

Vx Extract

Vx Extract™

GC/MS and LC/MS Information Extraction

Adron Systems LLC
Laporte, MN 56461
USA

June 15, 2010

Copyright© 2009 Adron Systems LLC. All rights reserved.

Motivation

The purpose of GC/MS and LC/MS data is to provide information. The task of extracting information is usually delegated to the data processing package provided by the instrument vendor. Data processing packages include Thermo Scientific Xcalibur, Varian MS WorkStation, and Agilent MSD ChemStation.

These GUI (graphical user interface) based applications are designed and optimized for predefined tasks. These targeted applications accomplish their tasks well; with a minimal learning curve. However, novel and cutting-edge data analysis methods don't fit within this framework; requiring capabilities beyond the scope of the vendor's package. The fields of metabolomics (metabolite profiling), proteomics (study of proteins), petroleum profiling, and pharmaceuticals often require specialized data processing.

Alternatives? In some cases, GC/MS and LC/MS data can be exported to numerical analysis, database, or spreadsheet packages for analysis and information extraction. Analysis packages such as MathWorks MatLab®, Wolfram Research Mathematica®, Maplesoft Maple®, and Microsoft Excel® have been used for extracting information from GC/MS and LC/MS data. These packages allow for programmable, customized solutions.

An alternate mechanism for extracting information from mass spectrometry data files is the Vx Extract™ software package from Adron Systems. Vx Extract provides an API (application programming interface) for accessing information from GC/MS and LC/MS data files. Vx Extract is a set of routines and classes that facilitate the building of custom data processing applications.

History

Vx Capture™ is Adron Systems' GC/MS and LC/MS data conversion package. In the conversion process, Vx Capture “reads” information from a vendor's data file and then “writes” this extracted information out to the desired target file format.

Vx Capture allows organizations to standardize and optimize their data processing needs; independent of GC/MS and LC/MS instrumentation. Vx Capture facilitates the export of GC/MS and LC/MS data to packages such as MathWorks MATLAB®, Maplesoft Maple®, and Microsoft Excel®.

The data access (“read”) routines from the Vx Capture file converter are the basis for Vx Extract. Routines and classes from Vx Capture are made accessible through the Vx Extract library package and its API.

Introduction

Vx Extract is a set of routines and classes callable from the Ruby scripting language; allowing simplified access and manipulation of GC/MS and LC/MS information. Combined with the power of the Ruby programming language, Vx Extract produces concise, adaptable solutions for the analysis of mass spectrometric data.

Vx Extract classes and methods are grouped into the Ruby module "ASE". The Vx Extract classes are:

<u>Class</u>	<u>Description</u>
ASE::Reader	Data access class (entry class) into GC/MS and LC/MS data files. Provides "reader" functionality for extracting information analogous to the Vx Capture file converter.
ASE::MI	Representation of a mass / intensity point. An array of these items represents a mass spectrum.
ASE::TI	Representation of a time / intensity point. An array of these items represents an ion chromatogram.
ASE::Time	Representation of time, i.e., a specific point in time. The acquisition (injection) time of data file is an example.

Each of these classes is described in dedicated sections in this document along with usage examples.

Vx Extract contains several Ruby scripts that illustrate the use of Ruby and the Vx Extract programming library. Reviewing these sample scripts is an effective way to learn Ruby and Vx Extract programming. These sample scripts can also be used as a foundation for your own programs.

In the following sections, you'll learn about the Ruby programming language, the installation of Vx Extract and Ruby, and how to get started with Vx Extract. Using Vx Extract and Ruby isn't difficult, especially if you study the various examples.

Ruby was designed to enhance programmer productivity and to be a pleasant, "fun" language to use. In this vein, we encourage you to experiment and play with Ruby and Vx Extract.

Let us know about your experiences with Vx Extract. If you have suggestions about possible enhancements, want to share your experiences or scripts, feel free to contact Adron Systems. You can e-mail Adron Systems at:

support@adronsystems.com

Adron Systems' contact information is available on the Internet at the following site:

<http://www.adronsystems.com/contactus.htm>

The Ruby Language

The Ruby programming language originated in Japan during the mid-1990s. Ruby was conceived, designed and initially developed by Yukihiro "Matz" Matsumoto as an improvement over the scripting languages Perl and Python. Ruby's syntax is similar to the "C" programming language but borrows aspects from a variety of other languages; notably Perl, Smalltalk, Eiffel, Ada, and Lisp.

Ruby is an object-oriented programming language but does not force this paradigm on the developer. As an interpreted language, it facilitates the rapid development of code. Ruby is designed to be extensible; allowing the addition of modules such as Vx Extract.

Ruby continues to gain popularity, in part due to the "Ruby on Rails" (Rails or ROR) open source web application framework. Its main competitors are the Python and Perl scripting languages.

The primary website for the Ruby programming language information is:

<http://www.ruby-lang.org/>

The Ruby installer download page is located at:

<http://www.ruby-lang.org/en/downloads/>

Vx Extract utilizes the Ruby on Windows version labeled as the "One-Click Installer". This version of Ruby is compatible with the development environment used for building the Vx Extract Ruby extension.

Ruby code can be developed in a simple text editor such as Windows Notepad. However, we recommend the use of a programmer's text editor, such as the free and open-source editor [Komodo Edit](#) from [ActiveState](#).

Documentation for Ruby is available on the above website. The book "*Programming Ruby: The Pragmatic Programmers' Guide*", known as the "Pick Axe" due to its cover art, is the recommend reference book on Ruby. The first edition of the book is available on-line at the following site:

<http://www.rubycentral.com/book/>

We relied on the second edition of the book (ISBN: 978-0974514055) while developing the Vx Extract library. This edition includes coverage of Ruby "1.8". As a suggestion, read Chapter 22, "The Ruby Language" before writing code; especially those sections related to naming conventions.

Another resource is *The Ruby Cookbook* from O'Reilly Media (ISBN: 978-0596523695); full of concise examples on how to use Ruby's standard libraries and popular extensions.

Installation

The Vx Extract and Ruby installers can be obtained from the Internet or directly from Adron Systems on CD-ROM disc.

- 1) Install the Ruby interpreter. On the Internet, go to the following website:

<http://www.ruby-lang.org/en/downloads/>

Select the Ruby package labeled as the “One-Click Installer.” As of this writing, this is the [ruby186-27_rc2.exe](#) package.

Note: Adron Systems' goal is to track Vx Extract versions with the “One-Click Installer” versions of Ruby available on the Internet.

Alternatively, on the Vx Acquisition System CD-ROM disc, select one of the Ruby installers located in the Ruby directory.

- 2) Run the Vx Acquisition System installer. The Vx installer can be obtained on the Internet by filling out the following form:

http://www.adronsystems.com/form_vxext.htm

Note: Vx Extract is a component of the Vx Acquisition System.

Alternatively, on the Vx CD-ROM disc, run the `Setup.exe` program found on the disc's root directory.

- 3) If you purchased or have an evaluation copy of Vx Extract on CD-ROM disc, install the Aladdin HASP driver. The installer, `HASPUserSetup.exe`, is located under the `Aladdin` directory on the Vx CD-ROM disc. After installing the driver, insert the software key into an available USB slot. The software key will light red when successfully installed.

Installation can be verified by using the Vx Configure program. A shortcut to this program is located under the “Adron Systems” folder. Double-click the “Software Key” item to view the key settings.

Without the software key, Vx Extract runs in “demo” mode with restricted, limited access to the GC/MS and LC/MS data file set.

Environment Variables

Vx Extract requires additions to Windows' environment variables. As an option, these modifications can be performed by the Vx installer.

Note: The Vx installer modifies environment variables for all users.

