `getAxisDescriptor()` depends on the axis name passed: If you specify a plain axis name as input to the routine, the field *triggerAxisName* of the axis descriptor data structure returned will also contain a plain name. Likewise, passing a qualified axis name will cause the routine to fill a qualified axis name into the field *triggerAxisName.*

In the above code fragment, various techniques for retrieving information from an axis descriptor are demonstrated and obviously it can be laborious to determine all details required for processing the contents of a Bricklet. Sometimes, it might thus appear more convenient to simply query the value of an Experiment Element instance parameter instead of analysing the axis hierarchy configuration (for example, one could locate an Experiment Element instance of type *XYScanner* and query the value of its parameter `Width` in order to determine the scan area width at Bricklet creation time), however, this approach to analysing Bricklet contents is actually not always future-proof: while current MATRIX experiments might use a particular Experiment Element in always the same way, this might not be true for the experiments supported by forthcoming MATRIX versions. For example, consider an experiment utilising *two* instances of the Experiment Element *XYScanner* — As it is actually hard to determine which of the two instances is associated with the data channel through which a particular Bricklet has been delivered, you would not be able to select the correct instance for the parameter query and thus might end up with the wrong width value.

Axes can have associated "meta" information referred to as "axis parameters". Axis parameters store data related to the specific semantics of a particular axis and can be important for certain operations, see chapter "Axis Parameters" on page 42. (For example, some axis types used in conjunction with electron spectroscopy experiments are associated with calibration information such as dispersion correction and normalisation factors that play an important role in quantitative analysis procedures.)

Table sets associated with the axes of an axis hierarchy provide another important aspect when analysing the contents of a Bricklet, as they restrict the number of clocks that triggered a data channel during experiment execution. The MATRIX system currently utilises table sets for defining the sub-grid used during raster spectroscopy operations only (in other words, in all other cases when there is no table set defined, a plug-in module can treat the missing table set just like a table set of the form [1, *clocks*, 1] which means that a trigger will be produced for all clocks configured for a particular axis). As a result, when developing a plug-in module, it is currently safe to assume that the table set configuration must be considered only when processing Bricklets that have a dimensionality of three and are associated with a View of type *Spectroscopy* (type code `vtc_Spectroscopy`) or for which the `getType()` service returns the identification code `btc_VolumeCITS`.

The table set concept is very powerful but also quite complex and using it properly requires some careful thought. The below example code demonstrates how to utilise table set information that has been attached to the SPM spectroscopy axis and the *X*-axis of an axis hierarchy; the hierarchy consists of three axes that have been ordered in the typical manner of a raster spectroscopy experiment: The root axis is *Y* which triggers *X* which in turn triggers the spectroscopy axis. A table set attached to the *X*-axis restricts the clock positions of the *Y*-axis at which *X* gets actually triggered. In the same manner, a table set attached to the spectroscopy axis restricts the clock positions of the *X*-axis at which a spectroscopy operation gets initiated.

The example implements an algorithm that iterates through the clock positions of the *Y*– and *X*-axis, checking for all possible combinations whether they will trigger the spectroscopy axis.

```
Vernissage::Session::AxisDescriptor specAxis =
  pSession->getAxisDescriptor(pBricklet,
                              pSession->getTriggerAxisName(pBricklet));
Vernissage::Session::AxisDescriptor xAxis =
  pSession->getAxisDescriptor(pBricklet,
                              specAxis.triggerAxisName);
Vernissage::Session::AxisDescriptor yAxis =
  pSession->getAxisDescriptor(pBricklet,
                              xAxis.triggerAxisName);
Vernissage::Session::AxisTableSets sets =
  pSession->getAxisTableSets(pBricklet,
                             pSession->getTriggerAxisName(pBricklet));

Vernissage::Session::TableSet xSet = sets[specAxis.triggerAxisName];
Vernissage::Session::TableSet ySet = sets[xAxis.triggerAxisName];

Vernissage::Session::TableSet::const_iterator yIt = ySet.begin();
Vernissage::Session::TableSet::const_iterator xIt = xSet.begin();

int tx = (*xIt).start;
int ty = (*yIt).start;
int x, y = 1;

// Use the axis configuration data for iterating the axis hierarchy
while (y <= yAxis.clocks) // Root axis
{
  x = 1;
  while (x <= xAxis.clocks) // Axis triggered by root axis
  {
    if ((x == tx) && (y == ty))
    {
      // Table set and axis "positions" match --> The spectroscopy axis
      // will be triggered!

      // ...

      // Advance to the next position described by the interval
      tx = tx + (*xIt).step;
      // End of the X-axis interval reached? If so, select the next
      // interval of the table set (if any)
      if (tx > (*xIt).stop)
      {
        ++xIt;
        if (xIt != xSet.end()) // End of table set not yet reached?
        {
          // We have selected the next X-axis interval, so we
          // reinitialise our coordinate variable now.
          tx = (*xIt).start;
        }
        else
        {
          // We have traversed all X-axis intervals specified. Thus,
          // advance the Y-axis interval and reset the X-axis interval.
          ty = ty + (*yIt).step;
          xIt = xSet.begin();
          tx = (*xIt).start;
          // End of the Y-axis interval reached? If so, select the next
          // interval (if any)
          if (ty > (*yIt).stop)
          {
            ++yIt;
            if (yIt != ySet.end())
            {
              // We have selected the next Y-axis interval, so we
              // reinitialise our coordinate variable now.
              ty = (*yIt).start;
            }
          }
        }
      }
    }
    x = x + 1; // Next point
  }
  y = y + 1; // Next line
}
```

Accessing Raw Data

By default, Vernissage will not load the raw data content of any Bricklet. Thus, when your software requires access to actual raw data items, you must direct the Vernissage core to first load the "payload" of a Bricklet.

Consider the following C++ code fragment:

```
void *pBricklet = pSession->getNextBricklet(&pContext,pFilterSet);
const int *pBuffer;
int rawDataItems;
loadBrickletContents(pBricklet,&pBuffer,rawDataItems);
int firstRawDataItem = *pBuffer;

...

unloadBrickletContents(pBricklet);
```

The above code fragment demonstrates how to gain access to the raw data stored by a particular Bricklet: The Vernissage API routine `loadBrickletContents()` will allocate a data buffer large enough for holding the Bricklet contents, load the raw data into the buffer and return the buffer address to the calling software. When the routine returns, the buffer will hold the number of raw data items returned in the variable passed as third argument to `loadBrickletContents()`; the buffer will contain the raw data items in the exact order in which they were originally acquired during experiment execution.

The contents of the buffer allocated by `loadBrickletContents()` remain valid until the Vernissage API routine `unloadBrickletContents()` gets called with the same Bricklet argument. As Bricklets can become quite large, you should double-check whether your plug-in module code contains corresponding calls to `unloadBrickletContents()` for each call to `loadBrickletContents()`; missing calls to `unloadBrickletContents()` will cause "memory leaks" of probably significant size.

See section **Raw Data Organisation** for more information on the order of raw data items in a Bricklet.

Raw values represent data exactly as acquired by the instrument electronics, thus such values must often be converted into their corresponding physical quantities. The Vernissage API offers routines that can compute a physical quantity from a raw value and vice versa. For this purpose, the respective routines will utilise the same data transfer functions that were used by the MATRIX software at the time the result set was created.

Routine	Description
`toPhysical`	Transforms a raw value into its physical equivalent.
`toRaw`	Transforms a physical quantity into its raw equivalent.

Table 12. Data transformation routines.

Although most information required for interpreting the contents of a Bricklet can be obtained by calling the various Bricklet and axis information routines, in some scenarios it can also be of interest to inquire details on the Experiment Element instances of a specific experiment, for example, the value of some Experiment Element instance parameter.

In general, one may either want to check the value of a particular deployment parameter of an Experiment Element instance, or the value of an Experiment Element instance parameter at the time a specific Bricklet was created:

- Deployment parameter values can be of interest if details regarding the *static* configuration of an experiment are required. This should be a very rare case, but can be useful for special purposes.
- Experiment Element instance parameter values provide background information on the dynamic aspects of an experiment configuration, e.g. the gap voltage used, the effective feedback loop gain, or the configured scan area offset.

The Vernissage API provides the following routines for inquiring Experiment Element information:

Routine	Description
`getExperimentElementInstanceNames`	Returns the names of all Experiment Element instances of an experiment.
`getExperimentElementDeploymentParameter`	Returns the (character string) value of a deployment parameter of an Experiment Element instance.
`getExperimentElementDeploymentParameters`	Returns the (character string) values of all deployment parameters of an Experiment Element instance.
`getExperimentElementParameter`	Returns the value of a single parameter of an Experiment Element instance at Bricklet creation time.
`getExperimentElementParameters`	Returns the value of all parameters of an Experiment Element instance at Bricklet creation time.

Table 13. Experiment element information routines.

The parameters of an Experiment Element can be of different data types; the Vernissage API functions will hence return any parameter value as a wide character string. Although this simplifies the calling sequence, it may require the implementation of dedicated value conversion mechanisms. Consider the following C++ code fragment:

```
// Get the names of all instances of Experiment Element type "XYScanner"
std::vector<std::wstring> instanceNames(
  pSession->getExperimentElementInstanceNames(pBricklet,L"XYScanner"));

// If there is only one instance, obtain its parameters
if (instanceNames.size() == 1)
{
  std::wstring scannerName = instanceNames[0];
  std::map<std::wstring, Vernissage::Session::Parameter> parameters =
    pSession->getExperimentElementParameters(pBricklet,scannerName);

  // Determine the scan area offset at the time the Bricklet referred
  // to by 'pBricklet' was created
  std::wstring xOff = parameters[L"X_Offset"].value;
  std::wstring yOff = parameters[L"Y_Offset"].value;

  // This is not really required, as we know that the data type of the
  // parameters is "double-precision floating point"
  if (parameters[L"X_Offset"].valueType ==
        Vernissage::Session::Parameter::vt_Double)
  {
    double xOffset = _wtof(xOff.c_str());
    double yOffset = _wtof(yOff.c_str());

    // Next conditional is only for demonstration purposes, as the unit
    // of the scanner parameters "X_Offset" and "Y_Offset" is always
    // "Meter".
    if (parameters[L"X_Offset"].unit == L"Meter")
    {
      // Turn into Nanometers
      double xOffsetNm = xOffset / 1.0e-9;
      double yOffsetNm = yOffset / 1.0e-9;

      ...
```

In the above example, the names of all Experiment Element instances of type *XYScanner* are inquired; afterwards, the code fetches the complete parameter set of the (only) instance. Finally, the values of the parameters `X_Offset` and `Y_Offset` are converted into double-precision floating point values by means of the Microsoft Visual C++ run-time library function `_wtof()`.

Please note that "Foreign Parameters" you may have added to an experiment by means of the respective mechanisms of the MATRIX Automated Task Environment (MATE) are treated as parameters of the pseudo-Experiment Element instance "-Foreign" (of type "Foreign"). The following example demonstrates how to access the foreign parameter "My_Temperature_Parameter":

```
std::vector<std::wstring> instanceNames(pSession->
  getExperimentElementInstanceNames(pBricklet,L"Foreign"));
if (instanceNames.size() == 1)
{
  std::wstring foreign = instanceNames[0];
  std::map<std::wstring, Vernissage::Session::Parameter> parameters =
    pSession->getExperimentElementParameters(pBricklet,foreign);
  std::wstring temperature =
    parameters[L"My_Temperature_Parameter"].value;
```

For special purposes, the Vernissage API also provides an option for obtaining the value of a deployment parameter utilised by an Experiment Element instance. As already mentioned, accessing the Experiment Element deployment parameters is rarely required. However, you may use the `getExperimentElementDeploymentParameter()` service function for obtaining the value of some deployment parameter, as shown in the below C++ code fragment:

```
std::wstring actuatorName =
    getExperimentElementDeploymentParameter(pBricklet,
                                            L"Spectroscopy",
                                            L"Actuator_Axis_1");
```

The above code fragment queries the value of the deployment parameter `Actuator_Axis_1` of the SPM Experiment Element instance *Spectroscopy*. The returned character string could, for example, be "*V*" or "Z", depending on the configuration of the Experiment Element.

You may also inquire a list of all deployment parameters and their associated values used by a specific Experiment Element instance by calling the service routine `getExperimentElementDeploymentParameters()`.

Please note that all values of Experiment Element deployment parameters are of type "character string" and have no associated information (such as a unit).

Miscellaneous Services

Some functions of the Vernissage API serve very special purposes or provide information not related to the processing of Bricklets. In particular, these functions are:

- `getSession()` and `releaseSession()` — Third party applications must utilise these routines for managing the object providing access to the remaining Vernissage API functions.
- `getExperimentInfo()` and `getMetaData()` — These routines inquire information about the experiment that produced a result set and about the result set itself.
- `getPlugInInfo()` — This routine returns information about the plug-in modules loaded.
- `showWorkInProgress()` — This routine allows plug-in modules to indicate the progress of an ongoing operation.

The Vernissage API also provides a number of convenience functions not related to the processing of MATRIX result files or result data files. These functions, however, are often useful when developing plug-in modules.

Routine	**Description**
`ansiToUnicode`	Converts an 8-bit ANSI multi-byte character string into its Unicode equivalent.
`unicodeToAnsi`	Converts an Unicode character string into its 8-bit ANSI multi-byte equivalent.
`createOutputFile`	Creates a new output file and opens it for writing.
`closeOutputFile`	Closes an output file.
`createDirectory`	Creates a new directory.
`directoryExists`	Checks whether a particular directory already exists.
`makePath`	Appends a file specification component to an existing path specification.
`splitPath`	Splits an existing path specification into its components.

Table 14. Convenience routines.

6. Flat File Format Reference

This part details the *Flat File Format* (FFF) used by default for exporting MATRIX result data sets. Note that you must be familiar with the concepts and terminology outlined in chapter **Understanding the Result File System** in order to understand this specification.

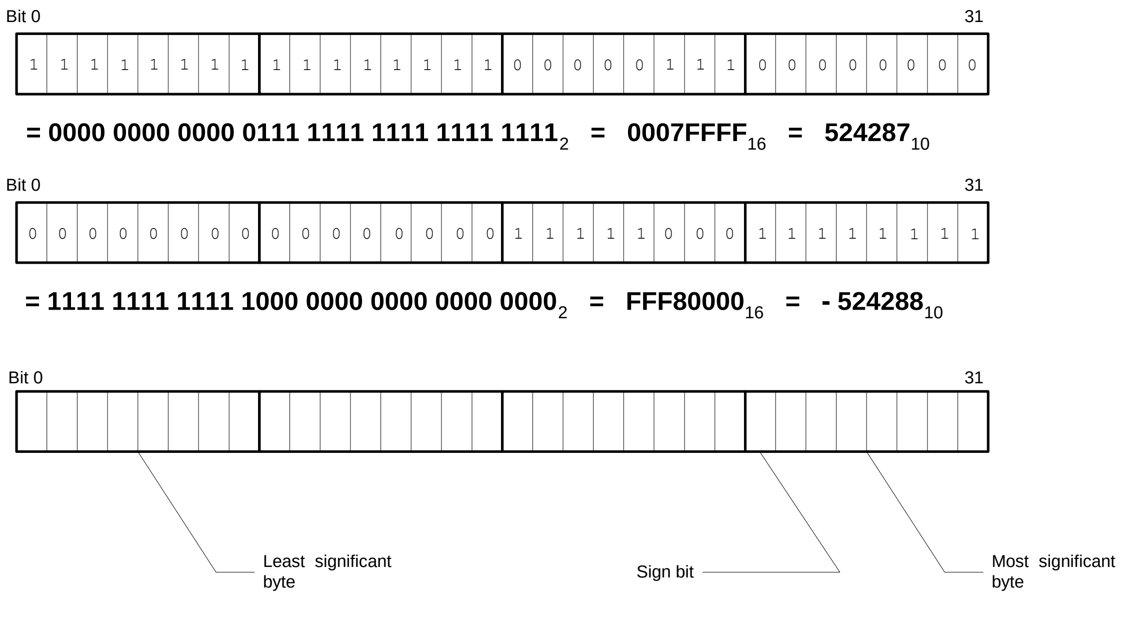
The FFF is a *flattened* representation (hence the name) of the native MATRIX result set layout called *Data Space Persistence Format* (DSPF). Unlike the DSPF, the FFF combines the contents of each Bricklet (i.e. acquired data) with related information into a single file in a way that the file is self-contained and can be processed without referring to additional information. The main advantages of the FFF are:

- Simple structure — The FFF can be interpreted by straightforward file parsers that can be developed easily.
- Self-contained contents — Each file contains all the information required to interpret the acquired raw data, thus the files can be used, stored and shared individually.
- Stores original raw data — The acquired data are stored exactly as provided by MATRIX and unchanged with respect to the original result data set.

The major drawback of the FFF is that it requires more disk space than the DSPF. In addition, the *logbook* character of the DSPF is not retained by the FFF.

General Structure

The Flat File Format is a binary data format storing raw measurement data as well as additional information required to interpret the raw data correctly. Raw data items are copied exactly as originally acquired by the MATRIX software and are thus 32-bit signed integer figures (in 2-th complement format) and with "little endian" byte order (i.e. the least significant byte is stored first).



$= 0000\ 0000\ 0000\ 0111\ 1111\ 1111\ 1111\ 1111_2 = 0007FFFF_{16} = 524287_{10}$

$= 1111\ 1111\ 1111\ 1000\ 0000\ 0000\ 0000\ 0000_2 = FFF80000_{16} = -524288_{10}$

Figure 37. Raw data item format

Besides the raw data, each Flat file stores the following details:

- Experiment and user information.
- Bricklet creation information such as timestamp, creation comment, utilised channel, etc.
- The axes associated with the Bricklet as well as the configuration and characteristics of each axis.
- The transfer function required to transform a raw value into a physical quantity.
- Important experiment and axis parameters.

