



# **Agilent VnmrJ 4 User Programming**

**Reference Guide**



**Agilent Technologies**

# Notices

© Agilent Technologies, Inc. 2013

No part of this manual may be reproduced in any form or by any means (including electronic storage and retrieval or translation into a foreign language) without prior agreement and written consent from Agilent Technologies, Inc. as governed by United States and international copyright laws.

## Manual Part Number

G7446-90520

## Edition

Revision A, March 2013

This guide is valid for 4.0 and 4.0i revisions of the Agilent VnmrJ software, until superseded.

Printed in USA

Agilent Technologies, Inc.  
5301 Stevens Creek Boulevard  
Santa Clara, CA 95051 USA

## Warranty

**The material contained in this document is provided “as is,” and is subject to being changed, without notice, in future editions. Further, to the maximum extent permitted by applicable law, Agilent disclaims all warranties, either express or implied, with regard to this manual and any information contained herein, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Agilent shall not be liable for errors or for incidental or consequential damages in connection with the furnishing, use, or performance of this document or of any information contained herein. Should Agilent and the user have a separate written agreement with warranty terms covering the material in this document that conflict with these terms, the warranty terms in the separate agreement shall control.**

## Technology Licenses

The hardware and/or software described in this document are furnished under a license and may be used or copied only in accordance with the terms of such license.

## Restricted Rights Legend

If software is for use in the performance of a U.S. Government prime contract or sub-contract, Software is delivered and licensed as “Commercial computer software” as defined in DFAR 252.227-7014 (June 1995), or as a “commercial item” as defined in FAR 2.101(a) or as “Restricted computer software” as defined in FAR 52.227-19 (June 1987) or any equivalent agency regulation or contract clause. Use, duplication or disclosure of Software is subject to Agilent Technologies’ standard commercial license terms, and non-DOD Departments and Agencies of the U.S. Government will receive no greater than Restricted Rights as defined in FAR 52.227-19(c)(1-2) (June 1987). U.S. Government users will receive no greater than Limited Rights as defined in FAR 52.227-14

(June 1987) or DFAR 252.227-7015 (b)(2) (November 1995), as applicable in any technical data.

## Safety Notices

### CAUTION

A **CAUTION** notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in damage to the product or loss of important data. Do not proceed beyond a **CAUTION** notice until the indicated conditions are fully understood and met.

---

### WARNING

A **WARNING** notice denotes a hazard. It calls attention to an operating procedure, practice, or the like that, if not correctly performed or adhered to, could result in personal injury or death. Do not proceed beyond a **WARNING** notice until the indicated conditions are fully understood and met.

---

# Contents

## 1 MAGICAL II Programming

Working with Macros	12
Writing a macro	12
Executing a macro	14
Transferring macro output	15
Loading macros into memory	16
Programming with MAGICAL	18
Tokens	18
Variable types	23
Arrays	26
Input arguments	30
Name replacement	31
Conditional statements	31
Loops	32
Macro length and termination	33
Command and macro tracing	34
Relevant VnmrJ Commands	35
Spectral analysis tools	35
Input/Output tools	37
Regression and curve fitting	39
Mathematical functions	40
Creating, modifying, and displaying macros	41
Miscellaneous tools	43

## 2 Pulse-Sequence Programming

Overview of Pulse-Sequence Programming	50
Overview of pulse-sequence execution	50
Compiling a pulse sequence with the PSG	51
Troubleshooting a new pulse sequence	54
Creating a parameter table for a new sequence	55
C framework for pulse sequences	56
Global PSG and real-time variables in multidimensional arrays	59
Assigning transmitters and receivers	60
Customizing the PSG software	62

Pulse-Sequence Statements	64
Creating a time delay	64
Assigning transmitters and receivers to channels	65
Transmitter pulses	66
Pulsing channels simultaneously	69
Setting quadrature phase shifts	71
Setting small-angle phase shifts	72
Controlling the frequency offset	73
Controlling the transmitter power	74
Explicit transmitter gating	77
Transmitter blanking and unblanking	78
The status statements	80
Receiver gating	81
Interfacing to external user devices	82
Real-Time Control of Pulse Sequences	83
Real-time integer variables and constants	83
Calculating with real-time integer math	84
Real-time loops and conditionals	87
Real-time integer tables	90
The format of a table file	93
Shaped Pulses and Waveforms	95
Executing shaped pulses	95
Calculation of shaped pulses	98
Programmed waveforms for decoupling	99
Waveforms with an automatic frequency offset	101
Using spinlock pulses	102
Calculation of programmed waveforms	103
Calculating gradient shapes	104
Statements that create shape, waveform and gradient files	105
Waveform interpolation	106
Programming for Acquisition Control	110
Implicit acquisition	110
Standard acquisition with the VNMRS digital receiver	110
Setting the receiver phase	112
Setting the receiver offset	112
Programming explicit acquisition	112
Windowed acquisition	116
Acquisition with multiple FIDS	119
Multidimensional NMR and Arrays	121
Variables for nD dimensions	121
Compressed arrays and nD dimensions	123
Switchable loops for nD imaging experiments	123

Arrayed data acquisition	125
Managing arrays	126
Multidimensional phases	127
Syntax for Controlling Parallel Channels	130
Synchronous Timing Syntax	130
A <code>parallelstart - parallelend</code> Section	130
Looping in Parallel Sections	134
Allowed Statements and Calculation in a Parallel Section	136
Programming the Receiver and Gradients	137
The <b>nowait</b> Attribute	138
Waveform Control with <code>obsprgon-obsprgoff</code>	139
Parameters and Variables	141
Categories of parameters and variables	141
Statements used to handle parameters	142
PSG Variables	145
Global PSG Variables	145
Imaging and Other Variables	149
Pulse Sequence Output	155
Output messages	155
Pulse sequence display	157
Setting the Amplitude, Phase, and Gate from Tables	159
Gradient Control for PFG and Imaging	161
Single-axis gradient control for spectroscopy	163
Shaped gradients with <code>nowait</code> capability	165
Creating a gradient table	165
Controlling magic angle gradients	166
Oblique and phase-encoded-oblique gradients for imaging	166
Controlling slice-selective RF shaped pulses for imaging	167
Controlling lock field correction	169

### 3 Pulse Sequence Statement Reference

Pulse Sequence Statements	172
A	177
C	181
D	187
E, F	250
G	256
H	261
I	263

K	273
L	275
M	278
O	285
P	298
R	328
S	342
T	375
U	383
V	387
W	400
X	401
Z	404

## 4 Linux Level Programming

Linux and VnmrJ	406
Linux Reference Guide	407
Command entry	407
File names	408
File handling commands	408
Directory names	408
Directory handling commands	408
Text commands	409
Other commands	409
Special characters	410
Linux Commands Accessible from VnmrJ	411
Opening a text editor from VnmrJ	411
Opening a shell from VnmrJ	411
Background VNMR	412
Running VNMR Command as a Linux background task	412
Running VNMR processing in the background	413
Shell Programming	414
Shell variables and control formats	414
Shell scripts	414
Data backup under Linux	416
General considerations	416
Data backup on CDs/DVDs	417

Installation of a second hard disk	419
Data mirroring using VnmrJ tools	423
Automatic data backup using rsync	424
Backup on a second hard disk	426
Backup on a remote mounted filesystem	428
Backup on a remote computer via Secure Shell (ssh)	432

## 5 Parameters and Data

VnmrJ Data Files	440
Binary data files	440
Data file structures	442
VnmrJ use of binary data files	447
Storing multiple traces	448
Header and data display	450
Binary files in VnmrJ and byte order	450
FDF (Flexible Data Format) Files	452
File structures and naming conventions	452
File format	453
Header parameters	454
Transformations	457
Creating FDF files	457
Splitting FDF files	458
Reformatting Data for Processing	459
Standard and compressed formats	460
Compress or decompress data	461
Move and reverse data	461
Table convert data	462
Reformatting spectra	462
Creating and Modifying Parameters	464
Parameter types and trees	464
Tools for working with parameter trees	465
Format of a stored parameter	468
Modifying Parameter Displays in VNMR	473
Display template	473
Conditional and arrayed displays	475
Output format	477
Modules	478
Creating modules	478
Viewing modules parameters	479
Module example—Och_adiabatic	479
User-Written Weighting Functions	480

Writing a weighting function	481
Compiling the weighting function	482
User-Written	484
FID files	484

## 6 User Space Customization

Customize User Space with dousermacro	488
Customizing Default Experiment Parameters using user<pslabel>	489
Customizing Output - Plotting	490

## 7 Panels, Toolbars, and Menus

Parameter Panel, Toolbar, and Menu Properties	494
Using the Panel Editor	495
Starting the panel editor	495
Editing existing panel elements	497
Adding and removing panel elements	499
Using the Item Editor	501
Adding user-defined sampltags	502
Saving panel changes	504
Exiting the Panel Editor	506
Panel Elements	508
Element style	508
Panel element attributes	508
Panel elements	510
Advanced panel elements	527
Creating a New Panel	534
Writing commands	534
Creating a new panel layout	534
Creating a new page	535
Defining and populating a page	536
Saving and retrieving a panel element	536
Files associated with panels	537
Sizing panels	539
Panel Style Guidelines	540
Graphical Toolbar Menus	545
Editing the Toolbar menu	545
Graphics toolbar parameters	545
Icons	546
Menu file description example, dconi	546



## A Status Codes

Status Codes 554





# 1 MAGICAL II Programming

Working with Macros 12  
Programming with MAGICAL 18  
Relevant VnmrJ Commands 35

Many actions are performed multiple times on an NMR spectrometer, daily. To make these actions easier for the user, the VnmrJ software incorporates a high-level macro programming language designed for NMR called the NMR language, MAGICAL II (MAGnetics Instrument Control and Analysis Language, version II - referred to as MAGICAL in this chapter). Many commands used in VnmrJ are in fact macros (see `/vnmr/mac.lib`).



## Working with Macros

- Writing a macro
- Executing a macro
- Transferring macro output
- Loading macros into memory

A *macro* is a user-defined command that can duplicate a long series of commands and parameter changes that need to be entered one after another. For example, to plot a spectrum, a scale under the spectrum, and the parameters on a page, the following sequence of commands is required:

```
pl
pscale
ppa
page
```

A macro called `plot`, containing the aforementioned commands, can be written. Consider another example. To routinely plot 2D spectra using certain parameters, a macro called `plot_2d` can be written using the following sequence of commands:

```
wc=160
sc=20
wc2=160
sc2=20
pcon(10,1.4)
page
```

MAGICAL provides an entire series of programming tools, such as `if` statements and loops that can be used as part of macros. MAGICAL also provides other NMR-related tools, which provide access to NMR information such as peak heights, integrals, and spectral regions. Using these two sets of tools, *NMR algorithms* are easily implemented with MAGICAL.

### Writing a macro

Consider the following scenario. You need to locate the largest peak in a spectrum in which the peaks may be positive or negative (such as an APT spectrum), and adjust the vertical scale of the spectrum so that the tallest peak is 180 mm high. The following macro (or MAGICAL program), which we call `vsadj`, illustrates how the MAGICAL tools can be used to quickly and simply find a solution:

```
"vsadj --- Adjust scale of spectrum"
peak:$height,$frequency          "Find largest peak"
```

```

if $height<0 then           "If negative, make positive"
$height=-$height endif
vs=180*vs/$height          "Adjust the vertical scale"

```

As written, the macro `vsadj` contains four lines:

The content enclosed in double-quotation marks (the first line and parts of other lines) are comments. `MAGICAL` permits comments, and as a good programming practice, this example is filled with comments to explain the macro.

- The second line of the macro ("`peak:$height,...`") illustrates the ability of `MAGICAL` to extract spectral information. The `peak` command looks through the spectrum and returns to the user the height and frequency of the tallest peak in the spectrum, which are then stored (in this example) in temporary variables named `$height` and `$frequency`.
- The third line of the macro ("`if $height<0...`") illustrates that `MAGICAL` is a high-level programming language, with conditional statements (for example, `if...then...`), loops, etc. This particular line ensures that the peak height we measure is always a positive value, which is necessary for the calculation in the next line.
- The last line ("`vs=180*vs...`") illustrates the use of NMR parameters (for example, `vs` sets the vertical scale) as simple variables in our macro. This line accomplishes the task of calculating a new value of `vs` that will make the height of the tallest peak equal to 180 mm.

Part of the power of the `MAGICAL` macro language is its ability to build on itself. For example, we can create first-level macros from existing commands, second-level macros from first-level macros and commands, and so on. Consider the following scenario: we create a macro `plot`, we might also create a macro `setuph`, another macro `acquireh`, and yet another macro `processh`. Now we can create a higher-level macro, `H1`, which is equivalent to `setuph acquireh processh plot`. Another example is that we create two more similar macros, `C13` and `APT`. Now we can create a higher-level macro `HCAPT`, equivalent to `H1 C13 APT`. At every step, the power of the macro increases, without increasing the complexity.

Many macros are part of the standard `VnmrJ` software. These macros are discussed in the relevant chapters of the manual *Getting Started*-processing macros are discussed along with processing commands, acquisition setup macros are discussed along with acquisition setup commands, and

so on. Refer to the *VnmrJ Command and Parameter Reference* for a concise description of standard macros. The examples used here are instructive examples and do not necessarily represent standard Agilent software.

## Executing a macro

When any program is executed, the command interpreter first checks if it is a standard VnmrJ command. If the program is not a command, the command interpreter then attempts to find a macro with the program name. Unlike a built-in VnmrJ command, which is a built-in procedure containing code that normally cannot be changed by users, the code inside a macro is text that is accessible and can be changed by users as needed.

If a VnmrJ command and a macro have the same name, the VnmrJ command always takes precedence over a macro. Consider the following example; there is a built-in VnmrJ command named `wft`. If someone writes a macro also named `wft`, then the macro `wft` will never get executed because the VnmrJ command `wft` takes precedence. To get around this restriction, the `hidecommand` command can be used to rename a command so that a macro with the same name as a command is executed instead of the built-in command. If the user who wrote the `wft` macro enters `hidecommand('wft')`, the command is renamed to `Wft` (first letter made upper case) and the macro `wft` can be directly executed. The new `wft` macro can access the hidden `wft` built-in command by calling it with the name `Wft`. To go back to executing the command `wft` first, enter `hidecommand('Wft')`.

VNMR supports “Application directories” or `appdirs`. An Application directory is a place where VNMR will look for `maclib` and other directories. A macro can exist in any of the `appdir` `maclibs`, and the precedence is determined by the order of the application directories. Generally, a user’s `vnmrSYS` directory is the `appdir` with the highest precedence, so macros located in the user’s `vnmrSYS/maclib` directory will be found first.

For example, `rt` is a standard VNMR macro in the system `maclib`. If a user puts a macro named `rt` in the user’s `maclib`, the user’s `rt` macro takes precedence over the system `rt` macro.

The which macro can search these locations and display the information about locations that contain a macro. For

example, entering which('rt') determines the location of the macro `rt`.

The system macro directory `/vnmr/maclib` can be changed by the system operator only, but the changes are available to all users. Each user also has his or her own private macro directory `maclib` in the user's `vnmrsys` directory. These macros take precedence over the system macros if a macro of the same name exists in both directories. Thus, users can modify a macro to their own needs without affecting the operation of other users. If the command interpreter does not find the macro, it displays an error message to the user

Macros are executed in exactly the same way as normal system commands, including the possibility of accepting optional arguments (shown by angled brackets "`<...>`"):

```
macroname<(argument1< , argument2 , ... >)>
```

Arguments passed to commands and macros can be constants (for example, `5.0` and `'apt'`), parameters and variables (`pw` and `$ht`), or expressions (`2*pw+5.0`). Recursive calls to procedures are allowed. Single quotes must be used around constant strings.

Macros can also be executed in three other ways:

- When the `VnmrJ` program is first run, a system macro `bootup` is run. This macro, in turn, runs a user macro named `login` in the user's local `maclib` directory if such a macro exists.
- When any parameter `x` is entered, if that parameter has a certain "protection bit" set (see `Format of a Stored Parameter`), a macro by the name `_x` (that is, the name of the parameter with an underline as a prefix) is executed. For example, changing the value of `sw` executes the macro `_sw`.
- Whenever parameters are retrieved with the `rt`, `rtp`, or `rtv` commands, a macro named `fixpar` is executed.

## Transferring macro output

Output from many commands and macros, in addition to being displayed on the screen or placed in a file, can also be transferred to any parameter or variable of the same type. To receive the output of a program of this type, the program name (and arguments, if any) are followed by a colon (`:`) and one or more names of variables and parameters that are to

take the output:

```
macroname<(arg1<,arg2,...)>:variable1,variable2,...
```

For example, the command `peak` finds the height and frequency of the tallest peak. Entering the command:

```
peak:r1,r2
```

results in `r1` containing the height of the tallest peak and `r2` containing its frequency. Therefore, entering the command

```
peak:$ht,cr
```

would set `$ht` equal to the height of the tallest peak and set the cursor (parameter `cr`) equal to its frequency, and therefore, would be the equivalent of a “tallest line” command (similar to but different than the command `n1` to position the cursor at the nearest line).

It is not necessary to receive all of the information. For example, entering

```
peak:$peakht
```

puts the height of the tallest peak into the variable `$peakht`, and does not save the information about the peak frequency.

The command that displays a line list, `d11`, also produces one output—the number of lines. Entering

```
d11:$n
```

reads the number of lines into the variable `$n`. `d11` alone is perfectly acceptable although the information about the number of lines is then *lost*.

## Loading macros into memory

Every time a macro is used, it is parsed before it is executed. This parsing takes time. If a macro is used many times or if a faster execution speed is required, the parsed form of the macro, user or system, can be loaded into the memory by the `macrold` command. When that macro is executed, it runs substantially faster. One or more macros can be “pre-loaded” to run automatically when VnmrJ is started by inserting some `macrold` commands into your login macro.

Macros are also loaded into memory by the `macrovi` or `macroedit` commands to edit the macro. The only argument in each is the name of the macro file; for example, enter `macrovi('pa')` or `macroedit('pa')` if the macro name is `pa`. The choice depends on the type of macro and the text editor required:



- For a user macro, use `macrovi`, a vi based editor.
- For a user macro from an editor, select `macroedit`.
- To edit a system macro, copy the macro to your personal macro directory and edit it there with `macrovi` or `macroedit`.

To select the editor for `macroedit`, set the operating system's (OS) variable `vnmreditor` to its name (`vnmreditor` is set through the OS `env` command). A script for the editor in the `bin` subdirectory of the VnmrJ system directory must also exist. For example, to select Emacs, set `vnmreditor=emacs` and have a script `vnmr_emacs`.

Several minor problems need to be considered while loading macros into memory:

- These macros consume a small amount of memory. In memory-critical situations, remove one or more macros from memory using the `purge<(file)>` command, where `file` is the name of the macro file to be removed from memory. Entering `purge` with no arguments removes all macros loaded into memory.

### CAUTION

The `purge` command with no arguments should never be called from a macro, because it will remove all macros from memory, including the macro containing `purge`. Furthermore, `purge`, where the argument is the name of the macro containing the `purge` command, should never be called.

- A macro loaded in memory and modified from a separate terminal window leaves the copy in memory as unchanged. Executing this memory causes VNMR to execute the old copy in memory. Use `macrovi` or `macroedit` to edit the macro, or if the macro has been edited in another window, then use `macrold` to replace the macro loaded in memory with the new version.

A personal macro created with the same name as a system macro already in memory requires the use of `purge` to clear the system macro from memory so that the personal version in the `maclib` directory can be subsequently executed.

Performance improves if a macro inside a macro loop is called before entering the loop and executing the loop. Remove the called macro from memory with the `purge` command after exiting the loop.

## Programming with MAGICAL

- Tokens
- Variable Types
- Arrays
- Expressions
- Input Arguments
- Name Replacement
- Conditional Statements
- Loops
- Macro Length and Termination
- Command and Macro Tracing

MAGICAL has many features, including tokens, variables, expressions, conditional statements, and loops. To program in MAGICAL, you need to know the main features as described in this section.

### Tokens

A token is a character or characters that are considered by the language as a single entity or unit. There are five classes of tokens in MAGICAL: identifiers, reserved words, constants, operators, and separators.

#### Identifiers

An identifier is the name of a command, macro, parameter, or variable, and is a sequence of letters, digits, and the special characters `_ $ #`. The underline `_` is considered a letter. Upper and lower case letters are different. The first character of identifiers, except for temporary variable identifiers, must be a letter. Temporary variable identifiers start with the dollar-sign (`$`) character. Identifiers can be of any length (within reason). Examples of identifiers are `pcon`, `_pw`, or `$height`.

#### Reserved words

The identifiers listed in [Table 1](#) Reserved words in MAGICAL are reserved words and may not be used otherwise. Reserved words are recognized in both upper and lower case formats (for example, do not use either `and` or `AND` except as a reserved word).

**Table 1** Reserved words in MAGICAL

abort	else	not	trunc
abortoff	elseif	or	typeof
aborton	endif	repeat	then
and	endwhile	return	until
break	if	size	while
do	mod	sqrt	

## Constants

Constants can be either floating or string.

- A floating constant consists of an integer part, a decimal point, a fractional part, the letter E (or e) and, optionally, a signed integer exponent. Both integer and fraction parts consist of a sequence of digits. Either the integer part or the fraction part (but not both) may be missing; similarly, either the decimal point, or the E (or e) and the exponent may be missing. Some examples are 1.37E-3, 4e5, .2E2, 1.4, 5.
- A string constant is a sequence of characters surrounded by single-quote characters ('...') or by backward single-quote characters (`...`). 'This is a string' and `This is a string` are examples of string constants.

To include a single-quote character in a string, place a backslash character (\) before the single-quote character, for example:

```
'This string isn\'t permissible without the backslash'
```

To include a backslash character in the string, place another backslash before the backslash, for example:

```
'This string includes the backslash \\'
```

Alternatively, the two styles of single quote characters can be used. If backward single quotes are used to delimit a string, then single quotes can be placed directly within the string, for example:

```
`This isn't a problem`
```

Or the single-quote styles can be exchanged, for example:

```
'This isn`t a problem'
```

The single quote style that initiates the string must also terminate the string.

## Operators

Table 2 lists the operators available in MAGICAL. Each operator is placed in a group, and groups are shown in order of precedence, with the highest group precedence shown first. Within each group, operator precedence in expressions is from left to right, except for the logical group, in which the respective members are listed in order of precedence.

**Table 2** Order of operator precedence (highest first) in MAGICAL

Group	Operation	Description	Example
special	<code>sqrt()</code>	square root	<code>a = sqrt(b)</code>
	<code>trunc()</code>	truncation	<code>\$3 = trunc(3.6)</code>
	<code>typeof()</code>	return argument type	<code>if typeof('\$1')</code> <code>then...</code>
	<code>size()</code>	return argument size	<code>r1 = size('d2')</code>
unary	<code>-</code>	negative	<code>a = -5</code>
multiplicative	<code>*</code>	multiplication	<code>a = 2 * c</code>
	<code>/</code>	division	<code>b = a / 2</code>
	<code>%</code>	remainder	<code>\$1 = 4 % 3</code>
	<code>mod</code>	modulo	<code>\$3 = 7 mod 4</code>
additive	<code>+</code>	addition	<code>a = x + 4</code>
	<code>-</code>	subtraction	<code>-b = y - sw</code>
relational	<code>&lt;</code>	less than	<code>if a &lt; b then...</code>
	<code>&gt;</code>	greater than	<code>if a &gt; b then...</code>
	<code>&lt;=</code>	less than or equal to	<code>if a &lt;= b</code> <code>then...</code>
	<code>&gt;=</code>	greater than or equal to	<code>if a &gt;= b</code> <code>then...</code>
equality	<code>=</code>	equal to	<code>if a = b then...</code>
	<code>&lt;&gt;</code>	not equal to	<code>if a &lt;&gt; b</code> <code>then...</code>
logical	<code>not</code>	negation	<code>if not (a=b)</code> <code>then...</code>
	<code>and</code>	logical and	<code>if r1 and r2</code> <code>then...</code>

**Table 2** Order of operator precedence (highest first) in MAGICAL

Group	Operation	Description	Example
	or	logical (inclusive) or (true if one or both conditions are true)	if (r1=2) or (r2=4) then...
assignment	=	equal	a = 3

There are four “built-in” special operators:

- `sqrt` returns the square root of a real number.
- `trunc` truncates real numbers.
- `typeof` returns an identifier (0, or 1) for the type (real, or string) of an argument. The `typeof` operator will abort if the identifier does not exist.
- `size` returns the number of elements in an arrayed parameter.

The unary, multiplicative, and additive operators apply only to real variables. The + (addition) operator can also be used with string variables to concatenate two strings together. The mathematical operators can not be used with mixed variable types.

If the variable is an array, the mathematical operators try to do simple matrix arithmetic. If two matrices of the same size are equated, added, subtracted, multiplied, divided, or one matrix is taken as a modulus, each element of the first matrix is operated on with the corresponding element of the second. If two matrices of the same size are compared with an `and` operator, the resulting Boolean is the AND of each individual element. If two matrices of the same size are ORed together, the resulting Boolean is the OR of each individual element. If the two matrices have unequal sizes, an error is thrown.

An arrayed variable *cannot* be operated on (added, multiplied, etc.) by a single-valued constant or variable. For example, if `pw` is an array of five values, `pw=2*pw` does *not* double the value of each element of the array.

## Comments

MAGICAL programming provides three ways to enter comments:

- Create a comment by putting characters between double quotation marks ("..."), except when the double quotation marks are in a literal string, for example,

```
'The word "and" is a reserved word'
```

Comments based on double quotation marks can appear anywhere—at the beginning, middle, or end of a line—but cannot span multiple lines. At the end of a comment, place a second double quotation mark; otherwise, the comment is automatically terminated at the end of a line.

- Create a single-line comment with two slash marks (//). The comment starts with the // and ends on the line., for example,

```
// This is a comment
```

As with the double quotation marks, // in a literal string does not signify a comment. This type of comment is often used for a brief description of the preceding command, for example,

```
cdc // clear drift correction
```

- Create a single-line or multiple-lines comment with a slash and asterisk (/\*), which begins the comment, and an asterisk and a slash (\*/\*), which ends the comment, for example,

```
/* The comment
   can span
   multiple lines
*/
```

This type of comment is useful for longer descriptions. It is also useful for “commenting out” sections of a macro for debugging purposes.

Again, if the /\* or \*/ are in a literal string, they do not serve as comment delimiters. These comments do not nest; that is, the following construct will fail:

```
/*
/* Comment does not nest
   This will cause an error
*/
*/
```

In this example, the first /\* starts the comment. The second /\* is ignored because it is part of the comment. The first \*/ terminates the comment, which causes the second \*/ to generate an error.

## Separators

Blanks, tabs, new lines, and comments serve to separate tokens and are otherwise ignored.

## Variable types

As with many programming languages, MAGICAL provides two classes of variables:

- Global variables (also called external) that retain their values on a permanent or semi-permanent basis. These are present in parameter sets and `~/vnmrsys/global`, for example.

Global variables in this section refer to variables that retain their values upon exiting a macro and not specifically to the variables present in `~/vnmrsys/global`.

- Local variables (also called temporary, dummy, or automatic) that are created for the time taken to execute the macro in question, after which the variables no longer exist.

Global and local variables can be of two types: real and string. Global real variables are stored as double-precision (64-bit) floating point numbers. The command `real(variable)` creates a real variable without a value, in which `variable` is the name of the variable to be created and stored in the current parameter set.

Although global real variables have potential limits from  $1e308$  to  $1e-308$ , when such variables are created, they are given default maximum and minimum values of  $1e18$  and  $-1e18$ ; these can subsequently be changed with the `setlimit` command. For example, `setlimit('r1',1e99,-1e99,0)` sets variable `r1` to limits of  $1e99$  and  $-1e99$ . Local real variables have limits slightly less than  $1e18$  ( $9.999999843067e17$ , to be precise) and cannot be changed.

String variables can have any number of characters, including a null string that has no characters. The command `string(variable)`, in which `variable` is the name of the variable to be created, creates a string variable without a value, and is stored in the current parameter set.

Both real and string variables can have either a single value or a series of values (also called an array).

Global and local variables have the following set of attributes associated with them:

name	group	array size
basictype	display group	enumeration
subtype	max./min. values	protection status
active	step size	

The variable's attributes are used by programs when manipulating variables.

### Global variables

The most important global variables used in macros are the VnmrJ parameters themselves. Thus parameters like `vs` (vertical scale), `nt` (number of transients), `at` (acquisition time), etc., can be used in a MAGICAL macro. Like any variable, they can be used on the left side of an equation (and hence their value changed), or they can be used on the right side of an equation (as part of a calculation, perhaps to set another parameter).

The real-value parameters `r1`, `r2`, `r3`, `r4`, `r5`, `r6`, and `r7`, and the string parameters `n1`, `n2`, and `n3` can be used by macros. These are experiment-based parameters. Setting these parameters in one experiment, `exp1` for example, and running a macro that changes experiments, using the command `jexp3` for example, causes a new set of such parameters to appear. Similarly, recalling parameters or data with the `rt` or `rtp` commands overwrites the current values of these parameters, just as it overwrites the values of all other parameters.

Within a single experiment, and assuming that the `rt` and `rtp` commands are not used, these parameters act like global parameters such that all macros can read or write information into these parameters, and hence information can be passed from one macro to another. Thus, they provide a useful place to store information that must be retained for some time or must be accessed by more than one macro—make sure that some other macro does not change the value of this variable in the meantime.

Variables stored in `~/vnmrsys/global` are not experiment-based and retain their values even when `jexp(experiment_number)`, `rt('file'<, 'nolog'>)`, or `rtp('file')>` are used.

### Local variables

Any number of local variables can be created within a macro. These temporary variables begin with the dollar-sign (\$) character, such as `$number` and `$peakht`. The type of variable (real or string) is decided by the first usage—there is no variable declaration, as in many languages. Therefore, setting `$number=5` and `$select='all'` establishes `$number` as a real variable and `$select` as a string variable. Every



macro is provided with some local variables.

`$0` is the name of the macro. This can be used if a single macro has multiple aliases (see `jexp1` macro for an example).

`$#`  is the number of passed arguments. If no arguments are passed,  `$#`  will be 0.

`$1, $2, ... $n`  contain the values of the first, second, ..., n-th argument respectively. For example, If  `$# = 0` , there will be no  `$1`  variable. If  `$# = 3` , there will be  `$1, $2, and $3`  variables.

`$##`  is the number of return values the calling macro is requesting.

For example, if macro A calls macro B as  `B(pw,$filename):$res` , the macro B will be provided with the following local variables

```
$0='A'
```

```
 $# =2
```

```
 $1=pw
```

```
 $2=$filename
```

`$##=1` A special initialization is required in one situation. When the first use of a string variable is used as the return argument from a procedure, it must be initialized first by being set to a null string. For example, the following line:

```
input('Input Your Name: '):$name
```

produces an error. Instead, use the following:

```
$name=' ' input('Input Your Name: '):$name.
```

By definition, local variables are lost on the completion of the macro. Furthermore, they are completely local, which means that each macro, even a macro that is being run by another macro, has its own set of variables. If one macro sets  `$number=5`  and then runs another macro that sets  `$number=10` , when the second macro completes operation and the execution of commands returns to the first macro,  `$number`  equals 5, not 10. If the first macro is run again at a later time,  `$number`  starts with an undefined value. It is a good practice to use local variables whenever possible.

Local variables can also be created on the command input line. These variables are automatically created but are not deleted, and hence this is not a recommended practice; use  `r1, r2, etc.,`  instead.

Accessing a variable that does not exist displays the following error message:

Variable “variable\_name” doesn’t exist.

## Arrays

Both global and local variables, whether real or string, can be arrayed. Array elements are referred to by square brackets ([...]), such as `pw[1]`. Indices for the array can be fixed numbers (`pw[3]`), global variables (`pw[r1]`), or local variables (`pw[$i]`). Of course, the index must not exceed the size of the array. Use the size operator to determine the array size. For example, the statement `r1=size('d2')` sets `r1` to the number of elements in variable `d2`. If the variable has only a single value, `size` returns a 1; if the variable doesn’t exist, it returns a 0.

Some arrays, such as a pulse width array, are user-created. Other arrays, such as `llfrq` and `llamp`, are created by the software (in this case, when a line list is performed). In both these cases, a macro can refer to any existing element of the array, for example, `pw[4]` or `llfrq[5]`.

A MAGICAL macro can also create local variables containing arrayed information by itself. No dimensioning statement is required; the variable just expands as necessary. The only constraint is that the array must be created in order: element 1 is first, element 2 second, and so on. The following example shows how an array might be created and all values initialized to 0:

```
$i=1
repeat
  $newarray[$i]=0
  $i=$i+1
until $i>10
```

### Arrays of string variables

Arrays of string variables are identical in every way to arrays of real variables, except that the values are strings. If, for example, a user has entered `dm='nny', 'yYy'`, the following macro plots each spectrum with the proper label:

```
$i=1
repeat
  select($i)
  pl
  write('plotter',0,wc2max-10,'Decoupler mode: %s',dm[$i])
  page
  $i=$i+1
until $i>size('dm')
```

### Arrays of listed elements

Arrays can be constructed by simply listing the elements, separated by commas. For example,

```
pw=1,2,3,4
```

creates a `pw` array with four elements. Select the initial array element when using this list mechanism by providing the index in square brackets. For example,

```
pw[3]=5,6
```

results in `pw` having elements 1,2,5,6. Extend arrays as in

```
pw[5]=7,8,9
```

which yields a `pw` array of 1,2,5,6,7,8,9. Change existing values and extend the array, as in

```
pw[6]=6,7,8,9,10
```

which yields a `pw` array of 1,2,5,6,7,6,7,8,9,10

Comma separated lists can also include expressions. For example,

```
d2=0,1/sw1,2/sw1,3/sw1
```

The square brackets can also be used on the right side of the equal sign in order to construct arrays. The `[ ]` can enclose a single value or expression or an array of values or expressions. Any mathematics applied to the `[ ]` element is applied individually to each element within the `[ ]`.

Some examples.

Enter	Result
<code>nt=[1]</code>	<code>nt=1</code>
<code>nt=[1,2,3]</code>	<code>nt=1,2,3</code>
<code>nt=[1,2,3]*10</code>	<code>nt=10,20,30</code>
<code>nt=22*[2*3,r2+6,trunc(r3)]+2</code>	<code>nt=22*2*3+2,22*(r2+6)+2,22*trunc(r3)+2</code>
<code>d2=[0,1,2,3]/sw1</code>	<code>d2=0/sw1,1/sw1,2/sw1,3/sw1</code>

Use `[ ]` to give precedence to expressions, just like `()`.