Vx Extract users can choose to modify these variables manually. (For directions, use a search engine on the phrases "Environment Variables" and "System Properties" along with the name of your PC's operating system.)

- 1) The environment variable VxDIR is set to the installation (root) directory for Vx Extract. Vx programs use VxDIR to locate directories and settings. VxDIR is usually set to one of the following values:

```
VxDIR=c:\Adron Systems\Vx
```

or

```
VxDIR=d:\Adron Systems\Vx
```

Depending on your installation choices, this can vary.

- 2) The RUBYLIB environment variable aids the Ruby interpreter in locating source and library files. For Vx Extract, this variable is set as:

```
rubylib=%VxDIR%\Programs;%VxDIR%\Macros;
```

If you create Ruby scripts in other directories, you may want to add these directories to the RUBYLIB variable.

- 3) The PATH environment variable contains a semicolon-delimited list of directories where the command interpreter will search for executable and script files. The PATH environment variable is modified by appending the following items:

```
%VxDIR%\Programs;%VxDIR%\Macros;
```

These modifications allow the command interpreter to locate Ruby scripts without entering complete path information. For example:

```
FileInfo.rb "d:\My Data\Westman\C8485R12.TKF"
```

instead of

```
d:\Adron Systems\Vx\Macros\FileInfo.rb "d:\My Data\Westman\C8485R12.TKF"
```

If you create Ruby scripts in other directories, you may want to append these directories to the PATH environment variable.

Getting Started

At this point, the “One-Click Installer” version of Ruby and the Vx Extract library should be installed. The appropriate environment variables should be configured as described on the preceding page. If you purchased or have an evaluation version of Vx Extract, then the Aladdin HASP device driver and license key should also be installed.

Open a command prompt, by using the Windows “Run...” dialog and entering “cmd” or by opening the Accessories folder, and selecting the “Command Prompt” choice. Type “FileInfo” or “FileInfo.rb” at the command prompt. This generates output similar to the following:

```
→ Data file header information display utility. (Release : 0.90.123)
→
→ FileInfo.rb InFileName
→
→ Copyright by Adron Systems LLC. All rights reserved.
```

Note: In this document, output is preceded with the “→” symbol.

Note: The extension “.rb” indicates the Ruby file type. This extension is optional when entering a Ruby script name at the Windows command prompt.

Note: If an error message is generated, verify your installation.

- 1) At the command prompt, enter “irb” for the “interactive Ruby shell”. If Ruby is installed correctly, this generates the following text:

```
→ irb(main):001:0>
```

Enter “quit” to leave the interactive Ruby shell. If you receive an error message, then Ruby wasn't installed correctly.

- 2) At the command prompt, enter “start placfg” to launch the “Vx Configure” program. If Vx Extract is installed correctly, a window entitled “Vx Configure” will appear.

Note: In Vx Configure, select the “Software Key” item to view the installed Vx options. This is useful if you installed the Vx USB software key.

- 3) Review the environment variables by entering “set|sort” at the command prompt. Review the RUBYLIB, PATH and VxDIR environment variables as described on the [prior page](#).

Alternatively, use the commands “set rubylib”, “set path” and “set vxdir” to review environment variables.

Vx Extract includes the sample GC/MS data file, "voldata.tkf". At the command prompt, enter the following:

```
fileinfo %vmdir%\Samples\voldata.tkf
```

The FileInfo.rb Ruby script extracts "header" information from the passed data file. The above command generates the following output:

```
→ d:\vx\Samples\voldata.tkf
→ Instrument      :
→ Injection Time  : 1991/02/08 12:29:20
→ Sample Info    : volstd 100ng
→ Sample Account :
→ Quality Control :
→ Operator       :
→ Calibration Name: D:\VECTOR2\CAL\tune.CAL
→ Method Name    : D:\VECTOR2\CAL\vol.mth
→ Solvent Scans  : 6
→ Scans (spectra) : 2538
→ Vial Number    : 0
→ Injection volume: 1
→ Description >>
→ >> Volatile Organic standard 100 ng
→ >> CyroTrap Analysis
→ >>
```

Try using the FileInfo.rb script on your own GC/MS and LC/MS data files.

The following Ruby code is taken from the ExtractDemo.rb script, with line numbering added:

```
1.  #!/usr/bin/env ruby
2.
3.  require 'ExtractLoad'
4.
5.  printf("Vx Extract version  : %s\n", ASE::EXTRACT_RELEASE)
6.  printf("Development company : %s\n", ASE::ADRON_COMPANY)
7.  puts
8.
9.  tmNow = ASE::Time.new
10. strNow = tmNow.imageLocal(ASE::Time.FORMAT_LONG_DATETIME)
11. printf("Current, local time : %s\n", strNow)
```

The *very first line* of your Vx Extract program must contain the text shown in line #1. On Windows, the command processor looks for the Ruby interpreter in the "env", i.e., along the PATH environment variable. (In the Unix world, this is called the "shebang" line.)

Line #3 loads the Vx Extract package. All Ruby scripts that utilize the Vx Extract library require the text of line #3.

Here is sample output from running the ExtractDemo.rb script:

```
→ Vx Extract version   : 0.90.123
→ Development company : Adron Systems LLC
→
→ Current, local time : Thr May 28, 2009 13:42:06.203
```

Lines #5 and #6 output the current version of Vx Extract, along with the company name. `EXTRACT_RELEASE` and `ADRON_COMPANY` are string constants defined within the ASE module.

Line #9 creates a Ruby object representing the current time. Line #10 turns this time object into a nicely formatted string. Finally, line #11 outputs this string.

In summary, each Vx Extract program must contain the “shebang” line and “ExtractLoad” require line. Application specific code then follows these lines.

Vx Extract comes with a number of sample scripts. These are located in the `Macros` directory:

```
c:\Adron Systems\Vx\Macros
or
d:\Adron Systems\Vx\Macros
```

Depending on your installation choices, this can vary. *We suggest adding your own scripts to this directory.*

To become familiar with Vx Extract, we suggest study of these sample scripts along with the descriptions of the Vx Extract classes and their member functions.

<u>Script</u>	<u>Description</u>
<u>AdronText.rb</u>	Convert a data file to the Adron Text (ASC) format.
<u>Channels.rb</u>	Extract monitor channel information.
<u>ChromCalc.rb</u>	Ion chromatogram extraction using RPN logic.
<u>ExtractDemo.rb</u>	A “getting started” demonstration program illustrating Ruby and Vx Extract usage.
<u>FileInfo.rb</u>	Extract file (header) information.
<u>Spectra.rb</u>	Extract one or more spectra from a GC/MS or LC/MS data file.
<u>TimeFormats.rb</u>	Output all the time image formatting strings from the ASE::Time class.

The remainder of this document describes the Vx Extract classes, its member functions, along with usage examples.

Description

An object of the ASE::MI class holds a “mass / intensity” data point. The “MI” name is derived from the first letters of the words “mass” and “intensity.”

A mass spectrum in the Vx Extract software package is an array of ASE::MI points.

The base peak of a mass spectrum is a single ASE::MI object.

Class Methods

new

ASE::MI.new → *mi*

ASE::MI.new(*ion_mass*, *counts*) → *mi*

Return an ASE::MI mass / intensity object.

The first form generates an ASE::MI object initialized to zero.

The second form initializes the ASE::MI object to the passed ion m/z (*ion_mass*) and intensity (*counts*) values.