For storing character sequences (such as parameter names), the FFF utilises *string descriptors* consisting of a 32-bit length field, followed by a sequence of 16-bit values: while the length field determines the number of characters the sequence comprises, the sequence of 16-bit values describes the characters in Unicode UTF-16 wide-character encoding.

Length	**H**	**e**	**l**	**l**	**o**
(5)	(0048_{16})	(0065_{16})	($006C_{16}$)	($006C_{16}$)	($006F_{16}$)

Figure 38. Character sequence example — Encoding of "Hello"

If a string is *empty*, its length field is zero. In this case, the string descriptor consists of the length field only.

Real figures are expressed as double-precision floating point numbers utilising the IEEE floating point number representation. Each value is represented as an eight byte sequence consisting of a sign bit, an 11-bit excess-1023 binary exponent (containing the order of magnitude of the value) and a 52-bit mantissa (containing the value itself). The most significant bit of any double-precision floating point value is always the sign bit. If it is 1, the number is considered negative; otherwise, it is considered a positive number.

Because exponents are stored in an unsigned form, the exponent is biased by half its possible value, i.e. it is actually 1023. One can compute the actual exponent value by subtracting the bias value from the exponent value.

The mantissa is stored as a binary fraction greater than or equal to 1 and less than 2. (Note that there is an implied leading 1 in the mantissa in the most-significant bit position, so mantissas are actually 53 bits long. However, the most-significant bit is not stored.)

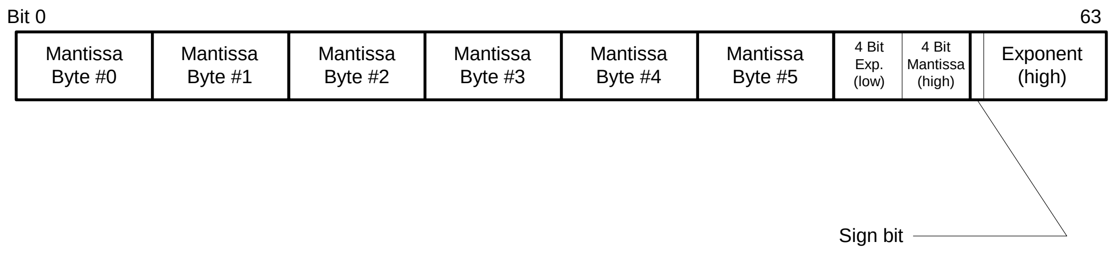


Figure 39. Floating point figure representation

Notice

Note that the FFF real figure representation is identical to the one used by the C++ data type `double` on an Intel-CPU-based PC, thus you can read a floating point figure from an FFF-encoded file directly into a C++ variable of type `double` without the need to run any conversion operation.

Bricklet Container File Structure

Basically, each Flat file consists of a series of *sections* each storing a stream of 32-bit integer values. The sequence of sections is outlined in the below table.

No.	Name	Description
1	File identification	This section stores a "magic word" identifying the file format and the structure level identification of the format.
2	Axis hierarchy description	Stores a description of the axis hierarchy associated with the Bricklet and the complete axis configuration information. (Note that axis parameters are part of the "Experiment parameters list".)
3	Channel description	Stores information about the data channel through which the Bricklet was delivered and the transfer function required for transforming raw data into physical quantities. In addition, this section lists the types of all Data Views that the experiment had associated with the channel.
4	Creation information	This section contains the Bricklet creation timestamp and the various textual information items (sample name, comments, etc.) associated with the Bricklet.
5	Raw data	The acquired raw data of the Bricklet.
6	Sample position information	If applicable, this section stores coordinates describing the sample positions at which the raw data were acquired.
7	Experiment information	Stores information about the experiment that has generated the Bricklet and related information.
8	Experiment parameter list	This section stores the values of all monitored Experiment Element parameters as well as all axis parameters that were valid at Bricklet creation time.
9	Experiment Element deployment parameter list	This section stores information about the deployment parameters of all Experiment Element instances.

The different file section types have an individual structure; the structure of each type is detailed below.

File Identification

The file identification section contains only two 32-bit integer values depicted below:

	Byte 0	Byte 1	Byte 2	Byte 3	
"Magic word"	70	76	65	84	= 0x54414C46
File structure level	48	49	48	48	= 0x30303130

The first four bytes are the so-called "magic word" allowing file reader software to identify the file contents. When interpreted as ASCII characters, these four bytes form the word "FLAT".

The subsequent four bytes describe the file structure level, i.e. the "format version number" of the FFF. The structure level is stored as ASCII character sequence and will currently be "0100".

Note: The structure level of the Flat File Format is tightly coupled to the structure level of the MATRIX Data Space Persistence Format. If the DSPF structure changes, the structure of the FFF will most likely also change.

Axis Hierarchy Description

The axis description section stores information about the axis hierarchy (starting with the trigger axis) associated with a Bricklet and details the configuration of all axes in the hierarchy. Besides a counter specifying the number of axes the axis hierarchy consists of, the section comprises a set of axis descriptions.

Byte	0	1	2	3	
Integer	Axis Count				Number of axes
Axis Description	Trigger Axis Description #1				
Axis Description	Axis Description #2				
	...				

Each axis description takes the form depicted below (note that all axis names are qualified axis names):

Byte	0	1	2	3	
Wide-character String	Axis Name Length				Name of the axis
	Character #1		Character #2		
	...				
Wide-character String	Parent Axis Name Length				Name of the parent axis (empty string in case of the root axis)
	Character #1		Character #2		
	...				
Wide-character String	Axis Unit Name Length				Physical unit name
	Character #1		Character #2		
	...				
Integer	Clock Count				Number of clock positions
Integer	Axis Start Value ("Raw")				Start value as "raw" figure
Integer	Axis Increment ("Raw")				Increment as "raw" figure
Double	Axis Start Value (Physical)				Start value as physical quantity

Double	Axis Increment (Physical)		Increment as physical quantity
Boolean	Mirrored		"Mirrored" characteristics flag
Integer	Table Set Count		Number of associated table sets
Wide-character String	Set #1: Axis Name Length		Name of the axis the first table set is associated with
	Character #1	Character #2	
	...		
Integer	Interval Count		Number of intervals
Integer	Interval #1: Start Clock		Start clock position
Integer	Interval #1: Stop Clock		Last clock position
Integer	Interval #1: Step		Clock increment of interval
	...		(Repeated for all intervals)
Wide-character String	Set #2: Axis Name Length		Name of the axis the second table set is associated with
	Character #1	Character #2	
	...		
	...		(Repeated for all table sets)

Notice

Please note that "Table Set Count" can also be zero; in this case, no table set description fields will be generated.

Channel Description

The next section in a file utilising the FFF is dedicated to the data channel through which the Bricklet described by the file has been delivered. The channel description section contains all information required for identifying the channel and for interpreting the raw data contents of the Bricklet; the structure of the section is shown below.

Byte	0	1	2	3	
Wide-character String	Channel Name Length				Logical name of the channel
	Character #1		Character #2		
	...				
Wide-character String	Transfer Function Name Length				Name of the transfer function
	Character #1		Character #2		
	...				
Wide-character String	Unit Name Length				Physical unit name
	Character #1		Character #2		
	...				

Integer	Parameter Count		Number of function parameters
Wide-character String	Parameter #1: Name Length		Name of first transfer function parameter
	Character #1	Character #2	
	...		
Double	Parameter #1: Value		Parameter value
Wide-character String	Parameter #2: Name Length		Name of second transfer function parameter
	Character #1	Character #2	
	...		
	...		(Repeated for all parameters)
Integer	Data View Type Count		Number of Data View types
Integer	Data View#1: Type Identifier		View type code
	...		(Repeated for all Views)

The channel description section provides information on the *transfer function* required for computing physical quantities from the raw value content of the Bricklet. Besides the name of the transfer function to be used, the section stores also the function's parameter values.

Currently, there a three different types of transfer functions; the table below details these functions.

Name	**Calculation rule**	**Parameter names**
`TFF_Linear1D`	$physical = \frac{raw - offset}{f}$	Offset Factor
`TFF_MultiLinear1D`	$physical = \frac{(raw_1 - offset_{pre}) \cdot (raw - offset)}{f_{neutral} \cdot f_{pre}}$	Raw_1 PreOffset Offset NeutralFactor PreFactor
`TFF_Identity`	$physical = raw$	—

The calculation rules given in the above table must be used for transforming a raw data item from a Bricklet (to be inserted as parameter *raw* of the respective rule) into a physical quantity. The parameter names in the third column specify the names of the remaining function parameters as used in a data channel description.

The channel description also stores a list of Data View types; Data Views of the types denoted by the list were attached to the channel at experiment execution time and hint at the nature of the data stored by a Bricklet. (Most often, the dimensionality of a Bricklet — that is, the number of axes it is associated with — together with the Data View types provide a good means for determining the type of data the Bricklet contains. For example, a dimensionality of three and a Data View of type `vtc_Spectroscopy` indicate that the Bricklet stores spectroscopy curves acquired during a volume CITS experiment. As another example, a dimensionality of two together with a Data View of type `vtc_ForwardBackward2D` would hint at a topography image or a similar image type.)

See section **Understanding Data Views** for more information on Data Views and the Data View type codes currently supported.

Creation Information

The creation information section stores the date and time when the Bricklet was originally stored by the MATRIX system. This information is provided as a single 64-bit integer specifying the creation time as the number of seconds since midnight of January, 1st 1970; the respective value is structurally compatible with the standard C-library data type *time_t*.

The second data item of the Bricklet creation information section is a wide-character string storing the sample name, the data set name, and all comments specified by the MATRIX user during experiment execution time.

Byte	0	1	2	3	
64-bit Integer	Timestamp				Creation timestamp
Wide-character String	Information String Length				Bricklet information as specified by user
	Character #1		Character #2		
	...				

Raw Data

Besides the raw data contents of a Bricklet, this section stores two 32-bit integers determining the total number of data items the Bricklet can store and the actual number of data items stored. If the Bricklet results from a completed data acquisition operation, the two figures will be identical. If the data acquisition process was stopped by the user while progressing, the *Data Item Count* figure will represent the number of data items actually acquired before the process was stopped.

The structure of the raw data section is shown below.

Byte	0	1	2	3	
Integer	Bricklet Size				Bricklet size in data items
Integer	Data Item Count				# of data items actually stored
Integer	"Raw" Data Item #1				Bricklet contents
	...				

See section **Raw Data Organisation** for information on the order of raw data items in a Bricklet.

Sample Position Information

A Bricklet might store information about the sample location(s) at which it was acquired. (The most prominent case is single point spectroscopy; Bricklets storing single point spectroscopy curves contain information about the sample location at which the spectroscopy operation was run. However, a Bricklet can also be associated with more than one sample location, such as a curve consisting of a number of data points each of which was acquired at a different sample location.)

The sample positions are provided as offsets (in metres) to the centre of the configured scan area. If no position information is available, the *Offset Count* field will be zeroed.

Byte	0	1	2	3	
Integer	Offset Count				# of offsets (*X/Y* pairs) stored
Double	Offset #1 *X*-direction				First *X*-axis offset to centre of scan area
Double	Offset #1 *Y*-direction				First *Y*-axis offset to centre of scan area
	...				

Experiment Information

The experiment information section stores several (wide-)character strings dedicated to details on the experiment which generated a particular Bricklet at execution time, about the software products that were used for result file and Flat file generation and the user who conducted the experiment. In particular, the section comprises the following information:

- The name, version identification and description text of the MATRIX experiment that has produced the Bricklet.
- The file specification of the description file defining the experiment.
- The identification of the software product that has created the Flat file.
- The identification of the MATRIX software that has created the original result file.
- The MATRIX-specific user name (usually "default") of the user who conducted the experiment.
- The Microsoft Windows account name of the user who conducted the experiment.
- The file specification of the original result data file storing the Bricklet.
- The run and scan cycle identification numbers associated with the Bricklet.

The structure of the experiment information section is shown below.

Byte	0	1	2	3	
Wide-character String	Experiment Name Length				Experiment name
	Character #1		Character #2		
	...				
Wide-character String	Experiment Version Length				Experiment version identifier
	Character #1		Character #2		
	...				
Wide-character String	Experiment Description Length				Description text associated with experiment
	Character #1		Character #2		
	...				

Wide-character String	Experiment File Specification Length		File specification of original XML experiment description
	Character #1	Character #2	
	...		
Wide-character String	File Creator Identification Length		Identification of Flat file creator software
	Character #1	Character #2	
	...		
Wide-character String	Result File Creator Identification Length		Identification of MATRIX result file creator software
	Character #1	Character #2	
	...		
Wide-character String	User Name Length		MATRIX user identification
	Character #1	Character #2	
	...		
Wide-character String	Account Name Length		Microsoft Windows account name
	Character #1	Character #2	
	...		
Wide-character String	Result Data File Specification Length		File specification of original result data file
	Character #1	Character #2	
	...		
Integer	Run Cycle Identification		Experiment run cycle
Integer	Scan Cycle Identification		Experiment scan cycle

Experiment Parameter List

Each Flat file also contains a snapshot of the Axis and Experiment Element instance parameters at the time the Bricklet was created.

The parameters are grouped by Axis or Experiment Element instance; for each parameter, the following information is provided:

- The parameter name
- The SI unit associated with the parameter value. (See section **Data Types and Formats** for more information on the supported units.)
- The identification code of the data type used for encoding the parameter value. (See section **Data Types and Formats** for more information on the parameter value type codes supported.)
- The parameter value, provided as character string.

A schematic view of the Experiment parameter list is depicted below.

Byte	0	1	2	3	
Integer	Instance Count				# of Axis and Experiment Elements instances
Wide-character String	Instance #1: Name Length				Name of the first Axis or Experiment Element instance
	Character #1		Character #2		
	...				
Integer	Parameter Count				# of parameters
Wide-character String	Parameter #1: Name Length				Parameter name
	Character #1		Character #2		
	...				
Integer	Parameter #1: Value Type Code				Data type code of parameter
Wide-character String	Parameter #1: Unit Length				Parameter (SI) unit
	Character #1		Character #2		
	...				
Wide-character String	Parameter #1: Value Length				Parameter value, encoded as character string
	Character #1		Character #2		
	...				
	...				(Repeat for all parameters and Experiment Element instances)

Experiment Element Deployment Parameter List

For special purposes, the last section in a Flat file stores a list of the deployment parameters of each Experiment Element instance. As deployment parameters use character string values only, there is no type or unit information for any of these parameters.

The structure of the Experiment Element deployment parameter list is depicted below.

Byte	0	1	2	3	
Integer	Experiment Element Instance Count				# of Experiment Elements
Wide-character String	Experiment Element Instance #1: Name Length				Name of the first Experiment Element instance
	Character #1		Character #2		
	...				
Integer	Deployment Parameter Count				# of parameters
Wide-character String	Parameter #1: Name Length				Deployment parameter name
	Character #1		Character #2		
	...				
Wide-character String	Parameter #1: Value Length				Deployment parameter value
	Character #1		Character #2		
	...				
	...				(Repeat for all parameters and Experiment Element instances)

7. Service Routines Reference

This part provides detailed descriptions of the routines that constitute the Vernissage Application Programming Interface (VAPI).

Notice

Most of the VAPI routines do not have a means to signal error conditions, thus the results of calling a service routine with invalid parameters are unpredictable.

addMessage

Adds a message to the global message buffer.

Syntax

addMessage (*message*)

Argument	Data Type	Access
message	Wide-character string	Read

C++ Binding

```
#include "Vernissage.h"
void Session::addMessage(std::wstring message);
```

Arguments

message
An arbitrary message to be added to the global message buffer.

Description

Vernissage provides a global message buffer that can be used by plug-in modules to issue arbitrary messages to the user. The message buffer mechanism is most useful for issuing error messages if a plug-in detects some problem that prevents it from completing its operations successfully.

Calling `addMessage()` will place the message string passed into the global message buffer. The Vernissage core software will display the contents of this buffer at some point, however, `addMessage()` will usually not cause the message buffer contents to be displayed immediately.

Omicron recommends to restrict the use of the global message buffer to error messages only.

Return Values

—

Associated Routines

```
getMessages()
clearMessages()
```

ansiToUnicode

Converts an 8-bit ANSI multi-byte character string into its Unicode equivalent.

Syntax	*unicodeString* := ansiToUnicode (*ansiString*)		
	Argument	**Data Type**	**Access**
	ansiString	Multi-byte character string	Read

C++ Binding

```
#include "Vernissage.h"
std::wstring Session::ansiToUnicode(std::string str);
```

Arguments

ansiString
The 8-bit multi-byte character string to be converted into Unicode.

Description

This routine converts a character string encoded as ANSI/ASCII 8-bit character sequence into the corresponding Unicode wide-character equivalent.

This is a convenience routine for simplifying the management of string data.

Return Values

Returns the wide-character Unicode-equivalent of the input character string.

Associated Routines

`unicodeToAnsi()`

clearMessages

Deletes the contents of the global message buffer.

Syntax

clearMessages ()

C++ Binding

```
#include "Vernissage.h"
void Session::clearMessages();
```

Arguments

—

Description

Vernissage provides a global message buffer that can be used by plug-in modules to issue arbitrary messages to the user. The message buffer mechanism is most useful for issuing error messages if a plug-in detects some problem that prevents it from completing its operations successfully.

Calling `clearMessages()` will erase the entire global message buffer, i.e. all messages will be deleted.

Return Values

—

Associated Routines

```
addMessage()
getMessages()
```

closeOutputFile

Closes an output file created by a previous call to the *createOutputFile* service.