Enter	Result
<code>nt=[2*[3+4]]</code>	<code>nt=14</code>

There are a couple of limitations if the `[ ]` element is used as part of a mathematical expression. When used in expressions, only a single `[ ]` element is allowed. Also, when used in expressions, the `[ ]` element cannot be mixed with the standard comma (,) arraying element. For example, `nt=[1,2]*[3,4]` is not allowed and generates the error message:

"No more than one [--.--]"

`nt=1, [2,3,4]*10` is not allowed and generates the error message:

"Cannot combine, with [--.--]"

These restrictions only occur if mathematical operators are used and the [ ] element itself contains a comma. Simply listing multiple [ ] elements, or combining them with the comma element is okay.

Enter	Result
<code>nt=[1,2],3</code>	<code>nt=1,2,3</code>
<code>nt=[1,2],[3,4]</code>	<code>nt=1,2,3,4</code>

### Array error messages

Accessing an array element that does not exist displays the error message:

```
variable_name['index'] index out of bounds
```

Using a string as an index, rather than an integer, displays the error message:

```
Index for variable_name['index'] must be numeric
```

or

```
Index must be numeric
```

Finally, using an array as an index displays the error message:

```
Index for variable_name must be numeric scalar
```

or

```
Index must be numeric scalar.
```

### Expressions

An *expression* is a combination of variables, constants, and operators. Parentheses can be used to group together a combination of expressions. Multiple nesting of parentheses is allowed. In making expressions, combine only variables and constants of the same type:

- Real variables and constants only with other real variables and constants.
- String variables and constants only with other string variables and constants.

The type of a local variable (a variable whose name begins with a \$) is determined by the context in which it is first used. The only ambiguity is when a local variable is first used as a return argument of a command such as `input`, as discussed in the previous section on local variables.

If an illegal combination is attempted, an error message is displayed:

```
Can't assign STRING value "value" to REAL variable \
"variable_name"
```

or

```
Can't assign REAL value (value) to STRING variable \
"variable_name"
```

### Mathematical expressions

Expressions can be classified as mathematical or Boolean. Mathematical expressions can be used in place of simple numbers or parameters. Expressions can be used in parameter assignments, such as in `pw=0.6*pw90`, or as input arguments to commands or macros, such as in `pa(-5+sc,50+vp)`.

When parameters are changed as a result of expressions, the normal checks and limits on the entry of that particular parameter are followed. For example, if `nt=7`, the statement `nt=0.5*nt` will end with `nt=3`, just as directly entering `nt=3.5` would have resulted in `nt=3`. Other examples of this include the round-off of `fn` entries to powers of two, limitation of various parameters to be positive only, etc.

### Boolean expressions

Boolean expressions have a value of either TRUE or FALSE. Booleans are represented internally as 0.0 for FALSE and 1.0 for TRUE, although in a Boolean expression, any number other than zero is interpreted as TRUE. Boolean expressions can only compare quantities of the same type—real numbers with real numbers, or strings with strings. Some examples of Boolean expressions include `pw=10`, `sw>=10000`, `at/2<0.05`, and `(pw<5) or (pw>10)`.

The explicit use of the words “TRUE” and “FALSE” is not allowed. All Boolean expressions are implicit—they are evaluated when used and given a value of TRUE or FALSE for the purpose of some decision.

### Input arguments

Arguments passed to a macro are referenced by `$n`, in which `n` is the argument number. An unlimited number of arguments (`$1`, `$2`, and so on) can be passed. The name of the macro itself may be accessed using the special name `$0`. For example, if the macro `test1` is running, `$0` is given the value `test1`. A second special variable `$#` contains the number of arguments passed and can be used for routines having a variable number of arguments. `##` is the number of return values requested by the calling macro. Arguments can be either real or string types, as with all parameters.

The following example uses an input argument, such as `$1`:

```
"vsmult(multiplier) "  
"Multiply vertical scale (vs) by input argument"  
vs=$1*vs
```

The following example uses two input arguments:

```
"offset(arg1,arg2) "  
"Increment vertical position (vp) and horizontal position  
(sc) "  
vp=$1+vp  
sc=$2+sc
```

The `typeof` operator returns a 0 if the variable is real. It returns a 1 if the variable is a string. It will abort if the variable does not exist. For example, in the conditional statement `if typeof('$1') then ...`, the `then` part is executed only if `$1` is a string.

## Name replacement

An identifier surrounded by curly braces (`{...}`) results in the identifier being replaced by its value before the full expression is evaluated. If the name replacement is on the left side of the equal sign, the new name is assigned a value. If the name replacement is on the right side of the equal sign, the value of the new name is used. The following are examples of name replacement:

```
$a = 'pw           '"variable $a is set to string 'pw'"
{$a} = 10.3       "pw is set to 10.3"
pw = 20.5         "pw is set to 20.5"
$b = {$a}         "variable $b is set to 20.5"
{$a}[2]=5        "pw[2] is set to 5.0"
$b = {$a}[2]     "variable $b is set to 5.0"
$cmd='wft'       "$cmd is set to the string 'wft'"
{$cmd}           "execute wft command"
```

The use of curly braces for command execution is subject to a number of constraints. In general, using the VNMR command `exec` for the purpose of executing an arbitrary command string is recommended. In this last example, this would be `exec($cmd)`.

## Conditional statements

The following forms of conditional statements are allowed:

```
if booleanexpression then ... endif
if booleanexpression then ... else ... endif
if booleanexpression then ... {elseif booleanexpression
then... }[else...]endif
```

The `elseif` subexpression in braces can be repeated any number of times. The `else` subexpression in brackets is optional.

Any number of statements (including none) can be inserted in place of the ellipses (...). If `booleanexpression` is `TRUE`, the `then` statements are executed; if `booleanexpression` is `FALSE`, the `else` statements (if any) are executed instead. Note that `endif` is required for both forms and that no other delimiters (such as `BEGIN` or `END`) are used, even when multiple statements are inserted. Nesting of `if` statements (the use of `if` statement as part of another `if` statement) is allowed, but make sure that each `if` has a corresponding `endif`. Nested `if...endif` statements tend to result in long, confusing lists of `endif` keywords. Often, this can be avoided by using the `elseif` keyword. Any number of `elseif`

statements can be included in an `if...endif` expression. Only one of the `if`, `elseif`, or `else` clauses will be executed.

The following example uses a simple `if ... then` conditional statement:

```
"error --- Check for error conditions"
if (pw>100) or (dl>30) or ((tn='H1') and (dhp='y'))
    then write('line3','Problem with acquisition
parameters')
endif
```

The following example adds an `else` conditional statement:

```
"checkpw --- Check pulse width against predefined limits"
if pw<1
    then pw=1 write('line3','pw too small')
    else if pw>100
        then pw=100 write('line3','pw too large')
    endif
endif
```

The following example illustrates the use of `elseif` conditional statements:

```
if ($1='mon') then
    echo('Monday')
elseif ($1 = 'tue') then
    echo('Tuesday')
elseif ($1 = 'wed') then
    echo('Wednesday')
elseif ($1 = 'thu') then
    echo('Thursday')
elseif ($1 = 'fri') then
    echo('Friday')
else
    echo('Weekend')
endif
```

## Loops

Two types of loops are available. The `while` loop has the following syntax:

```
while booleanexpression do ... endwhile
```

This type of loop repeats the statements between `do` and `endwhile`, as long as `booleanexpression` is `TRUE` (if `booleanexpression` is `FALSE` from the start, the statements are not executed).

The other type of loop is the `repeat` loop, which has the following syntax:

```
repeat ... until booleanexpression
```

This loop repeats statements between `repeat` and `until`, until `booleanexpression` becomes `TRUE` (if



booleanexpression is TRUE at the start, the statements are executed once).

The essential difference between `repeat` and `while` loops is that the `repeat` type always performs the statements at least once, while the `while` type may never perform the statements. The following macro is an example of using the `repeat` loop:

```
"maxpk(first,last) -- Find tallest peak in a series of
spectra"
$first=$1
repeat
  select($1) peak:$ht
  if $1=$first
    then $maxht=$ht
    else if $ht>$maxht then $maxht=$ht endif
  endif
  $1=$1+1
until $1>$2
```

Both types of loops are often preceded by `$n=1`, then have a statement such as `$n=$n+1` within the loop to increment some looping condition. Beware of endless loops!

## Macro length and termination

Macros have no restriction on length. Execution of a macro is terminated either after the last instruction / line is executed, or when the command `return` is encountered. This is usually inserted into the macro after testing some condition, as shown in the following example:

```
"plotif--Plot a spectrum if tallest peak less than 200 mm"
peak:$ht
if $ht>200 then return else pl endif
```

The syntax `return(expression1,expression2,...)` allows the macro to return values to another calling macro, just like `do` commands. This information is captured by the calling macro using the format `:argument1,argument2,...`. The following example returns a value to the calling macro:

```
"abs(input):output -- Take absolute value of input"
if $1>0 then return($1) else return(-$1) endif
```

In nested macros, `return` terminates the currently operating macro, but not the macro that called the current macro.

To terminate the action of the calling macro (and all higher levels of nesting), the `abort` command is provided. `abort` can be made to act like `return` at any particular level by using the `abortoff` command. Consider the following sequence:

```
abortoff macro1 macro2
```

If `macro1` contains an `abort` command and it is executed,

`abort` terminates `macro1`; however, `macro2` will still be executed. If the macro sequence did not contain the `abortoff` statement, however, execution of an `abort` command in `macro1` would have prevented the operation of `macro2`. The `aborton` command nullifies the operation of `abortoff` and restores the normal functioning of `abort`.

An alternative mechanism to the `abort` mechanism is to use the `exec` command. This allows a macro to execute another macro and determine if the called macro aborted or not. The calling macro can then decide whether it should abort or continue. See the CPR entry for the `exec` command for additional details.

## Command and macro tracing

In VnmrJ, we send the output to any terminal window. In the terminal window type `'tty'`; reply is `/dev/pts/xx`, in which `xx` is a number. Use this on the VnmrJ command line `jFunc(55, '/dev/pts/xx')`. Replace `xx` with the correct number. Alternatively, the tracing output can be routed into a file, e.g., with `jFunc(55, userdir+'vnmrj_trace')`.

The commands `debug('c')` and `debug('C')` turn on and off, the VnmrJ command and macro tracing, respectively. When tracing is on, a list of each executed command and macro is displayed in the terminal window from which VnmrJ was started. Nesting of the calls is shown by indentation of the output. A return status of “returned” or “aborted” can help track down the macro or command that failed.

The `debug` command has options ‘c2’ and ‘c3’ for additional output. The ‘c2’ option prints the arguments and return values for all command and macro calls. The ‘c3’ option additionally shows all parameter assignments. The Magical debugger will display the line number of an error if the macro fails. If a command called from a macro fails, the debugger will show the line number if the `debug` option is turned on. The ‘c0’ option to `debug` causes the line number of a failed command to be displayed, without all the other tracing output. See the CPR entry for `debug` for more details.

## Relevant VnmrJ Commands

- Spectral Analysis tools
- Input/output tools
- Regression and Curve fitting
- Mathematical functions
- Creating, Modifying, and Displaying macros
- Miscellaneous tools

Many VnmrJ commands are particularly well-suited for use with MAGICAL programming. This section lists some of those commands with their syntax (if the command uses arguments) and a short summary taken from the *VnmrJ Command and Parameter Reference*. Refer to that publication for more information. (Remember that string arguments must be enclosed in single quotes.)

### Spectral analysis tools

<b>dres</b>	<b>Measure linewidth and digital resolution</b>
Syntax:	<code>dres&lt;(&lt;frequency&lt;,fractional_height&gt;&gt;)&gt; \</code> <code>:linewidth,resolution</code>
Description:	Analyzes line defined by current cursor position ( <code>cr</code> ) for linewidth and digital resolution. <code>frequency</code> overrides <code>cr</code> as the line frequency. <code>fractional_height</code> specifies the height at which linewidth is measured.
<b>dsn</b>	<b>Measure signal-to-noise</b>
Syntax:	<code>dsn&lt;(low_field,high_field)&gt;:signal_to_noise</code> <code>,noise</code>
Description:	Measures signal-to-noise of the tallest peak in the displayed spectrum. Noise region, in Hz, is specified by supplying <code>low_field</code> and <code>high_field</code> frequencies or it is specified by the positions of the left and right cursors.
<b>dsnmax</b>	<b>Calculate maximum signal-to-noise</b>
Syntax:	<code>dsnmax&lt;(noise_region)&gt;</code>
Description:	Finds best signal-to-noise in a region. <code>noise_region</code> , in Hz, can be specified, or the cursor difference ( <code>delta</code> ) can be used by default.

<b>getll</b>	<b>Get line frequency and intensity from line list</b>
Syntax:	<code>getll(line_number)&lt;:height,frequency&gt;</code>
Description:	Returns the height and frequency of the specified line number.
<b>getreg</b>	<b>Get frequency limits of a specified region</b>
Syntax:	<code>getreg(region_number)&lt;:minimum,maximum&gt;</code>
Description:	Returns the minimum and maximum frequencies, in Hz, of the specified region number.
<b>integ</b>	<b>Find largest integral in specified region</b>
Syntax:	<code>integ&lt;(highfield,lowfield)&gt;&lt;:size,value&gt;</code>
Description:	Finds the largest absolute-value integral in the specified region or the total integral if no reset points are present between the specified limits. The default values for <code>highfield</code> and <code>lowfield</code> are parameters <code>sp</code> and <code>sp+wp</code> , respectively.
<b>mark</b>	<b>Determine intensity of the spectrum at a point</b>
Syntax:	<code>mark&lt;(f1_position)&gt;</code> <code>mark&lt;(left_edge,region_width)&gt;</code> <code>mark&lt;(f1_position,f2_position)&gt;</code> <code>mark&lt;(f1_start,f1_end,f2_start,f2_end)&gt;</code> <code>mark&lt;('trace',&lt;options&gt;)&gt;</code> <code>mark('reset')</code>
Description:	1D or 2D operations can be performed in the cursor or box mode for a total of four separate functions. In the cursor mode, the intensity at a particular point is found. In the box mode, the integral over a region is calculated. For 2D operations, this is a volume integral. In addition, the <code>mark</code> command in the box mode finds the maximum intensity and the coordinate(s) of the maximum intensity.
<b>nll</b>	<b>Find line frequencies and intensities</b>
Syntax:	<code>nll&lt;('pos'&lt;,noise_mult)&gt;&gt;&lt;:number_lines&gt;</code>
Description:	Returns the number of lines using the current threshold, but does not display or print the line list.
<b>numreg</b>	<b>Return the number of regions in a spectrum</b>
Syntax:	<code>numreg:number_regions</code>

Description: Finds the number of regions in a previously divided spectrum.

**peak Find tallest peak in specified region**

Syntax: `peak<(min_frequency,max_frequency)><:height ,freq>`

Description: Finds the height and frequency of the tallest peak in the selected region. `min_frequency` and `max_frequency` are the frequency limits, in Hz, of the region to be searched; default values are the parameters `sp` and `sp+wp`.

**select Select spectrum or 2D plane without displaying it**

Syntax: `select<(<'f1f3'|'f2f3'|'f1f2'><,'proj'> \ <'next'|'prev'|plane>)><:index>`

Description: Sets future actions to apply to a particular spectrum in an array or to a particular 2D plane of a 3D data set. `index` is the index number of spectrum or 2D plane.

## Input/Output tools

**apa Plot parameters automatically**

Description: Selects the appropriate command on different devices to plot the parameter list.

**banner Display message with large characters**

Syntax: `banner(message< ,color>< ,font>)`

Description: Displays the text given by `message` as large-size characters on the VNMR graphics windows.

**clear Clear a window**

Syntax: `clear<(window_number)>`

Description: Clears window given by `window_number`. With no argument, clears the text screen. `Clear(2)` clears the graphics screen.

**echo Display strings and parameter values in text window**

Syntax: `echo<(<'-n',>string1,string2,...)>`

Description: Functionally similar to the UNIX `echo` command. Arguments to VNMR `echo` can be strings or parameter values, such as `pw`. The `'-n'` option suppresses advancing to the next line.

<b>format</b>	<b>Format a real number or convert a string for output</b>
Syntax:	<code>format(real_number,length,precision):string_var</code> <code>format(string,'upper' 'lower' 'isreal'):return_var</code>
Description:	Using first syntax, takes a real number and formats it into a string with the given length and precision. Using second syntax, converts a string variable into a string of characters, all upper case or all lowercase, or tests the first argument to verify that it satisfies the rules for a real number (1 is returned if the first argument is a real number, otherwise a zero is returned).
<b>input</b>	<b>Receive input from keyboard</b>
Syntax:	<code>input(&lt;prompt&gt;&lt;,delimiter&gt;):var1,var2,...</code>
Description:	Receives characters from the keyboard and stores them into one or more string variables. <code>prompt</code> is a string that is displayed on the command line. The default <code>delimiter</code> is a comma.
<b>lookup</b>	<b>Look up and return words and lines from text file</b>
Syntax:	<code>lookup(options):return1,return2,...,number_re</code> <code>turned</code>
Description:	Searches a text file for a word and returns to the user subsequent words or lines. <code>options</code> is one or more keywords ('file', 'seek', 'skip', 'read', 'readline', 'count', and 'delimiter') and other arguments.
<b>nrecords</b>	<b>Determine number of lines in a file</b>
Syntax:	<code>nrecords(file):\$number_lines</code>
Description:	Returns the number of "records," or lines, in the given file.
<b>psgset</b>	<b>Set up parameters for various pulse sequences</b>
Syntax:	<code>psgset(file,param1,param2,...,paramN)</code>
Description:	Sets up parameters for various pulse sequences using information in a file from the user or system <code>parlib</code> .
<b>write</b>	<b>Write output to various devices</b>

**Syntax:** `write('graphics'|'plotter'<,color|pen> \<,<'reverse'>,x,y<,template><:height> write('alpha'|'printer'|"line3"|'error',template) write('reset'|"file",file<,template>)`

**Description:** Displays strings and parameter values on various output devices.

## Regression and curve fitting

**analyze**                    **Generalized curve fitting**

**Syntax:**                    (Curve fitting)  
`analyze('expfit',xarray<,options>)`  
 (Regression)  
`analyze('expfit','regression'<,options>)`

**Description:** Provides an interface to the curve fitting program `expfit`, supplying input data in the form of the text file `analyze.inp` in the current experiment.

**autoscale**                    **Resume autoscaling after limits set by scalelimits**

**Description:** Returns to autoscaling in which the scale limits are determined by the `expl` command such that all the data in the `expl` input file is displayed.

**expfit**                        **Least-squares fit to exponential or polynomial curve**

**Syntax:**                    `expfit options <analyze.inp >analyze.list`

**Description:** A command that takes a least-squares curve fitting to the data supplied in the file `analyze.inp`.

**expl**                         **Display exponential or polynomial curves**

**Syntax:**                    `expl(<options,>line1,line2,...)>`

**Description:** Displays exponential curves resulting from  $T_1$ ,  $T_2$ , or kinetic analyses. Also displays polynomial curves from diffusion or other types of analysis.

**pexpl**                        **Plot exponential or polynomial curves**

**Syntax:**                    `pexpl(<options><,line1,line2,...)>`

**Description:** Plots exponential curves from  $T_1$ ,  $T_2$ , or kinetics analysis. Also plots polynomial curves from diffusion or other types of analysis.

**poly0**                        **Display mean of the data in the file regression.inp**

Description: Calculates and displays the mean of data in the file `regression.inp`.

**rinput** **Input data for a regression analysis**

Description: Formats data for regression analysis and places it into the file `regression.inp`.

**scalelimits** **Set limits for scales in regression**

Syntax: `scalelimits(x_start,x_end,y_start,y_end)`

Description: Causes the command `expl` to use typed-in scale limits.

## Mathematical functions

**abs** **Find absolute value of a number**

Syntax: `abs(number)<:value>`

Description: Finds the absolute value of a number.

**acos** **Find arc cosine of a number**

Syntax: `acos(number)<:value>`

Description: Finds the arc cosine of a number. The optional return `value` is in radians.

**asin** **Find arc sine of a number**

Syntax: `asin(number)<:value>`

Description: Finds the arc sine of a number. The optional return `value` is in radians.

**atan** **Find arc tangent of a number**

Syntax: `atan(number)<:value>`

Description: Finds the arc tangent of a number. The optional return `value` is in radians.

**atan2** **Find arc tangent of two numbers**

Syntax: `atan2(y,x)<:value>`

Description: Finds the arc tangent of  $y/x$ . The optional return argument `value` is in radians.

**averag** **Calculate average and standard deviation of input**

Syntax: `averag(num1,num2,...)\  
:average,sd,arguments,sum,sum_squares`

Description: Finds average, standard deviation, and other characteristics of a series of numbers.



<b>cos</b>	<b>Find cosine value of an angle</b>
Syntax:	<code>cos(angle)&lt;:value&gt;</code>
Description:	Finds the cosine of an angle given in radians.
<b>exp</b>	<b>Find exponential value of a number</b>
Syntax:	<code>exp(number)&lt;:value&gt;</code>
Description:	Finds the exponential value (base $e$ ) of a number.
<b>ln</b>	<b>Find natural logarithm of a number</b>
Syntax:	<code>ln(number)&lt;:value&gt;</code>
Description:	Finds the natural logarithm of a number. To convert to base 10, use <code>log10x = 0.43429 * ln(x)</code> .
<b>sin</b>	<b>Find sine value of an angle</b>
Syntax:	<code>sin(angle)&lt;:value&gt;</code>
Description:	Finds the sine of an angle given in radians.
<b>tan</b>	<b>Find tangent value of an angle</b>
Syntax:	<code>tan(angle)&lt;:value&gt;</code>
Description:	Finds the tangent of an angle given in radians.

## Creating, modifying, and displaying macros

<b>crcom</b>	<b>Create a user macro without using a text editor</b>
Syntax:	<code>crcom(file,actions)</code>
Description:	Creates a user macro file in the user's macro directory. The actions string is the contents of the new macro.
<b>delcom</b>	<b>Delete a user macro</b>
Syntax:	<code>delcom(file)</code>
Description:	Deletes a user macro file in the user's macro directory. The actions string is the contents of the new macro.
<b>hidecommand</b>	<b>Execute macro instead of command with same name</b>
Syntax:	<code>hidecommand(command_name)&lt;:\$new_name&gt;</code> <code>hidecommand('?')</code>

**Description:** Renames a built-in VNMR command so that a macro with the same name as the built-in command is executed instead of the built-in command. `command_name` is the name of the command to be renamed. '?' displays a list of renamed built-in commands.

**macrocat** **Display a user macro on the text window**

**Syntax:** `macrocat(file1<,file2><,...>)`

**Description:** Displays one or more user macro files, in which `file1`, `file2...` are names of macros in the user macro directory.

**macrocp** **Copy a user macro file**

**Syntax:** `macrocp(from_file,to_file)`

**Description:** Makes a copy of an existing user macro.

**macrodir** **List user macros**

**Description:** Lists the names of user macros.

**macroedit** **Edit a user macro with user-selectable editor**

**Syntax:** `macroedit(file)`

**Description:** Modifies an existing user macro or creates a new macro. To edit a system macro, copy it to a personal macro directory first.

**macrold** **Load a macro into memory**

**Syntax:** `macrold(file)<:dummy>`

**Description:** Loads a macro, user or system, into memory. If the macro already exists in memory, it is overwritten by the new macro. Including a return value suppresses the message on line 3 that the macro is loaded.

**macrorm** **Remove a user macro**

**Syntax:** `macrorm(file)`

**Description:** Removes a user macro from the user macro directory.

**macroscopy** **Display a system macro on the text window**

**Syntax:** `macroscopy(file1<,file2><,...>)`

**Description:** Displays one or more system macro files, in which `file1`, `file2...` are names of macros in the system macro directory.

**macroscopy** **Copy a system macro to become a user macro**

**Syntax:** `macroscopy(from_file,to_file)`

**Description:** Makes a copy of an existing system macro.

**macroscopy** **List system macros**

**Description:** Lists the names of system macros.

<b>macrosysrm</b>	<b>Remove a system macro</b>
Syntax:	<code>macrosysrm(file)</code>
Description:	Removes a system macro from the macro directory.
<b>macrovi</b>	<b>Edit a user macro with vi text editor</b>
Syntax:	<code>macrovi(file)</code>
Description:	Modifies an existing user macro or creates a new macro using the <code>vi</code> text editor. To edit a system macro, copy it to a personal macro directory first.
<b>mstat</b>	<b>Display memory usage statistics</b>
Syntax:	<code>mstat&lt;(program_id)&gt;</code>
Description:	Displays the memory usage statistics of macros loaded into memory.
<b>purge</b>	<b>Remove a macro from memory</b>
Syntax:	<code>purge&lt;(file)&gt;</code>
Description:	Removes a macro from memory, freeing extra memory space. With no argument, removes all macros loaded into memory through <code>macrold</code> .
<b>record</b>	<b>Record keyboard entries as a macro</b>
Syntax:	<code>record&lt;(file 'off')&gt;</code>
Description:	Records keyboard entries and stores the entries as a macro file in the user's <code>maclib</code> directory.

## Miscellaneous tools

<b>axis</b>	<b>Provide axis labels and scaling factors</b>
Syntax:	<code>axis('fn' 'fn1' 'fn2')&lt;:\$axis_label, \ \$frequency_scaling,\$factor&gt;</code>
Description:	Returns axis labels, the divisor to convert from Hz to units defined by the <code>axis</code> parameter with any scaling, and a second scaling factor determined by any <code>scalesw</code> type of parameter. The parameter <code>'fn' 'fn1' 'fn2'</code> describes the Fourier number for the axis.
<b>beepoff</b>	<b>Turn beeper off</b>
Description:	Turns the beeper sound off. The default is beeper sound on.
<b>beepon</b>	<b>Turn beeper on</b>
Description:	Turns the beeper sound on. The default is beeper sound on.

<b>bootup</b>	<b>Macro executed automatically when VnmrJ is started</b>
Syntax:	<code>bootup&lt;(foreground)&gt;</code>
Description:	Displays a message, runs a user login macro (if it exists), starts Acqstat and acqi (spectrometer only), and displays the menu system. bootup and login can be customized for each user (login is preferred because bootup is overridden when a new VNMR release is installed). foreground is 0 if VNMR is being run in foreground, non-zero otherwise.
<b>exec</b>	<b>Execute a VnmrJ command</b>
Syntax:	<code>exec (command_string)</code>
Description:	Takes as an argument a character string constructed from a macro and executes the VNMR command given by <code>command_string</code> .
<b>exists</b>	<b>Determine if a parameter, file, or macro exists</b>
Syntax:	<code>exists(name,type):\$exists</code>
Description:	Checks for the existence of a parameter, file, or macro with the given name. type is 'parameter', 'file', 'maclib', 'ascii', 'directory' or filename. See the <i>Command and Parameter Reference</i> manual for a detailed description of the use of exists for Applications Directory usage.
<b>focus</b>	<b>Send keyboard focus to VNMR input window</b>
Description:	Sends the keyboard focus to the VNMR input window.
<b>gap</b>	<b>Find gap in the current spectrum</b>
Syntax:	<code>gap(gap,height):found,position,width</code>
Description:	Looks for a gap between lines of the currently displayed spectrum, in which gap is the width of the desired gap and height is the starting height. found is 1 if search is successful, or 0 if unsuccessful.
<b>getfile</b>	<b>Get information about directories and files</b>
Syntax:	<code>getfile(directory,file_index):\$file,\$file_extension</code> <code>getfile(directory):\$number_files</code>

**Description:** If `file_index` is specified, the first return argument is the name of the file in the directory with the index `file_index`, excluding any extension, and the second return argument is the extension. If `file_index` is not specified, the return argument contains the number of files in the directory (dot files are not included in the count).

**graphis**      **Return the current graphics display status**

**Syntax:** `graphis(command):$yes_no`  
`graphis:$display_command`

**Description:** Determines what command currently controls the graphics window. If no argument is supplied, the name of the currently controlling command is returned.

**length**      **Determine length of a string**

**Syntax:** `length(string):$string_length`

**Description:** Determines the length in characters of the given string.

**listenoff**      **Disable receipt of messages from send2Vnmr**

**Description:** Deletes the file `$vnmruser/.talk`, disallowing UNIX command `send2Vnmr` to send commands to VNMR.

**listenon**      **Enable receipt of messages from send2Vnmr**

**Description:** Writes files with VNMR port number that UNIX command `send2Vnmr` needs to talk to VNMR. Then, the command to send commands to VNMR is  
`/vnmr/bin/send2Vnmr $vnmruser/.talk`  
`command`  
 in which `command` is any character string (commands, macros, or if statements) normally typed into the VNMR input window.

**login**      **User macro executed when VnmrJ activated**

**Description:** When VNMR starts, the `bootup` macro executes, and then, if the `login` macro exists, `bootup` executes the `login` macro. By creating and customizing the `login` macro, a VNMR session can be tailored for an individual user. The `login` macro does not exist by default.

**off**      **Make a parameter inactive**

<b>Syntax:</b>	<code>off(parameter 'n'&lt;,tree&gt;)</code>
<b>Description:</b>	Makes a parameter inactive. tree is 'current', 'global', 'processed', or 'systemglobal'.
<b>on</b>	<b>Make a parameter active or test its state</b>
<b>Syntax:</b>	<code>on(parameter 'y'&lt;,tree&gt;&lt;:\$active&gt;</code>
<b>Description:</b>	Makes a parameter active or tests the active flag of a parameter. tree is 'current', 'global', 'processed', or 'systemglobal'.
<b>readlk</b>	<b>Read current lock level</b>
<b>Syntax:</b>	<code>readlk&lt;:lock_level&gt;</code>
<b>Description:</b>	Returns the same information as would be displayed on the digital lock display using the manual shimming window. It cannot be used during acquisition or manual shimming, but can be used to develop automatic shimming methods such as shimming via grid searching.
<b>rtv</b>	<b>Retrieve individual parameters</b>
<b>Syntax:</b>	<code>rtv&lt;(file,par1&lt;,index1&lt;,par2,index2...&gt;&gt;&gt;&lt;:val&gt;</code>
<b>Description:</b>	Retrieves one or more parameters from a parameter file to the experiment's current tree. If a return argument is added, <code>rtv</code> returns values to macro variables instead, which avoids creating additional parameters in the current tree. For arrayed parameters, array index arguments can specify the elements to be returned to the macro. The default is the first element.
<b>shell</b>	<b>Start a UNIX shell</b>
<b>Syntax:</b>	<code>shell&lt;(command)&gt;:\$var1,\$var2,...</code>
<b>Description:</b>	If no argument is given, opens a normal UNIX shell. If a UNIX command is entered as an argument, <code>shell</code> executes the command. Text lines usually displayed as a result of the UNIX command given in the argument can be returned to <code>\$var1</code> , <code>\$var2</code> , etc. <code>shell</code> calls involving pipes or input redirection (<) require either an extra pair of parentheses or the addition of ; cat to the shell command string, such as: <code>shell('ls -t grep May; cat')</code> <b>or</b> <code>shell('(ls -t grep May))</code>

- solppm**      **Return ppm and peak width of solvent resonances**  
 Syntax:      `solppm:chemical_shift,peak_width`  
 Description: Returns information about the chemical shift in ppm and peak spread of solvent resonances in various solvents for either  $^1\text{H}$  or  $^{13}\text{C}$ , depending on the observe nucleus `tn` and the solvent parameter `solvent`. This macro is used “internally” by other macros only.
- substr**      **Select a substring from a string**  
 Syntax:      `substr(string,word_number):substring`  
               `substr(string,index,length):substring`  
 Description: Picks a substring from a string. If two arguments are given, `substring` returns the `word_number` word in `string`. If there are three arguments, it returns a substring from `string` in which `index` is the number of the character at which to begin and `length` is the length of the substring.
- textis**      **Return the current text display status**  
 Syntax:      `textis(command):$yes_no`  
               `textis:$display_command`  
 Description: Determines what command currently controls the text window. If no argument is supplied, the name of the current controlling command is returned.
- unit**      **Define conversion units**  
 Syntax:      `unit<(suffix,label,m<,tree><,'mult'|'div'>,\ \`  
               `b<,tree><,'add'|'sub'>>`  
 Description: Defines a linear relationship that can be used to enter parameters with units. The unit is applied as a suffix to the numerical value (e.g., 10k, 100p). `suffix` identifies the name for the unit (e.g., 'k'). `label` is the name to be displayed when the axis parameter is set to the value of the suffix (e.g., 'kHz'). `m` and `b` are the slope and intercept, respectively, of the linear relationship. A convenient place to put unit commands for all users is in the `bootup` macro. Put private unit commands in a user's `login` macro.

## 1 MAGICAL II Programming





## 2 Pulse-Sequence Programming

Overview of Pulse-Sequence Programming	50
Pulse-Sequence Statements	64
Real-Time Control of Pulse Sequences	83
Shaped Pulses and Waveforms	95
Programming for Acquisition Control	110
Multidimensional NMR and Arrays	121
Syntax for Controlling Parallel Channels	130
Parameters and Variables	141
Pulse Sequence Output	155
Setting the Amplitude, Phase, and Gate from Tables	159
Gradient Control for PFG and Imaging	161



## Overview of Pulse-Sequence Programming

This section provides an overview of pulse-sequence programming in VnmrJ.

### Overview of pulse-sequence execution

Pulse sequences are written in C, a high-level programming language that allows considerable sophistication in the way pulse sequences are created and executed. A new pulse sequence is written as a C function called `pulsesequenc`. The file containing this function is compiled and linked with an object library that contains the definitions for all pulse-sequence statements, the PSG.

A compiled C sequence is executed on the Linux workstation at the start of acquisition, so-called run-time. At *run-time*, the sequence reads values from a parameter table and constructs a second *real-time* program of *acodes*, whose purpose will be to run on the *controllers* in the acquisition computer of the VNMRS.

At run-time, the compiled C-program may also use text files in the directory `shapelib`, whose members have the extensions `.RF`, `.DEC`, and `.GRD`, so-called *shape*, *waveform*, or *gradient* files. These files contain amplitude, phase and gate information for shaped pulses, waveforms, and gradients. The files in `shapelib` are read at run-time and stored in the controllers in binary form before the first scan. For modern pulse programs, *shape*, *waveform* and *gradient* files are usually created by C functions in the pulse program itself, but they might also be prewritten by another program, for example `Pbox`, or simply with a text editor.

The PSG library contains C code to run the `pulsesequenc` function in multidimensional loops of up to four dimensions and to run spectra corresponding to arrays of parameter values. The user need not program these loops explicitly.

At the start of the acquisition, all of the variables and statements of the compiled C-program, including the C-loops and conditionals, are resolved and fixed into *acodes*, without the possibility of further input or calculation. A complex program with many choices, using the C `if-else-endif` statement, may resolve into a very few *acodes*, because the *acodes* in the non-selected branches are never created. A pulse program with multidimensional looping and/or parameter arrays will produce a separate set of *acodes* for

every increment and therefore, may use considerable memory. The same is true for the C `for` and `while` statements.

The real-time acode program is automatically looped over the number of scans for each multidimensional increment or array element. Special *real-time* integer tables, `t1` to `t60` and *real-time integer variables*, `v1` to `v42`, are used to increment phases and other values that might change scan-to-scan. These tables and variables are initialized and manipulated by special pulse-sequence statements, the real-time math statements.