Note: The ASE::MI constructors are generally not needed because various ASE::Reader methods generate ASE::MI objects.

```
# Create ASE::MI objects using the two constructor forms.
mi = ASE::MI.new
mi_C10H8 = ASE::MI.new( 128.0626, 1000 )

# Print out contents of the ASE::MI objects.
printf("mi -> %10.4f : %8d\n", mi.mass, mi.intensity)
printf("mi -> %10.4f : %8d\n", mi_C10H8.mass, mi_C10H8.intensity)

-> mi      ->    0.0000 :      0
-> mi_C10H8 ->  128.0626 :   1000
```

mass *mi.mass* → *ion_mass*

mass=

mi.mass = ion_mass

ASE::MI methods to retrieve and to set the ion m/z value for the ASE::MI object referenced by *mi*.

```
# Create an ASE::MI object and initialize fields.
mi_C12H8S = ASE::MI.new
mi_C12H8S.mass = 184.0347
mi_C12H8S.intensity = 1000
```

```
# Print out contents of the ASE::MI object.
printf("mi_C10H8 -> %10.4f : %8d\n", mi_C12H8S.mass, mi_C12H8S.intensity)

-> mi_C10H8 -> 184.0347 : 1000
```

intensity*mi.intensity* → *counts***intensity=***mi.intensity = counts*

ASE::MI methods to retrieve and to set the intensity value for the ASE::MI object referenced by *mi*.

```
# Create an ASE::MI object and initialize fields.
mi_C25H3805 = ASE::MI.new
mi_C25H3805.mass = 418.272
mi_C25H3805.intensity = 1000

# Print out contents of the ASE::MI object.
printf("mi_C10H8 -> %10.4f : %8d\n", mi_C25H3805.mass, mi_C25H3805.intensity)

-> mi_C10H8 -> 418.2720 : 1000
```

MIN_MASSASE::MI::MIN_MASS → *ion_mass***MAX_MASS**ASE::MI::MAX_MASS → *ion_mass***SCALE_FACTOR_MASS**ASE::MI::SCALE_FACTOR_MASS → *res_factor*

The MIN_MASS and MAX_MASS constants specify the allowable mass (m/z) range for the ASE::MI class.

The inverse of the SCALE_FACTOR_MASS constant specifies the mass resolution supported by the ASE::MI class.

```
printf("Mass resolution factor      : %g\n", 1.0 / ASE::MI::SCALE_FACTOR_MASS)
printf("Minimum ion mass (m/z) value : %6d\n", ASE::MI::MIN_MASS)
printf("Maximum ion mass (m/z) value  : %6d\n", ASE::MI::MAX_MASS)

-> Mass resolution factor      : 0.0001
-> Minimum ion mass (m/z) value : 0
-> Maximum ion mass (m/z) value : 429496
```

Description

An object of the ASE::TI class holds a “time / intensity” data point. The “TI” name is derived from the first letters of the words “time” and “intensity.”

An ion chromatogram in the Vx Extract software package is an array of ASE::TI points.

Class Methods

new

ASE::TI.new → *ti*

ASE::TI.new(*itv*, *counts*) → *ti*

Return an ASE::TI time / intensity object.

The first form generates an ASE::TI object initialized to zero.

The second form initializes the ASE::TI object to the passed interval (*itv*) and intensity (*counts*) values.

Note: The ASE::Reader.ExtractChromatogram method, described below, creates ASE::TI objects when extracting ion chromatograms. The ASE::TI constructors are generally not used because other methods create these objects.

```
# Generate ASE::TI objects.
ti_zero = ASE::TI.new
ti_set = ASE::TI.new( 60, 1000 )

# Print out contents of ASE::TI objects.
printf("ti_zero -> %10.4f : %8d\n", ti_zero.interval, ti_zero.intensity)
printf("ti_set -> %10.4f : %8d\n", ti_set.interval , ti_set.intensity )

-> ti_zero -> 0.0000 : 0
-> ti_set -> 60.0000 : 1000
```

interval*ti.interval* → *itv***interval=***ti.interval = itv*

These are the ASE::TI methods used to retrieve and set the interval value for the ASE::TI object named *ti*.

```
# Create an ASE::TI object and set the field values.
ti = ASE::TI.new
ti.interval = 232.0440
ti.intensity = 1029414

# Print out contents of ASE::TI ti object.
# This retrieves the ASE::TI fields "interval" and "intensity".
printf("ti -> %10.4f : %8d\n", ti.interval, ti.intensity)

→ ti -> 232.0440 : 1029414
```

intensity*ti.intensity* → *counts***intensity=***ti.intensity = counts*

These are the ASE::TI methods used to retrieve and set the intensity value for the ASE::TI object named *ti*.

```
# Create an ASE::TI object and set the field values.
ti = ASE::TI.new
ti.interval = 70.0620
ti.intensity = 101444

# Print out contents of ASE::TI ti object.
# This retrieves the ASE::TI fields "interval" and "intensity".
printf("ti -> %10.4f : %8d\n", ti.interval, ti.intensity)

→ ti -> 70.0620 : 101444
```

MIN_INTENSITYASE::MIN_INTENSITY → *counts***MAX_INTENSITY**ASE::MAX_INTENSITY → *counts*

The MIN_INTENSITY and MAX_INTENSITY constants specify the allowable intensity range for the ASE::MI and ASE::TI classes.

```
printf("Minimum allowed intensity value : %d\n", ASE::MIN_INTENSITY)
printf("Maximum allowed intensity value : %d\n", ASE::MAX_INTENSITY)

→ Minimum allowed intensity value : -9223372036854775808
→ Maximum allowed intensity value : 9223372036854775807
```

Description

The ASE::Reader class is the primary interface for extracting information from GC/MS and LC/MS data files using the Vx Extract software package. The ASE::Reader class is designed to effectively and easily extract various types of information from your MS data files.

The total-ion-chromatogram (TIC), single ion chromatograms, and multiple ion chromatograms are readily extracted. The Ruby language allows you to manipulate these results in a myriad of ways. Extraction can be done on a per scan basis or over all scans.

Mass spectra can be extracted with either nominal integer masses or with data file stored mass values.

Additionally, information fields, such as injection time, operator name, and so forth can be extracted.

Class methods

new ASE::Reader.new(*kindtype*, *filename*) → *rd*
 ASE::Reader.new(*filename*) → *rd*

Return an ASE::Reader object, *rd*, by opening the data file(s) associated with *filename*. The type of data file to open is specified by *kindtype*. Possible values for *kindtype*:

<u><i>kindtype</i></u>	<u>Description</u>
TKF	Adron Systems, ProLab, and Teknivent Vector/2 and Vx (TKF)
ASC	Adron Systems Text File (ASC)
HPD	Agilent & HP ChemStation (.D/Data.MS, .D/Data.CDF)
CDF	AIA Andi netCDF (CDF)
MIMX	Finnigan Incos Mass Intensity Files (MI/MX)
TITX	Finnigan Incos Time Intensity Files (TI/TX/CT)
ITS	Finnigan ITS40, Magnum & Saturn I, II, III (MS)
SMS	Varian Workstation (SMS)
XMS	Varian Workstation (XMS)

```
# Open a GC/MS data file for reading.
rd = ASE::Reader.new('TKF', 'd:\my data\westman\c9493s3.tkf')
```

```
# Alternate create mechanism; easier but not as efficient.
rd = ASE::Reader.new('d:\my data\westman\c9493s3.tkf')
```

basePeakOfScan*rdr.basePeakOfScan(boolean, index) → mi*

Returns the base peak for the specified spectrum as an ASE::MI value.

If *boolean* is `true`, then a nominal m/z value is returned. If *boolean* is `false`, then the m/z value stored in the data file is returned.