Syntax

closeOutputFile (*fileHandle*)

Argument	**Data Type**	**Access**
fileHandle	Opaque reference	Read/Modify

C++ Binding

```
#include "Vernissage.h"
void Session::closeOutputFile(FILE **pHandle);
```

Arguments

fileHandle
A pointer variable storing the address of an output file handle returned by a previous call to `createOutputFile()`.

Description

This convenience routine closes an output file created by a previous call to the `createOutputFile()` service.

After this routine has returned, the specified file handle must not be used for accessing the file any longer.

Return Values

—

Associated Routines

`createOutputFile()`

createDirectory

Creates a directory or sub-directory.

Syntax

createDirectory (*dirSpec, exceptionsFlag*)

Argument	**Data Type**	**Access**
dirSpec	Wide-character string	Read
exceptionsFlag	Boolean	Read

C++ Binding

```
#include "Vernissage.h"
bool Session::createDirectory(std::wstring dirSpec,
                              bool raiseExceptions = true);
```

Arguments

dirSpec
An absolute or relative path specification determining the path to and the name of the directory to be created.

exceptionsFlag
A flag indicating whether the service will raise exceptions if any error condition is encountered.

Description

This routine creates a new directory or sub-directory. The **dirSpec** argument determines the path and name of the directory to be created and can contain a Microsoft Windows absolute (e.g. "C:\Temp\NewDir") or relative (e.g. "..\..\NewDir") path specification. If the path specification passed includes non-existing intermediate directories these will be created also.

The `createDirectory()` service will return *true* if the directory or sub-directory has been created. If the **exceptionsFlag** argument is set to *false*, the service will return *false* if the directory creation procedure fails for some reason, however, the calling module will receive no indication regarding the source of the error condition. If the **exceptionsFlag** argument is set to *true* (which is the default), the `createDirectory()` service will raise an exception of class `IOException` in case of an error condition; you may provide a standard C++ exception handler (`catch` block) for catching and processing the exception object.

I/O exception codes (stored in an exception object of type `IOException`) that can occur are:

- `ioerr_FileAlreadyExists` — Raised if the **dirSpec** argument actually refers to an existing file.
- `ioerr_DirAlreadyExists` — Raised if the specified directory already exists.
- `ioerr_UnableToCreate` — Raised if the directory creation operation has failed for some other reason.

Return Values

True if the directory has been created, *false* otherwise.

Associated Routines

`createOutputFile()`

createOutputFile

Creates a text or binary file intended for output.

Syntax

createOutputFile (*fileSpec, binaryFlag, overwriteFlag, fileHandle, exceptionsFlag*)

Argument	Data Type	Access
fileSpec	Wide-character string	Read
binaryFlag	Boolean	Read
overwriteFlag	Boolean	Read
fileHandle	Opaque reference	Modify
exceptionsFlag	Boolean	Read

C++ Binding

```
#include "Vernissage.h"
bool Session::createOutputFile(std::wstring fileSpec,
                               bool isBinary,
                               bool overwrite,
                               FILE **pHandle,
                               bool raiseExceptions = true);
```

Arguments

fileSpec
An absolute or relative file specification determining the path to and the name of the file to be created.

binaryFlag
A flag indicating whether the new file will store printable text (e.g. Unicode characters) only, or any type of data.

overwriteFlag
A flag indicating whether an existing file with the same path/name shall be silently overwritten.

fileHandle
A pointer variable for storing the address of an output file handle associated with the new file.

exceptionsFlag
A flag indicating whether the service will raise exceptions if any error condition is encountered.

Description

This routine creates a new file and opens it for write access. The **fileSpec** argument determines the path and name of the file to be created and can contain a Microsoft Windows absolute (e.g. "C:\Temp\Out.txt") or relative (e.g. "..\..\Out.txt") path specification.

If **overwriteFlag** is set to *true* and the specified file already exists, the existing file will be overwritten if possible. When **overwriteFlag** is *false* and the specified file already exists, the `createOutputFile()` routine will signal an error condition.

The type of data the new file can store is determined by the value of the **binaryFlag** argument: If set to *false*, the new file will be capable of storing text data (Unicode characters) only and certain control characters (such as "line feed") will be translated into Microsoft Windows-compliant control sequences automatically. Set the **binaryFlag** argument to *false* if the output file is intended for storing human-readable data that can be processed with an arbitrary text editor. Set the **binaryFlag** argument to *true* for creating an output file that can be used for storing arbitrary data, including binary-encoded figures, control characters and other.

Upon successful completion, the `createOutputFile()` service will store the address of a file descriptor representing the newly created file in a variable provided by the calling code. The **fileHandle** argument contains the address of this variable.

You may pass the file descriptor to standard routines for writing to the new file; for C++, common routines include but are not limited to:

- `fwprintf()` for writing wide-character encoded text. This routine is also useful for writing textual representations of binary data to the file.
- `fputc()` for writing an arbitrary character or data byte to the file.
- `fputwc()` for writing a wide-character encoded character to the file.

The `createOutputFile()` service will return *true* if the file has been created and opened successfully. If the **exceptionsFlag** argument is set to *false*, the service will return *false* if the file creation procedure fails for some reason, however, the calling module will receive no indication regarding the source of the error condition. If the **exceptionsFlag** argument is set to *true* (which is the default), the `createOutputFile()` service will raise an exception of class `IOException` in case of an error condition; you may provide a standard C++ exception handler (`catch` block) for catching and processing the exception object.

I/O exception codes (stored in an exception object of type `IOException`) that can occur are:

- `ioerr_FileAlreadyExists` — Raised if the specified file already exists and the **overwriteFlag** argument has been set to *false*.
- `ioerr_NoFileWritePermission` — Raised if the specified file already exists and the **overwriteFlag** argument has been set to *true* but the file access permissions prevent the service from overwriting the file contents.
- `ioerr_NoDirWritePermission` — Raised if the access permissions of the target directory prevent the service from creating a file.
- `ioerr_NoSuchDirectory` — Raised if the specified file path is invalid, i.e. includes a directory that does not exist.
- `ioerr_UnableToCreate` — Raised if the file creation operation has failed for some other reason.

Use the `closeOutputFile()` service for closing a file that has been created and opened by this routine.

Return Values

True if the file has been created and opened for write access successfully, *false* otherwise.

Associated Routines

```
closeOutputFile()
createDirectory()
```

directoryExists

Determines whether a specific directory already exists.

Syntax *status* := directoryExists (*dirSpec*)

Argument	**Data Type**	**Access**
dirSpec	Wide-character string	Read

C++ Binding

```
#include "Vernissage.h"
bool Session::directoryExists(std::wstring dirSpec);
```

Arguments

dirSpec
An absolute or relative path specification determining the path to and the name of the directory.

Description

This routine checks whether the specified directory exists, i.e. is known at the file system-level.

The **dirSpec** argument determines the path and name of the directory to be checked and can contain a Microsoft Windows absolute (e.g. "C:\Temp\ThisDir") or relative (e.g. "..\..\ThisDir") path specification.

Return Values

Returns *true* if the specified directory exists, *false* otherwise.

Associated Routines

`createDirectory()`

eraseResultSets

Erases all result sets currently loaded from the internal database.

Syntax	eraseResultSets ()

C++ Binding	`#include "Vernissage.h"`
	`void Session::eraseResultSets();`

Arguments	—
Description	This routine erases the entire internal result set database of the Vernissage software and thus reverses any previous call to the `loadResultSet()` and `loadAllResultSets()` functions. When this routine returns, all Bricklet descriptions have bee removed. This routine is only useful for special third party applications managing result sets in their own context.
Return Values	—

Associated Routines	`loadResultSet()`
	`loadAllResultSets()`
	`loadBrickletContents()`
	`unloadBrickletContents()`

getAxisClocks

Returns the number of clocks configured for the trigger axis of a Bricklet.

Syntax *count* := getAxisClocks (*bricklet, axisName*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read
axisName	Wide-character string	Read

C++ Binding

```
#include "Vernissage.h"
int Session::getAxisClocks(void *pBricklet,
                                 std::wstring axisName);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

axisName
The name of the axis to be queried; both plain and qualified axis names are supported.

Description

This routine returns the number of clocks configured for the specified axis associated with the data channel through which a particular Bricklet was acquired.

If the specified axis is the trigger axis of the channel through which the Bricklet passed has been acquired, the number of clocks returned would e.g. be identical to the number of data points on a curve (one-dimensional Bricklets), or the number of data acquisition points on a scan line (for Bricklets generated during a spatial scan operation).

Note, however, that the number of clocks returned is twice the number originally entered by the user if the specified axis is *mirrored*; you can use the `getAxisDescriptor()` service routine to verify whether an axis has been assigned the *mirrored* characteristic.

If the axis name passed to this function does not exist or is not part of the axis hierarchy associated with the specified Bricklet, the returned value is unpredictable.

Return Values

Returns the number of clocks configured for the specified axis.

Associated Routines

```
getAxisUnit()
getTriggerAxisName()
getTriggerAxisQualifiedName()
getRootAxisName()
getRootAxisQualifiedName()
getAxisDescriptor()
getAxisTableSets()
```

getAxisDescriptor

Returns the configuration description of an axis.

Syntax

axisConfiguration := getAxisDescriptor (*bricklet, axisName*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read
axisName	Wide-character string	Read

C++ Binding

```
#include "Vernissage.h"
struct AxisDescriptor {
  int rawStart;
  int rawIncrement;
  double physicalStart;
  double physicalIncrement;
  std::wstring physicalUnit;
  bool mirrored;
  int clocks;
  std::wstring triggerAxisName;
};
AxisDescriptor getAxisDescriptor(void *pBricklet,
                                 std::wstring axisName);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

axisName
The name of the axis to be queried; both plain and qualified axis names are supported.

Description

This routine returns the configuration of the specified axis, including its raw and physical start value and increment, the physical unit, the number of clock positions and the name of the triggering axis (if any).

Together with the `getAxisTableSets()` service, this routine can be used to determine all details on the configuration of a particular axis.

If the axis name passed to `getAxisDescriptor()` does not exist or is not part of the axis hierarchy associated with the specified Bricklet, the results of calling this routine are unpredictable.

Return Values

Returns information on the configuration of the specified axis. The data structure returned consists of the following entries:

- *rawStart* — The start value of the axis, expressed as raw device value.
- *rawIncrement* — The increment between two clock positions on the axis, expressed as raw device value.
- *physicalStart* — The start value of the axis, expressed as physical value.
- *physicalIncrement* — The increment between two clock positions, expressed as physical value.
- *physicalUnit* — The name of the unit used by *physicalStart* and *physicalIncrement*. See section **Data Types and Formats** for a list of supported unit names.
- *mirrored* — Flag indicating whether the axis has been assigned the *mirrored* characteristic. In this case, *clocks* specifies twice the number of clock positions the user has originally configured; the end value of the axis is thus *physicalStart* + (*clocks / 2 – 1) • physicalIncrement.*
- *clocks* — The number of clock positions on the axis.

If *mirrored* is *false*, the end value of the axis can be computed as follows: *physicalStart* + (*clocks – 1*) • *physicalIncrement.*

- *triggerAxisName* — The name of the triggering axis. If the axis passed to `getAxisDescriptor()` is a root axis, this field will contain an empty character string.

Associated Routines

```
getAxisClocks()
getAxisUnit()
getTriggerAxisName()
getTriggerAxisQualifiedName()
getRootAxisName()
getRootAxisQualifiedName()
getAxisTableSets()
```

getAxisParameter

Returns information on the value of a particular axis parameter.

Syntax

valueDescriptor := getAxisParameter (*bricklet, axisName, parameterName*)

Argument	Data Type	Access
bricklet	Opaque refererence	Read
axisName	Wide-character string	Read
parameterName	Wide-character string	Read

C++ Binding

```
#include "Vernissage.h"
struct Parameter {
  enum ValueType {
    vt_Special, vt_Integer, vt_Double,
    vt_Boolean, vt_Enum, vt_String
    } valueType;
  std::wstring unit;
  std::wstring value;
};
Parameter getAxisParameter(void *pBricklet,
                           std::wstring axisName,
                           std::wstring parameterName);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

axisName
The name of the axis to be queried; both plain and qualified axis names are supported.

parameterName
The name of the axis parameter of which information shall be returned.

Description

This routine can be used for retrieving the value of an axis parameter associated with the axis specified by the plain or qualified name passed. The information returned will refer to the parameter value at the time a particular Bricklet (passed as first argument) was created.

The `getAxisParameter()` service routine will create a value descriptor including the actual parameter value represented as a character string, a type code identifying the value type and a unit name.

If the axis name passed refers to a non-existing axis, or an axis that is not associated with the Bricklet passed as first argument, or if the specified parameter does not exist or is not associated with the specified axis, the contents of the value descriptor returned by this function are unpredictable.

See section **Axis Parameters** on page 42 for more information on axis parameters.

Return Values

Returns a data structure consisting of the following entries:

- *valueType* — An enumeration type code denoting the type of the value. The character string stored in field *value* represents a data item of this type.
- *unit* — The name of the SI unit of the parameter value. See section **Data Types and Formats** on page 37 for more information on supported units.
- *value* — The parameter value, represented as character string.

Associated Routines

`getAxisParameters()`

getAxisParameters

Returns information on all axis parameter values of an axis at Bricklet creation time.

Syntax

valueDescriptors := getAxisParameters (*bricklet*, *axisName*)

Argument	Data Type	Access
bricklet	Opaque refererence	Read
axisName	Wide-character string	Read

C++ Binding

```
#include "Vernissage.h"
struct Parameter {
  enum ValueType {
    vt_Special, vt_Integer, vt_Double,
    vt_Boolean, vt_Enum, vt_String
    } valueType;
  std::wstring unit;
  std::wstring value;
};
std::map<std::wstring, Parameter> getAxisParameters(void
*pBricklet,

std::wstring axisName
                                                    );
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

axisName
The name of the axis to be queried; both plain and qualified axis names are supported.

Description

This routine can be used for retrieving information on all axis parameters associated with the axis specified by the plain or qualified name passed. The information returned will refer to the parameter values at the time a particular Bricklet (passed as first argument) was created.

The `getAxisParameters()` service routine will create a collection of value descriptors including the actual parameter value represented as a character string, a type code identifying the value type and a unit name.

If the axis name passed refers to an axis that has no associated axis parameters, an empty collection will be returned.

If the axis name passed refers to a non-existing axis, or an axis that is not associated with the Bricklet passed as first argument, the contents of the value descriptor returned by this function are unpredictable.

See section **Axis Parameters** on page 42 for more information on axis parameters.

Return Values

Returns a collection of value descriptors representing the axis parameter values of an axis at Bricklet creation time. The collection uses the parameter name as key for the respective value descriptor. The value descriptor itself consists of the following entries:

- *valueType* — An enumeration type code denoting the type of the value. The character string stored in field *value* represents a data item of this type.
- *unit* — The name of the SI unit of the parameter value. See section **Data Types and Formats** on page 37 for more information on supported units.
- *value* — The parameter value, represented as character string.

Associated Routines

`getAxisParameter()`

getAxisTableSets

Returns a list of all table sets associated with an axis.

Syntax *tableSets* := getAxisTableSets (*bricklet, axisName*)

Argument	Data Type	Access
bricklet	Opaque reference	Read
axisName	Wide-character string	Read

C++ Binding

```
#include "Vernissage.h"
struct IntervalDescriptor {
  int start, stop, step;
};
typedef std::vector<IntervalDescriptor> TableSet;
typedef std::map<std::wstring, TableSet> AxisTableSets;
AxisTableSets getAxisTableSets(void *pBricklet,
                               std::wstring axisName);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

axisName
The name of the axis to be queried; both plain and qualified axis names are supported.

Description

This routine returns all table sets being associated with the specified axis; the axis must be part of the axis hierarchy that is associated with the data channel through which the specified Bricklet has been acquired.

`getAxisTableSets()` will check the entire axis hierarchy for axes that directly or indirectly trigger the specified axis and return all relevant table sets.

Each table set consists of a list of interval description structures (storing the start and end values of the interval and the increment), `getAxisTableSets()` will return a separate interval description list for each trigger axis.

Return Values

Returns a collection of table sets, ordered by trigger axis. If the specified axis is the root axis, an empty collection will be returned. If no trigger axis has defined a table set, an empty collection is returned also.

Associated Routines

```
getAxisClocks()
getAxisUnit()
getTriggerAxisName()
getTriggerAxisQualifiedName()
getRootAxisName()
getRootAxisQualifiedName()
getAxisDescriptor()
```

getAxisUnit

Returns the name of the physical unit associated with an axis.

Syntax

unit := getAxisUnit (*bricklet, axisName*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read
axisName	Wide-character string	Read

C++ Binding

```
#include "Vernissage.h"
std::wstring Session::getAxisUnit(void *pBricklet,
                                  std::wstring axisName);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

axisName
The name of the axis to be queried; both plain and qualified axis names are supported.

Description

This routine returns the name of the SI unit associated with an axis. The specified axis must be part of the axis hierarchy that is associated with the data channel through which the specified Bricklet has been acquired.

For example, the axis unit of the *Y*- and *X*-axis being associated with channels delivering topography (and similar) data use "Meter" as physical unit, while an SPM spectroscopy axis *V* will use "Volt".

Return Values

Returns a character string representing the unit name of an axis associated with a particular data channel. The channel is determined by the specified Bricklet.

See section **Data Types and Formats** for a list of supported unit names.

Associated Routines

```
getAxisClocks()
getTriggerAxisName()
getTriggerAxisQualifiedName()
getRootAxisName()
getRootAxisQualifiedName()
getAxisDescriptor()
getAxisTableSets()
```

getBrickletCount

Returns the total number of Bricklets currently loaded.