The `loop-endloop`, `rlloop-rlendloop`, and `kzloop-kzendloop` statements can execute an explicit *real-time* loop within a single scan and the `ifzero-elsenz-endif` statement can make a *real-time choice*, based upon a real-time integer variable as an argument.

Sections of the pulse sequence can be written as a parallel section, where individual hardware controllers are programmed independently. This provides a mechanism to program events that occur simultaneously on the different channels.

The stored shape and waveform files of `shapelib` can also be accessed and looped in real-time, though this mechanism is different from that used for `loop-endloop`. It should be recognized that the use of a real-time loop with tables and the use of a waveform or shape are alternative approaches to obtain the same result. Each approach has its own programming requirements and performance. Real-time looping has become the favored method for imaging applications, while the use of shapes and waveforms has been favored for spectroscopy.

A multidimensional pulse program with many waveforms requires considerable calculation and a large data-transfer to the acquisition computer at run-time, but this kind of program will place a lower burden on the controllers. A pulse program that makes heavy use of real-time loops and calculation will take less time to start but it could have lower performance in real-time.

## Compiling a pulse sequence with the PSG

Pulse-sequence text files, such as the listing of `hom2dj.c` in [Table 3](#), are stored in a directory named `psglib` in either the system directory (`/vnmr/psglib`) or in a user directory

(/home/vnmr1/vnmrsys/psglib for the user vnmr1). A pulse-sequence file has the extension `.c` to indicate that it contains C-language source code. Pulse sequences may also be saved in the `psglib` directory of an *applications directory*, which may have any name and path. An applications directory is made accessible through the Edit Applications tool of the Files pull down menu.

A pulse-sequence text file can be modified using the Linux tool *vi*, the standard Red Hat editor *gedit*, or by an available text editor or development package.

**Table 3** Partial listing for the `hom2dj.c` pulse sequence

```
#include <standard.h>
void pulsesequence()
{
    initval(4.0,v9); divn(ct,v9,v8);
    status(A);
    hsdelay(d1);
    status(B);
    add(zero,v8,v1); pulse(pw,v1);
    delay(d2/2.0);
    mod4(ct,v1); add(v1,v8,v1); pulse(p1,v1);
    delay(d2/2.0);
    status(C);

    mod2(ct,oph); dbl(oph,oph); add(oph,v8,oph);
}

```

Pulse-sequence source code is compiled by one of the following methods:

- By entering `seqgen(filename<.c>)` on the VnmrJ command line.
- By entering `seqgen` on the VnmrJ command line, with `seqfil='filename'`
- By entering `seqgen filename<.c>` from a Linux shell in the `psglib` directory

For example, enter `seqgen('hom2dj')` to compile the `hom2dj.c` sequence in VnmrJ. A full path is not necessary from the command line. Alternatively, you can enter `seqgen hom2dj` in the `psglib` directory in a Linux shell. The `seqgen` command will first search the user, then the available applications directories, and then the system for a file with a `.c` extension.

During compilation, the system performs the following steps:

- 1 Extensions are added to the pulse sequence to allow a graphical display of the sequence, using the `dps` command.
- 2 The source code is passed through the Linux program *lint* to check for syntax, variable consistency, and the correct usage of functions.
- 3 The source code is converted into compiled object code.
- 4 If the conversion is successful, the object code is combined with the necessary system PSG object libraries (`libparam.so` and `libpsglib.so`), to be linked at run-time. If the compilation of the pulse sequence with the `dps` extensions fails, the pulse sequence is recompiled without the `dps` extensions.

The executable code is stored in the user `seqlib` directory (for example, `/home/vnmr1/vnmrsys/seqlib`. If the user does not have a `seqlib` directory, it is automatically created. A copy of the source code is also saved in `vnmrsys/seqlib` of the user directory. If desired, you can copy the compiled sequence to `/vnmr/seqlib` or to the `seqlib` directory in an applications directory.

Many standard, compiled sequences are supplied in `/vnmr/seqlib` and the source code for each of these sequences is found in `/vnmr/psglib`. To recompile one of these sequences or to modify it, first copy the sequence into the user `psglib`, make the required modifications, and then recompile the sequence using `seqgen`. Sequences can only be compiled from a user directory. If you attempt to compile a system sequence, a local copy will be created. The `seqgenupdate` command performs a `seqgen` as the first step, and will then attempt to move the resulting `seqlib` entries back to the application directory from which they were taken.

The source files that are used to create the PSG object library are contained in the system directory `/vnmr/psg`. In principle, a user can customize and recompile the PSG source files, but most users do not do so. It is easier to add the user source code in a separate file using the standard C `#include` statement. User `#include` files should be stored in the `/vnmrsys/psg` directory of a user or an applications directory.

## Troubleshooting a new pulse sequence

During the compilation process, the user-written C procedure is passed through a utility to identify incorrect C syntax or potential coding problems. If an error occurs, messages are displayed in the Text panel of the Process tab of the VnmrJ interface. The error messages are also saved in a file with a `.err` extension in `psglib` and `seqlib`. A typical error report is shown below:

```
Pulse sequence did not compile.
The following errors can also be found in the
file /home/vnmr1/vnmrsys/psglib/name.errors:
```

Errors begin with the name of the pulse sequence enclosed in double quotes, followed by the line number and a brief description of the problem. For sequences that employ `#include` statements, the name of the included file will be identified. You should usually correct errors from the top down. Often, errors can result from mistakes in lines other than that identified. Sometimes simple errors in the placement of braces can generate many spurious error messages.

If a warning occurs, the following message is displayed:

```
Pulse sequence did compile but may not function properly.
The following comments can also be found in the
file /home/vnmr1/vnmrsys/psglib/name.errors:
```

A warning message reveals problems with the C-syntax of a compiled sequence that may or may not prevent operation of the sequence. It is usually a good practice to address all warnings.

At times, a sequence will compile properly in one version of VnmrJ but will give warnings in another. This is caused by the changing standards of the C compilers in different versions of VnmrJ.

You must keep a watch for the following three typical warnings:

```
warning: conversion from long may lose accuracy
warning: parameter_name may be used before set
warning: parameter_name redefinition hides earlier one
```

The first warning is often generated from older code that contains a calculation with the overall array index, `ix`. While the error itself is usually not a problem, it is usually appropriate to replace the use of `ix` with a call to one of the `nD` indexes. Some examples are shown in the section

discussing *Multidimensional Experiments and Arrays*.

The second warning indicates an un-initialized variable. All C variables must have an initial value before they are used. This error is often caused by variables that are first initialized within a loop or conditional. Assign these variables a dummy value such as 0, or change the logic of the loop or conditional.

The third warning indicates that a local variable defined in the pulse sequence has the same name as one of the standard PSG variables. The appendix provides a list of all *global* variables defined in the PSG. It is not necessary to redefine these values if their default behavior is required and the redefinition of one of these names for another purpose may have unpredictable consequences. This warning is normally avoided by removing the redefinition or by renaming the variable in the pulse-sequence file if it has a different purpose.

If the pulse-sequence program is syntactically correct, the following message is displayed:

```
Done! Pulse sequence now ready to use.
```

A compiled sequence may still have errors at run-time. These errors are caused by mistakes in the program logic and more subtle C-syntax errors. One of the most common is:

```
Segmentation Violation: Index overruns boundary of an array.
```

A segmentation violation is often caused by enclosing a string within single quotes. All C strings should be enclosed in double quotes. Segmentation violations are not identified with a line number and so they can be tedious to fix. The best approach is to comment out parts of the pulse sequence until the error is fixed. Then reinsert the code block-by-block to identify the location of the error. It is also helpful to insert `printf` statements into the sequence to print out parameter values or to reveal a particular location in code. The statement `printf` prints in the Text page of the Process panel.

## Creating a parameter table for a new sequence

A compiled pulse sequence requires a parameter table. If a sequence is written with a `getval` or a `getstr` statement for a new parameter name, that parameter must be added to the parameter table. If the parameter is absent, the system will output a warning and supply a default. The sequence may or

may not run with the default value.

The PSG object library contains a group of global PSG variables (designated `extern`) that are known to the `pulsesequenc` function. Many of these variables are initialized automatically from the parameter table. Their absence may cause both the sequence and the VnmrJ interface to fail.

It is a good practice to run a new pulse sequence first with the parameter table of a similar sequence. A parameter set can be loaded with the command `rtp` from the command line or you can select the related experiment from the Experiments pull-down menu or the Experiment Selector. Set `seqfil` to the new pulse-sequence name and attempt to run. The error messages will indicate the new parameters that are needed.

Most parameters that you enter from the interface, including those that initialize *global* PSG variables, are declared to be in the `current` tree. At the completion of acquisition, all `current` parameters are copied to the `processed` tree and the command `svf` saves the `processed` parameters. Both `current` and `processed` parameters are known only to a single workspace. A small number of `global` and `systemglobal` parameters are known to any parameter tree in a user or system-wide. The procedures for creating and managing parameters are discussed in [Chapter 5](#), “Parameters and Data” .

Note the difference between *global PSG variables* and `global` parameters. These are separate, unrelated entities, despite the fact that the word “global” is used in both names.

### C framework for pulse sequences

Each pulse sequence is a function in the C programming language named `pulsesequenc`. The `pulsesequenc` function contains pulse-sequence *statements* to control the spectrometer, which are defined in the PSG object library.

Every C function is followed by a pair of parentheses `()` for potential arguments and the code is included in a pair of braces `{}`. Every VnmrJ sequence must also contain the `#include` statement for `<standard.h>` at the top of the file. A sequence may also have additional `#include` statements to add optional user libraries of functions. Note that `largeuser` `#include` files may substantially lengthen compilation time.



```

#include <standard.h>
#include myinclude.h

void pulsedsequence()
{
.
.
    delay(d1);
.
}

```

The VnmrJ pulse-sequence language is a standard C-language compiler. Any statement that is described in a standard C manual can be used in a VnmrJ sequence. These statements include variable definitions, assignment statements, loops such as *for* and *while*, conditionals such as *if*, *else*, and the *switch* statement as well as advanced entities such as *structures* and *pointers*, if required. Standard C is used to control the structure of the program and do calculations. It has no effect on the program in real time.

VnmrJ pulse-sequence statements are defined in the PSG library and their purpose is to make codes that run in real-time. These statements control pulses, delays, frequencies, amplitudes, etc during the real-time execution of the sequence in the acquisition computer. Some pulse-sequence statements have an explicit time argument. Statements of this type are used to add a pulse or delay to the sequence. Other pulse-sequence statements simply set a state, for example a gate, the phase, or an amplitude. These statements add no time to the sequence.

There are no *AP-Bus* delays associated with the VNMRS hardware, as there were with Unity-series systems. There is one group of exceptions, the synthesizer frequency offset statements, each of which add about a 1.0  $\mu$ s delay to the sequence. The statements that set the coarse power also add 50 ns to the sequence.

The arguments of a pulse-sequence statement are variables of the C language as described in [Table 4](#).

**Table 4** C-Variable types used in pulse sequences

Type	Description	Length (bits)- use
<i>char</i>	character	8 - often used
<i>short</i>	short integer	16 - occasionally used in the PSG
<i>int</i>	integer	32- often used

**Table 4** C-Variable types used in pulse sequences

Type	Description	Length (bits)- use
<i>long</i>	long integer	32 - occasionally used in the PSG
<i>float</i>	floating point	32 - occasionally used in the PSG
<i>double</i>	double-precision floating point	64 - often used

Most pulse sequences make use of only the *int*, *char*, and *double* types. Strings are defined as arrays of *char* usually with the dimension `MAXSTR`. In C, strings are defined explicitly in double quotes, " ". In contrast, string parameter values in the VnmrJ interface are defined in single quotes, ' '. Failure to recognize this difference can be a source of programming errors.

Most variables are of type *double* and they are used to set values, such as delays, amplitudes, frequencies, etc, or they are used to do calculations. Arguments can be variable names (for example, `d1`), constants (for example, `3.4` or `20.0e-6`), expressions (for example, `2.0*pw`, `1.0-d2`) or functions (for example, `getval("pw");`). Delays with a value of zero are eliminated from the sequence. Negative delays are set to zero and give a warning.

Variables of the *int* type are usually used to index the C language loops and conditionals.

Arrays of *char* are used as flags (for example, `dmm`) or to represent parameter names as in the statement `getval("pw")`.

The VnmrJ PSG library describes a special set of *real-time integer variables*, named `v1` to `v42` (so called *v-variables*) and a set of *real-time integer tables*, `t1` to `t60`, that are provided to perform calculations scan-to-scan. These variables are special because their values in C point to memory locations in the acquisition computer. These *v-variables and tables* are initialized with specific pulse-sequence statements rather than C expressions. Other statements provide for calculation scan-to-scan using *integer real-time math* and *table math*.

V-variables and tables are used primarily to set the phase (for xample, `txphase(v1)` or `txphase(t1)`) and create phase tables. All phases and phase tables in the VnmrJ PSG are set as the product of a *double* phase-step and a real-time multiplier. Notably, there are no phase statements in the VnmrJ PSG that take a *double* as an argument.

V-variables are also used to index scans (*ct*, *ssctr*, and so on) and increments (*id2*, *id3* and so on).

The real-time integer variables and the real-time integer tables are represented in C as integer constants, whose values are internally defined indexes. Statements such as  $v1=v1+v2$  or  $t1=t3$  do math with the indexes, not the values, and have no meaning in the usual sense.

## Global PSG and real-time variables in multidimensional arrays

The PSG defines a set of *global C variables* to hold the values from standard parameters. The appendix lists many of these *global* variables. Usually a parameter that is associated with a *global* variable has the same name, and the value of the variable is set automatically. Global values are automatically known within the `pulsesequence` function and do not need a standard C definition. For multidimensional experiments and arrays, these variables hold their values increment-to-increment.

All other C *user variables* that are used in a pulse sequence must be explicitly defined. A `getval` or a `getstr` statement associates a parameter with the variable. User variables are updated with each new multidimensional increment or array element. They cannot store data increment-to-increment.

All of the variables that control multidimensional looping are *globals*. A user can create a new *global* user variable, by defining it as *static* in the sequence `.c` file, outside the `pulsesequence` function. For example the integer arrays that are used to create phase tables are defined in this way.

It is good practice to reinitialize *global* variables if you know that they should only have scope in a single increment. The extra `getval` statement does not hurt. Also beware of statements of the type `rof2=rof2+2.0e-6;` involving *globals*. Because `rof2` is a *global* variable it will increase by 2.0  $\mu$ s with each increment. This problem can be fixed by including the statement `getval("rof2")` in the `pulsesequence` function.

Real-time variables and tables, defined to exist on the acquisition computer, may persist increment to increment and a few real-time indexes (such as `id2`) count increments. However, most real-time variables and tables are designed for use within a single increment. It is good practice to reinitialize these variables with each increment and index v-variables to the scan counter `ct`. Never expect v-variables and tables to remain defined increment-to-increment.

Standard C-syntax is a linear list of C-language statements and pulse-sequence statements, referencing the RF channels, the gradients and the receivers. At run-time the compiled C program sorts the resulting acodes to their appropriate channels. By default, acodes are sorted with synchronous syntax. With this syntax any statement that takes time also places a corresponding delay on every other operational channel. Use of synchronous syntax ensures that acodes are executed sequentially. Statements that do not take time such as `xmtron` and `txphase` are all executed at the beginning of the next delay.

Alternatively the `parallelstart` and `parallelend` statements create parallel, *asynchronous* syntax. The `parallelstart` statement designates an individual channel to receive acodes and corresponding delays are not applied to other channels. Sequential `parallelstart` statements referencing different channels allow one to individually program the channels in parallel. The `parallelend` statement calculates a set synchronization delays to synchronously terminate the parallel channels and return to synchronous syntax. The `parallelsync` statement allows positioning of the synchronization delays in the parallel sections for each channel.

Parallel programming is also obtained through the use of waveforms and the `nowait` attribute of gradients.

The *interleave* function allows one to cycle through the increments of a multidimensional array for a value `bs` scans. Interleave requires that one save the real-time state of the increment and return to it later for the next `bs` scans and in this process real-time values can be lost. One can avoid any complexity from interleave by indexing all `v`-variable calculations to the scan counter `ct` and never expect `v`-variables to survive scan-to-scan.

### Assigning transmitters and receivers

Most pulse-sequence statements begin with the prefixes `obs`, `dec`, `dec2`, `dec3`, and `dec4`. The prefixes designate the *pulse-sequence channel* to which the statement applies. When used alone, they designate the pulse-sequence channel itself. A pulse-sequence channel is a logical structure in the pulse program that directs pulse-sequence statements to a particular transmitter, whose identity is decided at run-time.

A *transmitter* (in contrast to a channel) refers to the synthesizer, gating, and amplifier hardware that are used to

generate pulses at a particular probe port. A VNMR5 for spectroscopy can have up to five transmitters, labeled 1 to 5 from right to left in the RF card cage.

A *receiver* is RF hardware for signal detection including a digital receiver card in the acquisition computer. All NMR Systems for spectroscopy have at least one receiver, which can be associated with any transmitter. Some multi-receiver systems for spectroscopy have two or more receivers, up to a possible total of four.

Imaging systems may have an arbitrary number of transmitters and receivers. Systems with 16 and 32 transmitters and receivers are used. The software used to control imaging transmitters and receivers is described in the *Guide to Imaging*.

For VNMR5, all pulse-sequence channels are equivalent and they can be assigned to any transmitter. Most NMR Systems for spectroscopy are configured with a high-band transmitter 1 and a low-band transmitter 2. The uses of transmitters 3 to 5 depend upon the specific hardware configuration. By default, the `obs` and `dec` channels are designated as 1 and 2 if the spectrometer frequency is greater than 85% of the proton frequency. Otherwise they are designated 2 and 1. The channels `dec2` to `dec4` are assigned to transmitters 3 to 5 in order. The parameters `probeConnect` or `rfchannel` can be used to obtain any other assignment.

For single-receiver mode, the `obs` channel is designated as the channel for acquisition (it gets the receiver). Additional receivers are hardwired to specific transmitters. For a standard two-receiver system the second receiver is found on channel 3. Consult the documentation for the configuration of custom multi-receiver systems. Additional receivers are assigned to the same pulse-sequence channel as their respective transmitter\*, based on either the defaults or the use of `probeConnect` or `rfchannel`. \*Earliest versions of VnmrJ required that the user assign channel 3 with the second receiver to `dec`. This restriction is lifted in recent software versions.

The words `obs`, `dec`, `dec2`, `dec3` and `dec4`, despite their meaningful names, no longer strictly denote "observation" or "decoupling". Any channel can be assigned to any transmitter for either *pulsing* or *decoupling*. The `obs` channel is unique in that it is always used with a receiver, irregardless of the transmitter to which it is assigned. However the channels `dec`, `dec2`, `dec3` and `dec4` can also be used for observation if their assigned transmitter is hardwired to one of the

additional receivers.

The configuration file must designate a number of channels greater than or equal to that referred to in the sequence. A 3-channel sequence cannot be used on a 2-channel spectrometer. The configuration file identifies a system as *single-receiver* or *multiple-receiver for spectroscopy*. If multiple receivers are configured, the string parameter `rcvrs` designates the receivers to be used. Gradient statements are assigned to the appropriate gradient controller, either PFG or imaging, which is designated in the VnmrJ configuration file.

## Customizing the PSG software

Most of the `.c` and `.h` source-code files in the directory `/vnmr/psg` are precompiled as C-code object files with a `.o` extension and assembled in an object library `libpsglib.so`. A copy of this library `/vnmr/lib/libpsglib.a` is link-loaded with the pulse-sequence object file `pulsesequance.o` to create the fully compiled sequence whenever `seqgen` is used to compile a pulse sequence. A user must recompile the object library to make a change in one of the PSG source files. The recompiled object library is referred to as a *user PSG*. It is stored in the directory `~/vnmrsys/psg` of an individual Linux user and it only affects sequences for that user.

You can replace `/vnmr/psg/libpsglib.so` using the Linux copy command, but this action is usually not recommended. This action would affect every sequence that any user might subsequently compile (possibly to ill effect) and the new object library might be lost with a software upgrade.

To create a user PSG, run the Linux script `/vnmr/bin/setuserpsg` in a terminal window. This script creates the directory `~/vnmrsys/psg` for a user if it does not already exist and it initializes this user `psg` directory with all the object files from the system `psg` directory and an object library `libpsglib.a`.

To make a change, copy the source files that are to be changed to `~/vnmrsys/psg` and make the desired changes with a text editor.

The Linux script `/vnmr/bin/psggen` compiles the user source files, links them with the other object files in `~/vnmrsys/psg`, and creates a new file object library `libpsglib.a`. When `seqgen` is executed in a user, it first looks for `~/vnmrsys/psg/libpsglib.a`. The system library is

used if a user library is not found. Applications directories, designated from Edit Applications of the Edit pull-down menu, are not used by `seqgen`. Include files in applications directories are used by `seqgen`.

The Linux script `/vnmr/bin/fixpsg` recompiles the files in `/vnmr/psg` and places the resulting library in `/vnmr/lib`. One must have write permission to `/vnmr` in order to use this script. The `fixpsg` script is most often used during the installation of a patch for VnmrJ when the patch includes PSG files.

## Pulse-Sequence Statements

This section contains information on pulse-sequence statements in VnmrJ.

### Creating a time delay

The statements to provide time delays are `delay`, `hsdelay`, and `vdelay`. Table 5 summarizes these statements.

**Table 5** Statements for Creating Time Delays

<code>delay(time)</code>	Delay a specified time.
<code>hsdelay(time)</code>	Delay a specified time with a homospoil pulse.
<code>vdelay(timebase, count)</code>	Set a real-time variable delay.

Use `delay(time)` to set a specified time delay, where `time` is a *double*, for example `delay(d1)`. The value of `time` is rounded to the resolution depending on the transmitter version. Transmitters for the DD2 MR system, have a 25 ns minimum step and a 12.5 ns resolution. Transmitters for older systems have a 50 ns step and a 12.5 ns resolution. Consult the configuration file to determine which transmitter is present.

Use `hsdelay(time)` to create a delay with a homospoil pulse. To add a homospoil pulse to the delay, set the homospoil parameter `hs` to 'y'. A z1-axis homospoil pulse of length `hst` seconds is inserted at the beginning of the delay.

Use `vdelay(timebase, count)` to set a delay to the product of a fixed `timebase` and a real-time `count`, for which `timebase` is NSEC (nanoseconds), USEC (microseconds), MSEC (milliseconds), or SEC (seconds) and `count` is one of the real-time variables (`v1` to `v42`). For NSEC, the minimum delay is a `count` of either 2 or 4 (25 ns or 50 ns) depending on the transmitter, a `count` of less than the minimum corresponds to a delay of 0.0, and a `count` of  $n$  corresponds to a delay of  $12.5*n$  ns.

The *double* argument `time` of `delay` and `hsdelay` is fixed at run-time and so the `delay` statement cannot change its value scan-to-scan or in real-time loops and conditionals. The statement `vdelay` takes a real-time integer `count` as an argument and so its delay can change scan-to-scan or in real-time loops and conditionals.



## Assigning transmitters and receivers to channels

The labels `obs`, `dec`, `dec2`, `dec3`, and `dec4` refer to a possible five pulse-sequence, spectroscopic channels. These channels are assigned to transmitters at run-time by default or optionally by using the parameters `probeConnect` or `rfchannel`. The global variables `tn`, `dn`, `dn2`, `dn3`, and `dn4` set the nuclei associated with each pulse-sequence channel. The *global* variables `sfrq`, `dfrq`, `dfrq2`, `dfrq3`, and `dfrq4` store the digital-synthesizer frequencies for each channel. These frequencies are both transmitter frequencies of pulses and the center frequencies for channels with receivers (when `roff=0.0`). The *global* PSG variables `OBSch`, `DECch`, `DEC2ch`, `DEC3ch`, and `DEC4ch` store the transmitter assignment for each channel, where 1, 2, 3, 4, and 5 refer to transmitters from right to left in the RF card cage.

By default, the high-band transmitter, labeled '1', is assigned to `obs` if `sfrq` is greater than 85% of the proton frequency and in this case the low-band transmitter '2' is assigned to `dec`. If `sfrq` is less than 85% of the proton frequency, the assignment is reversed. The `dec2`, `dec3`, and `dec4` channels are assigned to transmitters '3', '4', and '5' respectively.

The *global* parameter `probeConnect` provides an automatic procedure to assign transmitters, based on the user configuration of transmitters to probe ports. The `probeConnect` string holds the designations of nuclei, separated by spaces, from left to right, that are associated with each transmitter from '1' to '5'. The user assigns this list based upon the probe, tuning configuration. The value of each nucleus parameter `tn`, `dn`, `dn2`, `dn3`, to `dn4` is compared with the entry in `probeConnect` to assign a transmitter to a pulse-sequence channel.

The current parameter `rfchannel` assigns transmitters from the parameter table. The parameter `rfchannel` is a string whose character-places, 0 to 4, represent the pulse-sequence channels, `obs` to `dec4`. The characters are '1' to '5', designating the VNMR5 transmitters. The parameter `rfchannel` should contain at least one character for each transmitter, up to the number of channels `numrfch`. The parameter `rfchannel` overrides `probeconnect` and to avoid confusion, `probeConnect` should not be created if `rfchannel` is used.

The transmitter assignments can be viewed on the Channels page of the Acquisition tab in the upper right corner of each

channel display.

Any of the first four transmitters can be designated obs and used with the first receiver. For two receivers, the second receiver is hardwired to transmitter '3'. The assignment of this receiver to a pulse-sequence channel follows that of transmitter '3'. For greater than two receivers you must consult the information about the hardware configuration to determine the association of transmitter-receiver pairs. The assignments of receivers to pulse-sequence channels for receivers greater than the first follow `rfchannel` or `probeConnect`.

The current `rcvrs` string parameter determines whether receivers are on or off. The placeholders of `rcvrs` refer to the first, second, third receivers, *etc*, up to the number of receivers. The character 'y' means acquisition takes place and 'n' means the receiver is not used. If `rcvrs` exists, at least one receiver must be selected. If `rcvrs` does not exist or is not set, acquisition takes place on obs only.

## Transmitter pulses

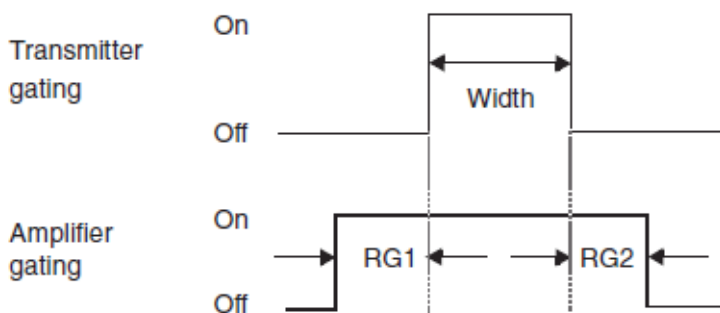
Statements to provide an RF pulse on a transmitter are described in [Table 6](#).

**Table 6** Statements to create transmitter pulses

<code>decpulse (width, phase)</code>	Pulse the dec channel with RG1=0 and RG2=0.
<code>decrgpulse (width, phase, RG1, RG2)</code>	Pulse the dec channel with RG1 and RG2 delays.
<code>dec2rgpulse (width, phase, RG1, RG2)</code>	Pulse the dec2 channel with RG1 and RG2 delays.
<code>dec3rgpulse (width, phase, RG1, RG2)</code>	Pulse the dec3 channel with RG1 and RG2 delays.
<code>dec4rgpulse (width, phase, RG1, RG2)</code>	Pulse the dec4 channel with RG1 and RG2 delays.
<code>obspulse ()</code>	Pulse the obs channel with RG1=rof1 and RG2=rof2, width=pw and phase=oph.
<code>pulse (width, phase)</code>	Pulse the obs channel with RG1=rof1 and RG2=rof2
<code>rgpulse (width, phase, RG1, RG2)</code>	Pulse the obs channel with RG1 and RG2 delays.

The statement `rgpulse(width,phase,RG1,RG2)` (See Figure 1) supplies a pulse to the obs transmitter, in which `width` is the pulse width in seconds, `phase` is a table or a real-time integer designating a quadrature phase, `RG1` is a predelay, and `RG2` is a postdelay in seconds. The phase is set and the amplifier is unblanked at the beginning of `RG1`. The transmitter is gated on at the beginning of `width` and gated off at the beginning of `RG2`. The transmitter is blanked at the end of `RG1` unless an explicit `rcvroff` statement has been executed sometime before the `rgpulse` statement. If `rcvroff` has been set, the transmitter remains unblanked after `RG1`.

The `rcvroff` statement is a compound statement that blanks the transmitter, sets the blanking mode of `rgpulse`, and turns off the receiver.



**Figure 1** Transmitter Gating for an Observe Pulse

The statement `pulse(width,phase)` is similar to `rgpulse` except that `RG1=rof1` and `RG2=rof2`.

The statement `decpulse(width,phase)` is similar to `decrpulse` except that `RG1=0.0` and `RG2=0.0`.

The statement `obspulse()` is similar to `pulse(width,phase)` except that `width=pw` and `phase=oph`.

The global variables `rof1`, `rof2`, and `pw` are *global* PSG variables for the standard amplifier unblanking delay, the standard receiver blanking delay and the standard pulse width, respectively, and they are automatically obtained from parameters of the same name. The phase variable `oph` is also the default receiver phase and it is set to 0, 1, 2, 3 unless `cp`, (*constant phase*) is 'Y'. In this latter case, `oph=0`. It is usual practice to set `oph` with an explicit phase table, using the `setreceiver` command.

The RF and phase behavior of the statements `decrpulse`, `dec2rgpulse`, `dec3rgpulse`, and `dec4rgpulse` are similar to

rgpulse for their respective channels dec, dec2, dec3, and dec4.

It is a good practice to use only the statements `rgpulse`, `decrgpulse`, `dec2rgpulse`, `dec3rgpulse` and `dec4rgpulse` and explicitly set the values of `width`, `phase`, `RG1`, and `RG2`. This practice eliminates ambiguity from the pulse-sequence code. The statements `pulse`, `decpulse`, and `obspulse` should be considered obsolete.

### Controlling blanking for observe pulses

Control of transmitter blanking is important for obtaining the best quality data. When *unblanked*, amplifiers are on and can amplify a pulse, when *blanked* they are off and produce no output. An unblanked amplifier can have noise output that can potentially reduce signal-to-noise during acquisition. Also an unblanked amplifier runs hotter than a blanked amplifier. The transition between extensive periods of blanking and other periods of unblanking can cause thermal amplitude instability.

For experiments for which pulses are sparse in time (most liquids experiments), it is best to blank all transmitters at every moment they are not producing RF. The standard `rof1=2.0` is adequate time to unblank before a pulse.

For experiments that use extensive spinlocking and decoupling (most solids experiments), it is best to unblank the amplifiers at all times other than the acquisition.

For most liquids sequences, it is desirable that the transmitter blank after `RG1` and `rcvroff` is rarely used. Usually `RG1` is set as `rof1` or `2.0` and `RG2` as `0.0`. One might set `RG2` as `rof2` if the pulse precedes an acquisition. A slightly better practice is to set `RG2=0.0` and explicitly add `obsblank(); delay(rof2);`. With this improvement, the transmitter blanks at a slightly earlier time before acquisition and `rof2` can be shorter.

To provide *back-to-back* pulses, with no intervening delay, set `RG1` and `RG2` as `0.0` and place the pulses adjacent to each other in the code.

Many solids sequences employ `rcvroff` to keep the observe transmitter unblanked until acquisition. In this case, one must place an `obsblank` statement after the last `rgpulse`, just before `delay(rof2)`.

### Controlling blanking for non-observe pulses

In default mode, only *observe* channels or channels in the same band as an observe channel can be blanked. Non-observe channels remain in continuous mode. To blank non-observe channels, you create the current string parameter `ampmode`, in which the character-places refer to transmitters, '1' to '5'. The character 'd' indicates *default*, the character 'p' indicates pulsed or “blinking allowed”, and the character 'c' indicates *continuous* or “blinking not allowed”.

If blanking is allowed, the statements `decrgpulse`, `dec2rgpulse`, `dec3rgpulse`, and `dec4rgpulse` behave as `rgpulse` on their respective channels `dec`, `dec2`, `dec3`, and `dec4`, with the exception that amplifier blanking always occurs (if allowed by `ampmode`) at the end of `RG1` (`rcvloff` has no effect). If blanking is not allowed, then these channels remain unblanked between pulses.

must be sufficiently long to allow the amplifier to stabilize after blanking is removed: 5 s is typically right.

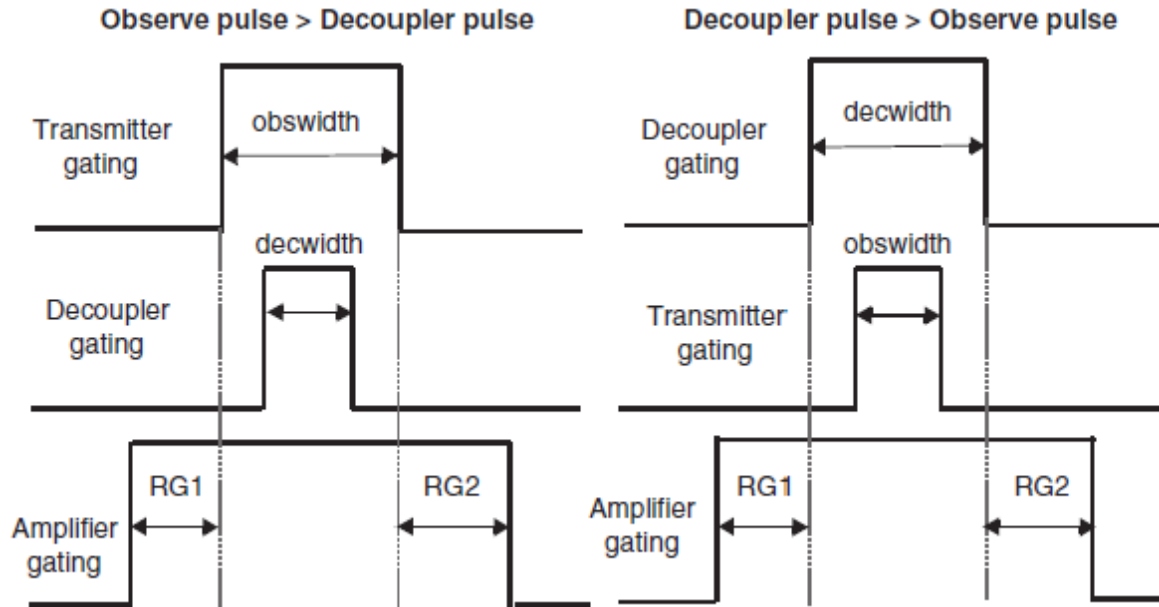
### Pulsing channels simultaneously

Statements for controlling simultaneous, non-shaped pulses are `simpulse`, `sim3pulse`, and `sim4pulse`. [Table 7](#) summarizes these statements. Simultaneous-pulse statements using shaped pulses are covered in a later section.