A particular mass scan (spectrum) is specified with the *index* value. Index values range from 0 to the scan count – 1. A negative index counts backward from the end of the data scans with -1 specifying the last scan.

rdr is an ASE::Reader object created using the ASE::Reader.new method.

```
# Print out base peaks for all scans in the data file.
if (cOfScans = rdr.numberOfScans)
  scan_range=0..cOfScans-1
  scan_range.each do |i|
    pk_base = rdr.basePeakOfScan(true, i)
    printf("Base peak for index %3d is ", i)
    printf("mass(%4g) : intensity(%8d)\n", pk_base.mass, pk_base.intensity)
  end
end
```

completeScanRange*rdr.completeScanRange → range_mass*

Return a Ruby range object containing the mass range of the data file. Nominal m/z values are returned for *range_mass*. If supported by the data file, the mass range returned is the experimental scan range. Otherwise, the returned mass range is the low and high m/z values found in the data file.

rdr is an ASE::Reader object created using the ASE::Reader.new method.

```
# Print mass range for acquired spectra.
range_mass = rdr.completeScanRange
printf("Data scan range is from %d to %d\n", range_mass.first, range_mass.last)

→ Data scan range is from 35 to 650
```

description*rdr.description* → *info_line*

Returns *info_line* of descriptive text, i.e., the comment field for the data file.

rdr is an ASE::Reader object created using the ASE::Reader.new method.

```
# Print the data file comment field.
printf("Description : %s\n", rdr.description)
```

```
→ Submitter : T.J. Taormino
```

descriptionLines*rdr.descriptionLines* → *ary_info_lines*

Some data file formats support multiple comment lines. This method returns *ary_info_lines* of descriptive text as an array of strings.

rdr is an ASE::Reader object created using the ASE::Reader.new method.

```
# Print out complete listing of description lines.
lines = rdr.descriptionLines
lines.each do |line|
  print( line, "\n" )
end
```

```
→ Submitter : T.J. Taormino
→ Misc Info : Pacific Green Tea Extract
→ Column : 30m X 0.25mmID X 0.20umdf SP-2380 (SUPELCO #7810-03B)
→ Carrier : He @ 3.1psi; 0.7ml/min;
→ Split Flow : 28 ml/min;
→ Split Ratio: 40:1
→ Avg Lin Vel: 30 cm/sec.
```

descriptionOfKind*ASE::Reader.descriptionOfKind(kindtype)* → *kind_info*

Returns *kind_info* descriptive text for the passed *kindtype* textual string. See the ASE::Reader::new method for *kindtype* values.

```
# Print descriptive text for the passed kindtype value.
printf("File type : %s\n", ASE::Reader.descriptionOfKind('MIMX'))
```

```
→ File type : Finnigan Incos (MI,MX)
```

extractChannel *rdr.extractChannel(indexChannel, indexScan) → channel_value*

Return the monitor channel value referenced by the passed channel and scan indices.

A monitor channel is specified with the *indexChannel* value. Index values range from 0 to the channel count – 1. A negative index counts backward from the end of the channels with -1 specifying the last channel.

A mass scan (spectrum) is specified with the *indexScan* value. Index values range from 0 to scan count – 1. A negative index counts backward from the end of the data scans with -1 specifying the last scan.

rdr is an ASE::Reader object created using the ASE::Reader.new method.

```
# Print out contents of all channels.
if (cOfScans = rdr.numberOfScans) && (cOfChannels = rdr.numberOfChannels)

  # Output a nice header.
  printf("%14s%3s", "Interval(s)", "")
  0.upto(cOfChannels-1) do |j|
    printf("%14s ", rdr.nameOfChannel( j ))
  end
  puts

  # Output scan interval plus all monitor channels.
  # This example does the upper 1/2 of all scans.
  (cOfScans>>1).upto(cOfScans-1) do |i|
    printf("%14.4f : ", rdr.intervalOfScan(i))
    0.upto(cOfChannels-1) do |j|
      printf("%14g ", rdr.extractChannel( j, i ))
    end
    puts
  end
end
end
```

```
→ Interval(s)      ScanIonTime      ScanPreTIC
→ 31.0630 :          8471          6409
→ 32.0470 :           540          3984
→ 33.0470 :           951          2087
→ 34.0470 :          1006          1966
→ 35.0470 :           975          2037
→ 36.0470 :           971          2031
→ 37.0470 :           955          2068
→ 38.0470 :           975          2032
→ 39.0470 :           979          2019
→ 40.0470 :           935          2127
→ 41.0470 :           947          2082
→ 42.0470 :           978          2027
→ 43.0470 :           954          2071
→ 44.0470 :           895          2209
→ 45.0470 :           967          2044
→ ...
```

extractChromatogram *rdr.extractChromatogram(ion_mass, ary_chrom) → count_points*

Extract an ion chromatogram from the data file using the *rdr* ASE:Reader object.

Returns the number of points extracted as *count_points*.

Extracted data points are stored in the passed *ary_chrom* array.

The m/z value to extract is specified by the nominal *ion_mass* m/z value. An *ion_mass* value of ASE::TIC specifies the TIC (total-ion-chromatogram).

Note: For efficiency, the *ary_chrom* array should be preallocated with the data file scan count. See the example below for suggested usage.

rdr is an ASE::Reader object created using the ASE::Reader.new method.

```
# Determine number of scans in data file.
cOfScans = rdr.numberOfScans

# Preallocate a "TIC" array, and an "Ion" array.
aTIC=Array.new(cOfScans)
aIon=Array.new(cOfScans)

# Extract the TIC and 131 m/z ion chromatograms.
rdr.extractChromatogram( ASE::TIC, aTIC )
rdr.extractChromatogram( 131, aIon )

# Print out the ion chromatogram time and intensity values.
0.upto(cOfScans-1) do |i|
  printf("%10.4f : ", aTIC[i].interval)
  printf("%8d %6d\n", aTIC[i].intensity, aIon[i].intensity)
end

→      0.0000 :    102323    4226
→      0.5603 :     82806    5530
→      1.1207 :     80203    4245
→      1.6810 :     77979    5411
→      2.2418 :     79630    4850
→      2.8018 :    100694         0
→      3.3623 :     90524    5841
→      3.9223 :    118383    4957
→      4.4828 :     99971    3979
→      5.0430 :    130733    5929
→      5.6033 :    101786    4125
→      6.1638 :    110569    3383
→      6.7245 :     95022    5962
→      7.2858 :     83734         0
→      7.8460 :     90499    5330
→      8.4065 :     91223    5805
→ ...
```

extractIon*rdr.extractIon(index, mass_value) → ion_sum*

Returns an ion intensity value or a summation of ion intensities depending on the contents of *mass_value*.

If *mass_value* contains a nominal, single m/z value, then that ion intensity is returned. An *mass_value* value of ASE::TIC extracts the TIC (total-ion-chromatogram) value.

If *mass_value* is a range of nominal m/z values, then the intensity summation for that ion range is returned.

If *mass_value* is an array of nominal m/z values, then the corresponding ion intensities are summed together and returned.

Note: A sorted array of m/z values optimizes extraction efficiency.