Syntax	*count* := getBrickletCount ()

C++ Binding	`#include "Vernissage.h"`
	`int Session::getBrickletCount();`

Arguments	—
Description	This routine returns the total number of Bricklets currently loaded, i.e. the number of Bricklets encountered in all result files read by Vernissage during a particular session.
Return Values	Returns the total number Bricklets loaded. If no result file has yet been read, this routine will return 0.
Associated Routines	`getNextBricklet()`

getBrickletDataItemCount

Returns the number of raw data items a Bricklet contains.

Syntax *count* := getBrickletDataItemCount (*bricklet*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
int Session::getBrickletDataItemCount(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine returns the actual number of raw values the specified Bricklet stores.

The number returned will be identical to the item count information returned by means of the **count** argument in a call to `loadBrickletContents()`.

Return Values

Returns an integer figure representing the number of raw values.

Associated Routines

`loadBrickletContents()`

getCalibrationInformation

Retrieves the calibration options for the instrument(s)

Syntax

calibrationInfo := getCalibrationInformation(bricklet [, instrumentName])

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read
instrumentName	Wide-character string	Read

C++ Binding

```
#include "Vernissage.h"
struct CalibrationInfo
{
  std::wstring dataSet;
  std::wstring dataSetVariant;
  std::wstring parameterSet;
  std::wstring parameterSetVariant;
};

std::map<std::wstring, Session::CalibrationInfo>
                        getCalibrationInformation (
                           void *pBricklet,
                           std::wstring instrumentName = L"");
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

instrumentName
The (optional) name of the instrument to be queried.

Description

This routine can be used for retrieving the calibration options for the instrument – or in case of TwoProbe or NanoProbe the calibration options for all instruments that were active when the particular Bricklet was created.

By specifying a valid instrument name (such as “Tip1”, “Tip2”, “Tip3”, “Tip4”,) as argument instrumentName, the service routine will be directed to return only the calibration for the given instrument.

The instrument name is part of the qualified axis name, e.g. You can find out, which instrument recorded the bricklet in hand by a call of either getTriggerAxisQualifiedName(bricklet) or getRootAxisQualifiedName(bricklet).
Note: You only get the name of the selected Data Set, Parameter Set, and Parameter Set Variant. The Data Set Variant is currently not used

Return Values

Returns the calibration options for the given instrument or for all instruments.

The calibration options for one instruments consist of the following entries:

- dataSet – name of data set
- dataSetVariant – (reserved for future usage)
- parameterSet – name of the parameter set
- parameterSetVariant – name of the parameter set variant

Associated Routines

getChannelGroupName

Returns the group name of the data channel through which the data of a Bricklet was acquired.

Syntax

name := getChannelGroupName (*bricklet*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
std::wstring Session::getChannelGroupName(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine returns the group name of the data channel through which the data of a particular Bricklet has been acquired.

In case of virtual channels (in ESpec) the channel name is constructed as:
`<ChannelGroupNameBase>[.<VirtExt>]`

Return Values

Returns a character string representing group name of the data channel through which the data contained by the specified Bricklet was acquired.

Associated Routines

```
getChannelInstanceName()
getChannelName()
getChannelUnit()
```

getChannelInstanceName

Returns the Experiment Element instance name of the data channel from which a Bricklet originated.

Syntax *name* := getChannelInstanceName (*bricklet*)

Argument	Data Type	Access
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
std::wstring Session::getChannelInstanceName(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine returns the instance name of the Experiment Element a Bricklet originated at. Currently, this will either be the instance name of an element of type *Channel* or of type *Detector*.

Return Values

Returns a character string representing the Experiment Element instance name of the data channel at which the specified Bricklet originated.

Associated Routines

```
getChannelName()
getChannelUnit()
```

getChannelName

Returns the logical name of the data channel through which the data of a Bricklet was acquired.

Syntax

name := getChannelName (*bricklet*)

Argument	Data Type	Access
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
std::wstring Session::getChannelName(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine returns the logical name of the data channel through which the data for a particular Bricklet has been acquired.

The logical name of a channel is often *not* identical to the instance name of its Experiment Element, for example, the logical name of a channel may be "*I(V)* " while the respective Experiment Element instance name is "*I_V* ".

The logical name of a channel is specified by the deployment parameter `Name` of its Experiment Element instance.

Return Values

Returns a character string representing the logical name of the data channel through which the data contained by the specified Bricklet was acquired.

Associated Routines

```
getChannelInstanceName()
getChannelUnit()
```

getChannelRawMax

Returns the maximum raw value supported by the data channel through which the specified Bricklet was acquired.

Syntax

rawMaximum := getChannelRawMax (*bricklet*)

Argument	Data Type	Access
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
int Session::getChannelRawMax(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine returns the maximum raw value supported by a data channel; the channel is identified by a Bricklet that it has acquired.

Return Values

Returns the minimum raw value supported by the data channel associated with the specified Bricklet. Usually, this will be +2,147,483,647 ($-2^{31} - 1$), as most MATRIX channels are 32-bit wide.

Associated Routines

```
getChannelRawMin()
getRawMin(),
getRawMax()
toPhysical()
toRaw()
```

getChannelRawMin

Returns the minimum raw value supported by the data channel through which the specified Bricklet was acquired.

Syntax

rawMinimum := getChannelRawMin (*bricklet*)

Argument	Data Type	Access
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
int Session::getChannelRawMin(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine returns the minimum raw value supported by a data channel; the channel is identified by a Bricklet that it has acquired.

Return Values

Returns the minimum raw value supported by the data channel associated with the specified Bricklet. Usually, this will be –2,147,483,648 (-2^{31}), as most MATRIX channels are 32-bit wide.

Associated Routines

```
getChannelRawMax()
getRawMin(),
getRawMax()
toPhysical()
toRaw()
```

getChannelUnit

Returns the unit name of the data acquired through a particular data channel.

Syntax *unit* := getChannelUnit (*bricklet*)

Argument	Data Type	Access
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
std::wstring Session::getChannelUnit(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine returns the name of the SI unit associated with the data contained by a Bricklet. (Because the data of a Bricklet has been acquired through a specific channel, the unit name returned is also a characteristic of the respective data channel.)

As Bricklets always store raw data (i.e. data that has not been transformed from its hardware representation into physical quantities), the unit name returned by a call to `getChannelUnit()` actually specifies the physical unit of the raw data after transformation.

Return Values

Returns a character string representing the unit name of the data acquired through a particular data channel. The channel is determined by the specified Bricklet.

See section **Data Types and Formats** for a list of supported unit names.

Associated Routines

```
getChannelInstanceName()
getChannelName()
```

getCreationComment

Returns the user's creation comment associated with a particular Bricklet.

Syntax

comment := getCreationComment (*bricklet*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
std::wstring Session::getCreationComment(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

The MATRIX result file format supports arbitrary user comments that can be stored at any time. Such comments are useful for providing short notes on the experiment set-up, details regarding the experiment execution, or other information.

The `getCreationComment()` service obtains the *active result set* comment when the specified Bricklet was created, i.e. the most recent comment entered by the user at the time the Bricklet was saved to disk. (Data-specific comments can be inquired by means of the `getDataComments()` service.)

Return Values

Returns a character sequence representing the creation comment specified by the user during experiment execution. If no comment was entered, a dash character ('-') will be returned.

Associated Routines

```
getCreationTimestamp()
getSampleName()
getDataSetName()
getDataComments()
```

getCreationTimestamp

Returns the date and time at which a specific Bricklet was acquired.

Syntax *dateTime* := getCreationTimestamp (*bricklet*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
struct tm Session::getCreationTimestamp(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

The `getCreationTimestamp()` service returns the date and time when the specified Bricklet was created, i.e. when the contents of the Bricklet were saved to disk. In other words, the service will return the date and time when the acquisition cycle that has produced the Bricklet was finished (*not* when it was started.)

Return Values

Returns the date and time of the creation of the Bricklet specified as argument, i.e. the date and time when the Bricklet data were being written to disk.

In case of the C++ binding, the date and time is returned by means of a structure of type `tm` as specified by the standard C header file `<ctime>` (or `<time.h>`, respectively.)

The date/time conversion run by the service is locale-aware, i.e. the date and time returned will reflect the Bricklet creation date/time with respect to the time zone, daylight saving time settings and similar information of the computer the Vernissage software is executing on.

Associated Routines `getCreationComment()`

getDataComments

Returns the Bricklet-specific comments entered by the MATRIX user.

Syntax

dataComments := getDataComments (*bricklet*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
std::vector<std::wstring> getDataComments(
                          void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine returns a list of character strings representing the *data comments* the user entered for a specific Bricklet during a MATRIX session. Data comments can be registered by means of the graphical user interface of MATRIX by right-clicking a data display and selecting "Enter comment…" from the context menu.

In contrast to the *creation comment* that applies to all Bricklets of an experiment run (until a new creation comment is entered), a data comment is attached to a particular Bricklet only. In addition, an arbitrary number of data comments can be associated with a single Bricklet.

Return Values

Returns a list of character strings, each string represents a data comment entered by the MATRIX user. If no data comment was entered, an empty list will be returned.

Associated Routines

```
getCreationComment()
getSampleName()
getDataSetName()
```

getDataSetName

Returns the data set name associated with a particular Bricklet.

Syntax *name* := getDataSetName (*bricklet*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
std::wstring getDataSetName(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine returns the data set name that was in effect when the Bricklet passed was created. (The data set name is an arbitrary character string that can help users identify a number of result data sets as related.)

Return Values

Returns the data set name associated with the specified Bricklet. If the MATRIX user specified no data set name, a single dash character ('–') will be returned.

Associated Routines

```
getCreationComment()
getDataComments()
getSampleName()
```

getDependingBricklets

Returns a list of Bricklets that have a "depends-on" relationship to a specific Bricklet.

Syntax

bricklets := getDependingBricklets *(bricklet, filterSet)*

Argument	Data Type	Access
bricklet	Opaque reference	Read
filterSet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
std::vector<void *> Session::getDependingBricklets(
  void *pBricklet, void *pFilterSet = 0);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

filterSet
The filter set passed to the `run()` method of the plug-in, or a null pointer. If a null pointer is passed, the list returned will contain all related Bricklets. Passing a filter set will cause the function to return only related Bricklets that are part of the filter set, i.e. that are subject to an export operation.

Description

This routine returns a list of Bricklets that have a "depends-on" relationship to the Bricklet passed as argument *bricklet*. An example for a "depends-on" relationship is a single point spectroscopy (SPS) curve depending on an SPM image because any SPS curve is associated with a particular sample location.

Please note that the `getDependingBricklets()` function can either return all Bricklets related to the specified Bricklet (if a null pointer is passed as argument *filterSet*), or only Bricklets that are subject to an export operation (if a pointer to a filter set data structure is passed as argument *filterSet*).

Please see section **Related Bricklets** for more information on Bricklet relationships.

Return Values

Returns a list of opaque pointers to Bricklets that have a "depends-on" relationship to the Bricklet passed to the routine. If the specified Bricklet has no dependant Bricklets, the returned list will be empty.

Associated Routines

```
getReferencedBricklets()
getSucessorBricklet()
getPredecessorBricklet()
```

getDimensionCount

Returns the dimensionality of the data contained by a Bricklet.

Syntax

count := getDimensionCount (*bricklet*)

Argument	Data Type	Access
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
int Session::getDimensionCount(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine returns the dimensionality of the data contained by a Bricklet.

MATRIX supports experiment set-ups generating arbitrary (i.e. *n*-dimensional) data spaces, hence it is important to know the dimensionality of the data a particular Bricklet contains in order to process it.

Return Values

Returns an unsigned integer figure indicating the dimensionality of the data contained by the specified Bricklet: "1" indicates one-dimensional data, i.e. a curve. "2" means that the respective Bricklet contains two-dimensional data, such as topography images, tunnelling current images, or similar. A return value of "3" indicates three-dimensional data such as a volume CITS cube resulting from a grid spectroscopy experiment; data from imaging XPS experiments is actually four-dimensional, hence the return value will be "4" for such Bricklets.

Associated Routines

```
getViewTypes()
getType()
```

getExperimentElementDeploymentParameter

Returns the value of a deployment parameter of an Experiment Element instance.

Syntax

value := getExperimentElementDeploymentParameter (*bricklet*, *instanceName*, *parameterName*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read
instanceName	Wide-character string	Read
parameterName	Wide-character string	Read

C++ Binding

```
#include "Vernissage.h"
std::wstring Session::getExperimentElementDeploymentParameter(
                                   void *pBricklet,
                                   std::wstring instanceName,
                                   std::wstring parameterName);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

instanceName
The name of an Experiment Element instance to be queried.

parameterName
The name of a deployment parameter supported by the respective Experiment Element instance.

Description

This routine can be used for retrieving the value of a deployment parameter used for deploying a specific Experiment Element instance. The Experiment Element instance must be a member of the experiment that has produced a particular Bricklet; a reference to this Bricklet must be passed to the routine as first argument.

Note that an Experiment Element deployment parameter is "typeless", i.e. its value will always be a character string.

Return Values

Returns the value of the specified deployment parameter of the Experiment Element instance with the instance name passed as second argument.

If the instance name does not refer to an existing Experiment Element instance, or the specified deployment parameter name is not known for the element instance, an empty string will be returned.

Associated Routines

```
getExperimentElementParameter()
getExperimentElementParameters()
getExperimentElementInstanceNames()
getExperimentElementDeploymentParameters()
```

getExperimentElementDeploymentParameters

Returns a list of deployment parameters and their respective values of an Experiment Element instance.

Syntax

parameters := getExperimentElementDeploymentParameters (*bricklet*, *instanceName*)

Argument	Data Type	Access
bricklet	Opaque refererence	Read
instanceName	Wide-character string	Read

C++ Binding

```
#include "Vernissage.h"
std::map<std::wstring, std::wstring>
  getExperimentElementDeploymentParameters(void *pBricklet,
                                    std::wstring instanceName);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

instanceName
The name of the Experiment Element instance to be queried.

Description

This routine can be used for retrieving all parameters and their associated values used for deploying a specific Experiment Element instance. The Experiment Element instance must be a member of the experiment that has produced a particular Bricklet; a reference to this Bricklet must be passed to the routine as first argument.

Note that all Experiment Element deployment parameters are "typeless", i.e. their values will always be character strings.

Return Values

Returns a collection of character strings representing the deployment parameters and their associated values. The collection uses the name of the deployment parameter as a key for the respective parameter value.

Associated Routines

```
getExperimentElementDeploymentParameter()
getExperimentElementParameter()
getExperimentElementParameters()
getExperimentElementInstanceNames()
```

getExperimentElementInstanceNames

Returns a list of Experiment Element instance names in use when a particular Bricklet was created.

Syntax

names := getExperimentElementInstanceNames (*bricklet, typeName [, catalogName]*)

Argument	Data Type	Access
bricklet	Opaque reference	Read
typeName	Wide-character string	Read

C++ Binding

```
#include "Vernissage.h"
std::vector<std::wstring> getExperimentElementInstanceNames(
                              void *pBricklet,
                              std::wstring typeName,
                              std::wstring catalogName = L"");
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

typeName
The name of an Experiment Element type.

catalogName
The (optional) name of the Experiment Element catalogue to be used.

Description

This routine determines all Experiment Element instances utilised by the experiment during which the specified Bricklet was created and returns their instance names.

By specifying a valid Experiment Element type name (such as *XYScanner* or *Channel*) as argument **typeName**, the service routine will be directed to return only the names of Experiment Element instances that match the given type.

In addition, you may use the optional third parameter **catalogName** to specify the name of an existing Experiment Element catalogue. If a catalogue name has been specified, the routine will only return the instance names of Experiment Elements from the respective catalogue.

See section **Understanding MATRIX Experiments** for more information on Experiment Elements.

Return Values

Returns a collection of character sequences describing the names of Experiment Element instances matching the query. If **typeName** is an empty string and **catalogName** has been omitted, the names of all Experiment Element instances utilised by an experiment will be returned.

If **typeName** specifies an unsupported Experiment Element type, or the name of an Experiment Element type not used by the respective experiment, an empty collection will be returned. Similarly, if **catalogName** specifies an unsupported Experiment Element catalogue, or the name of an Experiment Element catalogue not used by the respective experiment, an empty collection will result.

Associated Routines

```
getExperimentElementDeploymentParameter()
getExperimentElementDeploymentParameters()
```

getExperimentElementParameter

Returns information on the value of an Experiment Element instance parameter at Bricklet creation time.

Syntax

valueDescriptor := getExperimentElementParameter (*bricklet*, *instanceName*, *parameterName*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read
instanceName	Wide-character string	Read
parameterName	Wide-character string	Read

C++ Binding

```
#include "Vernissage.h"
struct Parameter {
  enum ValueType {
    vt_Special, vt_Integer, vt_Double,
    vt_Boolean, vt_Enum, vt_String
  } valueType;
  std::wstring unit;
  std::wstring value;
};
Parameter Session::getExperimentElementParameter(
                                   void *pBricklet,
                                   std::wstring instanceName,
                                   std::wstring parameterName);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

instanceName
The name of an Experiment Element instance to be queried.

parameterName
The name of an instance parameter supported by the respective Experiment Element type.

Description

This routine can be used for retrieving information about a parameter of an Experiment Element instance. The routine will return information about the value of that parameter at the time a particular Bricklet (passed as first argument) was created.

The `getExperimentElementParameter()` service will create a value descriptor including the actual parameter value represented as a character string, a type code identifying the value type and a unit name.

If the instance name passed refers to a non-existing Experiment Element instance, or the parameter name refers to a parameter not supported by the respective Experiment Element type, the results of calling this routine are unpredictable.

Return Values

Returns a data structure consisting of the following entries:

- *valueType* — An enumeration type code denoting the type of the value. The character string stored in field *value* represents a data item of this type.
- *unit* — The name of the SI unit of the parameter value. See section **Data Types and Formats** for more information on supported units.
- *value* — The parameter value, represented as character string. (For example "1.2e–9", "true", etc.)

Associated Routines

`getExperimentElementDeploymentParameter()`
`getExperimentElementParameters()`

getExperimentElementParameters

Returns information on all parameter values of an Experiment Element instance at Bricklet creation time.

Syntax

valueDescriptors := getExperimentElementParameters (*bricklet*, *instanceName*)

Argument	Data Type	Access
bricklet	Opaque reference	Read
instanceName	Wide-character string	Read

C++ Binding

```
#include "Vernissage.h"
struct Parameter {
  enum ValueType {
    vt_Special, vt_Integer, vt_Double,
    vt_Boolean, vt_Enum, vt_String
  } valueType;
  std::wstring unit;
  std::wstring value;
};
std::map<std::wstring, Parameter>
        Session::getExperimentElementParameters(
                                   void *pBricklet,
                                   std::wstring instanceName,
                                               );
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

instanceName
The name of an Experiment Element instance to be queried.