**Table 7** Statements for simultaneous pulses

<code>simpulse*</code>	Pulse the <code>obs</code> and <code>dec</code> channels simultaneously.
<code>sim3pulse*</code>	Pulse the <code>obs</code> , <code>dec</code> and <code>dec2</code> channels simultaneously.
<code>sim4pulse*</code>	Pulse the <code>obs</code> , <code>dec</code> , <code>dec2</code> and <code>dec3</code> channels simultaneously.
<code>*simpulse(obswidth, decwidth, obsphase, decphase, RG1, RG2)</code>	
<code>*sim4pulse(obswidth, decwidth, dec2width, obsphase, decphase, dec2phase, RG1, RG2)</code>	
<code>*sim4pulse(obswidth, decwidth, dec2width, dec3width, obsphase, decphase, dec2phase, dec3phase, RG1, RG2)</code>	

Use `simpulse(obswidth, decwidth, obsphase, decphase, RG1, RG2)` to simultaneously pulse the `obs` and `dec` channels with amplifier gating, for example, `simpulse(pw, pp, v1, v2, 0.0, rof2)`.



**Figure 2** Transmitter gating for simultaneous pulses

The shorter of the two pulses is centered on the longer pulse. The RG1 and RG2 delays surround the longer pulse. Phase shifting, blanking, and unblanking for both channels are similar to `rgpulse` and `decpulse` and occur simultaneously on both channels. If they are not identical, the absolute difference in the two widths must be greater than or equal to two times the minimum resolution of the transmitter. Otherwise, a timed event of less than the minimum value (25 ns or 50 ns) would be produced.

The statement `sim3pulse(obswidth, decwidth, dec2width, obsphase, decphase, dec2phase, RG1, RG2)` performs a simultaneous pulse on the `obs`, `dec`, and `dec2` transmitters, in which `obswidth`, `decwidth`, and `dec2width` are the pulse durations for the respective transmitters and `obsphase`, `decphase`, and `dec2phase` are the phases tables for the corresponding pulses, for example, `sim3pulse(pw, p1, p2, oph, v10, v1, rof1, rof2)`.

Two simultaneous pulses on `obs` and `dec2` can be achieved by setting `decwidth` to 0.0 and setting a dummy `v`-variable for `decphase`, for example, `sim3pulse(pw, 0.0, p2, oph, v10, v1, rof1, rof2)`. In this case, no events take place on the `dec` channel and functions such as `wave` from decoupling are not interrupted.

Use `sim4pulse(obswidth, decwidth, dec2width, dec3width,`

`obsphase`, `decphase`, `dec2phase`, `dec3phase`, `RG1`, `RG2`) to perform simultaneous pulses on as many as four different RF channels. If any pulse width is set to 0.0, no pulse is executed on that channel.

## Setting quadrature phase shifts

The statements `txphase`, `decphase`, `dec2phase`, `dec3phase`, `dec4phase` control transmitter phase in multiples of  $90^\circ$ , also known as *quadrature phase*. Table 8 summarizes these statements.

**Table 8** Statements for quadrature phase control

<code>decphase (phase)</code>	Set the phase of the <code>dec</code> channel in steps of $90^\circ$ .
<code>dec2phase (phase)</code>	Set the phase of the <code>dec2</code> channel in steps of $90^\circ$ .
<code>dec3phase (phase)</code>	Set the phase of the <code>dec3</code> channel in steps of $90^\circ$ .
<code>dec4phase (phase)</code>	Set the phase of the <code>dec4</code> channel in steps of $90^\circ$ .
<code>txphase (phase)</code>	Set the phase of the <code>obs</code> channel in steps of $90^\circ$ .

To set the `obs` transmitter phase, use `txphase(phase)` for which `phase` is a real-time variable (`v1` to `v42`, *etc*), a real-time constant (`zero`, `one`, *etc*), or a phase table `t1` to `t60`.

Use `txphase` to set the phase of the `obs` transmitter separately from a pulse. It is a good practice to set the phase at the beginning of the preceding delay, unless a value of `RG1` is set greater than zero for the `rgpulse`. The transmitter takes a finite time to settle (about 50 ns) and it is best if this change were to occur with the transmitter off. Use the same phase for `txphase` as that for `rgpulse`.

Use `txphase` to create generic back-to-back pulses, avoiding the use of `rgpulse`. The code: `txphase(t1); obsunblank(); delay(rof1); xmtron(); delay(pw); txphase(t2); delay(pw); xmtroff(); obsblank();` creates a composite pulse made of two widths `pw` using two phase tables `t1` and `t2`. It is necessary to unblank the transmitter explicitly in this case and `rof1` should be at least  $2.0 \mu\text{s}$ . Also it is a good practice to deliver the `txphase` command before the `rof1` period.

The statements `decphase`, `dec2phase`, `dec3phase` and `dec4phase` are similar to `txphase` for their respective channels.

## Setting small-angle phase shifts

The DD2 MR system has a phase resolution of 16 bits or  $360.0/65536$ , about 0.0055 degrees per step. The statements in [Table 9](#) are used to set the phase with full resolution. Current hardware sets the phase to the full resolution of the software.

**Table 9** Statements for small-angle phase control

<code>dcplrphase(multiplier)</code>	Set the small-angle phase of the dec channel.
<code>dcplr2phase(multiplier)</code>	Set small-angle phase of the dec2 channel.
<code>dcplr3phase(multiplier)</code>	Set small-angle phase of the dec3 channel.
<code>dcplr4phase(multiplier)</code>	Set small-angle phase of the dec4 channel.
<code>decstepsize(base)</code>	Set the step size of the dec channel.
<code>dec2stepsize(base)</code>	Set the step size of the dec2 channel.
<code>dec3stepsize(base)</code>	Set the step size of the dec3 channel.
<code>dec4stepsize(base)</code>	Set the step size of the dec4 channel.
<code>obsstepsize(base)</code>	Set the step size of the obs channel.
<code>xmtrphase(multiplier)</code>	Set the small-angle phase of the obs channel.

The `xmtrphase(multiplier)` statement is used to set the full small-angle phase where phase in degrees is  $\text{base} \times \text{multiplier}$  and `multiplier` is a real-time variable (`v1` to `v42`, etc), a real-time constant (`zero`, `one`, etc) or a phase table `t1` to `t60`. The variable `base` is a double whose value is the phase step in degrees.

The `obsstepsize(base)` statement uses the double parameter `base` (in degrees) to provide the stepsize of the small-angle phase shift for the obs transmitter. The `base` can be changed any time during the sequence.

The phase itself is never set directly in degrees, it is always a multiple of `base` and a real-time integer or table. If a non-quadrature phase cycle is required, for example 0,60,120,180,240,300 degrees, then `base = 60.0`, and the `multiplier` is a table containing the values 0,1,2,3,4,5. Note that the phase must be set with the `xmtrphase` statement because `rgpulse` does not accept a non-quadrature phase. It is good practice to set `xmtrphase` at the beginning of the preceding delay and then use a phase constant of zero in `rgpulse`.

The `xmtrphase` statement has precedence over the `txphase`



statement. If a `txphase` statement follows an `xmtrphase` statement, it will change only that portion of the phase that is a multiple of  $90^\circ$ . For example, if the phase is set to  $240^\circ$  with an `xmtrphase` statement, `txphase(zero)` will leave the phase at  $60^\circ$  and `txphase(two)` will leave the phase unchanged. If the `txphase` statements were delivered before the `xmtrphase` statement, the final phase would be  $240^\circ$  in both cases. Use `xmtrphase(zero)` to remove any remaining small-angle phase.

The VNMR5 can contain one of two different phase shifters, a 13-bit phase shifter with a resolution of  $360.0/8192 = \sim 0.043^\circ$  or a 16-bit phase shifter with a resolution of  $360.0/65536 = \sim 0.0055^\circ$ . The value of `base*multiplier` is set by the hardware to the nearest allowed phase step. Consult the configuration file to determine which device is present.

To set phases with full resolution, use `obsstepsize(360.0/65536)` (or `360.0/8192` for the 13-bit device). Phases can be set in apparent “degrees” by using  $1.0^\circ$  resolution with `obstepsize(1.0)`.

The statements `dcplrphase`, `decstepsize`, `dcplr2phase`, `dec2stepsize`, `dcplr3phase`, `dec3stepsize`, `dcplr4phase`, and `dec4phase` are similar to `xmtrphase` and `obstepsize`, for their respective channels.

## Controlling the frequency offset

Statements for frequency control are `obsoffset`, `decoffset`, `dec2offset`, `dec3offset` and `dec4offset`. [Table 10](#) summarizes these statements.

**Table 10** Statements to control frequency-offset

<code>decoffset (offset)</code>	Set the offset frequency of the <code>dec</code> channel.
<code>dec2offset (offset)</code>	Set the offset frequency of the <code>dec2</code> channel.
<code>dec3offset (offset)</code>	Set the offset frequency of the <code>dec3</code> channel.
<code>dec4offset (offset)</code>	Set the offset frequency of the <code>dec4</code> channel.
<code>obsoffset (offset)</code>	Set the offset frequency of the <code>obs</code> channel.

Each transmitter of a VNMR5 has a digital frequency synthesizer that sets both the frequency of the transmitter and the center frequency of the receiver (when `roff=0`). Each synthesizer is set automatically using the *global* PSG variables `sfrq`, `dfrq`, `dfrq2`, `dfrq3` and `dfrq4`, before the

first scan. These frequencies, in MHz, are set from a base frequency for the nucleus ( $t_n$ ,  $d_n$ ,  $d_{n2}$ ,  $d_{n3}$ , and  $d_{n4}$ ) plus a default offset in Hz, determined by the global PSG variables  $t_{of}$ ,  $d_{of}$ ,  $d_{of2}$ ,  $d_{of3}$ , and  $d_{of4}$ . If offset parameters are not present the default is 0.0.

The statement `obsoffset(offset)` is used to reset the offset during the sequence. The argument `offset` is *double* value in Hertz that sets the frequency with 0.1 Hz resolution. The statements that change the synthesizer affect both the transmitter and the receiver and they are not automatically reset during the sequence. One should explicitly initialize the offset to the default  $t_{of}$  at the beginning of the sequence if the offset is to be changed within the sequence. Failure to do so will cause scans greater than 1 to have the wrong offset.

The `obsoffset` statement adds about a 1.0 s delay to allow time to send the new frequency word to the synthesizer. If required, you should account for this delay in pulse-sequence timing.

It is a good practice to limit the use of the `obsoffset` statement in pulse sequences. Because a change in the synthesizer affects the receiver, an incorrect change of the offset can cause the receiver to lose phase coherence scan-to-scan. The frequency change caused by `obsoffset` is not guaranteed to be phase continuous. It is a good practice to change the offset only during delays where the spins are along a Z axis.

Current systems can create a specific transmitter offset through the calculation of phase-ramped shaped pulse or waveform. This latter procedure is recommended over the use of `obsoffset`.

The `decoffset`, `dec2offset`, `dec3offset` and `dec4offset` statements are similar to `obsoffset` for their respective channels.

### Controlling the transmitter power

The system controls the transmitter power through two devices:

- A step attenuator set in dB units of power, also called the “coarse attenuator”.
- A linear modulator set in units proportional to RF voltage, also called the “fine power” or “amplitude”.

Separate statements control each device. Table 11 summarizes these statements.

**Table 11** Statements for amplitude and power control

<code>decpower (value)</code>	Set the coarse attenuator for the <code>dec</code> channel.
<code>dec2power (value)</code>	Set the coarse attenuator for the <code>dec2</code> channel.
<code>dec3power (value)</code>	Set the coarse attenuator for the <code>dec3</code> channel.
<code>dec4power (value)</code>	Set the coarse attenuator for the <code>dec4</code> channel.
<code>decpwrf (value)</code>	Set the amplitude for the <code>dec</code> channel.
<code>dec2pwrf (value)</code>	Set the amplitude for the <code>dec2</code> channel.
<code>dec3pwrf (value)</code>	Set the amplitude for the <code>dec3</code> channel.
<code>dec4pwrf (value)</code>	Set the amplitude for the <code>dec4</code> channel.
<code>obspower (value)</code>	Set the coarse attenuator for the <code>obs</code> channel.
<code>obspwrf (value)</code>	Set the amplitude for the <code>obs</code> channel.

### Coarse attenuator control

The statement `obspower (value)` takes a *double* as an argument and can set `value` with resolution of up to 0.5 dB and a maximum of 63.0. The setting of 63.0 corresponds to maximum power of the associated amplifier. A change of -6.0 dB attenuates the power by x4 and the RF amplitude by x2.

The DD2 MR system contains a 100 dB attenuator with a minimum setting of -37.0 dB and steps of 0.5 dB. Older systems contain a 79.0 dB attenuator with a minimum of -16.0 dB and allowed steps of 1.0 dB. Consult the configuration file to determine which device is present.

If the 100 dB attenuator is present, `value` settings are rounded to the nearest 0.5 dB and steps below -37.0 are set to -37.0. If the 79 dB attenuator is present, `value` settings are rounded to the nearest whole number and settings below -16.0 are set to -16.0. A change of 0.5 dB can have a significant effect on the pulse width (about 6%). Be cautious when using calibrations obtained with 0.5 dB resolution on a system with 1.0 dB resolution.

Each transmitter is set with default values, using the *global* PSG variables `tpwr`, `dpwr`, `dpwr2`, `dpwr3`, and `dpwr4`, before the first scan. If parameters of the same name do not exist in the data set, the coarse power is initialized to 0.0. Individual statements that set the coarse power can be used to override the default values during the sequence.

The power is never reset to the default values automatically. It is necessary to explicitly initialize the power `tpwr` at the beginning of the sequence if it is changed within the sequence. Failure to do so will cause scans greater than 1 to have the wrong power. Despite the default, it is good practice to initialize the power explicitly with the statement `obspower(tpwr)`.

The `obspower` statement itself adds 50 ns to the pulse sequence. The setting time of the coarse attenuator is about 3.0  $\mu$ s after the statement and an attenuator change also causes a transient during this period. It is a good practice to allow a 3.0  $\mu$ s delay for settling after the `obspower` statement and to blank the transmitter during this time. You should never set `obspower` while the transmitter is on.

The `decpower`, `dec2power`, `dec3power` and `dec4power` statements are similar to `obspower` for their respective channels.

### CAUTION

Use caution when setting values of `power`, `obspower`, `decpower`, `dec2power`, and `dec3power` greater than 49 (about 2 watts). Performing continuous decoupling or long pulses at power levels greater than this can result in damage to the probe. Set a maximum safety limit for the `tpwr`, `dpwr`, `dpwr2`, and `dpwr3` parameters in the Utilities > System Settings window or use *global* variable `maxattnech1`, `maxattnech2`, to have PSG (the go command) check for power values in excess of defined limits.

### Amplitude control

The statement `obspwrf(value)` takes a *double* as an argument and sets the linear modulator in 16-bit linear steps of linear RF amplitude with a maximum of 4095.0 and a minimum of 0.0. The setting of 4095 corresponds to the power set by `tpwr`. A division of the setting by `x2` corresponds to attenuation of the RF amplitude by `x2` and the power by `x4`. The linear modulator hardware has a maximum attenuation of `x1000` for settings around 0.0. Settings around 0.0 cease to be linear.

A VNMRS may contain one of two linear modulators:

- The DD2 MR system contains a 16-bit device with a stepsize of  $4095/65536 = \sim 0.062$ .
- Older systems contain a 12-bit device with a stepsize of  $4095/4096 = \sim 1.0$ .

The setting of `value` is rounded to the appropriate resolution by the hardware. For the 12-bit attenuator, rounding always corresponds to a truncation to a whole number between 0 and 4095. The 16-bit attenuator accepts decimal values. Consult the configuration file to determine which device is present.

The *global* PSG variable `tpwrf`, `dpwrf`, `dpwrf2`, `dpwrf3`, and `dpwrf4` set the default amplitude for each channel before the first scan. If parameters of the same name do not exist in the data set, the amplitude is initialized to 4095.0. Individual statements that set the amplitude can be used to override the default values during the sequence.

The amplitude is never reset automatically during the sequence. It is necessary to explicitly initialize the amplitude at the beginning of the sequence if it is changed within the sequence. Failure to do so will cause scans greater than 1 to have the wrong amplitude. Despite the default, it is good practice to initialize the amplitude explicitly with the statement `obspwrf(tpwr)`.

The amplitude statements add no time to the sequence and amplitude changes have a settling time of 50 ns.

The `decpwrf`, `dec2pwrf`, `dec3pwrf` and `dec4pwrf` statements are similar to `obspwrf` for their respective channels.

## Explicit transmitter gating

The transmitter gating statements gate the RF on and off outside of pulses. [Table 12](#) summarizes these statements.

**Table 12** Statements for transmitter gating

<code>decoff()</code>	Gate the <code>dec</code> channel off.
<code>dec2off()</code>	Gate the <code>dec2</code> channel off.
<code>dec3off()</code>	Gate the <code>dec3</code> channel off.
<code>dec4off()</code>	Gate the <code>dec4</code> channel off.
<code>decon()</code>	Gate the <code>dec</code> channel on.
<code>dec2on()</code>	Gate the <code>dec2</code> channel on.
<code>dec3on()</code>	Gate the <code>dec3</code> channel on.
<code>dec4on()</code>	Gate the <code>dec4</code> channel on.
<code>xmtroff()</code>	Gate the <code>obs</code> channel off.
<code>xmtron()</code>	Gate the <code>obs</code> channel on.

The `xmtron` statement turns on the `obs` channel RF and the `xmtroff` statement turns it off. The `obs` transmitter must also be unblanked to obtain a pulse.

RF gating adds no time to the sequence. The RF gate has a 50 ns settling time.

The `xmtron` statement is used to deliver a generic pulse or decoupling period without the use of `rgpulse` or `status`.

The `decon`, `decoff`, `dec2on`, `dec2off`, `dec3on`, `dec3off`, `dec4on`, and `dec4off` statements are similar to `xmtron` and `xmtroff` for their respective channels, `dec`, `dec2`, `dec3`, and `dec4`.

### Transmitter blanking and unblanking

The statements for transmitter blanking and unblanking are shown in [Table 13](#).

**Table 13** Statements for amplifier blanking and unblanking

<code>decblank()</code>	Blank the amplifier associated with the <code>dec</code> channel.
<code>dec2blank()</code>	Blank the amplifier associated with the <code>dec2</code> channel.
<code>dec3blank()</code>	Blank the amplifier associated with the <code>dec3</code> channel.
<code>dec4blank()</code>	Blank the amplifier associated with the <code>dec4</code> channel.
<code>decunblank()</code>	Unblank the amplifier associated with the <code>dec</code> channel.
<code>dec2unblank()</code>	Unblank the amplifier associated with the <code>dec2</code> channel.
<code>dec3unblank()</code>	Unblank the amplifier associated with the <code>dec3</code> channel.
<code>dec4unblank()</code>	Unblank the amplifier associated with the <code>dec4</code> channel.
<code>obsblank()</code>	Blank the amplifier associated with the <code>obs</code> channel.
<code>obsunblank()</code>	Unblank the amplifier associated with the <code>obs</code> channel.

The `obsunblank` statement *gates on* the amplifier of the `obs` channel independently from `rgpulse`. The `obsblank` statement *gates off* the amplifier of the `obs` transmitter. See the discussion of amplifier blanking under the section Transmitter Pulses.

The blanking and unblanking statements add no time to the sequence but have a transition time of 2  $\mu$ s. The *global* PSG variable `rof1` and its associated parameter of the same name are often used to set a delay between an `obsunblank` statement and an `xmtron` statement. The value of `rof1` is often used to set `RG1` of an `rgpulse`.

The `decunblank`, `decblank`, `dec2unblank`, `dec2blank`, `dec3unblank`, `dec3blank`, `dec4unblank`, and `dec4blank` statements are similar to `obsunblank` and `obsblank` for their respective channels, `dec`, `dec2`, `dec3`, and `dec4`. See the discussion of `ampmode` in the section Transmitter Pulses.

## The status statements

The statements `status` and `statusdelay` set the state and modulation of the `obs`, `dec`, `dec2`, `dec3`, and `dec4` channels based on values in the parameter table. These statements execute statements for transmitter gating and/or set a waveform for modulation.

**Table 14** Status statements

<code>setstatus*</code>	Set the gate and modulation with arguments in the sequence.
<code>status (period)</code>	Set the gate and modulation from parameters.
<code>statusdelay (period, time)</code>	Execute the status statement within a delay.

\*`setstatus (transmitter, on, mode, sync, mod_freq)`

The `status (period)` statement accepts a single character argument `A`, `B`, `C`, *etc* that begins a region of the pulse sequence, a *status period*. The next statement ends the first region and begins the next region.

- The *global* string PSG variables `xm`, `dm`, `dm2`, `dm3`, and `dm4` (`obs` to `dec4`) and their parameters of the same name are used to set a transmitter gate on and off. A 'y' or an 's' indicates that the transmitter is on and a 'n' indicates that the transmitter is off during the appropriate status period. The character placeholders refer to the arguments `A`, `B`, `C`, *etc* in order. The modulation is applied synchronously if the value is 's'. If the value is 'y', the modulation is applied asynchronously. For the asynchronous mode, the starting phase of the modulation is adjusted randomly scan-to-scan to suppress decoupling sidebands.
- For example, it is common usage to set `status (A)` at the beginning of the sequence and to set `status (C)` during acquisition. Some sequences may also have a separate period `status (B)`. The parameter assignment `dm= 'nny'` sets decoupling only during acquisition. `dm= 'yy'` or `'yyy'` sets uninterrupted decoupling on `dec`. Note that a character for `B` must be present if `status (C)` is used, even though `status (B)` is not used in the sequence.
- The *global* string PSG variables variables `xmm`, `dmm`, `dmm2`, `dmm3`, and `dmm4` (`obs` to `dec4`) and their parameters of the same name are used to set a modulation pattern during the status periods. Allowed characters, for example, are 'c' for continuous decoupling, 'w' for WALTZ decoupling and 'p' for programmed decoupling. See the *Command and Parameter Reference* for more details about the choices.



- The statement `statusdelay(period, delay)` encapsulates the status statement in a delay that is determined by the second argument. This statement is usually unnecessary for VNMRS because the operations of `status` take no time, though it might be used to preserve compatibility with Unity-series systems.

The statement

`setstatus(transmitter, on, mode, sync, mod_freq)` is used to set the state and modulation of any channel from within the pulse sequence. The argument `channel` is one of the *global* PSG variables `OBSch`, `DECch`, `DEC2ch`, `DEC3ch`, or `DEC4ch` that assigns a transmitter; `on` is `TRUE` to set the transmitter on or `FALSE` to set it off, `mode` sets the modulation pattern ('c', 'g', 'p', *etc*), and `mod_freq` is the modulation frequency. The modulation frequency is the inverse of the minimum step size, as determined by the `dres` parameter used with the pattern. The argument `sync` is `TRUE` or `FALSE`. A value of `FALSE` randomizes the initial timing of the pattern relative to each scan to reduce decoupler sidebands. A value of `TRUE` lets the modulation pattern proceed deterministically. A common example is `setstatus(DECch, TRUE, 'w', FALSE, dmf)`.

Status periods are also used to control the presence or absence of a homospoil pulse during a delay set with `hsdelay`. The string parameter `hs` determines the state of the homospoil pulse. A character of 'y' enables the pulse and 'n' disables it. For example, if a particular pulse sequence uses `status(A)`, `status(B)`, and `status(C)` during acquisition, `hs='ny'` will disable homospoil in period A and enable it in B. Only two letters are needed because there would never be an `hsdelay` statement during an acquisition period.

## Receiver gating

Explicit receiver gating in the pulse sequence is controlled by the `rcvloff`, `rcvron`, `recoff`, and `recon` statements. These statements are described in [Table 15](#).

**Table 15** Statements for receiver gating

<code>rcvloff()</code>	Set the receiver off, the T/R switch on and unblank the obs amplifiers.
<code>rcvron()</code>	Set the receiver on the T/R switch off and blank the obs amplifiers.
<code>recoff()</code>	Set the receiver gate off and the T/R switch on.
<code>recon()</code>	Set the receiver gate on and the T/T switch off.

## Interfacing to external user devices

Table 16 shows statements that are used to trigger and receive triggers from external devices.

**Table 16** Statements to interact with external devices

<code>rotorperiod(value)</code>	Determine the period of an external signal and set a real-time value.
<code>rotorsync(periods)</code>	Delay for an integer number of periods of an external signal.
<code>sp1off()</code>	Set TTL port 1 off.
<code>sp2off()</code>	Set TTL port 2 off.
<code>sp3off()</code>	Set TTL port 3 off.
<code>sp1on()</code>	Set TTL port 1 on.
<code>sp2on()</code>	Set TTL port 2 on.
<code>sp3on()</code>	Set TTL port 3 on.
<code>triggerselect(select)</code>	Select ports 1,2 or 3 on the MRI interface board with A, B or C.
<code>xgate(pulses)</code>	Halt all controllers and start after an integer number of pulses.

The console provides User TTL BNC Outputs in the transmitter 1 section of the Console Interface Panel on the back of the console. The equivalent outputs on other transmitters are redundant. The commands `sp1on`, `sp2on`, and `sp3on` enable a positive TTL output on the respective ports, top to bottom. The commands `sp1off`, `sp2off`, and `sp3off` enable a negative TTL output.

The VNMRS provides a BNC external trigger for pulse-sequence operation on the EXT TRIG port by the USER OUTPUTS group on the Console Interface Panel. In addition, three other external trigger ports are located on the MRI User Panel of imaging systems. The MRI ports are labeled INPUT 1, 2, or 3 in the EXT TRIG group.

The statement `xgate(pulses)` halts all RF transmitters and the receivers simultaneously and holds them off until a positive trigger is received on the EXT TRIG port of the Console interface panel. The argument `pulses` is a *double* that sets the number of positive trigger pulses before resuming the sequence.

The statement `triggerSelect` with arguments A, B, and C selects one of the three BNC inputs on the MRI user panel.

## Real-Time Control of Pulse Sequences

This section contains information on programming real-time control of pulse sequences.

### Real-time integer variables and constants

The variables and constants listed in [Table 17](#) are used for real-time calculations, to set real-time indexes, and as integer multipliers to set the phase.

Real-time *v-variables* are C-integer constants whose values index 32-bit integer values on the acquisition computer. The values on the acquisition computer can be changed in real-time, scan-to-scan with a set of *real-time math* statements.

The real-time values of the variables `v1` to `v42` are automatically initialized to 0. The constants such as `zero`, `one`, *etc* are initialized to 0,1, *etc* as suggested by their names. The constants `ssval` and `bsval` are initialized based on the values of the parameters `ss` and `bs`. The counters `ct`, `bsctr`, `ssctr`, `id2`, `id3`, and `id4` are set based on the scan or increment number. The constants and counters are set automatically and their values should only be changed with caution.

It is not possible to declare a new v-variable. However, it is possible to reassign the integer index of a v-variable to a new name. For example, the assignment `int myvariable=v1;` assigns the index value of `v1` to the new name `myvariable`. After this assignment, both `myvariable` and `v1` can be used to access the values of the `v1` v-variable. A compiler definition can also be used to provide an alias for a v-variable. For example, use the statement `#define myvariable v1`. This statement is placed outside the pulse-sequence function and does not end with a semicolon.

**Table 17** Real-time variables and constants

<code>v1</code> to <code>v42</code>	Real-time v-variables point to 32-bit integer values that are used for real-time calculations, indexes of loops, and phase multipliers.
<code>ct</code>	The completed-transient counter points to a 32-bit integer that is incremented after each transient, starting with a value of 0 prior to the first experiment. This pattern (0,1,2,3,4,etc) is the basis for many scan-to-scan calculations.

**Table 17** Real-time variables and constants (continued)

<code>bsval, bsctr</code>	A blocksize counter <code>bsctr</code> points to a 32-bit integer that is decremented from <code>bsval</code> to 1 during each block of transients. After completing the last transient in the block, <code>bsctr</code> is set back to a value of <code>bs</code> and the block of data is transferred from the acquisition computer to the workstation.
<code>oph</code>	A phase multiplier <code>oph</code> controls the phase of the receiver in 90 increments (0=0 , 1=90 , 2=180 , and 3=270 ). If <code>cp='y'</code> , <code>oph</code> is set to the successive modulo values 0,1,2,3. If <code>cp='n'</code> , <code>oph</code> is set to zero. The pattern of successive values of <code>oph</code> can be obtained from a table through use of the <code>setreceiver</code> statement.
<code>zero, one, two, three</code>	Real-time constants with values of 0,1,2,3 that are often used to select the constant phases of 0 , 90 , 180 , and 270
<code>ssval, ssctr, ssct</code>	A steady-state counter <code>ssctr</code> is decremented from <code>ssval</code> to 1 during the steady-state period before the first scan <code>ct=1</code> . The counter <code>ssct</code> is incremented from 1 to <code>ssval</code> during the steady state period.
<code>id2, id3, id4</code>	Real-time counters of the first, second, and third indirect dimensions of multidimensional experiments. The value for each index is 0 for the first increment up to the dimension minus 1 for the last increment.

### Calculating with real-time integer math

The statements for integer mathematics as listed in [Table 18](#) are used to manipulate real-time `v`-variables. These are: `add`, `dbl`, `decr`, `divn`, `hlv`, `incr`, `mod2`, `mod4`, `modn`, `mult` and `sub`.

**Table 18** Statements for real-time integer math

<code>add(vi, vj, vk)</code>	Add integer values, set $vk = vi + vj$ .
<code>dbl(vi, vj)</code>	Double integer values, set $vj = 2*vi$ .
<code>decr(vi)</code>	Decrement an integer value, set $vj = vj - 1$ .
<code>divn(vi, vj, vk)</code>	Perform integer division, set $vj = vi \div vj$ .
<code>hlv(vi, vj)</code>	Perform integer division by 2, set $vk = vi \div 2$ .
<code>incr(vi)</code>	Increment an integer value, set $vk = vi + 1$ .
<code>mod2(vi, vj)</code>	Find and integer value modulo 2, set $vk = vi \% 2$ .
<code>mod4(vi, vj)</code>	Find and integer value modulo 4, set $vk = vi \% 4$ .
<code>modn(vi, vj)</code>	Find and integer value modulo n, set $vk = vi \% n$ .
<code>mult(vi, vj, vk)</code>	Multiply integer values, set $vk$ equal to $vi * vj$ .
<code>sub(vi, vj, vk)</code>	Subtract Integer values, set $vk$ equal to $vi - vj$ .

The result of an integer division is truncated. For the statement `hlv(vi, vj)`, if `vi=3` then `vj=1`.

The `mod#` statements yield the remainder of a division. For

the statement `modn(vi,vj,vk)`, if  $v_i=5$  and  $v_j=2$  then  $v_k=1$ . Negative integers are not allowed in the `mod#` statements.

It is always a good practice to document the result of a real-time statement, because its results are hard to visualize. The following example illustrates the action of several integer math statements and how comments are typically used:

```
hlv(ct,v1);      /* v1=0011223344... */
dbl(v1,v1);     /* v1=0022446688... */
mod4(v1,v1);    /* v1=0022002200... */
```

The same variable can be used as the input and output of a particular statement; for example, `dbl(v1,v1)` reassigns `v1`.

The parameter value `ss` is associated with real-time variables `ssval`, `ssctr`, and `ssct`. The value of `ssval` is set to the absolute value of `ss`. The value of `ssctr` is initialized to `ssval` and decremented on each steady-state scan until it reaches 0. When `ssctr` is 0, all subsequent transients are collected as data. The value of `ssct` is incremented from 1 to `ssval` during the steady state period. The steady state period occurs once before the first increment. If `ss` is greater than 0, the steady-state period is performed once at the beginning of any experiment. If `ss` is less than 0, the steady-state period is repeated before each increment of an arrayed or multidimensional experiment.

The parameter value `bs` is associated with real-time variables `bsval` and `bsctr`. The value of `bsval` is set to the value of `bs`. The value of `bsctr` is initialized to `bsval` and decremented until it reaches 1. When `bsctr` reaches 1, data is sent from the acquisition computer to the workstation and `bsctr` is reset to `bsval`.

The two statements in [Table 19](#) initialize a real-time variable with a value. The `assign` statement uses the value of another real-time variable. The `initval` statements truncates a *double* value to an integer and sets that value.

For a specific `v`-variable, the `nitval` statement should be used only once per increment. The `initval` occurs at run-time, and only the last `initval` for a given `v`-variable is ultimately used. For example,

```
initval(1.0,v1);
pulse(pw,v1);
initval(2.0,v1);
```

```
pulse(pw, v1);
```

will give two pulses, each with a phase of 2.

The v-variables can also be initialized with the assign statement. This assign statement is a real-time initialization, and can be used multiple times. For example,

```
assign(one, v1);
pulse(pw, v1);
assign(two, v1);
pulse(pw, v1);
```

Will give two pulses, the first with phase 1 and the second with phase 2. A v-variable can be reset to zero using `assign(zero, vi)` or by subtracting it from itself `sub(vi, vi, vi)`.

**Table 19** Statements to initialize real-time variables

<code>assign(vi, vj)</code>	Initialize a real-time value of <code>vj</code> with the value of <code>vi</code> .
<code>initval(varname, vi)</code>	Initialize a real-time value <code>vi</code> with rounded value of double <code>varname</code> .

## Real-time loops and conditionals

The statements for real-time loops and conditionals are described in [Table 20](#).

**Table 20** Statements for real-time loops and conditionals

<code>elsenz(testzero)</code>	Execute code if <code>testzero</code> is not equal to 0.
<code>endhardloop(count)</code>	End a hardware loop begun with <code>starthardloop</code> - obsolete.
<code>endif(testzero)</code>	End any real-time conditional.
<code>endloop(index)</code>	End a loop begun with <code>loop</code> .
<code>ifzero(testzero)</code>	Execute code if <code>testzero</code> is equal to 0.
<code>loop(count, index)</code>	Begin a loop of <code>count</code> cycles with <code>index</code> .
<code>starthardloop(count)</code>	Begin a hardware loop of <code>count</code> cycles.

A real-time loop consists of a set of statements placed between the statement `loop(count, index)` and the statement `endloop(index)`. The argument `count` is the loop count and `index` is incremented from 1 to the value of `count`. Both arguments must be real-time integers. All of the statements in between `loop` and `endloop` must be real-time math statements or statements that generate acodes. The argument `index` can be used in real-time math statements.