A particular mass scan (spectrum) is specified with the *index* value. Index values range from 0 to the scan count – 1. A negative index counts backward from the end of the data scans with -1 specifying the last scan.

rdr is an ASE::Reader object created using the ASE::Reader.new method.

```
# Example of ion m/z values.
masssingle=131
massrange=100..200
masslist=[55, 69, 83, 111, 126, 131, 252]

# Print ion intensity and ion summation values over all scans.
cOfScans = rdr.numberOfScans
0.upto(cOfScans-1) do |i|
  printf("%4d > ", i)
  printf("Single m/z : %6d ",    rdr.extractIon(i, masssingle))
  printf("Ion range sum : %6d ",  rdr.extractIon(i, massrange ))
  printf("Ion list  sum : %6d\n",  rdr.extractIon(i, masslist  ))
end

→   0 > Single m/z :   4226 Ion range sum : 12608 Ion list  sum : 14997
→   1 > Single m/z :   5530 Ion range sum : 15996 Ion list  sum : 14767
→   2 > Single m/z :   4245 Ion range sum : 10939 Ion list  sum : 13482
→   3 > Single m/z :   5411 Ion range sum :   5411 Ion list  sum : 12901
→   4 > Single m/z :   4850 Ion range sum : 14190 Ion list  sum : 11657
→   5 > Single m/z :     0 Ion range sum :   7790 Ion list  sum :   9126
→   6 > Single m/z :   5841 Ion range sum : 12310 Ion list  sum : 14940
→   7 > Single m/z :   4957 Ion range sum : 23903 Ion list  sum : 12523
→   8 > Single m/z :   3979 Ion range sum : 11185 Ion list  sum : 13537
→   9 > Single m/z :   5929 Ion range sum : 18283 Ion list  sum : 15733
→  10 > Single m/z :   4125 Ion range sum :   9822 Ion list  sum :   4125
→ ...
```

extractIons*rdr.extractIons(index, ary_mass, ary_intensity)*

Extract a list of ion intensities corresponding to a passed list of m/z values.

A particular mass scan (spectrum) is specified with the *index* value. Index values range from 0 to the scan count - 1. A negative index counts backward from the end of the data scans with -1 specifying the last scan.

Nominal ion m/z values are passed in the *ary_mass* array.

The ion intensity values are returned to the *ary_intensity* array.

rdr is an ASE::Reader object created using the ASE::Reader.new method.

```
# List of m/z values.
ary_mass=[55, 69, 83, 111, 126, 131, 252]

# Place to store retrieved ion intensities.
ary_intensity=Array.new(ary_mass.size)

# Loop through all scans, extracting intensity values.
0.upto(rdr.numberOfScans-1) do |i|
  rdr.extractIons(i, ary_mass, ary_intensity)
  printf("%4d > ", i)
  ary_intensity.each do |j| printf(" %8d", j) end
  puts
end

→ ...
→ 370 >      63560      45202      343330      15441      16762          0          0
→ 371 >     105940      61601      338270      34427      38510       2954      2699
→ 372 >     112310     117930     362730     33889     46520      8460     2355
→ 373 >     143400     106090     270420     30337     40988      9026          0
→ 374 >     237390     135740     196980     75540     67950      9268     2844
→ 375 >     229250     111300     201000     51360     61420     11682          0
→ 376 >     262350      99800     175260     45963     57210      7937     1493
→ 377 >     234870     107950     168370     47230     79860      8960          0
→ 378 >     214790      92830     133290     70750     55827      6497          0
→ 379 >       97712     117730     159090     84940     53660      7325          0
→ 380 >     215170     103620     113260     50300     59200      5055          0
→ ...
```

extractScan*rdr.extractScan(boolean, index, ary_peaks) → count_peaks*

Extract a mass spectrum from the data file using the ASE::Reader object *rdr*.

The returned *count_peaks* value is the count of mass / intensity pairs for the spectrum.

If *boolean* is `true`, then nominal m/z values are returned. If *boolean* is `false`, then the m/z values stored in the data file are returned.

A particular mass scan (spectrum) is specified with the *index* value. Index values range from 0 to the scan count – 1. A negative index counts backward from the end of the data scans with -1 specifying the last scan.

A mass spectrum is an array (list) of mass / intensity pairs. The retrieved pairs of mass / intensities are stored in the *ary_peaks* array as ASE::MI objects.

```
# Preallocate buffer to hold peaks.
ary_peaks=Array.new(50)

# Extract last spectrum (index = -1) and print the mass/intensity pairs.
if (count_peaks = rdr.extractScan( true, -1, ary_peaks))
  0.upto(count_peaks-1) do |i|
    printf( "%7.3f : %7d\n", ary_peaks[i].mass, ary_peaks[i].intensity )
  end
  puts
end
```

<u>rdr.ExtractScan(true, -1, ary_peaks)</u>	<u>rdr.ExtractScan(false, -1, ary_peaks)</u>
→ 39.000 : 3914	→ 39.029 : 3914
→ 40.000 : 79815	→ 39.956 : 79815
→ 41.000 : 19881	→ 41.018 : 19881
→ 43.000 : 3976	→ 42.889 : 3976
→ 44.000 : 6865	→ 43.961 : 7226
→ 50.000 : 3558	→ 44.070 : 6505
→ 52.000 : 2314	→ 50.046 : 3558
→ 55.000 : 3922	→ 52.071 : 2314
→ 57.000 : 1559	→ 54.606 : 3922
→ 63.000 : 2056	→ 57.046 : 1559
→ 68.000 : 2504	→ 63.103 : 2056
→ 69.000 : 8673	→ 68.113 : 2504
→ 77.000 : 4422	→ 69.079 : 8673
→ 78.000 : 11837	→ 77.092 : 4422
→ 119.000 : 1274	→ 78.098 : 11837
→ 131.000 : 7621	→ 118.970 : 1274
→ 137.000 : 3927	→ 131.032 : 7621
→ 146.000 : 4735	→ 137.150 : 3927
→ 181.000 : 5151	→ 146.100 : 4735
	→ 181.100 : 5151

injectionVolume*rdr.injectionVolume* → *vol*

Return the injection volume *vol* from the data file using the ASE::Reader object *rdr*.

```
# Print out sample injection volume.
printf("Injection volume : %g\n", rdr.injectionVolume)
```

```
→ Injection volume : 2.1
```

intervalOfScan*rdr.intervalOfScan(index)* → *itv*

Return the time interval *itv*, in seconds, for the specified scan.

A particular mass scan (spectrum) is specified with the *index* value. Index values range from 0 to the scan count – 1. A negative index counts backward from the end of the data scans with -1 specifying the last scan.

rdr is an ASE::Reader object created using the ASE::Reader.new method.

```
# Extract interval for last scan in data file.
itv = rdr.intervalOfScan( -1 )
printf("Interval for last scan is %g seconds or %g minutes", itv, itv/60.0)
puts
```

```
→ Interval for last scan is 4199.6 seconds or 69.9933 minutes
```

isProfile*rdr.isProfile* → *bool*

Return `true` if mass spectra are stored in “profile” or “raw” mode. In profile mode, mass spectra are not run through a peak detector.

Return `false` if mass spectra are stored as peak detected data. This is usually a peak height or a peak summation of intensity values.

rdr is an ASE::Reader object created using the ASE::Reader.new method.

```
# Print out the data storage mode for mass spectra.
printf("Scanning mode : %s\n", rdr.isProfile ? "Profile" : "Detected" )
```

```
→ Scanning mode : Detected
```

kind*rdr.kind* → *kindtype*

Return the *kindtype* for the data file. See the ASE::Reader.new method for possible values.

```
# Open a GC/MS data file for reading.
rdr = ASE::Reader.new('TKF', 'd:\\my data\\westman\\c9493s3.tkf')
printf("File type : %s\\n", ASE::Reader.DescriptionOfKind(rdr.kind))
puts
```

→ File type : Adron Systems Vector/2 and Vx (TKF)

nameOfCalibration*rdr.nameOfCalibration* → *cal_file*

Return the name of the calibration file associated with the data file. The data file is referenced through the ASE::Reader *rdr* object.

```
# Print out name of calibration file.
printf("Calibration file name : %s\\n", rdr.nameOfCalibration)
```

→ Calibration file name : D:\\VECTOR2\\INSTR2\\METHODS\\062799GC.CAL

nameOfChannel*rdr.nameOfChannel(index)* → *name_channel*

Return the data file monitor channel name specified by the index value.