Description

This routine can be used for retrieving information on all parameters of an Experiment Element instance. The routine will return information about the parameter values at the time a particular Bricklet (passed as first argument) was created.

The `getExperimentElementParameters()` service will create a collection of value descriptors including the actual parameter value represented as a character string, a type code identifying the value type and a unit name.

If the instance name passed refers to a non-existing Experiment Element instance, `getExperimentElementParameters()` will return an empty collection.

Return Values

Returns a collection of value descriptors representing the parameter values of an Experiment Element instance at Bricklet creation time. The collection uses the parameter name as key for the respective value descriptor. The value descriptor itself consists of the following entries:

- *valueType* — An enumeration type code denoting the type of the value. The character string stored in field *value* represents a data item of this type.
- *unit* — The name of the SI unit of the parameter value. See section **Data Types and Formats** for more information on supported units.
- *value* — The parameter value, represented as character string. (For example "1.2e–9", "true", etc.)

Associated Routines

`getExperimentElementDeploymentParameter()`
`getExperimentElementParameter()`

getExperimentInfo

Returns information about the experiment that has produced a particular Bricklet.

Syntax *experimentInfo* := getExperimentInfo (*bricklet*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
struct ExperimentInfo {
  std::wstring experimentName;
  std::wstring experimentVersion;
  std::wstring experimentDescription;
  std::wstring experimentFileSpec;
  std::wstring projectName;
  std::wstring projectVersion;
  std::wstring projectFileSpec;
};
ExperimentInfo Session::getExperimentInfo(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine returns information about the experiment that has produced a particular Bricklet, including the experiment name, its version identifier and the file specification of the experiment's description file.

Because MATRIX organises experiments into *Projects*, the name of the Project containing the respective experiment, the Project's version identifier and the file specification of the Project description file are also returned.

Return Values

Returns an experiment information data structure comprising the following entries:

- *experimentName* — The original name of the experiment (for example "STM_Spectroscopy").
- *experimentVersion* — The experiment version identifier.
- *experimentDescription* — A short text describing the experiment, as provided by the experiment author.
- *experimentFileSpec* — The full file specification of the original experiment description file.
- *projectName* — The name of the MATRIX Project containing the experiment.
- *projectVersion* — The Project's version identifier.
- *projectFileSpec* — The full file specification of the Project description file.

Associated Routines `getMetaData()`

getMessages

Returns the contents of the global message buffer.

Syntax *messages* := getMessages ()

C++ Binding

```
#include "Vernissage.h"
std::vector<std::wstring> Session::getMessages() const;
```

Arguments —

Description Vernissage provides a global message buffer that can be used by plug-in modules to issue arbitrary messages to the user. The message buffer mechanism is most useful for issuing error messages if a plug-in detects some problem that prevents it from completing its operations successfully.

Calling `getMessages()` will return the current contents of the global message buffer. The contents of the message buffer will not be modified.

This routine is only useful for special third party applications that need to display messages issued by plug-ins.

Return Values Returns a collection of messages that have been stored in the global message buffer. If the buffer is currently empty, an empty collection is returned.

Associated Routines

```
addMessage()
clearMessages()
```

getMetaData

Returns the creation meta data of a Bricklet.

Syntax

brickletMetaData := getMetaData (*bricklet*)

Argument	Data Type	Access
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
struct BrickletMetaData {
  std::wstring fileCreatorName;
  std::wstring fileCreatorVersion;
  std::wstring userName;
  std::wstring accountName;
};
BrickletMetaData Session::getMetaData(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

The `getMetaData()` service returns the "creation meta data" associated with a particular Bricklet. (More precisely, the creation meta data of the result file chain a Bricklet is associated with.)

The creation meta data comprises name and version identifier of the software product that has originally created the result file (and hence also the Bricklet) as well as information about the user accounts being utilised for running the experiment that has produced the Bricklet.

Return Values

Returns a data structure containing the creation meta data associated with the specified Bricklet. This structure consists of the following entries:

- *fileCreatorName* — The name of the software product that has created the result file associated with the Bricklet.
- *fileCreatorVersion* — The version identifier of the result file creator.
- *userName* — The (MATRIX) user name of the user who has run the experiment that produced the Bricklet. (Usually, this name will be "default".)
- *accountName* — The Microsoft Windows account name of the user who has run the experiment that produced the Bricklet.

Associated Routines

`getExperimentInfo()`

getNextBricklet

Returns the next Bricklet from the set of Bricklets currently loaded. Each call will retrieve a single Bricklet.

Syntax

bricklet := getNextBricklet (*context, filterSet*)

Argument	**Data Type**	**Access**
context	Opaque reference	Read/Modify
filterSet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
void* Session::getNextBricklet(void **pContext,
void *pFilterSet = 0);
```

Arguments

context
An opaque pointer variable into which the routine stores a context value for use by future calls to `getNextBricklet()` or `releaseBrickletContext()`. The **context** argument is the address of a pointer variable containing the address of the context. This pointer variable must be set to zero before the first call to `getNextBricklet()`. You can use the same **context** argument from one `getNextBricklet()` call to another provided you have not called `releaseBrickletContext()` for that **context** first. `getNextBricklet()` uses this argument to retain the context between subsequent calls.

You must not change the value of **context** in subsequent calls to `getNextBricklet()`.

filterSet
An opaque pointer to the filter set data structure passed to the `run()` method of the plug-in, or a null pointer. Provide a pointer to the filter set data structure for restricting the iteration process to the Bricklets that have been selected for export by the user. Specify a null pointer for iterating through all Bricklets currently loaded.

Description

This routine allows iterating through the set of Bricklets currently loaded. Upon each call, a pointer to an opaque Bricklet descriptor structure is returned; this pointer can then be passed to other service routines in order to access the Bricklet contents or to inquire information about the Bricklet.

The service will return the Bricklets by result set, i.e. if several result sets have been loaded, the Bricklets from the first set will be passed before the Bricklets from the second set. The order in which the service processes the result sets is unpredictable.

Bricklets from a particular result set will always be returned in the order they were created, i.e. in chronological order with the oldest Bricklet returned first.

When the last Bricklet from the set has been returned, a subsequent call to `getNextBricklet()` will return a null pointer. A further call with the same **context** argument will restart the iteration sequence.

Use `releaseBrickletContext()` for terminating an iteration sequence and for resetting the **context** parameter.

Return Values

Returns an opaque pointer to a Bricklet descriptor structure. This pointer cannot be used for referencing information directly, however, it can be passed to various information query services.

This routine returns a null pointer if the last Bricklet has already been returned by the preceding call to `getNextBricklet()`.

Associated Routines

```
releaseBrickletContext()
getBrickletCount()
```

getParentResultFileSpec

Returns the file specification of the parent result file of a Bricklet.

Syntax *fileSpec* := getParentResultFileSpec (*bricklet*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
std::wstring Session::getParentResultFileSpec(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine returns the file specification of the result file being associated with a particular Bricklet. As each Bricklet is actually referenced by a result file, there is always a *parent* result file.

The service routine will return the file specification of the first result file of a particular result file chain (the first result file of a chain has a name ending on "_0001.mtrx"), even if the specified Bricklet is referenced from another result file of the same result file chain.

Return Values

Returns the full file specification of the result file associated with the specified Bricklet, including drive identification letter and directory hierarchy.

Associated Routines `getResultDataFileSpec()`

getPlainAxisName

Returns the "plain" part of an axis name.

Syntax

name := getPlainAxisName (*qualifiedAxisName*)

Argument	Data Type	Access
qualifiedAxisName	Wide-character string	Read

C++ Binding

```
#include "Vernissage.h"
std::wstring getPlainAxisName (std::wstring qualifiedAxisName);
```

Arguments

qualifiedAxisName
The qualified axis name to be processed.

Description

The `getPlainAxisName()` service returns the "plain" part of a qualified axis name. Internally, the MATRIX software uses qualified names for all axes, as qualified axis names contain additional information about the instrument and the Experiment Element instance an axis is associated with. The `getPlainAxisName()` routine will remove this additional information from the qualified axis name passed; the remaining plain axis name will be returned.

Please see section **Axes and Axis Hierarchies** for additional information on plain axis names and qualified axis names.

Return Values

Returns a character string representing the plain name of the specified axis.

Associated Routines

```
getRootAxisQualifiedName()
getTriggerAxisQualifiedName()
```

getPlugInInfo

Returns information about all plug-in modules loaded.

Syntax

plugInInfoList := getPlugInInfo ()

C++ Binding

```
#include "Vernissage.h"
struct PlugInInfo {
  std::wstring fileSpec;
  std::wstring name;
  std::wstring version;
  std::wstring producer;
  std::wstring type;
  bool isLoaded;
  std::wstring loadMessage;
};
std::vector<PlugInInfo> Session::getPlugInInfo() const;
```

Arguments

—

Description

This routine collects information on all plug-in modules loaded and returns the result to the caller. For each plug-in module, a separate information structure is created.

`getPlugInInfo()` provides information about modules that have been loaded successfully but also about modules that could not be loaded due to some error condition.

Return Values

Returns a collection of information structures; each structure consists of the following entries:

- *fileSpec* — The complete file specification of the plug-in DLL.
- *name* — The name of the plug-in module as returned by the `getIdentity()` plug-in function.
- *version* — The version identifier of the plug-in module as returned by the `getIdentity()` plug-in function.
- *producer* — The name of the plug-in producer as returned by the `getIdentity()` plug-in function.
- *type* — A character string identifying the type of the plug-in module. Currently, the only supported identification string is *Exporter.*
- *isLoaded* — A flag indicating whether the plug-in module has been loaded successfully (*true*). If this flag is *false*, the respective plug-in module was not loaded.
- *loadMessage* — A character string providing the result of the plug-in load operation in human-readable form. If the module has been loaded successfully, *loadMessage* will be an empty string. If the module could not be loaded, *loadMessage* will contain a string describing the cause of the problem.

Associated Routines

—

getPlugInPath

Returns the path specification of the Vernissage plug-in modules directory.

Syntax	*pathSpec* := getPlugInPath ()

C++ Binding	`#include "Vernissage.h"`
	`std::wstring Session::getPlugInPath();`

Arguments	—
Description	This routine returns the path specification of the directory the Vernissage software uses for loading plug-in modules. All plug-ins to be used during a Vernissage session must be located in this directory.
Return Values	Returns the path to the directory from which the Vernissage software loads plug-in modules.
Associated Routines	`getPlugInInfo()`

getPlugModuleCount

Returns the count of all plug-in modules loaded.

Syntax	*plugInCount* := getPlugInModuleCount ()

C++ Binding	`#include "Vernissage.h"`
	`int Session::getPlugInModuleCount () const;`

Arguments	—
Description	This routine returns the count of all modules that have been loaded successfully.
Return Values	Returns the count of all plug-in modules loaded.

Associated Routines	`std::vector<Session::PlugInInfo> getPlugInInfo () const;`
	`std::wstring getPlugInPath () const;`

getPredecessorBricklet

Returns the predecessor of a specific Bricklet.

Syntax

bricklet := getPredecessorBricklet *(bricklet, filterSet)*

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read
filterSet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
void * Session::getPredecessorBricklet(void *pBricklet,
                                       void *pFilterSet = 0);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

filterSet
The filter set passed to the `run()` method of the plug-in, or a null pointer. If a null pointer is passed, the routine will return any predecessor Bricklet. Passing a filter set will cause the routine to return only predecessor Bricklets that are part of the filter set, i.e. that are subject to an export operation.

Description

This routine returns the predecessor Bricklet of the Bricklet passed to `getPredecessorBricklet()`. The specified Bricklet must have been acquired as part of an operation that either generated a series of consecutive Bricklets (such as signal sampling over a defined period of time) or that produced some logically connected Bricklets (such as a phase/amplitude curve consisting of data from a phase Bricklet and from an amplitude Bricklet).

Please note that the `getPredecessorBricklet()` function can either return any predecessor Bricklet of the specified Bricklet (if a null pointer is passed as argument *filterSet*), or only Bricklets that are subject to an export operation (if a pointer to a filter set data structure is passed as argument *filterSet*).

Please see section **Related Bricklets** for more information on Bricklet relationships.

Return Values

Returns an opaque pointer to the predecessor Bricklet of the Bricklet passed to the routine. If the specified Bricklet is the first of a consecutive Bricklet sequence, or has no associated predecessor Bricklets, a null pointer will be returned.

Associated Routines

```
getDependingBricklets()
getReferencedBricklets()
getSucessorBricklet()
```

getRawMax

Returns the maximum raw value the specified Bricklet contains.

Syntax *rawMaximum* := getRawMax (*bricklet*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
int Session::getRawMax(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine returns the actual maximum raw value which the specified Bricklet stores.

Note that for obtaining the maximum raw value, Vernissage will implicitly load the Bricklet data from disk and unload it afterwards. Thus, calling `getRawMax()` and/or `getRawMin()` frequently is inefficient and results in poor performance. If you want to use `getRawMin()`/`getRawMax()` without additional overhead, you can enclose the respective statements in calls to the `loadBrickletContents()` and `unloadBrickletContents()` service routines.

Note also that calling `getRawMax()` on large Bricklets can take a significant amount of time (i.e. several seconds).

Return Values Returns the maximum raw value the specified Bricklet stores.

Associated Routines

```
getRawMin()
getChannelRawMin()
getChannelRawMax()
toPhysical()
toRaw()
```

getRawMin

Returns the minimum raw value the specified Bricklet contains.

Syntax	*rawMinimum* := getRawMin (*bricklet*)		
	Argument	**Data Type**	**Access**
	bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
int Session::getRawMin(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine returns the actual minimum raw value which the specified Bricklet stores.

Note that for obtaining the minimum raw value, Vernissage will implicitly load the Bricklet data from disk and unload it afterwards. Thus, calling `getRawMin()` and/or `getRawMax()` frequently is inefficient and results in poor performance. If you want to use `getRawMin()`/`getRawMax()` without additional overhead, you can enclose the respective statements in calls to the `loadBrickletContents()` and `unloadBrickletContents()` service routines.

Note also that calling `getRawMin()` on large Bricklets can take a significant amount of time (i.e. several seconds).

Return Values

Returns the minimum raw value the specified Bricklet stores.

Associated Routines

```
getRawMax()
getChannelRawMin()
getChannelRawMax()
toPhysical()
toRaw()
```

getReferencedBricklets

Returns a list of Bricklets that have a "referenced" relationship to a specific Bricklet.

Syntax

bricklets := getReferencedBricklets *(bricklet, filterSet)*

Argument	Data Type	Access
context	Opaque reference	Read
filterSet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
std::vector<void *> Session:: getReferencedBricklets (
  void *pBricklet, void *pFilterSet = 0);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

filterSet
The filter set passed to the `run()` method of the plug-in, or a null pointer. If a null pointer is passed, the list returned will contain all related Bricklets. Passing a filter set will cause the function to return only related Bricklets that are part of the filter set, i.e. that are subject to an export operation.

Description

This routine returns a list of Bricklets that have a "referenced" relationship to the Bricklet passed as argument *bricklet*. An example for a "referenced" relationship is a SAM image of the sample area used for running electron spectroscopy curve acquisition operations because such curves can usually be associated with a particular sample location.

Please note that the `getReferencedBricklets()` function can either return all Bricklets related to the specified Bricklet (if a null pointer is passed as argument *filterSet*), or only Bricklets that are subject to an export operation (if a pointer to a filter set data structure is passed as argument *filterSet*).

Please see section **Related Bricklets** for more information on Bricklet relationships.

Return Values

Returns a list of opaque pointers to Bricklets that have a "referenced" relationship to the Bricklet passed to the routine. If the specified Bricklet has no associated reference Bricklets, the returned list will be empty.

Associated Routines

```
getDependingBricklets()
getSucessorBricklet()
getPredecessorBricklet()
```

getResultDataFileSpec

Returns the file specification of the data file storing a particular Bricklet.

Syntax *fileSpec* := getResultDataFileSpec (*bricklet*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
std::wstring Session::getResultDataFileSpec(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine returns the full file specification of the result data file storing a particular Bricklet.

In case the Bricklet passed to this service routine was embedded into the associated result file and thus no result data file storing the Bricklet is present, an empty string instead of a file specification will be returned.

Return Values

Returns the full file specification of the result data file storing the specified Bricklet, including drive identification letter and directory hierarchy. Returns an empty string if the Bricklet was embedded into its result file.

Associated Routines `getParentResultFileSpec()`

getRootAxisName

Returns the plain name of the root axis of the axis hierarchy associated with a Bricklet.

Syntax

name := getRootAxisName (*bricklet*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
std::wstring Session::getRootAxisName(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

The `getRootAxisName()` service obtains the plain name of the root axis of the axis hierarchy associated with the data channel through which the specified Bricklet was acquired.

The root axis is the *first* axis of an axis hierarchy, i.e. the axis that is not triggered by any other axis. For example, the root axis of a data channel delivering topography images is always the *Y*-axis (which triggers the *X*-axis, which in turn triggers the respective channel.)

Please see section **Axes and Axes Hierachies** for more information on plain axis names and qualified axis names.

Return Values

Returns a character string representing the plain name of the root axis of the axis hierarchy associated with the data channel through which the specified Bricklet was acquired.

Associated Routines