The following example executes a delay that cycles among four multiples of `d3`, scan-to-scan.

```
mod4(ct,v5); /* v5 is the loop counter: v5=01230123...*/
loop(v5,v3); /* v3 is the index for the loop          */
delay(d3);   /* executes delay(d3); v5 times        */
endloop(v3);
```

The `rlloop` and `rlendloop` are a variation on `loop` and `endloop`. Instead of using an `initval` statement to initialize the loop count for the `loop` element, it is passed as the first argument to `rlloop`. The following two examples yield the same results.

```
initval(7.0,v1)
loop(v1,v10);
delay(d3);
endloop(v10);
```

```
rlloop(7.0,v1,v10);  
delay(d3);  
rlendloop(v10);
```

The advantage of the `rlloop` pair is that the total time of the loop can be calculated at run-time and the `rlloop` `rlendloop` pair can therefore be used in parallel sections of a pulse sequence. The `loop` `endloop` pair cannot be used in parallel sections of a pulse sequence.

A third real-time loop is the `kzloop` `kzendloop` pair. In this case, instead of being passed a loop count, `kzloop` is passed a loop duration. The `kzendloop` then determines the total time of all the elements between the `kzloop` and `kzendloop` statements. From that, it calculates the loop count. Any leftover time is added as a delay as part of the `kzendloop` statement. This `kzloop` `kzendloop` pair is also allowed in parallel sections of a pulse sequence.

The statements `starhardloop` and `endhardloop` function as `loop` and `endloop`. The argument `count` is the loop count but `index` is not available. These statements should be replaced with `loop` and `endloop` or `rlloop` and `rlendloop` for VNMRS.

Statements within the pulse sequence can be executed conditionally by enclosing them within the statements `ifzero(testzero)`, `elsenz(testzero)` and `endif(testzero)` statements, in which `testzero` is a real-time variable to be tested for a value of zero. The first block is executed if the value of `testzero` is 0. The block after `elsenz` is executed if the value is nonzero. The `elsenz` statement may be omitted. The `endif` statement ends the second block. The first block might contain no statements.

The following example of a real-time conditional construction executes a pulse and delay of different lengths on alternate scans.



```

mod2(ct,v1);/* v1=010101...*/
  ifzero(v1);/* test for a v1 value of 0 (odd scans)*/
  pulse(pw,v2);/* execute a pulse pw on odd scans*/
  delay(d3);/* execute a delay of d3 on odd scans*/
elsenz(v1); /* test for a v1 value of not 0 (even scans*/
  pulse(2.0*pw,v2);/* execute a pulse pw/2.0 on even scans*/
  delay(d3/2.0);/* execute a delay of d3/2.0 on even scans*/
endif(v1)

```

It is important to remember that the statements of a real-time loop or conditional are executed differently at run-time and during real-time. At run-time, every statement is executed once and only once. The loop count and the conditional test have no effect at run-time. Real-time loops and conditionals have an effect only in real-time and only on the resolved acodes. All C calculations associated with the statements is complete before the start of acquisition. It is a bad practice to place C calculations in a real-time loop or conditional. This practice can create a misleading impression that the C code is actually looped or executed conditionally.

Additional statements for several other real-time conditionals are shown in [Table 21](#).

**Table 21** Statements for additional real-time conditionals

<code>ifrtEQ(vi,vj,testzero)</code>	Execute code if the value of <code>vi</code> equals <code>vj</code> .
<code>ifrtGE(vi,vj,testzero)</code>	Execute code if the value of <code>vi</code> is greater than or equal to <code>vj</code> .
<code>ifrtGT(vi,vj,testzero)</code>	Execute code if the value of <code>vi</code> is greater than <code>vj</code> .
<code>ifrtLE(vi,vj,testzero)</code>	Execute code if the value of <code>vi</code> is less than or equal to <code>vj</code> .
<code>ifrtLT(vi,vj,testzero)</code>	Execute code if the value of <code>vi</code> is less than or equal to <code>vj</code> .
<code>ifrtNEQ(vi,vj,testzero)</code>	Execute code if the value of <code>vi</code> is not equal to <code>vj</code> .

For the statement `ifrtEQ(vi,vj,testzero)`, the argument `testzero` is a real-time variable that is set to zero if the real-time value of `vi` is equal to the value of `vj` and the block of code before `elsenz(testzero)` or `endif(testzero)` is executed.

The other statements `ifrtGE`, `ifrtGT`, `ifrtLE`, `ifrtLT`, and `ifrtNEQ` have the same behavior for their respective conditionals, greater than equal, greater than, less than or

equal, less than, and not equal.

## Real-time integer tables

The real-time integer tables named `t1` to `t60` each store arrays of 32-bit integers on the acquisition computer. An integer table name can be used in any statement to control a phase, power, or amplitude in which a real-time integer, `v1` to `v42` might be used. Integer tables are used most often to create phase tables. For example, the statement `rgpulse(pw,t1,rof1,rof2)` performs an obs transmitter pulse whose quadrature phase is specified by the table `t1`. Alternatively, `rgpulse(pw,v1,rof1,rof2)` performs the same pulse with the value `v1`.

Integer table names cannot be used in the real-time math statements listed in [Table 16](#) on page 82.

A table name is initialized just once with the first increment of a multidimensional or arrayed pulse sequence. The `settable` and `loadtable` statements are ignored in subsequent increments. The initialized table values are available at the beginning of each increment.

By default, the index to a table is initialized to zero and incremented by one, at the end of each scan. The table element with index of 0 is used for all scans during a steady-state period.

[Table 22](#) shows the statements for handling tables.

**Table 22** Statements for handling tables

<code>getelem(tablename, Aindex, Aptest)</code>	Retrieve an element from a table.
<code>loadtable(file)</code>	Load table elements from a table, text file.
<code>setreceiver(tablename)</code>	Associate receiver phase with a table.
<code>settable*</code>	Initialize a table with an integer array.
<code>tsadd(tablename, scalarval, moduloval)</code>	Add an integer to table elements.
<code>tsdiv(tablename, scalarval, moduloval)</code>	Apply an integer division to table elements.
<code>tsmult(tablename, scalarval, moduloval)</code>	Multiply an integer by table elements.
<code>tssub(tablename, scalarval, moduloval)</code>	Subtract an integer from table elements.
<code>ttadd*</code>	Add two tables.
<code>ttdiv*</code>	Apply an integer division to two tables.
<code>ttmult*</code>	Multiply two tables.

**Table 22** Statements for handling tables (continued)

<code>ttsub*</code>	Subtract two tables.
<code>*settable(tablename, numelements, intarray)</code>	
<code>*ttadd(tablenamedest, tablenamemod, moduloval)</code>	
<code>*ttdiv(tablenamedest, tablenamemod, moduloval)</code>	
<code>*ttmult(tablenamedest, tablenamemod, moduloval)</code>	
<code>*ttsub(tablenamedest, tablenamemod, moduloval)</code>	

The `loadtable(file)` statement loads table elements from a table text file. The argument `file` specifies the name of a Linux text file in the user's personal `tablib` directory or in the VnmrJ system `tablib` directory. The statement `loadtable` can be used multiple times within a pulse sequence with different file names. However, you must ensure that a table name is not called more than once.

The `settable(tablename, numelements, intarray)` statement stores an array of integers in a real-time table. The argument `tablename` specifies the name of the table (`t1` to `t60`). The argument `numelements` specifies the size of the table. The argument `intarray` is a C array that contains the values of the table elements. The dimension of `intarray` should be greater than `numelements`. Like the `initval` statement, the `settable` is a run-time element. For a specific table variable, the `settable` statement should be used only once per increment. In contrast to `initval`, the first `settable` for a give table variable is used, unless the table size changes. For good programming practice, only use a single `settable` call for a given table variable.

The `getelem(tablename, APindex, APdest)` statement retrieves an element from a table. The argument `tablename` specifies the name of the Table (`t1` to `t60`). `APindex` is a variable (`v1` to `v42`, `oph`, `ct`, `bsctr`, `ssct`, or `ssctr`) that contains the index of the required table element. Note that the first element of a table has an index of 0. `APdest` is a real-time variable (`v1` to `v42` and `oph`) into which the retrieved table element is placed. Use `getelem` to customize the order of retrieval of table elements or to set a phase cycle during a steady-state period.

The `setreceiver(tablename)` statement assigns the `ctth` element of the table `tablename` to the receiver variable `oph`. If multiple `setreceiver` statements are used in a pulse sequence, or if the value of `oph` is changed by real-time

## 2 Pulse-Sequence Programming

math statements such as `assign`, `add`, *etc*, then the last value of `oph` prior to the acquisition of data determines the value of the receiver phase.

To perform scalar operations of an integer with table elements, use the following statements:

```
tsadd(tablename, scalarval, moduloval)
tssub(tablename, scalarval, moduloval)
tsmult(tablename, scalarval, moduloval)
tsdiv(tablename, scalarval, moduloval)
```

in which `tablename` specifies the name of the table (`t1` to `t60`) and `scalarval` is a C-integer added to, subtracted from, multiplied with, or divided into each element of the table. The result of the operation is taken modulo `moduloval` if `moduloval` is greater than 0. For the `tsdiv` statement, a `scalarval` of 0 causes a run-time error.

To perform run-time vector operations of one table with a second table, use the following table-to-table statements:

```
ttadd(tablenamedest, tablenamemod, moduloval)
ttsub(tablenamedest, tablenamemod, moduloval)
ttmult(tablenamedest, tablenamemod, moduloval)
ttdiv(tablenamedest, tablenamemod, moduloval)
```

in which `tablenamedest` and `tablenamemod` are the names of tables (`t1` to `t60`). Each element in `tablenamedest` is modified by the corresponding element in `tablenamemod`. The result, stored in `tablenamedest`, is taken modulo `moduloval` if `moduloval` is greater than 0. The number of elements in `tablenamedest` must be greater than or equal to the number of elements in `tablenamemod`.

## The format of a table file

Each table definition must start with the table name `t1` to `t60` and be followed by an enumeration of the table elements. Each table element must be written as an integer number and separated from the next by some form of white space, such as a blank space, tab, or carriage return. The table name is separated from the elements by an '=' sign. A table definition can occur only once in a table file.

A table that is defined as below starts with an index of 0 and increments by one, once per scan.

Special notation within the table modifies access to the table and it can simplify entering the table elements.

**Table 23** Formatting tables

<code>(...)</code> #	Indicates that the sequence of table elements within the parentheses are to be replicated in their entirety # times (# ranges from 1 to 64) before proceeding to any succeeding elements. For example: <code>t1=(0 1 2)3 /* t1 table=012012012 */.</code>
<code>[...]</code> #	Indicates that each element within square brackets is to be replicated # times (# ranges from 1 to 64) before going to the next element. For example: <code>t1=[0 1 2]3 /* t1 table=000111222 */.</code>

Nesting of `(...)` and `[...]` expressions is not allowed. Multiple `{...}` expressions within one table are not allowed.

The overhead operations preceding every transient are resetting the DTM (data-to-memory) control information. The overhead operations following every transient are error detection for the number of points and data overflow; detection for blocksize, end of scan, and stop acquisition, and resetting the decoupler status. `d0` does not take these delays into account.

The overhead operations preceding every array element are initializing the rf channel settings (frequency, power, *etc.*), initializing the high-speed (HS) lines, initializing the DTM, and if arrayed, setting the receiver gain. `d0` does not take into account arraying of decoupler status shims, VT, or spinning speed.

## Shaped Pulses and Waveforms

This section contains information on programming of shaped pulses and waveforms in VnmrJ.

### Executing shaped pulses

A *shaped pulse* is an RF pulse that is executed with a program of variable phase, amplitude and frequency offset in order to control the bandwidth of excitation. Execution of a shaped pulse requires the acquisition computer output time functions of phase and amplitude with maximum speed and minimum time resolution. Time functions of frequency offset are convoluted with the phase program, noting that a frequency-offset pulse can be produced with a linear time-ramp of phase.

Any pulse shape can be represented as a table of steps of duration, phase, amplitude and gate, a *pattern*. The gate is used to provide a true zero amplitude, should that be needed for the shape. VnmrJ represents the pattern as a .RF text file in the directory `shapelib` of the user, of an applications directory or of the system. Two related entities, *waveforms* and *gradient shapes*, have different file extensions, .DEC and .GRD, respectively, and are also stored in `shapelib`.

The acquisition computer has special software to optimize the output-speed of shaped pulses and waveforms. This software is accessed by including a shaped-pulse statement in the pulse sequence. If a shape is used, information from the appropriate .RF file is downloaded to the acquisition computer at run-time. The information in acquisition computer memory is accessed in real-time by the acodes associated with the shaped-pulse statement. Use of a shaped-pulse statement and a .RF file removes the need to program a standard loop of pulse-sequence statements. The shape usually also has better execution performance than a loop.

Statements to perform shaped pulses are listed in [Table 24](#).

**Table 24** Statements to create shaped pulses

<code>decshaped_pulse*</code>	Perform a shaped pulse on the dec channel.
<code>dec2shaped_pulse*</code>	Perform a shaped pulse on the dec2 channel.
<code>dec3shaped_pulse*</code>	Perform a shaped pulse on the dec3 channel.
<code>shaped_pulse*</code>	Perform a shaped pulse on the obs channel.
<code>simshaped_pulse*</code>	Perform two simultaneous pulses on obs and dec.
<code>sim3shaped_pulse*</code>	Perform three simultaneous pulses on obs, dec, and dec2.
<code>sim4shaped_pulse*</code>	Perform four simultaneous pulses on obs, dec, dec2, and dec3.

```
*decshaped_pulse(shape,width,phase,RG1,RG2)
```

```
*dec2shaped_pulse(shape,width,phase,RG1,RG2)
```

```
*dec3shaped_pulse(shape,width,phase,RG1,RG2)
```

```
*simshaped_pulse
(obsshape,decshape,obswidth,decwidth,RG1,RG2)
```

```
*sim3shaped_pulse
(obsshape,decshape,dec2shape,obswidth,decwidth,dec2width,
RG1,RG2)
```

```
*sim4shaped_pulse
(obsshape,decshape,dec2shape,dwc3shape,obswidth,decwidth,
dec2width,dec3width,RG1,RG2)
```

The statement `shaped_pulse(shape,width,phase,RG1,RG2)` executes a shaped pulse on the obs transmitter, in which `shape` is the pattern name and the root name of the `.RF` file in `shapelib`. The argument `width` is the duration of the pulse, `phase` is a real-time variable or a phase table to set the initial quadrature phase, and `RG1` and `RG2` are a pre-delay and post-delay similar to those of the `rgpulse` statement. For example, `shaped_pulse("gauss",pw,v1,rof1,rof2)` uses the `gauss.RF` file in the system `shapelib` to execute a pre-calculated standard Gaussian pulse.

The power of the shaped pulse is controlled by the existing attenuator setting and the maximum amplitude (1024) of the shaped pulse is scaled to the amplitude of the current fine power setting. The duration steps of the shaped pulse are scaled to fit the `width` argument. If the pattern name is `'`, the output will be a rectangular pulse.

The statements `decshaped_pulse`, `dec2shaped_pulse`, `dec3shaped_pulse` and `dec4shaped_pulse` have the same



behavior with the respective `dec`, `dec2`, `dec3`, and `dec4` channels.

The statement `simshaped_pulse`

(`obsshape`, `decshape`, `obswidth`, `decwidth`, `obsphase`, `decphase`, `RG1`, `RG2`) performs a simultaneous, two-pulse, shaped pulse on the `obs` and `dec` transmitters. The argument `obsshape` is the pattern name of shape executed on `obs` and `decshape` is the pattern name of the shape executed on `dec`. The arguments `obswidth` and `decwidth` are the respective pulse widths, `obsphase` and `decphase` are the respective quadrature phase variables or tables and `RG1` and `RG2` are defined for the `simpulse` statement. The two pulses are centered, as for `simpulse`. An example is:

```
simshaped_pulse("gauss", "hrm180", pw, p1, v2, v5, rof1, rof2).
```

If either `obswidth` or `decwidth` is 0.0, no pulse occurs on the corresponding channel. In this case the `simshaped_pulse` will not interfere with decoupling on that channel, if it is occurring. If either pattern name is '', the output will be a rectangular pulse.

The statement `sim3shaped_pulse` performs a simultaneous, three-pulse shaped pulse on the `obs`, `dec` and `dec2` transmitters. This statement is also used to provide a two-pulse shaped pulse on any two of these transmitters. Other behavior is similar to `simpulse`.

The statement `sim4shaped_pulse` performs a simultaneous, four-pulse, shaped pulse on the `obs`, `dec`, `dec2` and `dec3` transmitters. This statement is also used to provide two and three-pulse shaped pulse on any of these transmitters. Other behavior is similar to `simpulse`.

The statement

```
shapedpulseoffset("gauss", width, phase, RG1, RG2, offset)
```

performs a `shaped_pulse` with a frequency offset, determined by the argument `offset` (Hz). The offset is obtained by convoluting a linear phase ramp with the RF pattern obtained from the `.RF` file. The convolution is obtained by a combination of operations performed at both run time and real time. The statement is used to provide a frequency offset to a standard non-offset `.RF` pattern and avoids the need to explicitly program the offset into the shape.

## Calculation of shaped pulses

The .RF file for a shaped pulse (Table 25) has four columns that describe phase, amplitude, relative durations and the transmitter gate.

**Table 25** RF patterns

Column	Description	Limits	Default
1	Phase angle (degrees)	0.0044° resolution	Required
1	Phase angle (degrees)	0.0055° resolution	Required
2	Amplitude	0.0 to scalable max	max
3	Relative duration	0.0, or 1.0 to 255.0	1.0
4	Transmitter gate	0.0 or 1.0	1.0 (gate on)

Each line is a step in the shape. The first column sets the phase in degrees with four significant figures after the decimal to accommodate the DD2 MR 0.0055° (360.0°/65536) phase resolution. The second column is the relative amplitude with values of 0.0 to 1023.0 and at least 3 significant figures after the decimal for older systems that have 16-bit (1023.0/65536) amplitude resolution. The maximum amplitude of the shape is scaled at run-time to the current value of the fine attenuator 0.0 to 4095.0. The scaling preserves the 16-bit amplitude resolution. The third column is a relative duration for each step. The duration values are translated into time intervals at run time based upon the argument `width`. Time intervals are rounded to 0.025 μs (for DD2 MR) or 0.05 μs (for VNMRS) resolution. The fourth column is an optional gate. If the gate column is absent, the value is assumed to be 1.0 for transmitter on. If a shaped pulse has period in which the transmitter is off, then one must supply gate values for each step, in which 1.0 is for transmitter-on and 0.0 is for transmitter-off.

A row with a duration of 0.0 and an amplitude of 1023.0 can be used to make sure that the amplitude is scaled to the maximum step of 1023.0 rather than the maximum step of the pattern. One can also use a value greater than 1023.0 (for example 4095.0) to increase the range of amplitude values. This practice is common in imaging sequences.

Pre-calculated pattern files for pulse shapes can be supplied by any appropriate macro, external program, or by simply typing with a text editor. It is the user's responsibility to know the equation for shaped pulse and calculate the steps.

VnmrJ supplies an external C program called `Pbox` whose purpose is to calculate a large variety of the shapes used for spectroscopic applications. The information that `Pbox` uses for calculation of shapes is contained in the `wavelib` directory of the system. `Pbox` has its own programming language that is described in the *Spectroscopy User Guide*.

## Programmed waveforms for decoupling

A *waveform* is an RF pattern of phase amplitude, frequency, and offset that is repeated in a loop, usually to provide *decoupling* or a *spinlock*. Execution of a waveform requires that the acquisition computer output time functions of phase and amplitude with maximum speed and minimum time resolution and sustain that output for long periods of time. Time functions of frequency offset are convoluted with the phase function, noting that a frequency-offset can be produced with a linear time ramp of phase.

Statements related to programmable waveform control are listed in [Table 26](#).

**Table 26** Statements for waveform control

<code>decprgoff()</code> *	End waveform decoupling on the <code>dec</code> channel.
<code>dec2prgoff()</code> *	End waveform decoupling on the <code>dec2</code> channel.
<code>dec3prgoff()</code> *	End waveform decoupling on the <code>dec3</code> channel.
<code>decprgon()</code> *	Begin waveform decoupling on the <code>dec</code> channel.
<code>dec2prgon()</code> *	Begin waveform decoupling on the <code>dec2</code> channel.
<code>dec3prgon()</code> *	Begin waveform decoupling on the <code>dec3</code> channel.
<code>obsprgoff()</code> *	End waveform decoupling on the <code>obs</code> channel.
<code>obsprgon()</code> *	Begin programmable decoupling on the <code>obs</code> channel.
<code>*decprgon(pattern_name, 90_pulselength, tipangle_resoln)</code>	
<code>*dec2prgon(pattern_name, 90_pulselength, tipangle_resoln)</code>	
<code>*dec3prgon(pattern_name, 90_pulselength, tipangle_resoln)</code>	
<code>*obsprgon(pattern_name, 90_pulselength, tipangle_resoln)</code>	

The statement

`obsprgon(name, 90_pulselength, tipangle_resoln)` is used to start the waveform control of the `obs` transmitter. The argument `name` is the pattern name and the root name of the `.DEC` file in `shapelib`. The argument `90_pulselength` is a time duration for a step of 90 units of relative duration in

the .DEC file. The argument `tipangle_resoln` is the number of relative duration units associated with the minimum duration applied in the output of the pattern.

For simple waveforms, for example WALTZ16 decoupling, the relative duration units correspond to degrees of tip-angle for calibrated pulses. A statement for WALTZ16 is

```
obsprgon("waltz16",pw90,90.0),
```

which uses the pattern file `waltz16.DEC`. The pulses of WALTZ16 have tip angles of 90° and 180° and the respective, relative durations in the .DEC file are labeled 90.0 and 180.0. Argument 3 indicates that a step labeled 90.0 corresponds to one step in the RF pattern and argument 2 indicates that that step of 90 units has a pulse width of `pw90` μs.

Many waveforms have no obvious connection with tip angle and it is advantageous to set the minimum step to be the smallest duration that can be produced by the acquisition computer, usually 50 ns or 100 ns. To avoid complicated reasoning, it is best to *designate* the minimum step to have a *label* of 90.0 in the .DEC file. In this case, argument 3 is always 90.0 and argument 2 is the length of the minimum step, 50.0e-9 or 100.0e-9. When using this format, pulse lengths are designated as an integer number, N units of the minimum step, and they are labeled in the .DEC file with the number equal to  $N \times 90.0$ .

The `obsprgon` statement returns an *int* with the number of 50-ns ticks in one full pattern.

The statement `obsprgoff` is used to terminate a period of programmed decoupling on the `obs` transmitter. A programmed decoupling period can be terminated at any time. The pattern will halt in mid-cycle with no ill effects if the delay is not a multiple of the pattern time. This capability is of significant advantage when you need to decouple during an arbitrary period, say F1 of a 2-dimensional sequence, which does not require to synchronize the decoupler cycle with the F1 dwell. With a loop construction, you could only stop at multiples of the decoupler cycle.

The statements `decprgon`, `decprgoff`, `dec2prgon`, `dec2prgoff`, `dec3prgon`, `dec3prgoff`, `dec4prgon`, and `dec4prgoff` provide similar waveform capability with the `dec`, `dec2`, `dec3`, and `dec4` channels.

## Waveforms with an automatic frequency offset

The statements in [Table 27](#) begin a programmed waveform with an automatic frequency offset.

**Table 27** Statements for waveform control with an offset

<code>decprgonOffset()</code> *	Begin offset waveform decoupling on the dec channel.
<code>dec2prgonOffset()</code> *	Begin offset waveform decoupling on the dec2 channel.
<code>dec3prgonOffset()</code> *	Begin offset waveform decoupling on the dec3 channel.
<code>obsprgonOffset()</code> *	Begin offset waveform decoupling on the obs channel.
<code>*decprgonOffset(pattern_name, 90_pulselength, tipangle_resoln, frequency)</code>	
<code>*dec2prgonOffset(pattern_name, 90_pulselength, tipangle_resoln, frequency)</code>	
<code>*dec3prgonOffset(pattern_name, 90_pulselength, tipangle_resoln, frequency)</code>	
<code>*obsprgonOffset(pattern_name, 90_pulselength, tipangle_resoln, frequency)</code>	

The statement `obsprgonOffset` begins a waveform with an automatic frequency offset, for which `frequency` is a *double* that supplies the offset frequency in Hertz. Use `obsprgoff` to end the period of a programmed waveform. The first three arguments are similar to those of `obsprgon`.

The frequency offset is created by applying a linear phase ramp to the elements of the named pattern. The initial phase of the pattern is similar to that applied by `obsprgon`. The phase remains coherent throughout the repeated cycles of the entire pattern. The ending phase of each cycle is applied to the beginning of the next cycle. With `obsprgoff`, the phase reverts to the state before the `obsprgonOffset` was applied. You must explicitly program the starting phase of the next block of the sequence if there is a requirement that it be phase-coherent with the waveform.

Individual periods of duration in the named pattern are expanded into many elements each with a 50.0 ns duration and a linear increment or decrement of the phase. The expansion is carried out partially at run-time and the waveform stored in the acquisition computer will be substantially larger than the original named pattern. The rest of the expansion is carried out in real time by the RFcontroller.

A waveform created for use with `obsprgonOffset` will be stored in the acquisition computer with a size approximately 10-fold smaller than an equivalent offset waveform that

might be created for `obsprgon`. The last 10-fold expansion is carried out in real-time. The use of `obsprgonOffset` rather than `obsprgon` will decrease the startup time, because a smaller waveform is loaded, and it will allow a greater number of waveforms to be stored, in cases, for example, where the offset is arrayed.

The named patterns that are run with `obsprgonOffset` should be run with a minimum element size of 25 ns for DD2 MR systems and 50 ns for older systems. This means that  $90\_pulselength \geq 250.0e-9 * 90.0 / \text{tipangle\_resoln}$ . For example, the WALTZ sequence described above would have a minimum 90° pulse length of 25 ns for DD2 MR systems and 50 ns for older systems. You should be cautious in using waveforms for which `90_pulselength` is already a minimum step-size. The pattern files for such a waveform must be calculated for a minimum step-size greater than or equal to 25 ns for DD2 MR systems and 50 ns for older systems. .

User-libraries for the calculation of shaped pulses (such as Pbox) have the capability to apply frequency-offset waveforms, in which the entire offset pattern is represented in the `.DEC` pattern file. These patterns should be used with `obsprgon` and generally they will fail with `obsprgonOffset`, unless they have been purposefully calculated with a coarse stepsize.

The statements `deprgonOffset`, `dec2prgonOffset`, and `dec3prgonOffset` have a similar function on their respective channels.

## Using spinlock pulses

A *spinlock* pulse is a shaped pulse constructed from N cycles of programmed decoupling. Statements for creation of spinlocks are listed in [Table 28](#).

**Table 28** Statements for spinlock control

<code>decspinlock()</code> *	Perform a spinlock on the dec channel.
<code>dec2spinlock()</code> *	Perform a spinlock on the dec2 channel.
<code>dec3spinlock()</code> *	Perform a spinlock on the dec3 channel.
<code>spinlock()</code> *	Perform a spinlock on the obs channel.

\*(pattern, 90\_pulselength, tipangle\_resoln, phase, ncycles)

The statement `spinlock(name, 90_pulselength, tipangle_resoln, phase, n`

cycles) performs a spinlock pulse in which `pattern_name` is the name of the pattern and the root name of the `.DEC` file in `shapelib`. The arguments `90_pulselength` and `tipangle_resoln` have meaning similar to that for the statement `obsprgon`. The argument `phase` is a quadrature phase table or real-time variable and `ncycles` is the number of cycles of the pattern applied during the spinlock pulse. The spinlock has no predealy or postdelay. The spinlock sets the phase and applies transmitter unblinking coincident with the start of the pulse. An example is `spinlock('mlev16',pw90,90.0,v1,ncyc50)` which applies a pulse of `ncyc50` cycles of MLEV16, using the pattern file `mlev16.DEC`.

The statements `decspinlock`, `dec2spinlock`, `dec3spinlock`, and `dec4spinlock` provide similar spinlock capability with the `dec`, `dec2`, `dec3`, and `dec4` channels.

## Calculation of programmed waveforms

The `.DEC` file for programmed decoupling (Table 29) has four columns: relative duration, phase, amplitude, and the transmitter gate.

**Table 29** Elements of a waveform pattern

Column	Description	Limits	Default
1	Tip angle (degrees)	0.0 to max (unlimited)	Required
2	Phase (degrees)	0.0 to 360.0 (preferred)	Required
3	Amplitude	0.0 to 1023.0	1023.0
4	Transmitter gate	0.0 or 1.0	1.0 (gate on)

Each line is a step in the programmed decoupling pattern.

The first column sets the relative duration in "tip-angle" units. For patterns made of 90° and 180° pulses, this value should be the desired tip angle of the calibrated pulse. The time period corresponding to a tip angle of 90.0 is set by the `obsprgon` statement. For an arbitrary waveform, where there may be no reference to tip angle, the duration step should be the value `Nx90.0` in which `N` is the number of minimum step-sizes in the required pulse width. In this second format, `obsprgon` sets the minimum step size and the third argument should be `90.0`.

The second column sets the phase in degrees with four significant figures after the decimal to accommodate the

0.0055° (360.0/65536) phase resolution of VnmrJ. The actual phase is determined by the resolution of the hardware.

The third column is the relative amplitude with values of 0.0 to 1023.0. Three significant figures after the decimal are required to preserve the 16-bit resolution of VnmrJ. The actual amplitude is determined by the resolution of the hardware. The maximum amplitude of the shape is scaled at run-time to the current value of the fine attenuator 0.0 to 4095.0, preserving 16 bit resolution. If the third column is absent, the amplitude is set to 1023.0.

The fourth column is an optional gate. If the gate column is absent, the value is assumed to be 1.0 for transmitter-on. If a shaped pulse has a period in which the transmitter is off, then you must supply gate values for each step in which 1.0 is for transmitter-on and 0.0 is for transmitter-off.

The listing below shows the first 8 steps of a standard pattern, waltz16.DEC. The amplitude and gate columns are omitted

```

270.0 180.0
360.0 0.0
180.0 180.0
270.0 0.0
90.0 180.0
180.0 0.0
360.0 180.0
180.0 0.0

```

A row with a duration of 0.0 and an amplitude of 1023.0 can be used to ensure that the fine power is scaled to the maximum step of 1023.0 rather than the maximum step of the pattern.

Precalculated pattern files for pulse shapes can be supplied by any appropriate macro, external program or by simply typing with a text editor. It is the users responsibility to know the equation for their shaped pulse and calculate the steps. Many waveform patterns can be obtained from PBOX.

## Calculating gradient shapes

A gradient file has two columns, as shown in [Table 30](#). The first column is a gradient DAC level that should have the range -32767 to 32767 in units of 1. The second column is a



relative duration similar to that of the .RF file.

**Table 30** Elements of a gradient pattern

Column	Description	Limits	Default
1	Output amplitude	-32767 to 32767, 1 unit resolution	Required
2	Relative duration	1 to 255	1

## Statements that create shape, waveform and gradient files

Shape, waveform, and gradient text files can be created in shapelib with their respective statements, `init_rfpattern`, `init_decpattern`, and `init_grad_pattern` (Table 31).

**Table 31** Statements to create pattern files

<code>init_rfpattern(pattern, rfpattern_struct, nsteps)</code>	Create shaped-pulse pattern file.
<code>init_decpattern(pattern, decpattern_struct, nsteps)</code>	Create waveform pattern file.
<code>init_gradpattern(pattern, gradpattern_struct, nsteps)</code>	Create gradient pattern file
<code>userRFshape(pattern, rfpattern_struct, nsteps, channel)</code>	Create shaped-pulse pattern directly
<code>userDECshape(pattern, decpattern_struct, tipangle_reson, mode, nsteps, channel)</code>	Create waveform pattern directly

The statement

`init_rfpattern(pattern, rfpattern_struct, nsteps)` creates a text file in shapelib with a root name of `pattern` with a .RF extension. The information for this file is obtained from a C structure variable `rfpattern_struct` and `nsteps` is the number of elements in the file. To use `init_rfpattern`, declare a structure variable of `RFpattern` type with more than `nsteps` elements, for example:

```
RFpattern myshape(512);
```

in which the pattern `myshape` has 64 elements. Set values in the pulse sequence for `myshape.phase[n]`, `myshape.amp[n]`, and `myshape.time[n]` for each element `n=0` to 63 in the pattern file. The statement:

```
init_RFpattern('myshape', myshape, 64);
```

creates a text file in shapelib called `myshape.RF`.

For `init_decpattern(pattern, decpattern_struct, nsteps)`, the structure type is `DECpattern` and for a variable name `mywave`, they are `mywave.tip`, `mywave.amp`, `mywave.phase`, and `mywave.gate`.

For `init_gradpattern((pattern,rfpattern,nteps))`, the structure type is `DECpattern` and for a variable name `mygrad`, they are `mygrad.amp`, `mygrad.time`, and `mygrad.ctrl`.

The statement `userRFshape` creates a pattern for a shaped pulse directly in the acquisition computer and avoids the need to write a file into `shapelib`. The arguments are the same as `init_RFpattern` except that you must declare a fourth argument `channel` as `OBSch`, `DECch`, `DEC2ch`, `DEC3ch`, or `DEC4ch`. Direct creation of a pattern saves time at run-time.

The statement `userDECshape` creates a pattern for a waveform directly in the acquisition computer and avoids the need to write a file into `shapelib`. The arguments are the same as `init_DECpattern` except that you must declare a fourth argument `channel` as `OBSch`, `DECch`, `DEC2ch`, `DEC4ch`, or `DEC4ch`. The argument `tipangle_resoln` should be the same as the third argument of `obsprgon`.

## Waveform interpolation

### Automatic waveform interpolation

For MR systems with RF controllers, waveform execution is automatically enhanced by use of an *interpolation* processor, located on the RF controller between the main processor and the FIFO output. Interpolation is used to automatically execute any waveform duration designated to contain more than one duration step. For example, if the resolution (`tipangle_resoln`, argument 3 of `obsprgon`) is 90.0, a waveform element that is designated with a duration of 180.0 is executed by the interpolator as two steps of 90.0. Use of the interpolator saves time for the main processor, which only has to get one pattern element from memory, rather than two. The result is improved efficiency and a smaller probability for FIFO underflow errors. Interpolation was added with `VnmrJ2.1C`.

The interpolator can execute up to 256 steps using one pattern element. Waveform elements longer than 256 steps are automatically divided into multiple pattern elements. The execution speed of a 256-step element is only slightly larger than that of a single-step element. The break point for improved efficiency occurs with pattern elements of greater than about 3 to 5 steps, and the potential for improved efficiency is 50 to 100 fold.

Interpolation is beneficial when the duration step size (`90_pulselength`, argument 2 of `obsprgon`) is small (25 ns to 250 ns) and the typical width of a pulse in the waveform is large. The interpolator makes it possible to execute many small steps efficiently. The duration step size is also the pulse width resolution, so it is advantageous to keep the step size small. For example, a waveform such as GARP decoupling has long pulses that must be set with  $1^\circ$  flip-angle resolution. The pulses of GARP are executed by the interpolator each as a group of  $1^\circ$  steps. For many waveforms there is no obvious connection between the flip angle and the length of pulses. The practice for these waveforms is to choose a small step size (25 ns to 100 ns) and express the pulse length in steps. Interpolation makes this practice possible. See the section "Programmed Waveforms for Decoupling" for a discussion of `obsprgon` and its arguments. Interpolation with the statements `decprgon`, `dec2prgon`, `dec3prgon` and `dec4prgon` is similar to that with `obsprgon`.