nameOfFile*rdr.nameOfFile* → *data_file*

Return the name of the data file associated with ASE::Reader object *rdr*.

```
# Print out the data file name.
printf("Name of file : %s\\n", rdr.nameOfFile)
```

→ Name of file : d:\\my data\\westman\\c9493s3.tkf

nameOfInstrument*rdr.nameOfInstrument* → *name_instr*

Return the name of the instrument used to acquire the data. The data file is referenced through the ASE::Reader *rdr* object.

```
# Print the instrument name.
printf("Name of instrument : %s\\n", rdr.nameOfInstrument)
```

→ Name of instrument : Quad FM 4500, Rm A

nameOfMethod*rdr.nameOfMethod* → *meth_file*

Return the name of the acquisition method file associated with the data file.

rdr is an ASE::Reader object created using the ASE::Reader.new method.

```
# Print the name of the acquisition method file.
printf("Method file name : %s\n", rdr.nameOfMethod)

→ Method file name : D:\VECTOR2\INSTR2\METHODS\E9-65-GC.MTH
```

nameOfOperator*rdr.nameOfOperator* → *operator*

Return the name of the operator responsible for the GC/MS data file. The data file is referenced through the ASE::Reader *rdr* object.

```
# Print out the operator name.
printf("Name of operator : %s\n", rdr.nameOfOperator)

→ Name of operator : A. P. Taormino
```

nameOfSample*rdr.nameOfSample* → *sample*

Return the sample name stored in the data file associated with ASE::Reader object *rdr*.

```
# Print the sample name.
printf("Name of sample : %s\n", rdr.nameOfSample)

→ Name of sample : X22789-09-02
```

nameOfSampleAccount*rdr.nameOfSampleAccount* → *account*

Return "account" information associated with the sample. The data file is referenced through the ASE::Reader *rdr* object.

```
# Print the sample account information.
printf("Sample account : %s\n", rdr.nameOfSampleAccount)

→ Sample account : ACE-TRIS-0000-0047
```

numberOfChannels*rdr.numberOfChannels* → *count_channels*

Return the number of monitor channels stored in the data file.

numberOfScans*rdr.numberOfScans* → *count_scans*

Return the number of scans (spectra) in the data file. The data file is referenced through the ASE::Reader *rdr* object.

The *count_scans* value is used extensively in *Vx Extract* programs since the maximum scan index value is *count_scans*-1. Consider this example:

```
1.  if ( count_scans = rdr.numberOfScans )
2.    0.upto(count_scans-1) do |i|
3.      # Looping through all scans.
4.      # Do some action.
5.    end
6.  end
```

In line #1, the number of scans is stored in the *count_scans* variable. If this value is non-zero, i.e., there are scans present, then safely go onto line #2.

Line #2 iterates through all scan indices, starting at zero and going to the maximum value. Do data extraction and processing in this loop; replace lines #3 and #4 with your code.

```
# Print out number of scans (spectra) in data file.
count_scans = rdr.numberOfScans
printf("Number of scans : %d\n", count_scans)
```

→ Number of scans : 7495

numberOfSolventScans*rdr.numberOfSolventScans* → *count_solvent_scans*

Return the number of solvent scans in the data file.

Solvent scans are initial scans in the data file without any mass / intensity peaks. Most often this is because the MS filament is off while solvent from the column is passing through.

The data file is referenced through the ASE::Reader *rdr* object.

```
# Print out the number of solvent scans.
printf("Number of solvent scans : %d\n", rdr.numberOfSolventScans)
```

→ Number of solvent scans : 240

qcType*rdr.qcType* → *qc_string*

Return the quality control string associated with the sample. This value may be used by data processing or reporting packages.

This string is typically one of the following:

- "blank"
- "calibration"
- "CC"
- "duplicate"
- "unknown"

```
# Print out the quality control string.
printf("Quality control type : %s\n", rdr.qcType)
```

scanAtInterval*rdr.scanAtInterval(boolean, itv)* → *scan_index*

Return the scan index (spectral index) matching the passed criteria.

The time interval from the experiment start is specified by *itv* in seconds.

If *boolean* is *false*, then the scan interval exactly matching *itv*, or earlier is returned. (The interval of the scan returned is equal to or less than *itv*.)

If *boolean* is *true*, then the scan interval exactly matching *itv*, or later is returned. (The interval of the scan returned is equal to or greater than *itv*.)

```
# Locate scan indices for times 10 to 20 minutes.
# Note: Converting from seconds to minutes using 60*.
range_itv = (10..20)
index_first = rdr.scanAtInterval( false, 60*range_itv.first )
index_last  = rdr.scanAtInterval( true , 60*range_itv.last  )

# Retrieve actual scan intervals for found indices.
itv_first  = rdr.intervalOfScan( index_first )
itv_last   = rdr.intervalOfScan( index_last  )

# Print out found indices and actual scan interval in minutes.
# Note: Converting from minutes to seconds using /60.
printf( "At index %d, interval is %g minutes\n", index_first, itv_first / 60.0 )
printf( "At index %d, interval is %g minutes\n", index_last , itv_last  / 60.0 )
```

```
→ At index 1070, interval is 9.9952 minutes
→ At index 2142, interval is 20.0078 minutes
```

Notice that requested scan range of 10 to 20 minutes lies within the retrieved intervals. This results from the diligent use of the *boolean* variable in the *IntervalOfScan* call.

summationOfScan*rdr.summationOfScan(index) → sum*

Return the summation, *sum*, of spectral intensities for the specified scan. This is also called the total-ion-chromatogram, or TIC value.

A particular mass scan (spectrum) is specified with the *index* value. Index values range from 0 to the scan count – 1. A negative index counts backward from the end of the data scans with -1 specifying the last scan.

rdr is an ASE::Reader object created using the ASE::Reader.new method.

```
# Locate scan indices for times 7.8 to 8.0 minutes.
range_itv = (7.8..8)
index_first = rdr.scanAtInterval( false, 60*range_itv.first )
index_last = rdr.scanAtInterval( true , 60*range_itv.last )

# Print out table with scan index, interval and TIC values.
index_first.upto(index_last) do |index|
  itv = rdr.intervalOfScan(index)
  sum = rdr.summationOfScan(index)
  printf("Scan %5d : %8.4f : %8d\n", index, itv/60.0, sum )
end
```

```
→ Scan 834 : 7.7907 : 244256
→ Scan 835 : 7.8001 : 379461
→ Scan 836 : 7.8094 : 569242
→ Scan 837 : 7.8188 : 884815
→ Scan 838 : 7.8281 : 926530
→ Scan 839 : 7.8374 : 1153836
→ Scan 840 : 7.8468 : 1497164
→ Scan 841 : 7.8561 : 1694518
→ Scan 842 : 7.8654 : 2003765
→ Scan 843 : 7.8748 : 2323941
→ Scan 844 : 7.8841 : 2530807
→ Scan 845 : 7.8935 : 3017163
→ Scan 846 : 7.9028 : 3227340
→ Scan 847 : 7.9121 : 3656465
→ Scan 848 : 7.9215 : 3812601
→ Scan 849 : 7.9308 : 4272890
→ Scan 850 : 7.9402 : 4286320
→ Scan 851 : 7.9495 : 4678904
→ Scan 852 : 7.9589 : 5103814
→ Scan 853 : 7.9682 : 5325485
→ Scan 854 : 7.9776 : 4369473
→ Scan 855 : 7.9869 : 3305499
→ Scan 856 : 7.9962 : 1973595
→ Scan 857 : 8.0056 : 1369787
```

timeOfInjection*rdr.timeOfInjection* → *inj_time*

Return the injection time, *inj_time*, as an ASE::Time object. The returned time is UTC or Coordinated Universal Time.

rdr is an ASE::Reader object created using the ASE::Reader.new method.