```
getAxisClocks()
getAxisUnit()
getTriggerAxisName()
getTriggerAxisQualifiedName()
getRootAxisQualifiedName()
getAxisDescriptor()
getAxisTableSets()
```

getRootAxisQualifiedName

Returns the qualified name of the root axis of the axis hierarchy associated with a Bricklet.

Syntax

name := getRootAxisQualifiedName (*bricklet*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
std::wstring Session::getRootAxisQualifiedName(
                                void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

The `getRootAxisQualifiedName()` service obtains the qualified name of the root axis of the axis hierarchy associated with the data channel through which the specified Bricklet was acquired.

The root axis is the first axis of an axis hierarchy, i.e. the axis that is not triggered by any other axis. For example, the root axis of a data channel delivering topography images is always the Y-axis (which triggers the X-axis, which in turn triggers the respective channel.)

Please see section **Axes and Axis Hierarchies** for more information on plain axis names and qualified axis names.

Return Values

Returns a character string representing the qualified name of the root axis of the axis hierarchy associated with the data channel through which the specified Bricklet was acquired.

Associated Routines

```
getAxisClocks()
getAxisUnit()
getRootAxisName()
getTriggerAxisName()
getTriggerAxisQualifiedName()
getAxisDescriptor()
getAxisTableSets()
```

getRunCycleCount

Returns the run cycle identifier of a Bricklet.

Syntax *count* := getRunCycleCount (*bricklet*)

Argument	Data Type	Access
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
int Session::getRunCycleCount(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

The run cycle count is a running number that starts at one and will be incremented each time a new a data acquisition operation is started. What "operation" is actually counted depends on the configuration of the respective data channel through which a Bricklet was delivered.

For example, two Bricklets produced subsequently by a topography or similar channel will have different run counts if the scan process has been restarted after the first Bricklet was stored. However, Bricklets produced by a channel acquiring single point spectroscopy curves will have different run counts if they resulted from different single point spectroscopy operations. (This is usually true, except if the automatic repetition facility for single point spectroscopy operations was active. In this case, all Bricklets storing a curve acquired during the repetition process will have the same run count assigned.)

The run cycle count is also part of the file name of result data files, for example, in the result data file name "STM_Spectroscopy--4_2.Z_mtrx" the "4" is the run cycle count of the experiment during which the Bricklet stored in the data file was generated. (Here, the respective experiment was started four times, the Bricklet was produced during the fourth cycle.)

The result file name "STM_Spectroscopy--7_1.I(V)_mtrx", however, could be associated with a file storing a Bricklet from the seventh single point spectroscopy initiated during an experiment.

Return Values

Returns an unsigned integer figure indicating the run cycle of the operation during which the specified Bricklet was generated.

Associated Routines

```
getSequenceId()
getScanCycleCount()
```

getSampleName

Returns the sample name associated with a particular Bricklet.

Syntax *name* := getSampleName (*bricklet*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
std::wstring getSampleName(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description This routine returns the sample name that was in effect when the Bricklet passed was created.

Return Values Returns the sample name associated with the specified Bricklet. If the MATRIX user specified no sample name, a single dash character ('–') will be returned.

Associated Routines

```
getCreationComment()
getDataComments()
getDataSetName()
```

getScanCycleCount

Returns the scan cycle identifier of a Bricklet.

Syntax

count := getScanCycleCount (*bricklet*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
int Session::getScanCycleCount(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine returns the scan cycle count associated with a particular Bricklet.

The scan cycle count is a running number starting at "1"; it will be incremented for each Bricklet generated during the same experiment run cycle.

The scan cycle count is also part of the file name of result data files, for example, in the result data file name "STM_Spectroscopy--4_2.Z_mtrx" the "2" is the scan cycle count that generated the Bricklet stored in the data file. (Here, the respective Bricklet was the second generated during the fourth run cycle.)

Return Values

Returns an unsigned integer figure indicating the experiment scan cycle that generated the specified Bricklet.

Associated Routines

```
getSequenceId()
getRunCycleCount()
```

getSequenceId

Returns the sequence identifier of a Bricklet.

Syntax *id* := getSequenceId (*bricklet*)

Argument	Data Type	Access
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
int Session::getSequenceId(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine returns the sequence identifier associated with a particular Bricklet.

The sequence identifier (also referred to as *sequence ID*) is a running number starting at "1" which gets incremented for each new Bricklet delivered by a specific data channel. Please note, however, that the sequence ID will be reset after a change to the data space structure of an experiment, thus a specific sequence ID can be assigned to more than one Bricklet. (The structure of the experiment data space changes for example if the user modifies the number of points/lines of the scan area, the number of points per spectroscopy curve, enables or disables the scan sub-grid, selects a new scan mode, etc. However, changing parameters such as the scan area angle, the data acquisition raster time, the scan area offset and similar have no impact on the structure of the data space.)

Return Values

Returns an unsigned integer figure indicating the Bricklet sequence identifier associated with the specified Bricklet.

Associated Routines

```
getRunCycleCount()
getScanCycleCount()
```

getSession

Returns an interface object to the Vernissage session services.

Syntax	getSession ()

C++ Binding	`#include "Vernissage.h"`
	`Vernissage::Session* ::getSession();`

Arguments	—
Description	This routine returns an interface object providing access to the Vernissage session services. All service routines for loading, traversing and querying result data structures are available through this interface object only. Call `releaseSession()` when the interface object obtained through a call to `getSession()` is no longer required. Vernissage plug-in modules are not obliged to call `getSession()` as they receive the interface object to the session services automatically when invoked.
Return Values	Returns a pointer to an object of type *Vernissage::Session* to be used for accessing the Vernissage session services.
Associated Routines	`releaseSession()`

getSpatialInfo

Returns information about the sample location(s) at which a Bricklet was acquired.

Syntax

spatialInfo := getSpatialInfo (*bricklet*)

Argument	Data Type	Access
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
struct SpatialInfo {
  std::vector<double> physicalX;
  std::vector<double> physicalY;
  bool originatorKnown;
  std::wstring channelName;
  int sequenceId;
  int runCycleCount;
  int scanCycleCount;
  std::wstring viewName;
  int viewSelectionId;
  int viewSelectionIndex;
};
SpatialInfo Session::getSpatialInfo(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This service returns information about sample locations at which a particular Bricklet was acquired; the position information is based on the coordinate system of a reference Bricklet.

Information about sample locations is only available for certain operations such as single point spectroscopy (SPS) or atom manipulation (AtMa) procedures. It will be generated for Bricklets storing data acquired during such operations only. (Such Bricklets can be identified by calling the service routine `getType()`. Alternatively, you may also call `getDimensionCount();` the service will return "1" for curve data. In the latter case, to be sure that a Bricklet storing curve data was actually generated during an SPS or atom manipulation operation, it must be either associated with a View of type `vtc_Spectroscopy` (for SPS) or `vtc_1DProfile` (for AtMa.) You may obtain the View types associated with a Bricklet by calling the `getViewTypes()` service routine.)

In case of SPS, the information returned by `getSpatialInfo()` is useful for determining the sample position at which the spectroscopy curve was acquired with respect to the configured scan area. Each of the two arrays *physicalX* and *physicalY* returned as part of the information structure conveyed by `getSpatialInfo()` will contain a single entry describing the "physical" coordinate of the location as offset (in metres) to the centre of the configured scan area. (An offset of *X* = 0, *Y* = 0 means that the SPS operation was run at the centre of the scan area.)

In case of an atom manipulation operation, the information returned by the service routine can be used to determine the start and end point of the vector associated with the manipulation operation. In this case, the two arrays *physicalX* and *phyiscalY* returned will contain two values; the first X/Y values describe the "physical" coordinate of the vector's start point, while the second values describe the end point. Again, the values have to be interpreted as offset to the centre of the configured scan area.

If the Bricklet was generated by MATRIX version V1.0–4 or later, the `getSpatialInfo()` service routine will also return information allowing you to determine the exact image the MATRIX user has utilised as reference for initiating the operation: Besides the data channel through which the image data were acquired, identification information regarding the Bricklet containing the respective image will be returned. In addition, the information collected by `getSpatialInfo()` will also indicate the part of the scan cycle (e.g. "forward/up") that was in progress when the operation was launched.

By using the entire set of information items returned by `getSpatialInfo()` you can correlate the results of an operation with the original (image) position the user has clicked on.

Please note that the data fields *viewName* and *viewSelectionIndex* of the information structure returned by `getSpatialInfo()` are currently of limited or no use for third party software modules.

Return Values

Returns an information structure comprising the following entries:

physicalX — A list of physical *X*-axis values (in metres).

physicalY — A list of physical *Y*-axis values (in metres).

originatorKnown — A flag indicating whether additional information about the View/display combination that was used for initiating the acquisition operation is available.

If the flag *originatorKnown* is set to true, the following fields will be filled also: (Otherwise, the below fields will not contain meaningful information.)

channelName — The name of the channel that produced the reference image used for initiating the acquisition operation.

sequenceId — The sequence ID of the Bricklet storing the reference image.

runCycleCount — The run cycle count of the Bricklet storing the reference image.

scanCycleCount — The scan cycle count of the Bricklet storing the reference image.

viewName — The name of the View instance associated with the display that was used for selecting the sample location.

viewSelectionId — A code describing the type of information reduction the associated View used for data processing. This code hence determines during which part of the scan cycle the data acquisition operation was initiated. Valid codes are:

Type Code	Description
-1	Not applicable or no information available
0	Forward/Up scan sweep
1	Backward/Up scan sweep
2	Forward/Down scan sweep
3	Backward/Down scan sweep

viewSelectionIndex — Always zero.

Associated Routines

```
getDimensionCount()
getViewTypes()
getType()
```

getSuccessorBricklet

Returns the successor of a specific Bricklet.

Syntax

bricklet := getSuccessorBricklet *(bricklet, filterSet)*

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read
filterSet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
void * Session::getSuccessorBricklet(void *pBricklet,
                                      void *pFilterSet = 0);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

filterSet
The filter set passed to the `run()` method of the plug-in, or a null pointer. If a null pointer is passed, the routine will return any successor Bricklet. Passing a filter set will cause the routine to return only successor Bricklets that are part of the filter set, i.e. that are subject to an export operation.

Description

This routine returns the successor Bricklet of the Bricklet passed to `getSuccessorBricklet()`. The specified Bricklet must have been acquired as part of an operation that either generated a series of consecutive Bricklets (such as signal sampling over a defined period of time) or that produced some logically connected Bricklets (such as a phase/amplitude curve consisting of data from a phase Bricklet and from an amplitude Bricklet).

Please note that the `getSuccessorBricklet()` function can either return any successor Bricklet of the specified Bricklet (if a null pointer is passed as argument *filterSet*), or only Bricklets that are subject to an export operation (if a pointer to a filter set data structure is passed as argument *filterSet*).

Please see section **Related Bricklets** for more information on Bricklet relationships.

Return Values

Returns an opaque pointer to the successor Bricklet of the Bricklet passed to the routine. If the specified Bricklet is the last of a consecutive Bricklet sequence, or has no associated successor Bricklets, a null pointer will be returned.

Associated Routines

```
getDependingBricklets()
getReferencedBricklets()
getPredecessorBricklet()
```

getTriggerAxisName

Returns the plain name of the axis triggering the data channel through which a Bricklet was acquired.

Syntax

name := getTriggerAxisName (*bricklet*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
std::wstring Session::getTriggerAxisName(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

The `getTriggerAxisName()` service obtains the plain name of the axis that triggers the data channel through which the specified Bricklet was acquired.

The trigger axis is the *last* axis of an axis hierarchy, i.e. the axis that triggers a data acquisition channel. For example, the trigger axis of a channel delivering topography images is always the *X*-axis (which itself gets triggered by the *Y*-axis.)

Please see section **Axes and Axes Hierachies** for more information on plain axis names and qualified axis names.

Return Values

Returns a character string representing the plain name of the trigger axis of the axis hierarchy associated with the data channel through which the specified Bricklet was acquired.

Associated Routines

```
getAxisClocks()
getAxisUnit()
getRootAxisName()
getRootAxisQualifiedName()
getTriggerAxisQualifiedName()
getAxisDescriptor()
getAxisTableSets()
```

getTriggerAxisQualifiedName

Returns the qualified name of the axis triggering the data channel through which a Bricklet was acquired.

Syntax

name := getTriggerAxisQualifiedName (*bricklet*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
std::wstring Session::getTriggerAxisQualifiedName(
                                             void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

The `getTriggerAxisQualifiedName()` service obtains the qualified name of the axis that triggers the data channel through which the specified Bricklet was acquired.

The trigger axis is the last axis of an axis hierarchy, i.e. the axis that triggers a data acquisition channel. For example, the trigger axis of a channel delivering topography images is always the X-axis (which itself gets triggered by the Y-axis.)

Please see section **Axes and Axis Hierarchies** for more information on plain axis names and qualified axis names.

Return Values

Returns a character string representing the qualified name of the trigger axis of the axis hierarchy associated with the data channel through which the specified Bricklet was acquired.

Associated Routines

```
getAxisClocks()
getAxisUnit()
getRootAxisName()
getRootAxisQualifiedName()
getTriggerAxisName()
getAxisDescriptor()
getAxisTableSets()
```

getType

Returns a code identifying the contents of a Bricklet.

Syntax *ident* := getType (*bricklet*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
typedef enum {
  btc_Unknown,
  btc_SPMSpectroscopy,
  btc_AtomManipulation,
  btc_1DCurve,
  btc_SPMImage,
  btc_PathSpectroscopy,
  btc_ESpRegion,
  btc_VolumeCITS,
  btc_DiscreteEnergyMap,
  btc_ForceCurve,
  btc_PhaseAmplitudeCurve,
  btc_SignalOverTime,
  btc_RawPathSpectroscopy,
  btc_ESpSnapshotSequence,
  btc_ESpImageMap,
  btc_InterferometerCurve,
  btc_ESpImage
} BrickletTypeCode;
BrickletTypeCode Session::getType(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine analyses the contents of the Bricklet passed and returns a code identifying the type of data stored by the Bricklet. Calling `getType()` is thus basically a convenience function simplifying the procedure of identifying the kind of operation to be applied when processing the contents of a particular Bricklet.

The routine may return the following identification codes:

- `btc_SPMSpectroscopy` — The Bricklet stores a single SPM spectroscopy curve.

- `btc_AtomManipulation` — The Bricklet contains data (actually a one-dimensional object such as an I(r) curve) resulting from an Atom Manipulation operation.

- `btc_1DCurve` — The Bricklet stores an unspecific curve, i.e. the type of operation that produced the curve is arbitrary.

- `btc_SPMImage` — The Bricklet contains one (in case of a "forward only", "up only") to four (in case of a "forward/backward", "up/down") SPM images.

- `btc_VolumeCITS` — The Bricklet is a three-dimensional object storing the spectroscopy curves from a raster spectroscopy ("Volume CITS") experiment run.

- `btc_PhaseAmplitudeCurve` — The Bricklet stores one curve from a

Phase/Amplitude curve set, i.e. a phase curve or an amplitude curve.

- `btc_SignalOverTime` — The Bricklet contains a one-dimensional object representing one part of a curve acquired continuously over time.

- `btc_RawPathSpectroscopy` — The Bricklet stores "raw" counts for a number of data points at different energy levels. ("raw" means that all counts from all channels are stored separately without any preprocessing.) The structure of the Bricklet is three-dimensional, as each set of counts is represented *n* times with *n* being the number of energy levels used for the experiment

- `btc_PathSpectroscopy` — Similar to "raw" path spectroscopy, the Bricklet stores data acquired at various locations, however, the counts from the various channels will not be stored separately but as a single accumulated count value.

- `btc_ESpRegion` — The Bricklet represents electron spectroscopy result data, i.e. the electron counts of the various detector channels for each point of an electron spectroscopy curve.

- `btc_ ESpSnapshotSequence` — The Bricklet stores the electron counts of the various detector channels for a specific number of acquisition operations ("snapshots") at different energy levels.

- `btc_DiscreteEnergyMap` — The Bricklet contains a discrete energy map, i.e. a number of two-dimensional planes representing accumulated electron counts; each plane was acquired at a specific energy level.

- `btc_ESpImageMap` — The Bricklet consists of a four-dimensional data object representing a number of 2D-planes each of which has been acquired at a particular energy level. Each data point on a plane in turn consists of a set of "raw" data counts (one count value per channel)

- `btc_ForceCurve` — The Bricklet stores a single force/distance curve.

- `btc_InterferometerCurve` — The bricklet stores Force/distance curve

- `btc_ESpImage` — The Bricklet contains a forward/down Sp image (2D SEM image acquired by generic scanner)
- `btc_Unknown` — The data contents of the Bricklet could not be identified. This code can be returned if the Bricklet has been generated by a MATRIX software version that is not compatible with the Vernissage release in use.

Return Values

Returns an identification code determining the type of data stored by a particular Bricklet.

Associated Routines