The statement `obsprgonOffset` uses interpolation to create a frequency offset. The frequency designated by the offset argument is executed automatically by the interpolator as a ramp of small phase steps, executed in blocks. The interpolator can execute the ramped phase steps of `obsprgonOffset` with the same efficiency as the constant steps of `obsprgon`. The statements `decprgonOffset`, `dec2prgonOffset`, `dec3prgonOffset` and `dec4prgonOffset` are similar to `obsprgonOffset`.

### Custom waveform interpolation with `userDECshape`

The `userDecshape` statement and the `DECpattern` structure can be used to generate waveforms with explicit control of interpolation. This capability is not available for the `userRFshape` statement.

The statement `userDecshape` can generate waveforms for any of the pulse-sequence channels `obs`, `dec`, `dec2`, etc depending on the value of its sixth argument channel. These waveforms are executed by `obsprgon`, `decprgon`, etc. The `DECpattern` structure, contains a set of fields `tip`, `phase`, `amp`, and `gate` to describe the shape, and this structure provided as an argument to `userDecshape` to calculate the waveform. See "Statements that create shapes, waveform and gradient files" in this chapter, for more information about `userDECshape`.

The `DECpattern` structure has two additional fields for interpolation, `phase_inc` and `amp_inc`. The value of `phase_inc` is an increment to the value of `phase` that is applied with each step of the element. The first step is set as `phase` and each subsequent step adds `phase_inc`. Both `phase` and `phase_inc` are expressed in degrees with 16-bit resolution (one unit is  $\sim 0.0055^\circ$ ). The value of `amp_inc` is an increment to the value of `amp` that is applied with each step of the element. The first step is set as `amp` and each subsequent step adds `amp_inc`. Both `amp` and `amp_inc` are expressed in amplitude units (0.0 to 1023.0) with 16-bit resolution (one unit is  $\sim 0.0625$ ). It is possible to ramp phase and amplitude at the same time.

The number of duration steps to be interpolated is set as `tip/(tip_angle_resoln*90_pulselength`, where `tip` is the duration field of the `DECpattern` structure. The `tip_angle_resoln` and `90_pulselength` are arguments 2 and 3 of `obsprgon`, `decprgon`, etc. If the number of steps is greater than 256, multiple elements are calculated.

The use of `userDECshape` with control of interpolation is found with versions of VnmrJ 3 and greater. Sequences that use `userDECshape` in earlier versions of VnmrJ will give an error message in VnmrJ 3, indicating the absence of the increment values, `phase_inc` and `amp_inc`. Interpolation cannot be used with standard `.DEC` text, pattern files. Increment values in columns 5 and 6 of these text files, if generated, would be ignored.

The interpolation capability of the `userDECshape` command was developed to make faster waveforms available for solid-state NMR experiments and it is also available for future pulse sequence development. With VnmrJ 3.2 and later, the standard sequences for solids (see Chapter 13 of the "Spectroscopy User Guide") can make use of `userDECshape` through their access to solids pulse sequence modules in the `#include` file `/vnmr/psg/solidstandard.h`. Solids interpolation is turned off by default. See the documentation for the SolidsPack package appropriate to VnmrJ 3 for more information.

At present (vnmrJ3.2) no other sequences or packages make use of `useDECshape` for interpolation.

### Considerations for waveform programming with interpolation

A waveform that is constructed with `userDECshape` and interpolation can be viewed to be constructed from trapezoidal elements. The use of trapezoidal elements

improves waveform fidelity with only a small cost in efficiency, when it is not possible to further reduce the duration of elements. Interpolation becomes important for waveforms where the average duration of elements approaches 100 to 300 ns, where a controller could possibly give FIFO underflow errors.

Interpolated steps can also have a variable duration and therefore accommodate the simulation of pulse lengths with greater flip-angle resolution or time resolution. One must still consider two times, a *small time* and a *large time*. With interpolation the argument `90_pulselength` of `obsprgon` is set to the small time, the smallest step size that the controller can generate (25 ns or 50 ns). Individual durations vary with this same resolution. However, one must still be sure that the average size of an element does not fall below a large time of about 100 to 300 ns. A conventional alternative without interpolation would be to set `90_pulselength` to the large time, with corresponding loss of time resolution.

The phase or amplitude slope of a trapezoidal element must be the result of a constant increment. The interpolator cannot dither (vary) the phase or amplitude increment within an element to produce an arbitrary slope. However one can dither adjacent elements to achieve an average slope over a longer time. This practice still provides better fidelity than the use of constant steps.

A typical use of interpolation is to generate a phase-ramped offset. As noted above, the standard command `obsprgonOffset` does this and `obsprgon` uses interpolation with constant steps. For the sequence CPM, cosine phase modulation, used for solid-state NMR, the phase varies between +/- limits with a cosine function. One can calculate CPM with interpolation using piecewise linear steps and get effective performance approaching four elements per cycle. A chirp or single frequency sweep, SFS, is calculated piecewise as a parabolic phase ramp. The sequence DFS, double frequency sweep is a varying sinusoidal amplitude modulation and is calculated with piecewise amplitude steps.

## Programming for Acquisition Control

This section contains information on programming of acquisition control in VnmrJ.

### Implicit acquisition

If a pulse-sequence `.c` file contains no statements related to acquisition, VnmrJ will insert a standard acquisition at the end of the sequence. A large majority of VnmrJ sequences employ this *implicit* acquisition. When using implicit acquisition, the last statement of the sequence should be the receiver delay, the time after the last pulse, before the transmit/receive switch and the receiver are turned on. Usually that statement is `delay(rof2)`; for which `rof2` is the *global* PSG variable used for the receiver turn-on. The implicit acquisition then supplies a second acquisition delay `alfa`, followed by the acquisition of `np` points with a dwell-time of  $1/sw$ . No explicit statements can follow an implicit acquisition. The implicit acquisition automatically ends any `status` period, started before acquisition, and the sequence returns to the first statement.

For a default, implicit acquisition, the digital receiver adds a short period of acquisition (about  $1.0/sw$ ) following the time at to fully establish the value of the last point. The parameter `acqtm`, if created, will be set with the full acquisition time. To avoid this additional delay, it is a good practice create `acqtm` and set it equal to 'n'. For a usual acquisition, the last point is noise and does not need to be determined precisely.

The implicit acquisition also supplies an overhead period of about 200  $\mu$ s to complete the scan, save the data, and begin the next scan. This time, along with the additional acquisition time, is automatically placed in the recycle delay `d1` of the next scan. If `d1` is less than the overhead delay, an error is thrown.

The overhead delay and additional acquisition delay will be encapsulated in a delay with length `d0`, if this parameter is defined. Use the value of `d0` to set a minimum recycle time when `d1=0.0`.

### Standard acquisition with the VNMRS digital receiver

The VNMRS employs a digital receiver to obtain all FID data for spectral widths up to 5 MHz. The digital receiver

digitizes a 20 MHz NMR signal, Intermediate Frequency (IF), using a single 80-MHz Analog-to-Digital Converter, ADC. At the time of digitization, the center frequency of the spectrum is 20 MHz. Peaks to higher frequency, to the left of the spectrum, are just greater than 20 MHz and lower frequency peaks are just less than 20 MHz. The 20-MHz digitized signal is then processed with digital filters in order to down-shift the center frequency to zero. The result is a standard FID, containing real and imaginary parts, with the desired dwell time  $1.0/sw$ .

For all systems, all digital filters are *brick-wall* by default with a cutoff at  $\pm sw/2.0$ . The only spectral widths that are allowed are those for which filters exist. VnmrJ will set  $sw$  to the closest allowed value to that which is entered. All data from the system digital filters are *time-corrected*, so as to remove the effects of digital-filter delay. The result of a default time-correction, without further adjustment, is a standard FID whose first point represents the NMR signal as it was at the digitizer immediately following the acquisition delay  $alfa$  or  $ad$ .

The *global* PSG variable  $ddrtc$  is used to adjust the first-order phase correction for all spectral widths of 2.5 MHz and smaller. The value of  $ddrtc$  is automatically set from the parameter  $ddrtc$ , if it exists. If  $ddrtc$  does not exist, the default corresponds to  $ddrtc=0.0$ . With  $ddrtc=0.0$ , a single-pulse spectrum would require a first-order correction of roughly  $lp=360.0*(rof2+alfa)*sw$  degrees. If  $ddrtc=rof2+alfa$ , the correction would be roughly  $lp=0$ . A fine adjustment of  $ddrtc$ , to allow for finite-pulse bandwidth, can produce exactly  $lp=0$ . The value of  $ddrtc$  is set in 0.4  $\mu s$  increments. To achieve greater precision, fine-adjust  $rof2$  as well, which can be set in increments of 12.5 ns.

Use  $ddrtc='n'$  or set  $ddrtc=0.0$  to turn off time correction,

A standard macro  $setrc$  is available to set  $ddrtc$  and  $rof2$  exactly to produce  $lp=0.0$ . This macro is automatically called by most standard liquids protocols but it is not generally applicable to a new sequence. A writer of a new sequence should decide whether  $setrc$  is appropriate to be used with their sequence.

The local parameter  $ddrpm$ , if created, stands for the type of pulse sequence and controls the setting of  $ddrtc$  by the  $setrc$  macro.  $ddrpm$  can either be set to:

- "p" ("pulse" -  $ddrtc = rof2 + alfa + 2*pw/\Pi$ ),

- "e" ("echo" -  $\text{ddrtc} = \text{alfa}$  or
- "r" ("refocus" -  $\text{ddrtc} = \text{rof2} + \text{alfa}$ ).

These `ddrpm` values are present and set with all standard VnmrJ 3 pulse sequences.

The *global* PSG variable `ddrcr` and its parameter, if created, control the cutoff-width of the digital filter. The default corresponds to `ddrcr=75` and a brick-wall filter. Smaller values relax the cutoff and can improve the baseline in cases where there is a first-point error due to ring-down. A value of `ddrcr=7.0` provides a reasonable cutoff and often avoids first-point baseline roll.

### Setting the receiver phase

The statement `setreceiver(table)` is used to assign a quadrature receiver phase table, where `mult` is a table name (`t1` to `t63`) that is used to set the value the receiver phase `oph` scan-to-scan.

The statement `rcvrphase(mult)` is used to set the full  $360^\circ$  phase of the receiver in which `mult` is a real-time integer (`v1` to `v42`) or an integer table (`t1` to `t63`) whose values multiply a small-angle receiver stepsize. The stepsize is set with the statement `rcvrstepsize(value)` in which `value` is a *double* that sets the stepsize in degrees.

The total receiver phase is  $\text{oph} + \text{mult} * \text{value}$ . The `rcvrphase` statement can be used to set up non-quadrature or cogwheel phase cycling or it can be used to adjust the phase of the receiver to follow the magnetization from a frequency offset region of the sequence.

### Setting the receiver offset

A digital receiver offset can be set in Hertz with the *global* PSG variable `roff` and its associated parameter. The parameter `roff` can be arrayed. The variable `roff` is a *double* with a default of 0.0. A non-zero value applies that offset to any acquisition block that follows. As a *double*, `roff` cannot be changed scan-to-scan. A value of `roff` can be used to detect a small spectral width about a region of the spectrum other than the center.

### Programming explicit acquisition

It is necessary to explicitly program acquisition if you wish



to place pulse-sequence statements after the acquisition or if you wish to interleave acquisition of data points with pulse-sequence statements. The statements of [Table 32](#) are used to program acquisition explicitly.

**Table 32** Receiver programming statements

<code>acquire(points, dwell)</code>	Acquire points/2.0 real-imaginary pairs with a time spacing, dwell.
<code>endacq()</code>	End the acquisition of np/2.0 complex points.
<code>rcvroff()</code>	Set the receiver off, T/R switch to transmit and unblank the amplifier.
<code>rcvron()</code>	Set the receiver on, T/R switch to receive and blank the amplifier.
<code>startacq(delay)</code>	Initialize the digital receiver and perform <code>rcvron</code> during the time delay.

**The `rcvron` and `rcvroff` statements**

The `rcvron` statement is a compound statement, used to set the receiver gate on, the transmit/receive, T/R, gate to receive, and to blank the observe transmitter. This statement adds a delay to the pulse sequence determined by the PSG variable `rof3`. The default for `rof3` is 2.0  $\mu$ s. The variable `rof3` can be set to other values by defining the parameter `rof3`. The T/R switch and blanking are set at the start of the statement. The receiver is gated on after the delay `rof3`.

The purpose of the `rof3` delay is to ensure that there is always a delay before receiver turn-on, in case the preceding statement is a pulse with `rof2` or `rd` of 0.0. Without a delay, this condition can damage the mixer box in the Front End. The value of `rof3` can be set shorter in special cases when you can make sure that the preceding receiver delay will never be zero. An example of this case is when `rcvron` is used to start a windowed acquisition period.

The `rcvroff` statement adds no time to the sequence. It simultaneously gates the receiver off, the T/R switch to transmit, and unblanks the observe transmitter. The `rcvroff` statement also sets an internal gate to disable the automatic blanking at the end of `rgpulse`.

**The `startacq` statement**

The compound statement `startacq(delay)` prepares the digital receiver for acquisition during the time `delay`. This statement first executes an acquisition gate during a period of 50 ns. It then performs a `rcvron` statement with the delay `rof3`, followed by a delay of `delay-rof3` to allow the receiver to come fully on.

The value of `delay` must be at least  $50 \text{ ns} + \text{rof}3$ . If it is less, that delay will be executed any way. One should also add an additional 1.0 to 2.0  $\mu\text{s}$  to allow the receiver to come fully on. The statement `startacq(alfa)` duplicates the function of implicit acquisition and uses the PSG variable `alfa`, the *acquisition delay*, which is typically 4.0 to 6.0  $\mu\text{s}$ . An implicit `startacq(alfa)` is added if the statement is not found in the sequence.

The digital filter delay takes place *after* `alfa`, while the receiver is supplying points. This delay is ultimately corrected to zero by the digital filters. See the preceding discussion of `ddrtc`. `alfa` is always positive, and it is set just long enough to turn on the receiver fully.

The `ad` delay is replaced by `alfa` and it is set as a constant of 4.0 to 6.0  $\mu\text{s}$ , independent of `sw`. The statement `startacq(ad)` is used in some sequences. Here, the user variable `ad` is used in preference to `alfa`, but otherwise the function of `startacq(ad)` is identical to `startacq(alfa)`.

### The `acquire` and `sample` statements

The statement `acquire(points,dwell)` is used to produce  $\text{points}/2.0$  complex real/imaginary data-pairs with a time spacing of `dwell`. The default arguments, `acquire(np,1/sw)`, duplicate the function of implicit acquisition. The first argument `points` must be a *double* value, not an integer, even though it is set as a whole number.

The default `acquire` statement produces  $\text{np}/2.0$  complex values listed as alternating real/imaginary pairs, with a nominal total time  $\text{at}=\text{np}/(2.0*\text{sw})$  and a nominal time separation determined by  $1/\text{sw}$ . The `acquire` statement adds a time of  $(\text{points}*\text{dwell})/2.0$  to the sequence. During this time, the digital receiver collects and processes points, acquired at a rate of 80 MHz. These points are *downsampled* and digitally filtered to produce the requested `np` points with the dwell of  $1/\text{sw}$ .

For the system digital receiver, only the product  $\text{points}*\text{dwell}$  has an effect on pulse-sequence timing and the individual arguments have no relationship with `np` and  $1/\text{sw}$ .

The default acquisition `acquire(np,1/sw)` is recommended when the pulse-sequence compatibility is required.

The `sample(delay)` statement removes the redundancy of the arguments of `acquire`. The statement `sample(np/(2.0*sw))`

has a function identical to `acquire(np,1/sw)`. The use of `sample` avoids the misleading impression of `acquire` that the user is programming the dwell time. If `np` and `sw` are not used to directly set `acquire` or `sample`, it is the user's responsibility to make sure that the programmed values of `points*dwell` or `delay` are large enough to collect any required `np` points with the desired `sw`. If the time is too short, the pulse sequence will fail. There are no ill effects if the time is too long.

### The `endacq` statement

The `endacq` statement is placed after the last `sample` or `acquire` statement. This statement gates the receiver off and concludes sampling. The `endacq` statement adds no time to the sequence itself, but an overhead delay will take place in the next delay. An implicit `endacq()` statement is added if it is not present in the sequence.

## Windowed acquisition

A *windowed* acquisition is an explicitly-programmed acquisition for which multiple `acquire` or `sample` statements are separated by delays. Usually the delays contain refocusing pulses or multiple-pulses for homonuclear decoupling. Acquisition occurs in the windows between the pulses. The most common uses are solids multipulse experiments or Carr-Purcell Meiboom Gill, (CPMG) experiments. Windowed acquisition is also used for Echo-Planar Imaging (EPI) experiments and other imaging experiments.

A windowed acquisition can be performed in one of two modes, *constant-sampling* mode and *explicit-sampling mode*. For constant-sampling mode, the acquisition period begins with the first instance of `acquire` or `sample` and continues until the end of the last `acquire`. During the intervening delays, the receiver writes zeros. For explicit-sampling mode, the receiver discards the zeros and compresses the `acquire` statements into a single block.

Constant-sampling is the default. You set explicit sampling with the statement `setacqmode(WACQ|NZ)` at some position in the sequence before the beginning of acquisition.

The statements below show an example of a windowed acquisition following a standard pulse. The windows are separated by a delay `tdly`.

```
rgpulse(pw,t1,0.0,0.0);
obsblank();
delay(rof2);
startacq(alfa);
loop(v1,v2);    //v1 initialized with "nloops"
    acquire(2.0,tau);
    delay(tdly);
endloop(v2);
endacq()
```

The acquisition is initialized with `startacq`. The first data point is collected at time `rof2+alfa` after the pulse. Data collection occurs repeatedly during the windows of time `tau`, separated by the time `tdly`, for each element of the loop. The `endacq()` statement concludes sampling.

### Constant-sampling mode

For constant-sampling mode, the receiver stores data points during `tau` and stores zeros during `tdly`. The total time for acquisition is `nloops*(tau+tdly)`.

The expression `nloops = (double)((int)((np*(tau+tdly)/(2.0*sw))))` ensures that the acquisition period `at` will accommodate any values of `np` and `sw`. The truncation to an integer and then back to a *double* type ensures that `nloops` is a whole number and that the time for `nloops` will always be less than the required acquisition time `at`.

If the sample for this acquisition were a narrow-line proton spectrum, for example of water, the constant sampling mode would chop the FID with periods of zeros. For a large spectral width, such as 5 MHz, you will observe the chopped FID. The Fourier transform of a FID with a large spectral width is a center band surrounded by replicas each separated by the chopping frequency.

For a small spectral width, less than the chopping frequency, you would observe a normal proton spectrum. The brick-wall filter ensures that the replicas do not fold into the spectrum. For a spectral width of  $sw = \sim 1.0 / (\tau + tdly)$  the nominal dwell is equal to the chopping frequency. For this case the digital receiver produces one point per cycle, nominal "single point acquisition". This case is the usual use

of the constant-sampling mode.

Constant-sampling mode preserves the natural  $T_2$  of the water-line and chemical shifts remain unscaled. The signal to noise for windowed sampling is lower than that of a fully sampled spectrum where:

$$S/N = S/N_{100\%} * \sqrt{\tau/(\tau+tdly)}$$

The advantage of constant sampling mode is the simplicity of processing. Constant-sampling is now used for all solids multipulse and CPMG experiments that are supplied by Varian.

### Explicit-sampling mode

For explicit-sampling mode, the receiver discards the zeros during `tdly` and compresses the periods of acquisition into one block. For explicit sampling mode, the total time of acquisition is `nloops =`

```
(double)((int)((np*tau/(2.0*sw)))).
```

The explicit-mode sequence would need to begin with `setacqmode(WACQ|NZ)`.

For explicit sampling, the data is a piecewise collection of blocks from the FID in which the time between the centers of two adjacent blocks is `tau`. If the percentage sampling is low, as it would be in a multipulse experiment, the blocks can be considered to be single "points" and so the natural spectral width for this data is `sw = 1.0/tau`. The Fourier transformation of this data would be a scaled spectrum with a scaling factor of `tau/(tau+tdly)`. Both chemical shifts and linewidths are scaled.

Explicitly-sampled data is the default for Unity-series spectrometers. This type of acquisition was made necessary by the fact that the receiver could not digitize points (or zeros) between the acquire periods. Unity-series, multipulse sequences employed the statement `acquire(2.0,2.0e-7)` and required that `sw=5.0e6`, using the 5.0 MHz digitizer option. One then corrected the scale factor with the parameter `scalesw`. This approach can also be used with VNMRS (`tau=2.0e-7`) to produce a Unity compatible mutipulse sequence, but the approach is not recommended.

Imaging sequences that use the explicit-sampling method are described in the Imaging User Guide.

### Constant-sampling with pulses

When pulses are present in a windowed acquisition loop, you must add `rcvroff` and `rcvron` statements to surround the pulses. The following example adds a single pulse between each window, using the standard acquisition, receiver, and unblanking delays.

```
rgpulse(pw,t1,0.0,0.0);
obsblank();
delay(rof2);
startacq(alfa);
loop(v1,v2);          //v1 initialized with "nloops"
    acquire(2.0,tau);
    rcvroff();
    delay(rof1);
    rgpulse(pw,t1,0.0,0.0);
    obsblank();
    delay(rof2);
    rcvron();
    delay(alfa-rof3);
endloop(v2);
endacq()
```

Here, a `rcvroff` follows each `acquire` statement, with a time `rof1` to fully unblank the transmitter. An explicit `obsblank` statement follows the pulse immediately (though the `rcvron` statement will eventually supply one) and a standard `rcvron` follows the receiver delay `rof2`. After a time `alfa-rof3`, the receiver is ready for the `acquire` statement of the next loop element. You should remember that `rcvron` has a time delay `rof3`. For this example, `rof3` is subtracted from `alfa` for clarity. As with the previous example, you must determine `nloops` from `np` and `sw` for the chosen sampling mode.

### Acquisition with multiple FIDS

Multiple acquisition periods can be programmed within a single scan. The procedure is referred to as *compressed* acquisition. Place `startacq()` - `endacq()` pairs around each acquisition and allow 200  $\mu$ s between acquisition periods.

The PSG variable `nf` and its parameter determine the number of FIDS to be expected from a single scan. Each FID should have `np` points. The result is known as a *compressed array*. The parameter `cf` selects the required FID for

processing. For example, if a pulse sequence contains 4 acquisitions, `nf=4`, and enter `cf=2` to process the second acquisition.

Multiple acquisition periods can be used with both compressed arrays and multiple receivers. Select the appropriate value of `cf` to obtain the array for the required acquisition. The value of `nf` is set to the product of the number of explicitly acquired array elements and the number of receivers that are used.



## Multidimensional NMR and Arrays

The `pulsesequence` function is compiled with C-looping software to automatically perform multidimensional ( $nD$ ) experiments up to 4 dimensions in which the three *indirect* dimensions involve looping. In addition, any parameter can be assigned to an array of values (it can be "arrayed") and the arrays can be run in nested or simultaneous loops. An arbitrary number of arrayed parameters can be nested in loops.

When arrays and  $nD$  spectra are run at the same time, by default all parameter arrays are in the inner loops followed next by the three indirect dimensions, for 2D, 3D, and 4D spectra. One exception to this rule occurs for imaging experiments if the first indirect dimension is a compressed acquisition. In the case the compressed acquisition does not require looping.

The nesting of loops involving arrayed parameters is determined by the PSG *global* variable `array` and its associated parameter. The parameter `array` is a string containing the names of arrayed parameters punctuated by either commas ',' or nested parentheses '( )'. Parameter names enclosed in parentheses are incremented simultaneously in a single loop. Parameters or blocks of parentheses separated by commas are nested from right to left, with the innermost loop on the right.

The order of  $nD$  acquisition can be changed by explicitly arraying the parameters `d2`, `d3` and `d4`. A parameter can be simultaneously arrayed with an indirect dimension by including it in parentheses with one of these parameters.

### Variables for $nD$ dimensions

[Table 33](#) summarizes the *global* PSG variables associated with individual  $nD$  arrays.

**Table 33** Multidimensional PSG variables

PSG variable	PSG type	VnmrJ parameter	Description
<code>d2_index</code>	int	0 to $(ni-1)$	Current index of the <code>d2</code> array
<code>id2</code>	real-time	0 to $(ni-1)$	Current real-time index of the <code>d2</code> array
<code>inc2D</code>	double	$1.0/sw1$	Dwell time for the first indirect dimension

**Table 33** Multidimensional PSG variables (continued)

PSG variable	PSG type	VnmrJ parameter	Description
d3_index	int	0 to (ni2-1)	Current index of the d3 array
id2	real-time	0 to (ni2-1)	Current real-time index of the d3 array
inc2D	double	1.0/sw2	Dwell time for the second indirect dimension
phase2	int	phase2	Aquisition mode for the second indirect dimension
d4_index	int	0 to (ni3-1)	Current index of the d4 array
id4	real-time	0 to (ni3-1)	Current real-time index of the d4 array
inc4D	double	1.0/sw3	Dwell time for the third indirect dimension
phase3	int	phase2	Aquisition mode for the third indirect dimension
ix	int	1 to arraydim	Current element of an arrayed experiment

The *global* variable `arraydim` is initialized from its parameter with the total elements in all parameter and nD arrays. The *global* variable `ix` is an index from 1 to `arraydim`.

The successive three indirect nD dimensions each have total elements determined from `ni`, `ni2` and `ni3`. The global PSG variables `d2_index`, `d3_index` and `d4_index` are indexes to these three dimensions.

In addition the three indexes are used to initialize the real-time variables `id2`, `id3` and `id4`. These real-time indexes are available for real-time math and are used in real-time loops and conditionals.

The *global double* variables `d2`, `d3`, and `d4` provide standard delays for their associated indirect dimensions. These variables are initialized from their parameter values and then incremented by the dwell times `inc2D`, `inc3D`, and `inc4D`. Each dwell time is calculated from its indirect spectral width `sw1`, `sw2`, and `sw3` and rounded to 12.5 ns timing resolution. The allowed indirect spectral widths are only those that yield properly rounded dwell times. You can always calculate the dwell exactly from the inverse of the spectral width.

For many experiments, the indirect delays will contain continuous decoupling patterns or spinlocks. It has been the practice for most spectroscopic experiments to deliver these

patterns using programmed waveforms or spinlocks. The indirect delays `d2`, `d3` or `d4` are used directly to supply the length of a `spinlock` pulse or to supply an explicit delay statement between `#prgon` and `#prgoff` (`# = obs, dec...etc`). When using these spinlocks or waveforms, you should synchronize the cycle time of the pattern with the indirect dwell time. This synchronization can be accomplished most reliably by calculation of the `.DEC` pattern file within the pulse sequence.

The `loop-endloop` statements are also used to create synchronized spinlocks during the indirect delays. In this case, the real-time values of `id2`, `id3`, and `id4` are available as loop counts.

## Compressed arrays and nD dimensions

It is sometimes possible to obtain all the data for an arrayed experiment or an indirect dimension in a single increment through use of multiple periods of acquisition. For example, the Carr-Purcell-Meiboom-Gill, CPMG experiment can be used to obtain multiple echos from a single excitation and these echos can be summed to improve signal to noise. Imaging experiments commonly use compressed phase-encode and multislice dimensions.

For a compressed acquisition, the *global* PSG variable `nf` and its associated parameter should be set to the total number of acquisition periods. In this case, the total number of points of detected data is `np*nf`. The *Command and Parameter Reference* describes commands that are used to parse the data to obtain the original arrayed FIDs.

## Switchable loops for nD imaging experiments

The `msloop-endmsloop` and the `peloop#-endpeloop` (`# = 1, 2 or 3`) statements allow you to control the execution of pulse-sequence code in real-time loops in multidimensional spectra and arrays. Arguments `apv1` and `apv2` should be real-time variables to contain the loop count and the loop index, respectively. These loops have two states, 'c' for *compressed* mode, and 's' for *standard* mode, which are set by the first argument `state`. The argument `max_count` is a *double* that is used to set the value of the loop count `apv2`. The `state` argument controls how `max_count` is used.

For the `msloop` statement, if `state='c'`, then `max_count` is used to initialize the real-time value of `apv1` and a real-time loop is executed `max_count` times. If `state='s'`, then `apv1`

is assigned as one and the contents of the loop are executed only once. In general, `msloop` can be used to either set the real-time loop count from a *double* or to force it to be executed once, depending on `state`.

The statements `peloop` and `peloop2` are similar to `msloop` in compressed mode. If `state='s'`, then these statements initialize `apv2` from the index of the appropriate nD loop, `d2_index` for `peloop`, or `d3_index` for `peloop2`. In general, in standard mode, these statements can be used to set up an internal loop over a group of statements within an indirect dwell time. This construction can be useful, for example, if an indirect dimension is a looped multipulse sequence, rather than one of the delays `d2`, `d3`, or `d4`, though a similar result can be obtained with `loop-endloop` by the use `id2`, `id3`, and `id4`.

The switchable loops are used primarily for imaging experiments to optionally compress nD *multislice* or *phase-encode* dimensions into a single increment. In this case, an acquisition period is nested in the innermost loop. In compressed mode, all of the indirect data are combined into a single FID. The value of `nf` must be set correctly so that the data can be uncompressed for processing.

Imaging experiments also use the *global* PSG variable `seqcon` and its associated parameter to control nD array dimensions. The `seqcon` variable is a string of five characters that are used to setup the nD array structure of imaging experiments. Allowed characters are 'c', 's' and 'n' for compressed, standard, and `noloop`. The placeholders refer to echo, multislice, and 3 phase-encode dimensions, 1,2, and 3 respectively. When the multislice `seqcon[1]` and phase-encode characters `seqcon[N]`,  $N=2,3,4$  are set to 'c', imaging dimensions are set up with a single increment. For 's', a standard nD dimension is created. For imaging experiments, `state` is set with the appropriate character of `seqcon`. In general, users of switchable loops must set `ni`, `ni2`, and `ni3` with the appropriate values to correspond to their use of `state`.

The statement `loop_check` compares the total number of compressed FIDs with the value of `nf` and aborts if they are not equal.

**Table 34** Switchable loop statements

<code>endmsloop(apv2)</code>	End multislice loop.
<code>endpeloop(apv2)</code>	End any phase-encode loop.

**Table 34** Switchable loop statements (continued)

<code>loop_check()</code>	Check <code>nf</code> for compressed dimensions.
<code>msloop(state,max_count,apv1,apv2)</code>	Start multislice loop.
<code>peloop(state,max_count,apv1,apv2)</code>	Start phase-encode loop for first indirect dimension.
<code>peloop2(state,max_count,apv1,apv2)</code>	Start phase-encode loop for second indirect dimension.

## Arrayed data acquisition

The `array` parameter indicates which parameters are arrayed. For NUS data collection, the sparsely sampled indirect dimensions are treated as "diagonal" arrays. For example, for a 3D data set with `array='phase,phase2'` and both the `ni` and `ni2` dimensions sparsely sampled, the `array` parameter is treated by PSG as `array='(d2,d3),phase,phase2'`

The following functions can be used to obtain information about the looping elements specified by the `array` parameter and any implicit arrays.

**Table 35** `array` parameter functions and PSG variables

<code>int numLoops()</code>	Returns the number of looping elements. In the above case, it would return 3. Diagonal arrays are treated as a single looping element.															
<code>int varsInLoop(int index)</code>	Returns the number of parameters in the specified loop. For non-diagonal arrays, it always returns a 1. For diagonal arrays, it returns the number of parameters that are jointly arrayed. Note that the <code>index</code> goes from 0 to <code>numLoops-1</code> . In the case above, the return values would be <table border="1" data-bbox="812 1386 1055 1501"> <thead> <tr> <th>index</th> <th>varsInLoop</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>2</td> </tr> <tr> <td>1</td> <td>1</td> </tr> <tr> <td>2</td> <td>1</td> </tr> </tbody> </table>	index	varsInLoop	0	2	1	1	2	1							
index	varsInLoop															
0	2															
1	1															
2	1															
<code>char *varNameInLoop(int index, int index2)</code>	Returns the names of the parameters in the specified loop. The second <code>index</code> goes from 0 to <code>varsInLoop-1</code> . In the case above, the return values would be <table border="1" data-bbox="812 1617 1282 1764"> <thead> <tr> <th>index</th> <th>index2</th> <th>varNameInLoop</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>"d2"</td> </tr> <tr> <td>0</td> <td>1</td> <td>"d3"</td> </tr> <tr> <td>1</td> <td>0</td> <td>"phase"</td> </tr> <tr> <td>2</td> <td>0</td> <td>"phase2"</td> </tr> </tbody> </table>	index	index2	varNameInLoop	0	0	"d2"	0	1	"d3"	1	0	"phase"	2	0	"phase2"
index	index2	varNameInLoop														
0	0	"d2"														
0	1	"d3"														
1	0	"phase"														
2	0	"phase2"														

**Table 35** array parameter functions and PSG variables

<pre>int valuesInLoop(int index)</pre>	<p>Returns the number of elements in the specified loop. In the case above, the return values would be</p> <table border="1"> <thead> <tr> <th>index</th> <th>valuesInLoop</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>CSdensity*ni*ni2</td> </tr> <tr> <td>1</td> <td>array size of parameter phase</td> </tr> <tr> <td>2</td> <td>array size of parameter phase2</td> </tr> </tbody> </table>	index	valuesInLoop	0	CSdensity*ni*ni2	1	array size of parameter phase	2	array size of parameter phase2
index	valuesInLoop								
0	CSdensity*ni*ni2								
1	array size of parameter phase								
2	array size of parameter phase2								
<p><b>Additional PSG variables</b></p>									
<pre>int initializeSeq</pre>	<p>This parameter is set to 1 the very first time the <code>pulsesequence()</code> function is called. Otherwise, it is set to 0. Many sequences currently use the software idiom <code>if (ix == 1)</code></p> <p>One time initializations are often conditional upon this <code>if</code> statement. Depending on the sampling scheme, <code>ix</code> may not be equal to 1 on the first call to <code>pulsesequence()</code>. The <code>initializeSeq</code> parameter can always be used as an indicator that <code>pulsesequence()</code> is being called for the first time.</p> <p><code>d2_index</code>, <code>d3_index</code>, <code>d4_index</code> and their real-time counterparts <code>id2</code>, <code>id3</code>, and <code>id4</code> will reflect in increment of the indirect dimension. For example, if <code>ni=128</code> and <code>CSdensity=25</code>, there will be 32 distinct values of <code>d2</code>. The value of <code>d2_index</code> will reflect the current index from the sampling schedule. That is, it will not go from 0 to 31 but rather will take on 32 values randomly selected between 0 and 127, as per the sampling schedule.</p> <p>Note that many sequences use the idiom</p> <pre>if (ix == 1) d2_init = d2; t1_counter = (int) ((d2 - d2_init) * sw1 + 0.5);</pre> <p>The <code>d2_index</code> will make this obsolete.</p>								

## Managing arrays

There are no *global* PSG variables (similar to `d2_index` and `ni`) to supply the indexes and limits of array dimensions. However, the size of any array is available to a pulse

sequence as an integer return from the `getarray` statement. It is straightforward to parse the array variable to get the array loop structure and calculate the indexes from the array dimensions and `ix` index.