Note: See the ASE::Time section below for usage information.

```
# Open a GC/MS data file for reading.
rdr = ASE::Reader.new('TKF', 'd:\\my data\\westman\\c9494s3.tkf')

# Retrieve time-of-injection as an ASE::Time object
inj_time = rdr.timeOfInjection

# Display time in various example formats.
tm_str = inj_time.image(ASE::Time.FORMAT_ANDI_DATETIME)
printf("UTC injection time: %s\n", tm_str)

tm_str = inj_time.imageBiased(ASE::Time.FORMAT_VECTOR2_DATETIME, -300)
printf("Bias (-300 min) time: %s\n", tm_str)

tm_str = inj_time.imageLocal(ASE::Time.FORMAT_LONG_ORDERED_DATETIME)
printf("Local injection time: %s\n", tm_str)

→ UTC injection time: 19990526205447-0500
→ Bias (-300 min) time: Wed May 26 15:54:47 1999
→ Local injection time: 1999/05/26 15:54:47.000
```

vialNumber*rdr.vialNumber* → *vial_id*

Return the vial number, *vial_id*, associated with the data file. The data file is referenced through the ASE::Reader *rdr* object.

```
# Print out the vial number.
printf("Vial number : %d\n", rdr.vialNumber)

→ Vial number : 7
```

Description

An object of the ASE::Time class holds a time value. An ASE::Time object is returned when calling the ASE::Reader.TimeOfInjection method.

Class methods

new

ASE:Time.new → *tm*

Return an ASE::Time object, *tm*, set to the current time. The returned time is set to UTC (Coordinated Universal Time).

```
# Get the current time.
tm_now = ASE::Time.new

# Make and display a time string in local time.
img_local = tm_now.imageLocal(ASE::Time.FORMAT_LONG_DATETIME)
printf("Now as local time: %s\n", img_local)

# Make and display a time string in local time.
img_utc = tm_now.image(ASE::Time.FORMAT_LONG_DATETIME)
printf("Now as UTC time: %s\n", img_utc)

→ Now as local time: Tue May 12, 2009 11:07:41.953
→ Now as UTC time: Tue May 12, 2009 16:07:41.953
```

image

tm.image(*format_str*) → *time_str*

Return a string representation of the ASE::Time object, *tm*. The returned time string layout is determined by formatting information passed in the *format_str* string. See the *Format Specifiers* section below for usage.

```
# Retrieve the injection time from the data file.
rdr = ASE::Reader.new('TKF', 'd:\\my data\\westman\\c94794s2.tkf')
inj_time = rdr.timeOfInjection

# Use one of the predefined formatting strings.
format_str = ASE::Time.FORMAT_OPC_DATETIME

# Form the time string based on above format.
time_str = inj_time.image( format_str )

# Display the format string and the resulting time string.
printf("Format string : %s\n", format_str )
printf("UTC time string : %s\n", time_str )
puts

→ Format string : dd-mmm-yyyy hh:nn
→ UTC time string : 26-May-1999 20:54
```

imageBiased*tm.imageBiased(format_str, bias) → time_str*

Return a string representation of the ASE::Time object, *tm*, adjusted by the passed *bias* offset. The bias value, in minutes, allows for adjustment to different timezones. The value is signed, allowing for corrections east or west of UTC.

The returned time string layout is determined by formatting information passed in the *format_str* string. See the *Format Specifiers* section below for usage.

```
# Retrieve the injection time from the data file.
rdr = ASE::Reader.new('TKF', 'd:\\my data\\westman\\c94794s2.tkf')
inj_time = rdr.timeOfInjection

# Use one of the predefined formatting strings.
format_str = ASE::Time.FORMAT_LONG_ORDERED_DATETIME

# Form the time string based on above format.
time_utc_str = inj_time.image( format_str )
time_bias_str = inj_time.imageBiased( format_str, -300 )

# Display the format string and the resulting time strings.
printf("Format      string : %s\n", format_str )
printf("UTC      time string : %s\n", time_utc_str )
printf("Biased time string : %s\n", time_bias_str )
puts

→ Format      string : yyyy/mm/dd hh:nn:ss.zzz
→ UTC      time string : 1999/05/26 20:54:47.000
→ Biased time string : 1999/05/26 15:54:47.000
```

imageLocal*tm.imageLocal(format_str) → time_str*

Return a string representation of the ASE::Time object, *tm*, adjusted to local time.

The returned time string layout is determined by formatting information passed in the *format_str* string. See the *Format Specifiers* section below for usage.

```
# Retrieve the injection time from the data file.
rdr = ASE::Reader.new('TKF', 'd:\\my data\\westman\\c94794s2.tkf')
inj_time = rdr.timeOfInjection

# Form the local time string based on the passed format.
time_local_str = inj_time.imageLocal( ASE::Time.FORMAT_ORDERED_DATETIME )

# Display the resulting time string.
printf("Local time string : %s\n", time_local_str )

→ Local time string : 1999/05/26 15:54:47
```

theHour*tm.theHour* → *value*

Return the hour *value* of the ASE::Time object *tm*. Hours range from 0 to 23.

```
# Get the current time.
tm_now = ASE::Time.new

# Print out current UTC time plus 'TheHour' value.
printf("UTC time: %s\n", tm_now.image(ASE::Time.FORMAT_LONG_DATETIME))
printf("The Hour: %d\n", tm_now.theHour)

→ UTC time: Tue May 12, 2009 16:13:05.312
→ The Hour: 16
```

theMillisecond*tm.theMillisecond* → *value*

Return the millisecond *value* of the ASE::Time object *tm*. Milliseconds range from 0 to 999.

```
# Get the current time.
tm_now = ASE::Time.new

# Print out current UTC time plus 'TheMillisecond' value.
printf("UTC time: %s\n", tm_now.image(ASE::Time.FORMAT_LONG_DATETIME))
printf("The Millisecond: %d\n", tm_now.theMillisecond)

→ UTC time: Wed Apr 22, 2009 20:34:56.500
→ The Millisecond: 500
```

theMinute*tm.theMinute* → *value*

Return the minute *value* of the ASE::Time object *tm*. Minutes range from 0 to 59.

```
# Get the current time.
tm_now = ASE::Time.new

# Print out current UTC time plus 'TheMinute' value.
printf("UTC time: %s\n", tm_now.image(ASE::Time.FORMAT_LONG_DATETIME))
printf("The Minute: %d\n", tm_now.theMinute)

→ UTC time: Wed Apr 22, 2009 20:34:56.500
→ The Minute: 34
```

theMonth*tm.theMonth* → *value*

Return the month *value* of the ASE::Time object *tm*. Months range from 1 to 12 with the following value associations:

```
01 ↔ January
02 ↔ February
03 ↔ March
04 ↔ April
05 ↔ May
06 ↔ June
07 ↔ July
08 ↔ August
09 ↔ September
10 ↔ October
11 ↔ November
12 ↔ December
```

```
# Get the current time.
tm_now = ASE::Time.new

# Print out current UTC time plus 'TheMonth' value.
printf("UTC time: %s\n", tm_now.image(ASE::Time.FORMAT_LONG_DATETIME))
printf("The Month: %d\n", tm_now.theMonth)

→ UTC time: Wed Apr 22, 2009 20:34:56.500
→ The Month: 4
```

theMonthDay*tm.theMonthDay* → *value*

Return the month day *value* of the ASE::Time object *tm*. Month days range from 1 to 31.

```
# Get the current time.
tm_now = ASE::Time.new

# Print out current UTC time plus 'TheMonthDay' value.
printf("UTC time: %s\n", tm_now.image(ASE::Time.FORMAT_LONG_DATETIME))
printf("The Month Day: %d\n", tm_now.theMonthDay)

→ UTC time: Wed Apr 22, 2009 20:34:56.500
→ The Month Day: 22
```

theSecond*tm.theSecond* → *value*

Return the second *value* of the ASE::Time object *tm*. Seconds range from 0 to 59.

```
# Get the current time.
tm_now = ASE::Time.new

# Print out current UTC time plus 'TheSecond' value.
printf("UTC time: %s\n", tm_now.image(ASE::Time.FORMAT_LONG_DATETIME))
printf("The Second: %d\n", tm_now.theSecond)

→ UTC time: Wed Apr 22, 2009 20:34:56.500
→ The Second: 56
```

theWeekDay*tm.theWeekDay* → *value*

Return the week *value* of the ASE::Time object *tm*. Week days range from 1 to 7 with the following value associations:

- 1 ↔ Sunday
- 2 ↔ Monday
- 3 ↔ Tuesday
- 4 ↔ Wednesday
- 5 ↔ Thursday
- 6 ↔ Friday
- 7 ↔ Saturday

```
# Get the current time.
tm_now = ASE::Time.new

# Print out current UTC time plus 'TheWeekDay' value.
printf("UTC time: %s\n", tm_now.image(ASE::Time.FORMAT_LONG_DATETIME))
printf("The Week Day: %d\n", tm_now.theWeekDay)

→ UTC time: Wed Apr 22, 2009 20:34:56.500
→ The Week Day: 4
```

theYear*tm.theYear* → *value*

Return the year *value* of the ASE::Time object *tm*.

```
# Get the current time.
tm_now = ASE::Time.new

# Print out current UTC time plus 'TheYear' value.
printf("UTC time: %s\n", tm_now.image(ASE::Time.FORMAT_LONG_DATETIME))
printf("The Year: %d\n", tm_now.theYear)

→ UTC time: Wed Apr 22, 2009 20:34:56.500
→ The Year: 2009
```

theYearDay*tm.theYearDay* → *value*

Return the year day *value* of the ASE::Time object *tm*. Year days range from 1 to 365 except for leap years which range from 1 to 366.

```
# Get the current time.
tm_now = ASE::Time.new

# Print out current UTC time plus 'TheYearDay' value.
printf("UTC time: %s\n", tm_now.image(ASE::Time.FORMAT_LONG_DATETIME))
printf("The Year Day: %d\n", tm_now.theYearDay)

→ UTC time: Wed Apr 22, 2009 20:34:56.500
→ The Year Day: 112
```

Format Methods

FORMAT_...ASE::Time.FORMAT_... → *format_str*

The ASE::Time class has several predefined format strings for use with the “image” members defined above.

This Ruby snippet prints out a listing of the ASE::Time format methods on the left. These have been highlighted in blue. To the right are the returned format specifiers, highlighted in red, along with sample output “images.”

```
# Get the current time.
tmNow = ASE::Time.new

# These are the "singleton" methods in the ASE::Time class.
# This action generates a list of method names.
# The list is sorted alphabetically using the "sort!" action.
format_methods = ASE::Time.singleton_methods(false).sort!

# Generate a list with method names, specifier values, and example images.
format_methods.each do | method_name |
  m = ASE::Time.method(method_name)
  printf( "ASE::Time.%-28s== %s\n", method_name, m.call )
  printf( "%-38s=> %s\n", "", tmNow.image( m.call ) )
end

→ ASE::Time.FORMAT_ANDI_DATETIME == yyyy|mm|dd|hh|nn|ss|bbbb
→                               => 20090421213014-0500
→ ASE::Time.FORMAT_CLOCK        == h:nn AM
→                               => 9:30 PM
→ ASE::Time.FORMAT_DATETIME     == mm/dd/yyyy hh:nn:ss
→                               => 04/21/2009 21:30:14
→ ASE::Time.FORMAT_HPDI_MIL_DATETIME == dd mmm yy hh:nn
→                               => 21 Apr 09 21:30
→ ASE::Time.FORMAT_HPDI_STD_DATETIME == dd mmm yy hh:nn am
→                               => 21 Apr 09 09:30 pm
→ ASE::Time.FORMAT_INCOS_DATE   == mm/dd/yy
→                               => 04/21/09
→ ASE::Time.FORMAT_LONG_DATE    == ddd mmm d, yyyy
→                               => Tue Apr 21, 2009
→ ASE::Time.FORMAT_LONG_DATETIME == ddd mmm d, yyyy hh:nn:ss.zzz
→                               => Tue Apr 21, 2009 21:30:14.203
→ ASE::Time.FORMAT_LONG_ORDERED_DATETIME == yyyy/mm/dd hh:nn:ss.zzz
→                               => 2009/04/21 21:30:14.203
→ ASE::Time.FORMAT_OPC_DATETIME == dd-mmm-yyyy hh:nn
→                               => 21-Apr-2009 21:30
→ ASE::Time.FORMAT_ORDERED_DATETIME == yyyy/mm/dd hh:nn:ss
→                               => 2009/04/21 21:30:14
→ ASE::Time.FORMAT_TIMER        == hh:nn:ss.zzz
→                               => 21:30:14.203
→ ASE::Time.FORMAT_VECTOR2_DATETIME == ddd mmm dd hh:nn:ss yyyy
→                               => Tue Apr 21 21:30:14 2009
```

Format Specifiers

The ASE::Time class provides three member functions for creating string representations (images) based on a passed format string. These class members are:

- ASE::Time.image
- ASE::Time.imageBiased
- ASE::Time.imageLocal

The prior page lists several predefined format strings.

Format strings are created using the specifiers listed in the following table:

<u>Specifier</u>	<u>Description</u>
td	Displays the year day.
d	Displays the day as a number without a leading zero (1-31).
dd	Displays the day as a number with a leading zero (01-31).
ddd	Displays the day as an abbreviation (Sun-Sat).
dddd	Displays the day as a full name (Sunday-Saturday).
m	Displays the month as a number without a leading zero (1-12).
mm	Displays the month as a number with a leading zero (01-12).
mmm	Displays the month as an abbreviation (Jan-Dec).
mmmm	Displays the month as a full name (January-December).
yy	Displays the year as a two-digit number (00-99).
yyyy	Displays the year as a four-digit number (0000-9999).
h	Displays the hour without a leading zero (0-23).
hh	Displays the hour with a leading zero (00-23).
n	Displays the minute without a leading zero (0-59).
nn	Displays the minute with a leading zero (00-59).
s	Displays the second without a leading zero (0-59).
ss	Displays the second with a leading zero (00-59).
z	Displays the millisecond without a leading zero (0-999).
zzz	Displays the millisecond with a leading zero (000-999).
am or pm	Uses the 12-hour clock for the preceding h or hh specifier, and displays 'am' for any hour before noon, and 'pm' for any hour after noon.
a or p	Uses the 12-hour clock for the preceding h or hh specifier, and displays 'a' for any hour before noon, and 'p' for any hour after noon.
bbbb	Local time bias to UTC in hours and minutes with leading + or -.
bb	Local time bias to UTC in hours with leading + or -.

<u>Specifier</u>	<u>Description</u>
/	Displays the date separator character.
	Special separator character that isn't displayed.
:	Displays the time separator character.
.	Displays fraction separator character.
,	Displays comma separator character.
'xx' / "xx"	Characters enclosed in single or double quotes are displayed as-is, and do not affect formatting.

Note: Format specifiers may be written in upper or lower case letters. The results are the same.

Note: All "specifiers" are delineated by separator characters.