```
getDimensionCount()
getViewTypes()
```

getViewTypes

Returns a list of Data Views associated with a particular Bricklet.

Syntax

viewTypes := getViewTypes (*bricklet*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
typedef enum {
  vtc_Other, vtc_Simple2D, vtc_Simple1D, vtc_ForwardBackward2D,
  vtc_2Dof3D, vtc_Spectroscopy, vtc_ForceCurve, vtc_1DProfile,
  vtc_Interferometer, vtc_ContinuousCurve,
  vtc_PhaseAmplitudeCurve, vtc_CurveSet,
  vtc_ParameterisedCurveSet, vtc_DiscreteEnergyMap,
  vtc_ESpImageMap
} ViewTypeCode;
std::vector<ViewTypeCode> Session::getViewTypes(
                                void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine determines the Data Views associated with the data channel at which the specified Bricklet originated; the routine will return a list of View identification codes specifying the types of Views utilised.

The types of Views associated with a particular data channel can be of interest, because they hint at the use of the data acquired through that channel at experiment run-time. For example, the View type code `vtc_ForwardBackward2D` reveals that the data from the respective channel was used for feeding 2D displays rendering data from a spatial scan process. The View type code `vtc_Spectroscopy` together with `vtc_2Dof3D` reveals that the acquired data were fed into a display dedicated to rendering SPM spectroscopy curves and into another display showing spectroscopy curves *parameter-wise*. (The latter display would only be active when the grid spectroscopy option had been enabled.)

Whenever the inherent characteristics of the data contained by a particular Bricklet (for example, the dimensionality of the data) is not sufficient to determine how to process the Bricklet contents, checking the associated View types can provide additional hints: For example, when encountering a Bricklet storing one-dimensional data (i.e. a single curve), checking the associated Views will instantly reveal whether the respective data were generated during a single point spectroscopy operation, an AFM force/distance curve experiment, an AtMa operation etc.

See section **Understanding Data Views** for more information on View types and the View type codes.

Return Values

Returns a list of View type codes associated with the specified Bricklet as argument. If the Bricklet originated at a channel associated with several Views of the same type, the respective type code is only returned once. If no View was assigned to the channel at which the specified Bricklet originated, the returned list will be empty.

Associated Routines

```
getDimensionCount()
getSpatialInfo()
getType()
```

loadAllResultSets

Loads all result sets from a directory into the internal database.

Syntax

loadAllResultSets (*pathSpec, accumulative*)

Argument	**Data Type**	**Access**
pathSpec	Wide-character string	Read
accumulative	Boolean	Read

C++ Binding

```
#include "Vernissage.h"
bool Session::loadAllResultSets(std::wstring pathSpec,
                                bool accumulative);
```

Arguments

pathSpec
An absolute or relative path specification determining the directory containing the result set(s) to be loaded.

accumulative
A flag indicating whether the result sets should be added to the database (*true*), or will replace the current result sets in the database (*false*).

Description

This service routine allows third party applications to load all result sets from a specific directory into the internal database of the Vernissage software.

The **pathSpec** argument must specify the path to an existing directory storing the result sets to be loaded and can contain a Microsoft Windows absolute (e.g. "D:\Result Files\Today") or relative (e.g. "..\..\Today") path description.

The `loadAllResultSets()` function will ignore any file stored in the specified location that is not part of a result set. The function will not search sub-directories of the specified directory.

This routine is only useful for special third party applications that need to load result sets and **must not** be called by exporter plug-ins.

Return Values

True if the result sets have been loaded successfully, *false* if at least one of the specified result sets could not be loaded.

Note that if `loadAllResultSets()` returns *false*, this does not necessarily indicate that the Vernissage-internal database remained unchanged, as some result sets (or part of a result set) may have been loaded successfully.

Associated Routines

```
loadResultSet()
eraseResultSets()
loadBrickletContents()
unloadBrickletContents()
```

loadBrickletContents

Loads the data contained by a Bricklet into memory.

Syntax

loadBrickletContents (*bricklet, buffer, count*)

Argument	Data Type	Access
bricklet	Opaque reference	Read
buffer	Integer array reference	Modify
count	Signed integer	Modify

C++ Binding

```
#include "Vernissage.h"
void Session::loadBrickletContents(void *pBricklet,
                                   const int **pBuffer,
                                   int& count);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

buffer
A pointer variable into which the service routine will store the address of the data buffer containing the Bricklet contents.

count
A reference to an integer variable into which the service routine will store the number of data items contained by the Bricklet.

Description

As Bricklets can be quite large (and the raw data contained by a Bricklet can thus occupy significant amounts of memory), the Vernissage core software does *not* load Bricklet contents by default. When access to the raw data stored in a Bricklet is required, you must load the Bricklet contents into memory first.

The `loadBrickletContents()` service routine will allocate a buffer large enough to hold the contents of the Bricklet passed as argument to the routine and load the raw data into that buffer. If the Bricklet data are stored in a separate result data file, `loadBrickletContents()` will read the data from the respective file. To deallocate the memory used for storing the contents of the specified Bricklet, call the `unloadBrickletContents()` service routine.

The `loadBrickletContents()` service will load the raw data of a Bricklet only once, multiple calls to `loadBrickletContents()` with an identical **bricklet** argument will return the same buffer address for each call. Note that the Vernissage core will record all calls to `loadBrickletContents()` that refer to the same Bricklet internally, the buffer storing the raw data will not be deallocated unless the `unloadBrickletContents()` routine has been called as many times as `loadBrickletContents()` was used on the same Bricklet. (In practise, this generally means that each time your code calls `loadBrickletContents()` with a particular Bricklet as argument, it must call `unloadBrickletContents()` subsequently in order to deallocate the memory used for storing the raw data of the respective Bricklet.)

Upon successful completion, the `loadBrickletContents()` service will store the address of the buffer containing the raw data of the Bricklet into the pointer variable passed as **buffer** argument; the number of raw data items will be stored in the integer variable passed as argument **count**.

If `loadBrickletContents()` fails to allocate the buffer, or is not able to access the result data file storing the Bricklet, a null pointer will be returned.

Return Values

—

Associated Routines `unloadBrickletContents()`

loadResultSet

Loads one or more result sets into the internal database.

Syntax

loadResultSet (*pathSpec, fileName, accumulative*)

Argument	**Data Type**	**Access**
pathSpec	Wide-character string	Read
fileName	Wide-character string	Read
accumulative	Boolean	Read

C++ Binding

```
#include "Vernissage.h"
bool Session::loadResultSet(std::wstring pathSpec,
                            std::wstring fileName,
                            bool accumulative);
```

Arguments

pathSpec
An absolute or relative path specification determining the path to the result file(s) to be loaded. The path specification may not contain any wildcard characters.

fileName
The name of the result file to be loaded. The name may contain wildcard characters; if such characters are found, all files matching the wildcard pattern will be loaded.

accumulative
A flag indicating whether the result set(s) should be added to the database (*true*), or will replace the current result sets in the database (*false*).

Description

This service routine allows third party applications to load one or more result sets from a specific directory. The routine also allows to restrict the load operation to one or more result data files.

The **filePath** and **fileName** arguments determine the path and name of the file(s) to be loaded. The file path specification can be both, absolute (e. g. "C:\Temp\Today") or relative (e. g. "..\..\Today"). The file name specifies the name of the result set to be loaded and may contain wildcard characters (such as "*" or "?") for loading several result sets in one operation.

In case the file name part of the file specification uses wildcards, the `loadResultSet()` function will load any MATRIX result file or result data file matching the file specification pattern; non-MATRIX files matching the pattern will be ignored.

If a file specification pattern matches a particular result data file as well as its associated result file, the Bricklet data will nevertheless be loaded only once. Also, patterns matching several result files from the same result file chain will not cause multiple read operations.

This routine is only useful for special third party applications that need to load result sets and **must not** be called by exporter plug-ins.

Return Values

True if the result set(s) have been loaded successfully, *false* if at least one of the specified result sets could not be loaded.

Note that if `loadResultSet()` returns *false*, this does not necessarily indicate that the Vernissage-internal database remained unchanged, as some result sets (or part of a result set) may have been loaded successfully.

Associated Routines

```
loadAllResultSets()
eraseResultSets()
loadBrickletContents()
unloadBrickletContents()
```

makePath

Concatenates a file or directory path element to an existing path.

Syntax *pathSpec* := makePath (*pathSpec, pathElement*)

Argument	**Data Type**	**Access**
pathSpec	Wide-character string	Read
pathElement	Wide-character string	Read

C++ Binding

```
#include "Vernissage.h"
std::wstring Session::makePath(std::wstring path,
                               std::wstring element);
```

Arguments

pathSpec
An absolute or relative path specification.

pathElement
A file or directory path element to be joined with the given path specification.

Description

This convenience routine concatenates the specified file or directory path element (for example, the file name specification "Outfile.txt") to a given path specification (such as "C:\Temp"). The resulting path specification ("C:\Temp\Outfile.txt") will be returned.

Return Values

Returns the concatenated path specification.

Associated Routines

`splitPath()`

releaseBrickletContext

Terminates a Bricklet set iteration sequence and resets the context.

Syntax

releaseBrickletContext (*context*)

Argument	**Data Type**	**Access**
context	Opaque reference	Modify

C++ Binding

```
#include "Vernissage.h"
void Session::releaseBrickletContext(void **pContext);
```

Arguments

context
An opaque pointer variable from a previous call to the `getNextBricklet()` service.

Description

This routine terminates a Bricklet set iteration sequence initiated by a previous call to the `getNextBricklet()` service. The service context will be reset; subsequent calls to `getNextBricklet()` with the same **context** argument will restart the iteration.

Calling `releaseBrickletContext()` is not required if the entire Bricklet set has already been returned, i.e. the most recent call to the `getNextBricklet()` service returned a null pointer.

Return Values

—

Associated Routines

`getNextBricklet()`

releaseSession

Marks an interface object as obsolete for the Vernissage session services.

Syntax	releaseSession ()

C++ Binding	`#include "Vernissage.h"`
	`void ::releaseSession();`

Arguments	—
Description	This routine marks an interface object obtained through a call to `getSession()` as obsolete. After calling `releaseSession()`, you must not use the respective interface object for calling Vernissage services any longer. Vernissage plug-in modules are not obliged to call `releaseSession()` as the interface object management is done automatically by the Vernissage core software.
Return Values	—
Associated Routines	`getSession()`

showWorkInProgress

Renders a work-in-progress indicator for informing the user about the progress of data processing.

Syntax

showWorkInProgress (*percentage*)

Argument	Data Type	Access
percentage	Unsigned integer	Read

C++ Binding

```
#include "Vernissage.h"
void Session::showWorkInProgress(int percentage);
```

Arguments

percentage
The percentage to which data processing has already be completed.

Description

This routine will direct the Vernissage core software to render a work-in-progress indicator. The indicator will reflect the amount of data processing work a (plug-in) module has already completed; this amount must be passed to the routine as percentage.

The `showWorkInProgress()` service routine can be called multiple times with (presumably) increasing **percentage** argument values to keep a Vernissage user updated about the progress of the active module.

If **percentage** is smaller than zero, `showWorkInProgress()` will return immediately without updating the indicator.

The type of indicator rendered by the `showWorkInProgress()` service depends on the active user interface and other aspects of the execution environment; you should not assume a particular type of indicator (such as a work-in-progress box) to be rendered when calling this routine.

Note that this routine is intended for use by plug-in modules and is of no use for third party software linking against the Vernissage service API.

Return Values —

Associated Routines —

splitPath

Splits a file or directory path specification into its components.

Syntax

components := splitPath (*pathSpec*)

Argument	**Data Type**	**Access**
pathSpec	Wide-character string	Read

C++ Binding

```
#include "Vernissage.h"
std::vector<std::wstring> Session::splitPath(
                                      std::wstring path);
```

Arguments

pathSpec
An absolute or relative path specification.

Description

This convenience routine splits the given path specification into its components. The path specification can be absolute or relative, is not required to refer to an existing file or directory hierarchy and can therefore also be incomplete.

Return Values

Returns a collection of wide-character strings representing the components of the given path specification.

Associated Routines

`makePath()`

toPhysical

Returns the physical quantity representation of a raw value with respect to the specified Bricklet.

Syntax *physical* := toPhysical (*raw*, *bricklet*)

Argument	Data Type	Access
raw	Signed integer	Read
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
double Session::toPhysical(int rawValue,
                           void *pBricklet);
```

Arguments

rawValue
The raw value to be transformed.

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine transforms the specified raw value into a physical quantity. The transfer function used for the operation is determined by the data channel through which the specified Bricklet has been acquired.

Calling `toPhysical()` on a value that is not raw data from the Bricklet passed as second argument may result in an invalid physical value.

To obtain the name of the unit of the physical quantity, use the service routine `getChannelUnit()`.

Return Values

Returns the transformed raw value.

Associated Routines

```
getRawMin()
getRawMax()
getChannelRawMin()
getChannelRawMax()()
toRaw()
```

toRaw

Returns the raw representation of the specified physical value with respect to the specified Bricklet.

Syntax	*raw* := toRaw (*physical*, *bricklet*)		
	Argument	**Data Type**	**Access**
	physical	Double-precision floating point	Read
	bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
int Session::toRaw(double physicalValue,
                   void *pBricklet);
```

Arguments

physicalValue
The physical value to be transformed.

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

This routine transforms the specified physical value into its raw representation equivalent. The transfer function used for the operation is determined by the data channel through which the specified Bricklet has been acquired.

Calling `toRaw()` on a physical value that is not the physical representation of a raw data item from the Bricklet passed as second argument may result in an invalid raw value.

Return Values

Returns the transformed physical value.

Associated Routines

```
getRawMin()
getRawMax()
getChannelRawMin()
getChannelRawMax()()
toPhysical()
```

unicodeToAnsi

Converts a wide-character Unicode string into its 8-bit ANSI multi-byte equivalent.

Syntax *ansiString* := unicodeToAnsi *(unicodeString*)

Argument	**Data Type**	**Access**
unicodeString	Wide-character string	Read

C++ Binding

```
#include "Vernissage.h"
std::string Session::unicodeToAnsi(std::wstring str);
```

Arguments

unicodeString
The wide-character string to be converted into its 8-bit ANSI equivalent.

Description

This routine converts a wide-character Unicode-encoded string into the corresponding ANSI/ASCII 8-bit equivalent.

This is a convenience routine for simplifying the management of string data.

Return Values

Returns the 8-bit ANSI/ASCII-equivalent of the input character string.

Associated Routines

`ansiToUnicode()`

unloadBrickletContents

Unloads the data of a particular Bricklet from memory.

Syntax unloadBrickletContents (*bricklet*)

Argument	**Data Type**	**Access**
bricklet	Opaque reference	Read

C++ Binding

```
#include "Vernissage.h"
void Session::unloadBrickletContents(void *pBricklet);
```

Arguments

bricklet
An opaque pointer variable from a previous call to `getNextBricklet()`.

Description

The `unloadBrickletContents()` service routines deallocates the memory buffer used for storing the raw data of the specified Bricklet. After the routine returns, you must not access the buffer contents any longer.

If the `loadBrickletContents()` service was called multiple times with the same **bricklet** argument, a call to `unloadBrickletContents()` will only mark the respective raw data buffer for deletion, however, the buffer and its contents will be retained. The Vernissage core will deallocate the buffer when the number of calls to `unloadBrickletContents()` matches the number of calls to `loadBrickletContents()` for the same Bricklet.

Note that calling `loadBrickletContents()` without a subsequent call to `unloadBrickletContents()` will cause the Vernissage software to keep the raw data of the affected Bricklet in memory. As Bricklets can become quite large (and hence can utilise significant amounts of memory) failing to call `unloadBrickletContents()` can be the source of an unexpected memory depletion or software performance degradation.

Return Values —

Associated Routines `loadBrickletContents`()

Appendix: Raw Data Structures

This appendix briefly describes all Bricklet raw data structures the Vernissage software can process. Whether a particular exporter module is capable to convert each of these structures strongly depends on the exporter implementation and target data format.

SPM Bricklet Types

SPM Image

Bricklet type code:	`btc_SPMImage`
Dimensionality:	2
Common View types:	`vtc_ForwardBackward2D`

The raw data represents the signal samples as signed 32-bit values. Samples are stored in the order of acquisition; e.g. "forward" sweep of first scan line, "backward" sweep of first scan line (if enabled), etc. If the scan process included an "up" trace as well as a "down" trace, the Bricklet will not only include the samples obtained during the "upward" scan, but also store the data line by line in reversed order as acquired while scanning in "downward" direction.

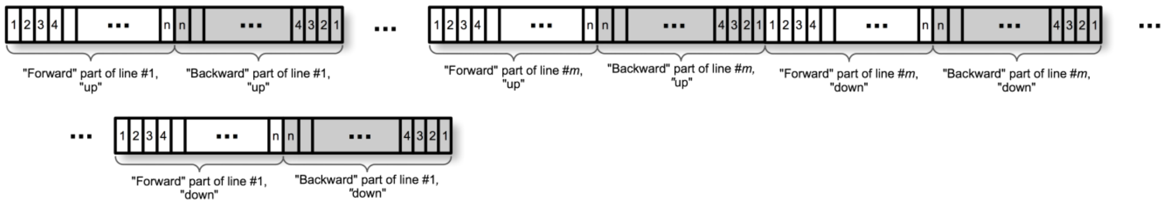

"Forward" part of line #1, "up"
"Backward" part of line #1, "up"
"Forward" part of line #m, "up"
"Backward" part of line #m, "up"
"Forward" part of line #m, "down"
"Backward" part of line #m, "down"
"Forward" part of line #1, "down"
"Backward" part of line #1, "down"

Figure 40. SPM image data of a forward/backward up/down scan.

SPS Curve

Bricklet type code:	`btc_SPMSpectroscopy`
Dimensionality:	1
Common View types:	`vtc_Spectroscopy`

The raw data represents the signal samples as signed 32-bit values; each data item corresponds to a point on the spectroscopy curve. If the "ramp reversal" option was enabled during the spectroscopy operation, the Bricklet will also contain the data acquired while retracing the ramp.