## Multidimensional phases

VnmrJ contains general Fourier transform commands `ft2d` or `wft2d` that allow the user to customize the addition of arrays of nD spectra for various purposes. These methods are fully described in the *Spectroscopy User Guide*. They require you to manipulate the phase tables in arrays.

It is a standard practice to obtain pure-phase quadrature detection for indirect dimensions by combining arrays of nD spectra, obtained with different phase tables, also called *Hypercomplex Fourier Transform*, or to cycle the phases of individual pulses from increment to increment, also called *time-proportional phase incrementation* or TPPI. You can also use this approach to manage artifactual peaks, also called *FAD* or *States TPPI*.

The *global* PSG variables `phase1`, `phase2`, and `phase3` are used to index the phase tables of the three indirect dimensions. These variables are automatically initialized from the parameters `phase`, `phase2`, and `phase3`. Note that the variable `phase1` is initialized from the parameter `phase`.

The values of the phase parameters are indexes 0,1,2,3, etc. that are associated with the initialization of particular phase tables in the pulse sequence. By convention, `phase=0` is used alone to represent an absolute value phase cycle, `phase=1` and `phase=2` represent the cosine and sine components of the hypercomplex method or States TPPI, and `phase=3` is used to represent simple TPPI.

It is the responsibility of the user to add the correct code to initialize phase tables. Some examples for States TPPI (the most common method) follow.

### States TPPI using tables and *global* PSG indexes

This example uses phase tables and the States TPPI is obtained using a scalar addition to the appropriate tables, using a *global* PSG index. This particular example is a fairly recently programmed solids heteronuclear correlation.

```
//Add STATES TPPI ("States with "FAD")
  tsadd(phRec,2*d2_index,4);
  if (phase1==2) {
```

```

        tsadd(phHtilt3,2*d2_index+1,4);
    }
    else {
        tsadd(phHtilt3,2*d2_index,4);
    }
    setreceiver(phRec);

```

The array `phase=1,2` produces two 2D data sets. In each set, the 2D detection pulse and the receiver phase (`ph3Htilt` and `phRec` in this example) are toggled by  $180^\circ$  on alternate increments using a `tsadd` statement of `2*d2_index`, modulo 4 (FAD). For `phase=2`, the detection pulse is incremented by 1 using the expression `2*d2index+1`, modulo 4 (States) to convert a cosine table into a sine table.

The `setreceiver` statement produces the receiver phase `oph` from the phase table, `phRec`.

The approach above is called "States TPPI" because the detection pulse is cycling through four phases every two complex increments and "States" because two arrayed data sets are used to produce the 2D data. Typically, this method is called "States + FAD".

### States TPPI using real-time math

The second example illustrates States TPPI using real-time variables. Here, the phase tables are represented in successive values of real-time variables that were obtained through real-time math calculation on each increment (not shown). This example comes from a standard two-pulse homonuclear correlation experiment for liquids.

```

/* Add States with phase=2 */
    mod4(v1,oph);
    if (iphase==2) {
        incr(v1);
    }
/* Add FAD for phase=1 or phase=2 */
    if ((iphase==1) || (iphase==2)) {

initval(2.0*(double)((int)(d2*getval("sw1")+0.5)%2),v13);
        add(v1,v13,v1); add(oph,v13,oph);
    }

```

The array `phase=1,2` produces two 2D data sets. For `phase=2`, the prep pulse `v1` is incremented by 1 using `incr(v1)` (States) to convert a cosine table into a sine table. In both sets, the prep pulse and the receiver phase (`v1` and



oph in this example) are toggled by  $180^\circ$  on alternate increments using an add statement of v13 (0,2,0,2...) (FAD).

In this older sequence, v13 is assigned using a calculation of the number of dwell times from d2. An alternative that exists now is to use `mod2(id2,v13)`. Here, the receiver phase oph has been calculated directly, so `setreceiver` is not needed.

## Syntax for Controlling Parallel Channels

Each transmitter, receiver and gradient device is controlled by a separate parallel processor (a *controller*). At run-time the compiled C program assigns each controller an identity (a *channel*), and generates the appropriate codes for each. The identity may be one of the RF channels, `obs`, `dec`, `dec2`, etc for transmitters, one of the receivers or one or more streams for gradient output, all loosely referred to as *channels*. During acquisition the controllers run in parallel and are synchronized with a common 80 MHz clock, but otherwise there is no communication between controllers during acquisition. The C-code syntax of the pulse program must completely describe the timing relationship between the codes of each channel at run time.

The PSG provides four syntaxes to control timing of channels. These are

1. Synchronous Timing
2. A `parallelstart-parallelend` Section
3. The `nowait` Attribute
4. Waveform Control with `obsprgon-obsprgoff`

This section describes the four syntax methods and provides information about their use.

### Synchronous Timing Syntax

Any pulse sequence statement that executes a delay causes a synchronous delay to be executed on all other channels. For example an `rgpulse` statement executes three delays, `pw`, `RG1` and `RG2` along with statements to control the transmitter gate, the phase and blanking (See figure 1). In synchronous mode the `rgpulse` statement also executes a delay with a value `time = pw + RG1 + RG2` on every other operational controller, including the other RF channels, the receivers and the gradients. Synchronous syntax assures that the next statement occurs after the `rgpulse` on any channel. Synchronous syntax is the default.

### A `parallelstart - parallelend` Section

The statement `parallelstart(chnl)` begins a parallel section on the designated channel where `chnl` can be `"obs"`, `"dec"`, `"dec2"`, etc or `"rcvr"` or `"grad"`. For example

`parallelstart("obs")` designates the `obs` channel as a parallel section. With the start of a parallel section, only statements for that channel are allowed. Statements that cause a delay do *not* place a synchronous delay on other channels.

**Table 36** Statements to create parallel sections

<code>parallelend()</code>	End parallel section of pulse sequence
<code>parallelstart(chnl)</code>	Start parallel section of pulse sequence
<code>parallelsync()</code>	Position parallel synchronization delays

The most important statements that are allowed for `obs` are `rgpulse`, `obspower`, `obsoffset`, `obspwrf`, `obsblank`, `obsunblank`, `txphase`, `xmtrphase`, `obsstepsize`, `obsprgon` and `obsprgoff`. For the `obs` parallel section the `delay` and `hsdelay` statements are executed only on `obs`. Consult the section "Pulse-Sequence Statements" in this chapter for more information about statements for the `obs` channel.

A second `parallelstart` statement begins a section for a new channel, for example `parallelstart("dec")` begins a section for `dec`. Only statements for `dec` are allowed. Statements that add time add it to `dec`, beginning synchronously with the time of `parallelstart("obs")`. One may start parallel sections for as many channels as required. One must create sections for at least two channels and for any channel that has activity during the section.

All of the parallel sections are ended by single `parallelend` statement. The `parallelend` statement places synchronization delays in each section so that its duration will match the duration of the section with the longest duration. The synchronization delay of the longest section is 0.0. The total delay is also applied synchronously to channels that were not included as parallel sections.

It is important to note that the `parallelend` statement is executed at run time, before the acquisition begins. This restriction precludes the use of real-time statements such as `loop-endloop`, `ifzero-elsenz-endif` and `vdelay` in parallel sections. Real-time statements might change the total delay of a parallel section after the synchronization delay has been calculated. For this reason they are forbidden. Special *fixed* loops are available for use in a parallel section. See the discussion of "Looping in Parallel Sections" below.

Parallel sections have asynchronous syntax. There is no mandatory timing-syntax that associates statements in two different parallel sections. It is good practice to arrange the delays of parallel sections to sum to the same value. With this practice one does not have to depend upon `parallelend` to synchronize the sections. However that practice is not mandatory.

Asynchronous syntax frees the programmer from unnecessary C coding. One can more readily program custom *simultaneous pulses* where, for example, pulses of two channels "pass through each other". Another example is the opportunity to use *incommensurate loops* (with different cycle times) on two channels. For example one might program a spinlock or decoupler pattern on one channel with a loop and a pattern of pulses on the other. While spinlocks can also be executed with waveforms (for example `obsprgon-obsprgoff`, or `obsspinlock`, loops often have much less programming overhead than waveforms.

A `parallelstart-parallelend` construction can be placed multiple times, anywhere in a sequence. It can be placed within another loop, real-time or fixed, or within a real-time conditional. One is free to construct the entire sequence as a set of parallel sections.

```
parallelstart("obs");
    delay(d2_max/2.0);
    rgpulse(pw, oph, 0.0, 0.0);
    delay(d2_max/2.0);
parallelstart("dec");
    delay(d2);
    decrgpulse(pw, oph, 0.0, 0.0);
parallelstart("dec2");
    delay(d2_max-d2);
    dec2pulse(pw, oph);
parallelend();
```

In the example a pulse on `obs` is centered in a delay `d2_max`. As the delay `d2` increases from 0.0 to `d2_max` the pulse on `dec` moves from the beginning to the end of the `d2_max` delay. The pulse on `dec2` moves from the end of the delay to the beginning. To obtain this result with synchronous syntax would require complicated conditionals.

The `parallelsync` statement is used to specify when the synchronization delay is executed. For example, if `parallelsync` is placed at the beginning of the section, the synchronization delay is executed at the beginning of the section. The `parallelsync` statement can occur anywhere in a section and can occur only once. If no `parallelsync` statement is present the synchronization delay is executed at the end of the section. Even though it is redundant, some programmers may still wish to include `parallelsync` at the end of parallel sections for clarity.

The following two uses of `parallelsync` give the same result.

```
parallelstart("obs");
    delay(d2_max);
parallelstart("dec")
    delay(d2_max-d2-pw);
    decrgpulse(pw, oph, 0.0, 0.0);
parallelend();
```

```
parallelstart("obs");
    delay(d2_max);
parallelstart("dec")
    parallelsync();
    decrgpulse(pw, oph, 0.0, 0.0);
    delay(d2);
parallelend();
```

The amount of time in each parallel section must be equal. However, there is also a minimum time event for the system. If the time difference between parallel sections is nonzero but less than the minimum time event, the minimum time event is added to the `parallelsync` delays on every channel. One cause of timing errors may be due to round-off errors, often caused by correcting the phase during a pulse by multiplying the pulse by  $2.0/\text{PI}$ . Two pulse elements help avoid these round-off problems.

```
calcDelay(delay)
pwCorr(delay)
```

Each of these return the delay, rounded correctly to the minimum time event. For example:

```
delay( tauhx - gt5 -gstab - pwC +
calcDelay(2.0*pwC/PI) );
```

The second pulse element is `pwCorr(delay)`, which does the multiplication by  $2/\text{PI}$  and rounds the result correctly. The above delay could be rewritten as

```
delay( tauhx - gt5 -gstab - pwC + pwCorr(pwC) );
```

## Looping in Parallel Sections

Special loop statements are provided for use in parallel sections, the `rlloop-rlendloop` and `kzloop-kzendloop` statements. These loops are *fixed* loops. The fixed loops operate in a manner similar to the `loop-endloop` statements, except that the number of repetitions is known at run time.

For the statement `rlloop(count,vcount,index)` the first argument `count` is an *int* containing the number of repetitions. The second argument `vcount` is the real-time variable, initialized with `count`. The third argument `index` is a real-time index of the loop. One should not change `vcount` or `index` with real-time math. In this case the pulse sequence may not operate or it may operate in an unpredictable manner. If `count` is 0 the loop will not be executed.

```
double cycletime = getval("cycletime");
int count = 4;
rlloop(count,v1,v2;)
    delay (cycletime);
rlendloop(v2);
```

In the example of `rlloop-rlendloop` the C integer `count` sets four repetitions of the delay `cycletime`. The real-time variable `v1` is initialized with a value of 4. Neither `v1` or `v2` should be changed with real-time math.

```
double cycletime = getval("cycletime");
double count = 4;
initval(count,v1);
loop(v1,v2;)
    delay (cycletime);
```

```
endloop(v2);
```

For synchronous syntax this use of loop-endloop is equivalent to the use of rllloop-rlendloop above, except that the value of v1 can be changed scan to scan.

The statement kzloop(time,vcount,index) is similar to rllloop with the exception that one specifies delay, the total duration of the loop rather than the count. The loop count is calculated from delay. The loop ends before the total duration and a delay will be added after the loop to complete the duration. If the duration is insufficient to execute one repetition only a delay will be executed.

In this example of kzloop-kzendloop, the loop count is calculated as  $\text{trunc}(14.0/3.0) = 4$  repetitions of 3 seconds. The remaining duration,  $14.0 - 4*3.0 = 2.0$  seconds will be added as a delay after the repetitions.

```
kzloop(14.0,v2,v11);
    delay(3.0);
kzendloop(v11);
```

For the statements rlendloop(index) and kzendloop(index) the v-variable argument index should be the same as the third argument of rllloop and kzloop. One should use separate v-variables for vcount in multiple instances of rllloop and kzloop.

**Table 37** Fixed loops for parallel sections

kzendloop(index)	End real-time loop with fixed duration
kzloop(time,vcount,index)	Start real-time loop with fixed duration
rlendloop(index)	End real-time loop with fixed count
rllloop(count,vcount,index)	Start real-time loop with fixed count

When placed in a parallel section the delays associated with statements in the loop will be added only to the designated channel. The fixed loops can also be used with synchronous syntax. For synchronous syntax the total delay of the loop is duplicated on the other channels.

The restrictions associated with the fixed loops are of little consequence in real pulse sequences. The cases of concern are those where the count v-variable of loop might be

changed with real-time math scan-to-scan, an unlikely circumstance. That construction is not possible with the fixed loops, but it is possible using `loop-endloop` with synchronous timing.

It is possible to derive delay of `kzloop` from the variable `d2`, the F1 delay or to derive count of `rlloop` from `d2_index`, the *int* F1 index. The same is true for the other dimensions. This use of fixed loops is analogous to the use of the v-variable `id2` as the count v-variable of `loop`, which can only be done with synchronous syntax. This kind of looping provides a method to generate a spinlock or decoupling pattern during an indirect dimension.

## Allowed Statements and Calculation in a Parallel Section

Any C-code is allowed in a parallel section, including those C statements such as `if-else`, `for` and `while` which hold pulse-sequence statements. All C-code is executed at run-time and so it presents no problem for synchronization at the `parallelend` statement. All other pulse sequence statements can be used in parallel sections unless they are designated real-time or unless they provide `acodes` on a channel other than that designated by `parallelstart`.

For most sequences the C `if-else` statement provides a fixed conditional and it is a reasonable replacement for `ifzero-elsnz-endif` statement. The C conditional is resolved at run time and so does not affect synchronization. Use of the C conditional prevents conditional choices that occur scan-to-scan but not increment-to-increment. If the former is required, for example a pulse width that changes scan-to-scan, one must use synchronous syntax.

Real-time math statements and other statements that set v-variables or tables are allowed as long as they do not change the v-variables used as `vcount` and `index` for fixed loops.

The existing simultaneous-pulse-sequence statements such a `simpulse` cannot be used in a parallel section because they provide `acodes` for multiple channels. However, parallel sections are designed for the creation of custom, more flexible simultaneous pulses to replace the standard simultaneous pulses that are supplied.

The single-channel shaped-pulse statement, the `spinlock` statements, `obsspinlock`, `decspinlock`, etc and the waveform statements such as `obsprgon-obsprgoff` are allowed. The pairs of statements `obsprgon` and `obsprgoff`



can have any relationship to a set of parallel sections, both-outside, both-inside or only one-inside and one-outside. The only requirements are that `obsprgoff` must exist somewhere in the sequence following `obsprgon` and that the channel designated by the statement agree with that designated for the section in which it resides.

The `xgate` and `rotorsync` commands are allowed in parallel sections with a strong caveat. The `xgate` command stops the output of all channels synchronously and resumes them when an external trigger arrives. That means that one must pay attention to the state of the other channels when `xgate` occurs. For example, one can program an `xgate` in one section that splits a pulse in another. This is a legal operation and it is up to the programmer to decide whether it should occur. Despite the fact that `xgate` occurs in real-time it adds the same delay to all channels, so it does not invalidate the synchronization of `parallelend`. The `rotorsync` statement is constructed from `xgate` statements and so has the same properties.

## Programming the Receiver and Gradients

The statement `parallelstart("rcvr")` creates a parallel section for the receiver. If programming a parallel receiver, the statements `startacq` and `endacq` should precede and follow the parallel sections. With (VnmrJ3.2) support of multiple receivers is not available. The reason for this is that the standard `startacq` and `endacq` pulse elements control gates on both the receiver and observe transmitter. A set of additional pulse elements separate these controls and can be used in parallel sections. Care must be taken so that the timing of events on the receiver are coordinated with the timing of events on the observe transmitter.

Statements for acquisitions in parallel sections.

```
startacq_obs(delay)
startacq_rcvr(delay)

acquire_obs(points,dwell)
acquire_rcvr(points,dwell)

endacq_obs()
endacq_rcvr()

parallelacquire_obs(delay,points,dwell)
parallelacquire_rcvr(delay,points,dwell)
```

These statements perform the appropriate actions on just

the observe transmitter or receiver.

The `parallelacquire_obs` and `parallelacquire_rcvr` pulse elements combine the operations of the corresponding `startacq_XXX`, `acquire_XXX`, and `endacq_XXX` statements. An example of use is:

```
parallelstart("obs");
  parallelacquire_obs(alfa,np,1.0/sw);

parallelstart("rcvr");
  parallelacquire_rcvr(alfa,np,1.0/sw);

parallelstart("dec");
  // provide a custom decoupling sequence here

parallelend();
```

If the time from the start of the parallel section to the start of the `startacq_XXX` statements do not match, an error will be given. If the time from the start of the parallel section to the start of the `endacq_XXX` statements do not match, an error will be given. If the time from the start of the parallel section to the start of the `parallelacquire_XXX` statements do not match, an error will be given. The `acquire_obs` does not control any gates on the observe transmitter and is provided to make it easy to synchronize the `endacq_XXX` elements. It really corresponds to a delay of duration  $0.5 * \text{points} * \text{dwell}$ . The statement `parallelstart("grad")` creates a parallel section for PFG applications of gradients. The statements `rgradient` and `zgradpulse` are allowed. With VnmrJ3.2 other gradients statements are not supported.

## The `nowait` Attribute

The `nowait` attribute is a property of gradient pulses that allows the parallel execution of non- gradient statements during a gradient pulse. Effectively, the `nowait` attribute creates a parallel section for the gradient pulse. Pulse-sequence statements following the gradient pulse are applied synchronously with the beginning of the gradient pulse. Synchronous timing occurs for all other channels. The gradient pulse ends after its designated duration, independently of the non-gradient channels. See the section "Gradient Control for PFG and Imaging" for more information about the `nowait` attribute.

## Waveform Control with `obsprgon-obsprgoff`

The waveform statement `obsprgon` also creates a parallel section for the `obs` channel through a mechanism very different from `parallelstart`. Waveform patterns are often the preferred method for decoupling or to create a spinlock on one or more channels simultaneously. Waveforms on multiple channels can be started and stopped synchronously on multiple channels with no pre-delays or post-delays. See the section "Shaped Pulses and Waveforms" in this chapter for more information about waveforms.

The `obsprgon` statement takes control of the output of the `obs` channel and places it under the control of a waveform pattern that is external to the pulse sequence. This pattern takes control of the amplitude, the phase and the gate of the `obs` channel and blocks the execution of any other pulse-sequence statements for this channel. The timing of the `obs` channel remains synchronous with all other channels. The waveform pattern can execute instructions that are asynchronous with other channels because these instructions are generated outside of the pulse sequence code.

The `obsprgoff` statement stops the waveform pattern and returns control to the pulse sequence. The timing of `obsprgoff` is synchronous with the pulse sequence, not the waveform, and so it might halt the waveform mid-cycle or even mid-element.

The ability to halt the waveform mid-cycle is a benefit when a waveform statement is used during an indirect delay. The cycle time of the decoupling need not be synchronous with the end of the indirect delay. A loop in a parallel section cannot be halted mid-cycle. The parallel section expands to accommodate the total duration of the loop, a value which may be inconsistent with the duration of the indirect delay.

The statements `decprgon-decprgoff`, `dec2prgon-dec2prgoff`, `dec3prgon-dec3prgoff` and `dec4prgon-dec4prgoff` have properties similar to `obsprgon-obsprgoff` for their respective channels.

Waveforms have access to the interpolation processor. They can potentially run with greater efficiency than loops and generate states faster in a sustained manner. See the discussion of interpolation and `userDECshape` in this chapter. When programmed properly, a waveform can generate new states in a sustained manner with a 25 ns or 50 ns step size. For a loop the minimum step size that can

be sustained is approximately 200-400 ns per step.

A waveform can be programmed to generate a phase-ramped, frequency offset, an essential element in many sequences. While it is possible to generate a phase ramp with a loop (see the section "Setting the Amplitude phase and Gate from Tables" in this chapter), the programming is also difficult and execution is less efficient than that of a waveform.

Waveform patterns are supplied from a source external to the pulse program, usually a .DEC file in the directory `~/vnmrsys/shapelib` or from a file passed directly from the run-time C program, as when `userDECshape` is used. Waveform patterns usually have a large programming overhead. They are often generated with an external program such as Pbox. With more recent sequences they are generated directly from calculation in the C program, which might be a call to Pbox with special pulse sequence elements, or through the use of included functions or an object library, as in `solidstandard.h` or SGL for imaging. Often the use of a loop has less programming overhead and is simpler to apply.

Waveforms are difficult to program for homonuclear decoupling because waveforms do not control of the amplifier blanking and T/R switch gate. While VnmrJ has a feature for automatic application of homonuclear waveforms, loops of shaped pulses are preferred to waveforms for user programming.

## Parameters and Variables

This section describes parameters and variables used in programming in VnmrJ.

### Categories of parameters and variables

At run-time, a compiled pulse sequence accesses a *parameter table* in the workspace in which it is to be run and uses those values to construct the real-time program of acodes that will run on the acquisition computer.

A set of standard parameters are automatically accessed to initialize a set of *global PSG variables*. The *global* PSG variables are declared `extern` in the PSG object library and they are automatically known to the `pulsesequences` function. The appendix lists the set *Global* PSG variables for spectroscopy and imaging, most of which are described in this manual. Parameters used to initialize these variables almost always have the same name as the variable. Parameter definitions are found in the *Command and Parameter Reference*.

A user may also define additional *user variables* in the `pulsesequences` function. These variables are local to the pulse-sequence function and they must be initialized for every increment of a multidimensional experiment or array. User variables should be explicitly defined as a C type. User variables are initialized from user parameters in the parameter table, usually of the same name. Table 35 shows the statements used to handle parameter values.

Parameters and their values are found in the file `vnmrsys/exp#/curpar` of the user in which `exp#` is the current workspace. A user must create the additional parameters they get from the `pulsesequences` function. If the parameters are not found, a warning will be delivered to the text window and a default value will be set. Chapter 5 of this manual *Creating and Modifying Parameters* describes the command-line statements that are used to create and control parameters.

The parameters of the parameter table are designated `current` and they must contain values to initialize all the *global* and user variables of the sequence. If the parameter table is empty, VnmrJ will fail. It is a good practice to initially choose a parameter set from a working sequence to ensure that all the standard parameters are present. Once it

has been modified for the sequence, save the parameter file as a `.par` file in the `vnmrsys/parlib` directory of the user. When transferring a sequence to another system, it is important to also to send the `.par` file.

### Statements used to handle parameters

The statements in Table 35 can be used to input or output parameter values to the pulse sequence. All input/output occurs at run-time before the acquisition of the first scan.

The command `go('check')` can be used to perform all run-time input/output without starting acquisition. The value of the *global* PSG integer `checkflag` is 1 if the 'check' argument is used. The value of `checkflag` can be used in a C conditional if input/output is required only with `go('check')`.

The value of the *global* integer `ix` is 1 (not 0!) on the first increment of any multidimensional or arrayed experiment. The value of `ix` can be used in a C conditional to reserve input/output to the first increment.

If a parameter name is not found, the variable will be given a default value, 0 for numerics and '' for strings. A message will be sent to the Text page of the Process tab of the VnmrJ interface.

**Table 38** Statements to handle parameter values

<code>pardim=getarray(parname, arrayname)</code>	Set a double array from an arrayed parameter.
<code>getstr(parname, varname)</code>	Obtain the value of a string parameter.
<code>varname=getval(parname)</code>	Obtain the value of a numeric parameter.
<code>putarray(parname, varname, pardim)</code>	Set an arrayed parameter from a double array.
<code>putCmd(command, varnames)</code>	Send a command string to the VnmrJ command line.
<code>putstring(parname, varname)</code>	Set a string parameter from a character array.
<code>putvalue(parname, varname)</code>	Set a numeric parameter from a double variable.

### Statements to load parameter values

The statement `getval` is used to obtain a numeric parameter value and return it to a *double* variable. A user `varname` must be defined and `parname` must be the parameter name placed in *double* quotes or a char pointer to the name. Some examples of the syntax are:

```
double mix;
mix = getval("mix");
```

or

```
double mix = getval("mix")
```

A `getval` statement can also be used as an argument to another statement, as shown in the following example:

```
delay(getval("mix"));
```

The statement `getstr` is used to obtain a string parameter `parname` and set a string variable `varname`. The variable `varname` must be defined as a C *char* array and `parname` must be the parameter name placed in double quotes or a char pointer to the name. An example is:

```
char[MAXSTR] myflag;
getstr("myflag",myflag);
```

The `getval` and `getstr` statements are executed with every increment. If a parameter is arrayed, the correct array element will be loaded.

The statement `getarray` is used to load all of the numeric values of an array at once into an array of *doubles*. This statement loads values that will be assigned to a real-time list.

The argument `arrayname` is the C array and `parname` must be the parameter name placed in double quotes or a char pointer to the name. An example is:

```
int myarraydim;
double myarray[256]; //myarraydim must be less than 256.
myarraydim = getarray("mylist",myarray);
```

Set protection bit 8(256) of the arrayed parameter `setprotect('myarray',256)` to suppress its loading with each array increment.

### Statements to output parameter values

The statement `putvalue` is used to return a numeric value to a parameter from the pulse sequence. The argument `varname` must be defined as a C *double* type and `parname` must be the parameter name in double quotes or a *char* pointer to that name. The `putvalue` statement might be used to return a value that has been altered by the pulse sequence.

```
double mix = getval("mix");
mix = (double) tau*((int) mix/tau + 0.5);
```

```
//round mix to multiples of tau  
putvalue("mix, mix); //update the parameter table
```

The statement `putstring` is used to return a string value to a parameter from the pulse sequence. The argument `varname` must be defined as a C array of characters and `parname` must be the parameter name in double quotes or a character pointer to that name.

```
getstr( "dm",dm); // originally dm='nnn'  
dm[2]='y';      // fix the decoupling on during  
                // acquisition  
putstring("dm",dm)// update the parameter table
```

The statement `putarray` is used to return all of the elements of a numeric array from the pulse sequence. The argument `varname` must be defined as a C array of *double* and `parname` must be the parameter name in double quotes or a char pointer to that name.

An example of the syntax is:

```
double mylist[6] = 1.0,2.0,3.0,4.0,5.0,6.0;  
putarray("list",mylist,6);
```

The numeric parameter `list` should exist in the parameter table.

The statement `putCmd` is used to execute an expression or a macro on the command line from the pulse sequence. The argument `command` must be a valid expression for the command line contained in double quotes or a pointer to that character array. One use of `putCmd` is to execute a more complicated parameter operation:

```
d1=1.0;  
putCmd("setvalue('d1',%g,'processed')",d1);
```

Here the macro `setvalue` is used to set the value of `d1` in the processed parameter tree.

The `putCMD` statement allows the use of format expressions, similar to the C `printf` command. The designation `varnames` refers to one or more variables to be formatted.



## PSG Variables

This section describes PSG variables used in programming VnmrJ.

### Global PSG Variables

The below table displays a list of Global PSG variables for spectroscopy.

**Table 39** Global PSG Variables

<b>Acquisition</b>			
<b>PSG Global Variable</b>	<b>C-Variable Type</b>	<b>PSG Variable</b>	<b>Description</b>
extern	char	il [MAXSTR]	interleaved acquisition parameter, 'y', 'n'
extern	double	inc2D	t1 dwell time in a 3D/4D experiment
extern	double	inc3D	t2 dwell time in a 3D/4D experiment
extern	double	sw	spectral width
extern	double	nf	Number of FIDs in pulse sequence
extern	double	np	Number of data points to acquire (real)
extern	double	nt	Number of transients
extern	double	sfrq	Observe frequency MHz
extern	double	dfrq	Decoupler frequency MHz
extern	double	dfrq2	2nd decoupler frequency MHz
extern	double	dfrq3	3rd decoupler frequency MHz
extern	double	dfrq4	4th decoupler frequency MHz
extern	double	fb	Filter bandwidth
extern	double	bs	Block size
extern	double	tof	Observe transmitter offset
extern	double	dof	Decoupler offset
extern	double	dof2	2nd decoupler offset
extern	double	dof3	3rd decoupler offset
extern	double	dof4	4th decoupler offset
extern	double	gain	Receiver gain value, or 'n' for autogain
extern	double	d1p	Decoupler low power value
extern	double	dhp	Decoupler low power value

## 2 Pulse-Sequence Programming

**Table 39** Global PSG Variables

extern	double	tpwr	Transmitter pulse power
extern	double	tpwrf	Transmitter fine linear attenuator for pulse
extern	double	dpwr	Decoupler pulse power
extern	double	dpwrf	Decoupler fine linear attenuator for pulse
extern	double	dpwrf2	2nd decoupler fine linear attenuator
extern	double	dpwrf3	3rd decoupler fine linear attenuator
extern	double	dpwrf4	4th decoupler fine linear attenuator
extern	double	dpwr2	2nd decoupler power course attenuator
extern	double	dpwr3	3rd decoupler power course attenuator
extern	double	dpwr4	4th decoupler power course attenuator
extern	double	filter	Pulse amp filter setting
extern	double	xmf	Observe transmitter pulse width
extern	double	dmf	Decoupler modulation frequency
extern	double	dmf2	Decoupler modulation frequency
		dmf3	
		dmf4	
extern	double	fb	Filter bandwidth
extern	double	vttemp	VT temperature setting
extern	double	vtwait	VT temperature time-out setting
extern	double	vtc	VT temperature cooling gas setting
extern	double	cpflag	Phase cycling; 1=no cycling, 0=quad detect
extern	double	dhpflag	Decoupler high power flag

### Pulse Widths

PSG Global Variable	C-Variable Type	PSG Variable	Description
extern	double	pw	Transmitter modulation frequency
extern	double	p1	A pulse width
extern	double	pw90	90° pulse width
extern	double	hst	Time homospoil is active

### Delays

PSG Global Variable	C-Variable Type	PSG Variable	Description
---------------------	-----------------	--------------	-------------

**Table 39** Global PSG Variables

extern	double	alfa	Time after receiver is turned on that acquisition begins
extern	double	beta	Audio filter time constant
extern	double	d1	Delay
extern	double	d2	An auto incremental delay, used in 2D experiments
extern	double	d3	An auto incremental delay, used in 3D experiments
extern	double	d4	An auto incremental delay, used in 4D experiments
extern	double	pad	Preacquisition delay
extern	double	padactive	Preacquisition delay active parameter flag
extern	double	rof1	Amplifier unblanking delay before pulse
extern	double	rof2	Amplifier blanking delay

**2D/3D/4D**

PSG Global Variable	C-Variable Type	PSG Variable	Description
extern	double	totaltime	Total timer events for an experiment duration estimate
extern	int	phase1	Used for 2D acquisition
extern	int	phase2	Used for 3D acquisition
extern	int	phase3	Used for 4D acquisition
extern	int	d2_index	d2 increment (from 0 to ni-1)
extern	int	d3_index	d3 increment (from 0 to ni2-1)
extern	int	d4_index	d4 increment (from 0 to ni3-1)

**Programmable Decoupling Sequences**

PSG Global Variable	C-Variable Type	PSG Variable	Description
extern	char	xseq [MAXSTR]	
extern	char	dseq [MAXSTR]	
extern	char	dseq2 [MAXSTR]	
extern	char	dseq3 [MAXSTR]	
extern	char	dseq4 [MAXSTR]	
extern	double	xres	Digit resolution prg dec
extern	double	dres	Digit resolution prg dec
extern	double	dres2	Digit resolution prg dec

**Table 39** Global PSG Variables

extern	double	dres3	Digit resolution prg dec
extern	double	dres4	Digit resolution prg dec
<b>Status Control</b>			
<b>PSG Global Variable</b>	<b>C-Variable Type</b>	<b>PSG Variable</b>	<b>Description</b>
extern	char	xm [MAXSTR]	Transmitter status control
extern	char	xmm [MAXSTR]	Transmitter modulation type control
extern	char	dm [MAXSTR]	1st decoupler status control
extern	char	dmm [MAXSTR]	1st decoupler modulation type control
extern	char	dm2 [MAXSTR]	2nd decoupler status control
extern	char	dmm2 [MAXSTR]	2nd decoupler modulation type control
extern	char	dm3 [MAXSTR]	3rd decoupler status control
extern	char	dmm3 [MAXSTR]	3rd decoupler modulation type control
extern	char	dm4 [MAXSTR]	4th decoupler status control
extern	char	dmm4 [MAXSTR]	4th decoupler modulation type control
extern	char	homo [MAXSTR]	1st decoupler homo mode control
extern	char	homo2 [MAXSTR]	2nd decoupler homo mode control
extern	char	homo3 [MAXSTR]	3rd decoupler homo mode control
extern	char	homo4 [MAXSTR]	4th decoupler homo mode control
extern	int	xmsize	Number of characters in xm
extern	int	xmmsize	Number of characters in xmm
extern	int	dmsize	Number of characters in dm
extern	int	dmmsize	Number of characters in dmm
extern	int	dm2size	Number of characters in dm2
extern	int	dmm2size	Number of characters in dmm2
extern	int	dm3size	Number of characters in dm3
extern	int	dmm3size	Number of characters in dmm3
extern	int	dm4size	Number of characters in dm4
extern	int	dmm4size	Number of characters in dmm4
extern	int	homosize	Number of characters in homo
extern	int	homo2size	Number of characters in homo2
extern	int	homo3size	Number of characters in homo3

**Table 39** Global PSG Variables

extern	int	homo4size	Number of characters in homo4
extern	int	hssize	Number of characters in hs