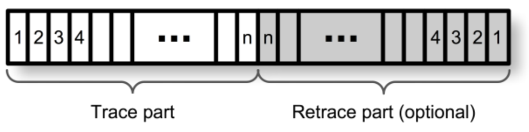


Figure 41. SPS curve data.

Volume CITS

Bricklet type code:	`btc_VolumeCITS`
Dimensionality:	3
Common View types:	`vtc_Spectroscopy`, `vtc_2Dof3D`

The raw data represents the signal samples as signed 32-bit values; each data item corresponds to a point on a spectroscopy curve. If the "ramp reversal" option was enabled during the spectroscopy operation, the Bricklet will also contain the data acquired while retracing the ramp. The Bricklet stores acquired data curve-by-curve in the order of acquisition; e.g. spectroscopy curve data acquired during the "forward" sweep of the first scan line, and then during the "backward" sweep of the first scan line (if enabled), etc. If the scan process included an "up" trace as well as a "down" trace, the Bricklet will not only include the spectroscopy curve data obtained during the "upward" scan, but also while scanning in "downward" direction.

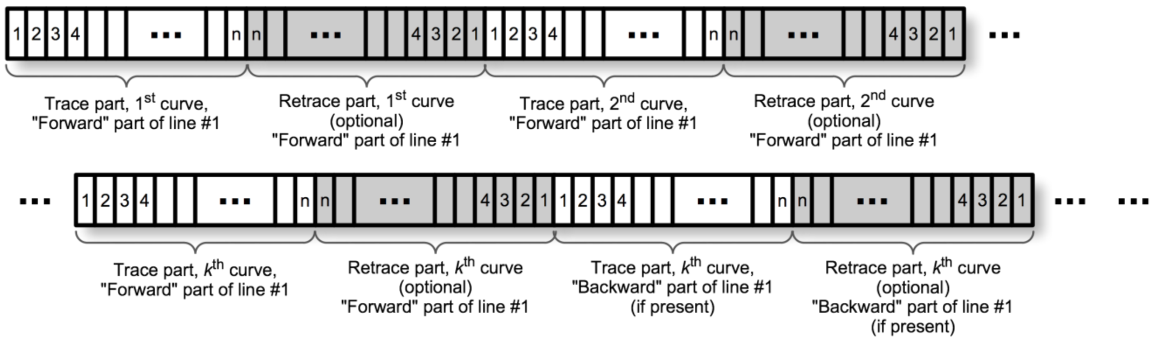

Trace part, 1st curve, "Forward" part of line #1
Retrace part, 1st curve (optional) "Forward" part of line #1
Trace part, 2nd curve, "Forward" part of line #1
Retrace part, 2nd curve (optional) "Forward" part of line #1
Trace part, kth curve, "Forward" part of line #1
Retrace part, kth curve (optional) "Forward" part of line #1
Trace part, kth curve, "Backward" part of line #1 (if present)
Retrace part, kth curve (optional) "Backward" part of line #1 (if present)

Figure 42. Volume CITS data of a forward/backward up/down scan.

Phase/Amplitude Curve

Bricklet type code:	`btc_PhaseAmplitudeCurve`
Dimensionality:	1
Common View types:	`vtc_PhaseAmplitudeCurve`

The raw data represents the phase or amplitude signal samples as signed 32-bit values; each data item corresponds to a point on the phase or amplitude curve. A Bricklet of this type either contains a phase curve or an amplitude curve but not both. Please refer to section **Related Bricklets** for more information on finding the corresponding amplitude or phase curve for a given Bricklet of type "Phase/Amplitude Curve".

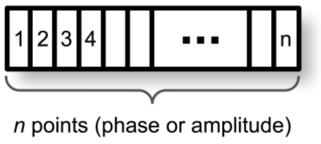


Figure 43. Phase/amplitude curve data.

Force Curve

Bricklet type code:	`btc_ForceCurve`
Dimensionality:	1
Common View types:	`vtc_ForceCurve`

The raw data represents the force signal samples as signed 32-bit values; each data item corresponds to a point on the force/distance curve. The Bricklet will contain the data acquired during the approach and retract phase of the operation in chronological order.

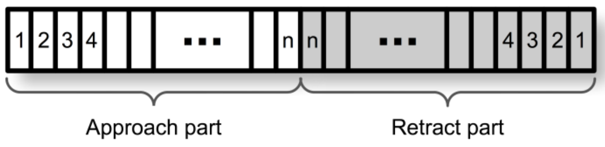


Figure 44. Force/distance curve data.

Atom Manipulation Curve

Bricklet type code:	`btc_AtomManipulation`
Dimensionality:	1
Common View types:	`vtc_1DProfile`

The raw data represents the signal samples as signed 32-bit values; each data item corresponds to a point on the atom manipulation curve.

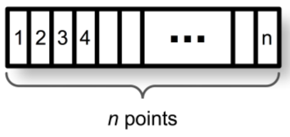


Figure 45. Atom manipulation curve data.

Continuous Signal Curve

Bricklet type code:	`btc_SignalOverTime`
Dimensionality:	1
Common View types:	`vtc_ContinuousCurve`

The raw data represents the respective signal samples as signed 32-bit values; each data item corresponds to a point on the curve. The Bricklet will contain the number of samples configured by the user and is usually part of a longer sequence of curve segment Bricklets resulting from sampling a signal for a specific period. Please refer to section **Related Bricklets** for more information on finding the related curve segments for a given Bricklet of type "Continuous Signal Curve".

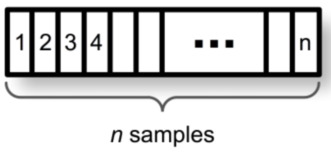


Figure 46. Curve data representing a segment from a continuous curve.

Arbitrary Curve

Bricklet type code:	`btc_1DCurve`
Dimensionality:	1
Common View types:	`vtc_Interferometer`

The raw data represents the respective signal samples as signed 32-bit values; each data item corresponds to a point on the curve. The "physical" interpretation of the curve is unspecific; however, the Bricklet type is currently used for storing data from an interferometer adjustment data channel only.

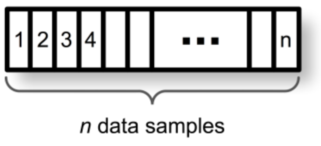


Figure 47. Unspecific curve data.

Electron Spectroscopy Bricklet Types

Region Sweep

Bricklet type code:	`btc_ESpRegion`
Dimensionality:	2
Common View types:	`vtc_CurveSet`

The raw data represents a set of electron counts; for each energy analyser detector channel a dedicated count is stored. The Bricklet represents a single energy range sweep and contains a set of electron counts for each energy level of the sweep.

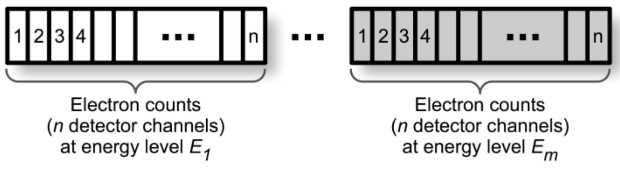

1
2
3
4
...
n
...
Electron counts (*n* detector channels) at energy level E_1
Electron counts (*n* detector channels) at energy level E_m

Figure 48. Electron spectroscopy data from a single energy range sweep.

Line/Multi-Point Energy Map

Bricklet type code:	`btc_PathSpectroscopy`
Dimensionality:	2
Common View types:	`vtc_DiscreteEnergyMap`

The raw data represents accumulated electron counts (i.e. the sum of all counts from the various energy analyser detector channels) acquired at a particular discrete energy level and at a particular sample location. The organisation of the counts depends on the energy switching policy configured; the counts will be either grouped by sample location (switching policy "Point-wise"), or by energy level (switching policy "Path-wise" or "Set-wise", respectively.)

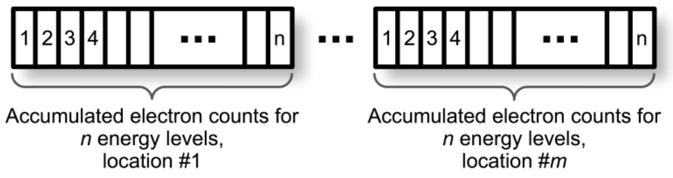

1
2
3
4
...
n
...
Accumulated electron counts for *n* energy levels, location #1
Accumulated electron counts for *n* energy levels, location #*m*

Figure 49. Path spectroscopy data, energy switching policy point-wise.

Line/Multi-Point "Raw" Energy Map

Bricklet type code:	`btc_RawPathSpectroscopy`
Dimensionality:	3
Common View types:	`vtc_DiscreteEnergyMap`

The raw data represents electron counts; unlike the line/multi-point energy map outlined above, a dedicated count value is stored for every energy analyser detector channel. Each set of counts has been acquired at a particular discrete energy level and at a particular sample location. The organisation of the count sets depends on the energy switching policy configured; the sets will be either grouped by sample location (switching policy "Point-wise"), or by energy level (switching policy "Path-wise" or "Set-wise", respectively.)

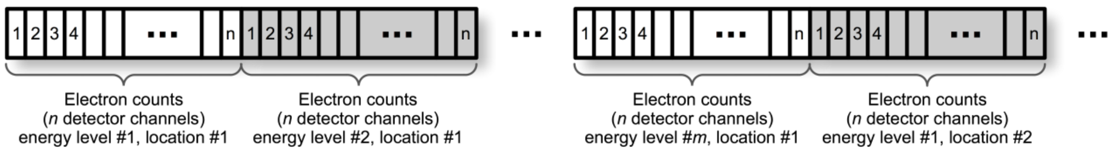

1 2 3 4 ... n 1 2 3 4 ... n ... 1 2 3 4 ... n 1 2 3 4 ... n ...
Electron counts (n detector channels) energy level #1, location #1
Electron counts (n detector channels) energy level #2, location #1
Electron counts (n detector channels) energy level #m, location #1
Electron counts (n detector channels) energy level #1, location #2

Figure 50. Path spectroscopy data, energy switching policy point-wise.

Discrete Energy Map (Raw Image)

Bricklet type code:	`btc_ESpImageMap`
Dimensionality:	4
Common View types:	`vtc_ESpImageMap`

The raw data represents electron counts acquired at different discrete energy levels; a dedicated count value is stored for every energy analyser detector channel. The Bricklet contents can be interpreted as a "layered" image where each "pixel" of a particular layer actually consists of *n* electron counts acquired at a specific energy level. The number of image layers would then correspond to the number of energy levels used. The actual organisation of the counts depends on the energy switching policy configured; the counts will be either grouped by sample location (switching policy "Point-wise"), scan line-by-scan line (switching policy "Line-wise") or by energy level (switching policy "Frame-wise".)

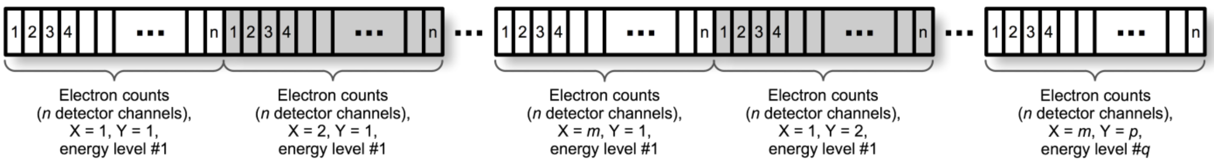

1 2 3 4 ... n 1 2 3 4 ... n ... 1 2 3 4 ... n 1 2 3 4 ... n ... 1 2 3 4 ... n
Electron counts (n detector channels), X = 1, Y = 1, energy level #1
Electron counts (n detector channels), X = 2, Y = 1, energy level #1
Electron counts (n detector channels), X = m, Y = 1, energy level #1
Electron counts (n detector channels), X = 1, Y = 2, energy level #1
Electron counts (n detector channels), X = m, Y = p, energy level #q

Figure 51. Raw image map data, energy switching policy frame-wise.

Energy Snapshot Sequence

Bricklet type code:	`btc_ESpSnapshotSequence`
Dimensionality:	3
Common View types:	`vtc_ParameterisedCurveSet`

The raw data represents electron counts resulting from a series of "snapshots" executed consecutively at different energy levels; the sequence of energy levels can be repeated an arbitrary number of times. The Bricklet contains a dedicated count value for every energy analyser detector channel. The Bricklet can contain up to 4,096 repetitions of the acquisition sequence for the various energy levels; as an actual snapshot sequence can consist of significantly more than 4,096 repetitions, a specific Bricklet of type "Energy Snapshot Sequence" might only be one part of a set of consecutive snapshot Bricklets. (Please refer to section **Related Bricklets** for more information on finding the related parts of a given Bricklet of type "Energy Snapshot Sequence".)

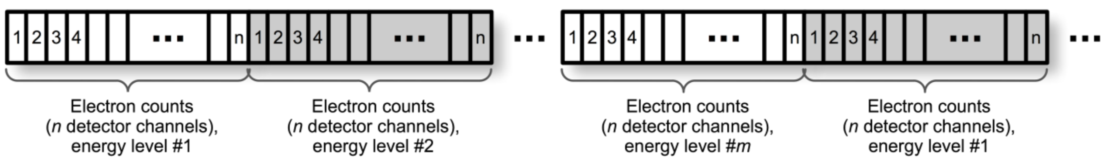

1 2 3 4 ... n 1 2 3 4 ... n ... 1 2 3 4 ... n 1 2 3 4 ... n ...
Electron counts (n detector channels), energy level #1
Electron counts (n detector channels), energy level #2
Electron counts (n detector channels), energy level #m
Electron counts (n detector channels), energy level #1

Figure 52. Snapshot sequence data.

Discrete Energy Map (Accumulated Counts)

Bricklet type code:	`btc_DiscreteEnergyMap`
Dimensionality:	3
Common View types:	`vtc_ESpImageMap`

The raw data represents accumulated electron counts (i.e. the sum of all counts from the various energy analyser detector channels) acquired at different discrete energy levels. The Bricklet contents can be interpreted as a "layered" image where each "pixel" of a particular layer actually consists of an accumulated electron count acquired at a specific energy level. The number of image layers would then correspond to the number of energy levels used. The actual organisation of the counts depends on the energy switching policy configured; the counts will be either grouped by sample location (switching policy "Point-wise"), line-by-line (switching policy "Line-wise") or by energy level (switching policy "Frame-wise".)

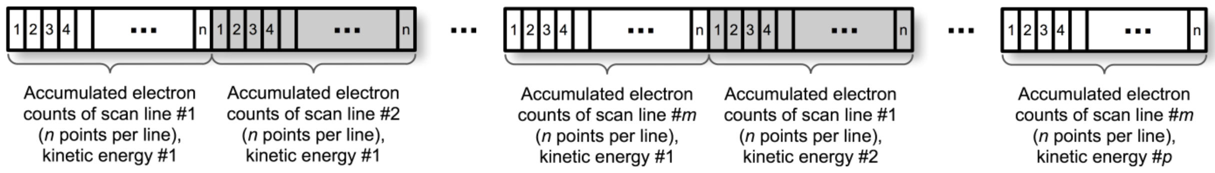

Accumulated electron counts of scan line #1 (n points per line), kinetic energy #1
Accumulated electron counts of scan line #2 (n points per line), kinetic energy #1
Accumulated electron counts of scan line #m (n points per line), kinetic energy #1
Accumulated electron counts of scan line #1 (n points per line), kinetic energy #2
Accumulated electron counts of scan line #m (n points per line), kinetic energy #p

Figure 53. Image map data, energy switching policy frame-wise.

Generic Image

Bricklet type code:	`btc_ESpImage`
Dimensionality:	2
Common View types:	`vtc_Downward2D`

The raw data items represent accumulated electron counts for each sample location in the order of acquisition. Hence, the raw data is organised as a series of 32-bit values representing the image left-to-right and line-by-line, starting with the pixel in the upper left corner of the image. (With respect to SPM image data, the scan direction is thus always "forward/down".)

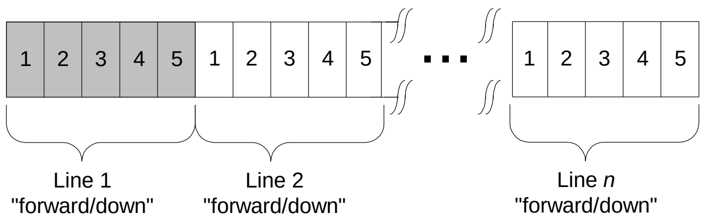


Figure 54. Electron spectroscopy image data of a generic image.

Service at Omicron

Should your equipment **require service**

- Please **contact OMICRON** headquarters or your local OMICRON representative to discuss the problem. An up-to-date address list is available on our website

  http://www.omicron.de/

- Make sure all necessary information is supplied. Always **note the serial number(s)** of your instrument and related equipment (e.g. head, electronics, preamp…) of your instrument or have it at hand when calling.

If you have to **send any equipment back to OMICRON**

- Please contact **OMICRON headquarters** before shipping any equipment.
- Place the instrument it in a polythene bag and **use the original packaging and transport locks.**
- Take out a **transport insurance policy.**

For computer equipment only:

Notice
OMICRON reserves the right to restore the computer to its original state of delivery.

- OMICRON does not accept any liability for the conservation or recovery of any data present on the computer, hard disk or any supplied data storage devices (e.g. measured data or licence information, etc.). We expect our customers to perform data backup procedures regularly. **In addition**, please carry out the following steps before shipping any computer equipment:
- If at all possible make a complete backup of all data present on your hard disk before shipping. If you need to supply a storage device (tape, disk, etc.) send a copy and keep the original.
- Make sure the computer can be run up in a stand-alone mode. This may mean that you uninstall/deactivate network configurations or external devices.
- Make sure the original passwords are re-installed or supply the current passwords by fax or e-mail.