## Imaging and Other Variables

**Table 40** Imaging and Other Variables

RF Pulses			
extern	double	p2	Pulse length
extern	double	p3	Pulse length
extern	double	p4	Pulse length
extern	double	p5	Pulse length
extern	double	pi	Inversion pulse length
extern	double	psat	Saturation pulse length
extern	double	pmt	Magnetization transfer pulse length
extern	double	pwx	X-nucleus pulse length
extern	double	pwx2	X-nucleus pulse length
extern	double	ps1	Spin-lock pulse length
extern	char	pwpat [MAXSTR]	Pattern for pw, tpwr
extern	char	pw1pat [MAXSTR]	Pattern for p1, tpwr1
extern	char	pw2pat [MAXSTR]	Pattern for p2, tpwr2
extern	char	pw3pat [MAXSTR]	Pattern for pw3, tpwr3
extern	char	pw4pat [MAXSTR]	Pattern for pw4, tpwr4
extern	char	pw5pat [MAXSTR]	Pattern for pw5, tpwr5
extern	char	pipat [MAXSTR]	Pattern for pi, tpwri
extern	char	satpat [MAXSTR]	Pattern for pw, tpwr
extern	char	mtpat [MAXSTR]	Pattern for psat, satpat
extern	char	ps1pat [MAXSTR]	Pattern for spin-lock
extern	double	tpwr1	Transmitter pulse power
extern	double	tpwr2	Transmitter pulse power
extern	double	tpwr3	Transmitter pulse power
extern	double	tpwr4	Transmitter pulse power
extern	double	tpwr5	Transmitter pulse power

**Table 40** Imaging and Other Variables

extern	double	tpwri	Inversion pulse power
extern	double	satpwr	Saturation pulse power
extern	double	mtpwr	Magnetization transfer pulse power
extern	double	pwxlvl1	pw <sub>x</sub> pulse level
extern	double	pwxlvl2	pw <sub>x</sub> 2 power level
extern	double	tpwrs1	Spin-lock power level

### RF Decoupler Pulses

extern	char	decpat [MAXSTR]	Pattern for decoupler pulse
extern	char	decpat1 [MAXSTR]	Pattern for decoupler pulse
extern	char	decpat2 [MAXSTR]	Pattern for decoupler pulse
extern	char	decpat3 [MAXSTR]	Pattern for decoupler pulse
extern	char	decpat4 [MAXSTR]	Pattern for decoupler pulse
extern	char	decpat5 [MAXSTR]	Pattern for decoupler pulse
extern	char	dpwr1	Decoupler pulse power
extern	char	dpwr4	Decoupler pulse power
extern	char	dpwr5	Decoupler pulse power

### Gradients

extern	double	gro, gro2, gro3	Readout gradient strength
extern	double	gpe, gpe2, gpe3	Phase encode for 2D, 3D, and 4D
extern	double	gss, gss2, gss3	Slice-select gradients
extern	double	gror	Readout focus
extern	double	gssr	Slice-select refocus
extern	double	grof	Readout refocus fraction
extern	double	gssf	Slice-select refocus fraction
extern	double	g0, g1, ... g9	Numbered levels
extern	double	gx, gy, gz	X, Y, and Z levels
extern	double	gvox1, gvox2, gvox3	Voxel selection
extern	double	gdiff	Diffusion encode
extern	double	gflow	Flow encode
extern	double	gspoil, gspoil2	Spoiler gradient levels

extern	double	gcrush, gcrush2	Crusher gradient levels
extern	double	gtrim, gtrim2	Trim gradient levels
extern	double	gramp, gramp2	Ramp gradient levels
extern	double	gpemult	Shaped phase encode multiplier
extern	double	gradstepsz	Positive steps in the gradient DAC
extern	double	gradunit	Dimensional conversion factor
extern	double	gmax	Maximum gradient value (G/cm)
extern	double	gxmax	X maximum gradient value (G/cm)
extern	double	gymax	Y maximum gradient value (G/cm)
extern	double	gzmax	Z maximum gradient value (G/cm)
extern	double	gtotlimit	Limit combined gradient values (G/cm)
extern	double	gxlimit	Safety limit for X gradient (G/cm)
extern	double	gylimit	Safety limit for Y gradient (G/cm)
extern	double	gzlimit	Safety limit for Z gradient (G/cm)
extern	double	gxscale	X scaling factor for gmax
extern	double	gyscale	Y scaling factor for gmax
extern	double	gzscale	Z scaling factor for gmax
extern	char	gpatus [MAXSTR]	Gradient ramp-up pattern
extern	char	gpatusdown [MAXSTR]	Gradient ramp-down pattern
extern	char	gropat [MAXSTR]	Readout gradient pattern
extern	char	gpepat [MAXSTR]	Phase encode gradient pattern
extern	char	gsspat [MAXSTR]	Slice gradient pattern
extern	char	gpat [MAXSTR]	General gradient pattern
extern	char	gpat1 [MAXSTR]	General gradient pattern
extern	char	gpat2 [MAXSTR]	General gradient pattern
extern	char	gpat3 [MAXSTR]	General gradient pattern
extern	char	gpat4 [MAXSTR]	General gradient pattern
extern	char	gpat5 [MAXSTR]	General gradient pattern
<b>Delays</b>			
extern	double	tr	Repetition time per scan
extern	double	te	Primary echo time
extern	double	ti	Inversion time

## 2 Pulse-Sequence Programming

extern	double	tm	Mid-delay for STE
extern	double	at	Acquisition time
extern	double	tpe, tpe2, tpe3	Phase encode durations for 2D to 4D
extern	double	tcrush	Crusher gradient duration
extern	double	tdiff	Diffusion encode duration
extern	double	tdelta	Diffusion encode duration
extern	double	tDELTA	Diffusion gradient separation
extern	double	tflow	Flow encode duration
extern	double	tspoil	Spoiler duration
extern	double	hold	Physiological trigger hold off
extern	double	trise	Gradient coil rise time: sec
extern	double	satdly	Saturation time
extern	double	tau	General use delay
extern	double	runtime	User variable for total experiment time
<b>Frequencies</b>			
extern	double	resto	Reference frequency offset
extern	double	wsfrq	Water suppression offset
extern	double	chessfrq	Chemical shift selection offset
extern	double	satfrq	Saturation offset
extern	double	mtfrq	Magnetization transfer offset
<b>Physical Sizes and Positions (for slices, voxels, and FOV)</b>			
extern	double	pro	FOV position in readout
extern	double	ppe, ppe2, ppe3	FOV position in phase encode
extern	double	pos1, pos2, pos3	Voxel position
extern	double	pss[MAXSLICE]	Slice position array
extern	double	lro	Readout FOV
extern	double	lpe, lpe2, lpe3	Phase encode FOV
extern	double	lss	Dimension of multislice range
extern	double	vox1, vox2, vox3	Voxel size
extern	double	thk	Slice or slab thickness
extern	double	lpe, lpe2, lpe3	Phase encode FOV
extern	double	fovunit	Dimensional conversion factor



extern	double	thkunit	Dimensional conversion factor
<b>Bandwidths</b>			
extern	double	sw1, sw2, sw3	Phase encode bandwidths / spectral widths
<b>Counts and Flags</b>			
extern	double	nD	Experiment dimensionality
extern	double	ns	Number of slices
extern	double	ne	Number of echoes
extern	double	ni	Number of standard increments
extern	double	nv, nv2, nv3	Number phase encode views
extern	double	ssc	Compressed ss transients
extern	double	ticks	External trigger counter
extern	char	ir [MAXSTR]	Inversion recovery flag
extern	char	ws [MAXSTR]	Water suppression flag
extern	char	mt [MAXSTR]	Magnetization flag
extern	char	pilot [MAXSTR]	Auto gradient balance flag
extern	char	seqcon [MAXSTR]	Acquisition loop control flag
extern	char	petable [MAXSTR]	Name for phase encode table
extern	char	acqtype [MAXSTR]	Example: "full" or "half" echo
extern	char	exptype [MAXSTR]	Example: "se" or "fid" in CSI
extern	char	apptype [MAXSTR]	Keyword for parameter init, e.g. "imaging"
extern	char	seqfile [MAXSTR]	Pulse-sequence name
extern	char	rfspoil [MAXSTR]	rf spoiling flag
extern	char	satmode [MAXSTR]	Presentation mode
extern	char	verbose [MAXSTR]	Verbose mode for sequences and psg
<b>Miscellaneous</b>			
extern	double	rfphase	rf phase shift
extern	double	B0	Static magnetic field level
extern	double	slcto	Slice selection offset
extern	double	delto	Slice spacing frequency
extern	double	tox	Transmitter offset
extern	double	toy	Transmitter offset

## 2 Pulse-Sequence Programming

extern	double	toz	Transmitter offset
extern	double	griserate	Gradient rise rate

## Pulse Sequence Output

This section describes how to program output in VnmrJ.

### Output messages

The statements in [Table 41](#) control the operation of the pulse sequence at run-time and provide input output to the VnmrJ interface or the Linux file system. It should be emphasized that all C input/output occurs before the acquisition of the first scan. You can use `checkflag` and `ix` (described in the section [Parameters and Variables](#)) to control the circumstances of input and output.

Any C function for input/output can be used to communicate with the Linux file system, for example to open, close, and read or write from files. The `stdout` designation for statements such as `fprintf` or `printf` in pulse sequences is the Text page of the Process tab of the VnmrJ interface. Additional information about C can be obtained from a basic C programming manual.

**Table 41** Statements that control run-time output

<code>abort_message(message, varnames)</code>	Abort at run-time and send a message to error window.
<code>printf(message, varnames)</code>	Print formatted text to the VNMJRj Text page (C).
<code>psg_abort(1)</code>	Abort the pulse sequence at run-time.
<code>psg_abort(1)</code>	Abort the pulse sequence at run-time.
<code>text_error(message, varnames)</code>	Print formatted text to the VnmrJ Text page.
<code>text_message(message, varnames)</code>	Print formatted text to the VnmrJ Text page.
<code>warn_message(message, varnames)</code>	Send a warning message to the error window

The statement `psg_abort(1)` aborts a pulse sequence at run-time, before the first scan and causes a beep. The argument is an internal error index, which is always 1 for pulse sequences. Standard C commands to abort a program have variable behavior in different versions of C and they should no be used.

The statement `abort_message(message, varnames)` is similar to `psg_abort`, but it outputs the string message to the error window of VnmrJ, along with a beep. The argument message should be the message contents, contained in double quotes or be a pointer to a char array with the message.

The statement `warn_message(message, varnames)` outputs a string message to the error window of VnmrJ, along with a beep. The argument message should be the message contents, contained in double quotes or be a pointer to a char array with the message.

The standard C statement `printf` written as `printf(message, varnames)` writes formatted text to the Text page of the Process tab of the VnmrJ interface without a beep. The argument message is the message contents contained in double quotes or be a pointer to a *char* array with the message.

The statements `abort_message` and `warn_message` formats a value of a C variable as a string in manner similar to the standard C statement `printf`. Consult a basic C manual to learn `printf`. The code below gives an example.

An abort condition or the condition for a warning might be the result of an internal calculation in the pulse sequence, such as a duty-cycle calculation, or it might be in response to a parameter that is set incorrectly. The statements for a warning or an abort are usually placed in a C conditional. One might also write to the text window with a `printf` statement. For example:

```
double duty = pw;
double totaltime = d1 + pw + rof2 + alfa + at;
if (dm[2]='y') {
    duty = duty + rof2 + alfa + at;
}
duty = duty/totaltime;
if (duty > 0.1) {
    printf("Abort: the dutycycle of %f\% is too
high!", duty*100);
    psg_abort(1);
}
```

would check the duty cycle of the observe transmitter and decoupler together of a sequence such as `s2pul.c` and abort if it was greater than 10% with a message to the text window. One might replace the contents of the conditional with:

```
abort_message("Abort: the dutycycle of %f\% is too
high!", duty*100);
```

or

```
warn_message("Abort: the dutycycle of %f\% is too
high!", duty*100");
```

## Pulse sequence display

The Menu item, Display Sequence, in the Acquisition pull-down executes the command `dps`, which provides a display of the pulse sequence in a viewport of the VnmrJ interface. The pulse -sequence statements in [Table 42](#) control the output of this display.

The command `dps` runs the pulse sequence in a special program to represent the pulse sequence in a viewport. The immediate display represents the first increment along with most of its input/output, including shape and waveform calculations, as well as printed messages to the VnmrJ interface.

The `dps` command does not recognize `putCmd`, `putvalue`, `putstring`, and `putarray`.

The indexes that index the real-time variables `v1` to `v42`, constants, and real-time tables `t1` to `t60` are different for the `dps` program. Calculations that manipulate these indexes may cause the `dps` command to display incorrect phase-table values.

### Statements to control the DPS display

**Table 42** Statements to control the pulse-sequence display

<code>dps_off()</code>	Turn off the pulse-sequence display for subsequent statements.
<code>dps_on()</code>	Turn on the pulse-sequence display for subsequent statements.
<code>dps_skip()</code>	Turn off the next pulse-sequence statement.
<code>dps_show(options)</code>	Show a particular icon option in the display.

The `dps_off`, `dps_on`, `dps_skip`, and `dps_show` statements can be inserted into a pulse sequence to control the graphical display of the pulse-sequence statements by the `dps` command.

To turn off the `dps` display of statements, insert `dps_off()` into the sequence. All pulse sequence statements following `dps_off` will not be shown.

To turn on the `dps` display of statements, insert `dps_on()` into the sequence. All pulse-sequence statements following `dps_on` will be shown.

To skip the `dps` display of the next statement, insert `dps_skip()` into the sequence. The next pulse-sequence

statement will not be displayed.

To draw pulses for `dps` display, insert `dps_show(options)` statements into the pulse sequence. The pulses will appear in the graphical display of the sequence.

The icon options to `dps_show` are described fully in its entry in “[PSG Variables](#)” on page 145. These options include the ability to draw a line to represent a delay, draw a pulse picture and display the channel name below the picture, draw shaped pulses with labels, draw obs and dec pulses at the same time.

## Setting the Amplitude, Phase, and Gate from Tables

The statements `vobspwrf`, `vdecpwrf`, `vdec2pwrf`, `vdec3pwrf`, and `vdec4pwrf` (Table 43) are used to set the fine power or amplitude scan-to-scan or in a real time loop.

**Table 43** Statements to create and use real-time lists

<code>vdecpwrf (vamp)</code>	Set the amplitude of the dec channel using a real-time integer.
<code>vdec2pwrf (vamp)</code>	Set the amplitude of the dec2 channel using a real-time integer.
<code>vdec3pwrf (vamp)</code>	Set the amplitude of the dec3 channel using a real-time integer.
<code>vdec4pwrf (vamp)</code>	Set the amplitude of the dec4 channel using a real-time integer.
<code>vobspwrf (vamp)</code>	Set the amplitude of the obs channel using a real-time integer.

These statements take a real-time integer or table argument `vamp` with values of 0 to 4095 to set the fine power or amplitude. Note that real-time variables can have only integer values, so these statements set the amplitude with only 12-bit resolution.

Real-time amplitude statements can be used to create a pulse shape or waveform from within a real-time loop. To make a real-time shape, create an integer array containing the shape, using `getarray` or through a calculation in the `pulsesequences` function. Use `settable` to initialize a real-time table with the values. The values can be accessed as real-time variables in a loop with the `gettable` statement. For example:

```
sub(v1,v1,v1);    // initialize v1 to zero
loop(v2,v3);     // initialize v2 with the number of steps
  gettable(t1,v1,v4); // get the v1th element of table t1
                    // and set v4
  incr(v1);      // increment v1
  vobspwrf(v4);  // set the amplitude from v4
  delay(tau);    // set tau to be the pulse-length divided
                // by steps
endloop(v3);
```

The statements

`xmtrphase`, `dcplrphase`, `dcplr2phase`, `dcplr3phase`, and `dcplr4phase` (Table 9 on page 72) can be used to add phase modulation to the shape.

You can gate the transmitter based on a real-time argument

by including the commands `xmtron` and `xmtroff` within a real-time conditional. For example:

```
ifzero(v1);  
    xmtroff();  
elsnz(v1);  
    xmtron();  
endif(v1);
```

gates the transmitter on if  $v1 = 1$  and off if  $v1 = 0$ .

Any other pair of gate commands can be used in this manner.



## Gradient Control for PFG and Imaging

A system for spectroscopy will usually provide one of several different PFG gradient amplifiers to provide current to PFG probes. These include the Performa single-axis amplifiers (1 to 4 and Diffusion) and a three-axis PerformaXYZ amplifier. A gradient pulse is most-often used for coherence selection or as a homospoil pulse. Gradients are also used to provide spatial dispersion for gradient shimming or for microimaging or diffusion experiments with a PFG probe. All of the Performa amplifiers for PFG use a gradient controller in the PFG slot of the acquisition card cage.

A system equipped for imaging uses a set of three L500 imaging amplifiers in a separate bay and a variety of different probes for either horizontal or vertical-bore magnets. Imaging hardware is controlled by an imaging controller in the gradient slot of the card cage.

The shim coils can provide gradient pulses to any probe for either homospoil or gradient shimming. Commands that pulse the shim coils are executed through the master controller.

Table 44 provides all the pulse-sequence statements that control gradients, through PFG amplifiers, imaging amplifiers, or the shims. In principle, all of these statements can be used either with PFG or imaging hardware provided that the appropriate number of axes are present, though there may be some limitations for calculation-intensive imaging sequences. Gradient shapes that are executed through the PFG controller nominally execute at 25 ns for DD2 MR systems and at 50 ns for VNMRS systems time resolution of the RF system. Pulse sequences that use the imaging controller to provide gradient shapes for imaging are automatically limited to 4 s time resolution. It may be necessary to unconfigure the imaging controller when running some sequences for spectroscopy.

**Table 44** Gradient control statements

---

```
create_rotation_list(name, angle_array, num_angles)
```

---

Create rotation list.

---

```
magradient(gradlvl)
```

---

Magic-angle gradient.

---

```
magradpulse(gradlvl, width)
```

---

**Table 44** Gradient control statements (continued)

<b>Magic-angle gradient pulse.</b>
<code>mashapedgradient (pattern, gradlvl, width, loops, wait)</code>
<b>Shaped magic-angle gradient pulse.</b>
<code>mashapedgradpulse (pattern, gradlvl, width)</code>
<b>Shaped magic-angle gradient pulse.</b>
<code>obl_gradient (gradlvl1, gradlvl2, gradlvl3)</code>
<b>Oblique gradient</b>
<code>obl_shapedgradient (pattern, width, gradlvl1, gradlvl2, gradlvl3, wait)</code>
<b>Oblique, shaped gradient, one pattern.</b>
<code>obl_shaped3gradient (pat1, pat2, pat3, gradlvl1, gradlvl2, gradlvl3, wait)</code>
<b>Oblique shaped gradient, three patterns.</b>
<code>pe_gradient (stat1, stat2, stat3, step2, vmult)</code>
<b>Oblique gradient, phase-encode one axis.</b>
<code>pe2_gradient (stat1, stat2, stat3, step2, step3, vmult2, vmult3)</code>
<b>Oblique gradient, phase-encode two axes.</b>
<code>pe3_gradient (stat1, stat2, stat3, step1, step2, step3, vmult1, vmult2, vmult3)</code>
<b>Oblique gradient, phase-encode three axes.</b>
<code>pe_shapedgradient (pattern, width, stat1, stat2, stat3, step2, vmult2, wait)</code>
<b>Oblique shaped gradient pulse, one pattern, phase-encode one axis.</b>
<code>pe_shape3dgradient (pat1, pat2, pat3, width, stat1, stat2, stat3, step2, vmult2, wait)</code>
<b>Oblique shaped gradient pulse, three patterns, phase-encode one axis.</b>
<code>pe2_shapedgradient (pattern, width, stat1, stat2, stat3, step2, step3, vmult2, vmult3, wait)</code>
<b>Oblique shaped gradient pulse, one pattern, phase-encode two axes.</b>
<code>pe2_shaped3gradient (pat1, pat2, pat3, width, stat1, stat2, stat3, step2, step3, vmult2, vmult3, wait)</code>
<b>Oblique shaped gradient pulse, three patterns, phase-encode two axes.</b>
<code>pe3_shapedgradient (pattern, width, stat1, stat2, stat3, step1, step2, step3, vmult1, vmult2, vmult3, wait)</code>
<b>Oblique shaped gradient pulse, one pattern, phase-encode three axes.</b>
<code>pe3_shaped3gradient (pat1, pat2, pat3, width, stat1, stat2, stat3, step1, step2, step3, vmult1, vmult2, vmult3, wait)</code>
<b>Oblique shaped gradient pulse, three patterns, phase-encode three axes.</b>
<code>rgradient (axis, daclvl)</code>
<b>Set gradient level, any axis.</b>
<code>rot_angle (phi, theta, psi)</code>

**Table 44** Gradient control statements (continued)

Set user gradient, oblique rotation angles.
<code>rot_angle_list(listId, state, vindex)</code>
Set gradient, oblique rotation angles from a list.
<code>rotate()</code>
Set gradient, oblique rotation angles.
<code>shapedgradient(pattern, width, gradlvl, axis, loops, wait)</code>
Shaped gradient pulse.
<code>zero_all_gradients()</code>
Zero all gradients.
<code>zgradpulse(daclvl, width)</code>
Z-axis gradient pulse.

### Single-axis gradient control for spectroscopy

The statement `rgradient(axis, daclvl)` is commonly used to set Z-axis gradient levels for systems with single axis Performa amplifiers, for PFG or Diffusion. The argument `axis = 'z' or 'Z'` and `daclvl` is a whole number with a range from  $-32768$  to  $+32767$  for Performa amplifiers 2-4 and the Performa D for high-gradient-strength diffusion. The Performa 1 amplifier has a range from  $-4096$  to  $+4095$ . A `daclvl` of 0 produces no current.

A gradient pulse with `daclvl = 1327.0` would use the statements:

```
rgradient('Z',1327.0);
delay(width);
rgradient('Z',0.0);
```

in which `width` is the width of a gradient pulse.

The statement `zgradpulse(1327.0,width)` automatically applies the above gradient pulse along the Z axis.

The `rgradient` statement can be used to set X and Y gradients for systems with the PerformaXYZ amplifier or to an imaging system, in which `axis = 'x' or 'X' and 'y' or 'Y'`. The `rgradient` statement can also be used to generate homospoil pulses on X, Y, or Z by selecting the homospoil option on the configuration page, if it is provided as an option.

Use the `gradalt` parameter to reduce gradient recovery artifacts by inverting the gradients on alternate scans. `gradalt` acts as a multiplier for the gradient amplitude on alternating scans.

The `zgradpulse` and `rgradient` pulse elements use the value of `gradalt` to multiply the gradient amplitude.

No changes are made if:

the local (`curpar`) parameter `gradalt` does not exist

the local (`curpar`) parameter `gradalt` is set to "Not Used"

the local (`curpar`) parameter `gradalt` is set to "1"

Examples:

- `gradalt=1` Alternating scans will have gradient amplitudes multiplied by 1; no changes are made.
- `gradalt=-1` Alternating scans will have gradient amplitudes multiplied by "-1".
- `gradalt=-0.99` Alternating scans will have gradient amplitudes multiplied by "-0.99".  
Use this when positive and negative gradients are not exactly the same.
- `gradalt=0` Alternating scans will have gradient amplitude multiplied by "0".
- `gradalt=2` Alternating scans will have gradient amplitude multiplied by "2".
- `gradalt=-0.997, -0.998, -0.999, -1.0, -1.001, -1.002, -1.003` Alternating scans will have gradient amplitudes multiplied by the listed numbers.  
Use this to calibrate the negative gradient.
- `gradalt=-1,1` Alternating scans will have gradient amplitudes multiplied by "-1, 1".  
Use this to compare the effect of alternating gradients.

Pulse sequences can be coded to activate or deactivate gradient alternation, independently of the `gradalt` parameter. Adding the statement

```
gradalt = 1.0
```

will deactivate alternating gradients from following `zgradpulse` or `rgradient` calls.

Adding the statement

```
gradalt = -1.0
```

will activate alternating gradients to follow `zgradpulse` or `rgradient` calls. This mechanism also allows specific gradients in a pulse sequence to either alternate gradients or not. For example,

```
double gradaltorig;

gradaltorig = gradalt; /* remember the user
preference for gradalt */

gradalt= 1.0;          /* defeat alternating
gradients for the following */

zgradpulse(gzlvl1,gt1);

gradalt= -1.0;        /* force alternating
gradients for the following */

zgradpulse(gzlvl2,gt2);

gradalt= gradaltorig; /* return to user preference
for alternation for the following */

zgradpulse(gzlvl3,gt3);
```

## Shaped gradients with `nowait` capability

The statement `shapedgradient(pattern,width,daclvl,axis,loops,wait)` can be used to deliver a gradient shape to the Z axis of a Performa amplifier, one of the three axes of the Performa XYZ amplifier or an imaging system. The argument `pattern` is the root name of a text file with the extension `.GRD` in the directory `shapelib` containing the elements of the shape. The time `width` is in seconds, gradient level `daclvl` is set in DAC units and `axis` can be 'x' or 'X', 'y' or 'Y', and 'z' or 'Z'. The integer value `loops` is  $N = 1$  to 255 and it allows the `pattern` to execute multiple times with a total time of  $N \times \text{width}$ .

The string argument `wait` can have the values 'wait' or 'nowait'. If the value is 'nowait', instructions following the `shapedgradient` statement are executed immediately. If the value is 'wait', instructions following the statement are executed after the completion of the gradient pulse. This *nowait* capability allows you to execute shaped RF pulses simultaneously with the gradient pulse. Most gradient statements that execute a pattern have *nowait* capability.

## Creating a gradient table

All gradient statements other than `rgradient` and

`zgradpulse` require gradient amplifiers for three axes, either the Performa XYZ or hardware for imaging. All three-axis gradient statements set the gradient level `gradlvl` in Gauss/cm rather than DAC units and require the use of a gradient calibration table to convert between DAC units and Gauss/cm. This table is created with the macro `createtable` and it is stored in `/vnmr/imaging/gradtables`. Consult the *Command and Parameter Reference* for instructions in using `createtable`.

### Controlling magic angle gradients

The statement `magradient(gradlvl)` is used to set a gradient along the magic angle ( $54.7^\circ$ ) using the X,Y, and Z axes of the PerformaXYZ amplifier. The value of `gradlvl` is entered in Gauss/cm and converted to DAC units using the gradient table. The statement `magradient(0.0)` is used to end a magic-angle gradient pulse after a delay.

The statement `magradpulse(gradlvl,width)` can be used to apply a magic-angle gradient pulse with the time `width`, in seconds.

The statement `mashapedgradient(pattern,gradlvl,width,loops,wait)` is used to deliver a gradient shape along the magic angle. The value of `gradlvl` is entered in Gauss/cm and converted to DAC units using the Gradient Table. The `loops` and `wait` arguments are similar to those of `shapedgradient`.

The statement `mashapedgradpulse(pattern,gradlvl,width)` is used to deliver a gradient shape along the magic angle without looping and the *nowait* capability.

### Oblique and phase-encoded-oblique gradients for imaging

The gradient statements labeled with `obl_` and `pe#_` in [Table 38](#) on page 142 are static and phase-encoded oblique gradients that are used exclusively in imaging sequences. You should consult the *Guide to Imaging* for a thorough discussion of pulse-sequence programming with these statements.

The three axes of an oblique gradient are logical axes that are designated (1) *read-out*, (2) *phase-encode*, and (3) *slice-select*. The orientation of the logical axes relative to the physical X,Y, and Z axes is determined by three Euler angles `psi`, `theta`, and `phi` that are represented internally as *global* PSG variables. The `rotate()` statement is used to set the

gradient rotation software in the imaging controller with these values. The statement `rot_angle(psi,theta,phi)` can be used to set and apply the rotation angles in the pulse sequence.

The statement `obl_gradient(gradlvl1,gradlvl2,gradlvl3)` sets the gradient levels of the three logical axes and `obl_gradient(0.0,0.0,0.0)` ends an oblique gradient pulse after a delay. The `gradlvl#` arguments are set in Gauss/cm.

The statement `obl_shapedgradient(pattern,gradlvl1,gradlvl2,gradlvl3,wait)` delivers a shaped gradient pulse in which `pattern` is the root name of a text file with a `.GRD` extension in `shapelib`, to apply a gradient shape for all three logical axes. The argument `wait` provides *nowait* capability.

The statement `obl_shaped3gradient(pat1,pat2,pat3,gradlvl1,gradlvl2,gradlvl3,wait)` delivers a shaped gradient pulse that uses three different patterns, `pat1`, `pat2`, and `pat3` for the three logical axes. The argument `wait` provides *nowait* capability.

The statements `pe_gradient`, `pe2_gradient`, and `pe3_gradient` provide gradients that are phase-encoded along one, two, and three axes. For these gradients, the gradient levels consist of a static portion `stat1`, `stat2`, and `stat3` and a step size `step1`, `step2`, and `step3` in Gauss/cm. The arguments `vmult1`, `vmult2`, and `vmult3` are real-time indexes for the steps.

The statements `pe_shapedgradient`, `pe2_shapedgradient`, and `pe3_shapedgradient` deliver phase-encoded gradients with a single pattern for all three axes. The statements `pe_shaped3gradient`, `pe2_shaped3gradient`, and `pe3_shaped3gradient` deliver phase-encoded gradients that use three different patterns for the three axes. All of the phase-encoded shaped gradients have *nowait* capability, using the argument `wait`.

## Controlling slice-selective RF shaped pulses for imaging

The statements in [Table 45](#) on page 169 are used to provide frequency-offset shaped pulses to selectively excite positions in imaging experiments. The offset for an individual position is determined from the gradient level and a distance in cm. An offset is combined with a pattern to produce a selective phase-ramped, frequency-offset pulse that excites the position. A typical imaging experiment makes use of a list of

these pulses. For more information about the programming of slice-selective pulses, consult the *Guide to Imaging*.

The statement `poffset(pos,gradlvl)` returns a single frequency offset in Hz for the observe channel that depends upon a position `pos` in cm and a gradient level `gradlvl` in Gauss/cm. This statement automatically takes into account a *global* PSG variable `resto`, which should contain the spectral offset of the resonance of interest in Hz.

The statement `shapedpulseoffset(pattern,width,phase,rof1,rof2,offset)` applies a shaped RF pulse with a frequency offset created by phase ramping, in which `pattern` is the root name of a text file in `shapelib` with a `.RF` extension and `offset` is the frequency offset in Hz. The argument `width` is the pulse width and `rof1` and `rof2` are the predelay and postdelay. The argument `offset` can be the value returned from the statement `poffset`. The `pattern` file for the offset shape is executed directly in the RF controller and is not written into `shapelib`.

The statement `offsetlist(posarray,gradlvl,resfrq,offsetarray,ns,state)` returns an array of offsets `offsetarray` in Hz based upon an array of positions `posarray` in cm and a single gradient level `gradlvl`. The integer `ns` is the number of offsets. The string argument `state` is supplied from the *global* PSG variable `seqcon` that determines whether the offsets will be supplied in a compressed 'c' or a standard 's' loop. See the previous discussion of `peloop` and `msloop` in this chapter. The argument `resfrq` is the spectral offset of the resonance of interest. The statement `offsetglist` is similar, but it accepts an array of gradient levels `gradlvlarray` as well as positions.

The statement `shapelist(pattern,width,offsetarray,ns,state)` creates a list of phase-ramped, offset, shaped RF pulses using the argument `pattern`, for the shape and an array of offsets `offsetarray`. The argument `width` is the pulse width in seconds, the integer `ns` is the number of shapes, and `state` should be the same as that used for `offsetlist`. The statement `shapelist` returns an integer `listId` to identify the list of shapes. The `patterns` files for the offset shapes are executed directly in the RF controllers and are not written into `shapelib`.

The statement `shapedpulselist(listId,width,phase,rof1,rof2,state,i`



index) is used to apply offset, shaped pulses from the list generated by `shapelist`. The first argument `listId` is an integer designating the list returned by `shapelist`. The argument `width` is the pulse width and `rof1` and `rof2` are the `predelay` and `postdelay`. The argument `state` should be the same as that of `shapelist`, and `state` determines `index`. The `index` should be a real-time variable if the list is to be executed in compressed mode and it should be an integer 1 if the list is to be executed in standard mode.

**Table 45** Statements for slice-selective shaped pulses

---

```
offsetlist(posarray,gradlvl,resfrq,offsetarray,ns,state)
```

---

Create offset from position array.

---

```
offsetglist(posarray,gradarray,resfrq,offsetarray,ns,state)
```

---

Create offset array from position and gradient-level array.

---

```
offset=poffset(pos,gradlvl)
```

---

Return offset for observe channel from position.

---

```
poffset_list(posarray,gradlvl,ns,apvl)
```

---

Set observe-channel offsets from position array.

---

```
position_offset(pos,gradlvl,resfrq,device)
```

---

Set offset of device from position and resonance offset.

---

```
position_offset_list(posarray,gradlvl,ns,resfrq,device,
listId,apvl)
```

---

Set offsets of device from position array and resonance offset.

---

```
shapedpulselist(listId,width,phase,rof1,rof2,state,index)
```

---

Apply phase-ramped shaped pulses from a pre-computed list.

---

```
shapedpulseoffset(pattern,width,phase,rof1,rof2,offset)
```

---

Apply a phase-ramped shaped pulse with an offset.

---

```
listId=shapelist(pattern,width,offsetarray,ns,state)
```

---

Create phase-ramped shaped pulses from a pattern and offset array.

---

```
width=shapelistpw(pattern,width)
```

---

Return exact width of shaped pulse with offset.

---

## Controlling lock field correction

**Table 46** Statements for controlling lock field correction

---

```
lk_hold()          Set lock to hold correction.
```

---

```
lk_sample()       Set lock to sample lock signal.
```

---

The statements `lk_sample` and `lk_hold` (Table 46) control the lock field-correction circuitry. Use `lk_hold` to disable the field correction circuitry of the lock channel during gradient pulses or during RF pulses at the  $^2\text{H}$  lock frequency, which can disturb the lock level. Resume field correction with `lk_sample` after the disturbance is complete



## 3 Pulse Sequence Statement Reference

Pulse Sequence Statements 172

This chapter contains a reference to the statements used in programming VnmrJ pulse sequences.



## Pulse Sequence Statements

abort_message	Abort PSG at run-time and send message to VnmrJ
acquire	Acquire data explicitly
add	Add real-time integer values
acquire_obs	Acquire data explicitly in parallel section
acquire_rcvr	Acquire data explicitly in parallel section
assign	Assign real-time integer value using real-time integer
clearapdatatable	Zero all data in digital receiver memory
create_angle_list	Create real-time list of gradient coordinate rotation angles
create_offset_list	
create_rotation_list	Create list of gradient-coordinate rotation angles
dbl	Double real-time integer value
dcplrphase	Set small-angle phase of first decoupler
dcplr2phase	Set small-angle phase of second decoupler
dcplr3phase	Set small-angle phase of third decoupler
dcplr4phase	Set small-angle phase of fourth decoupler
decblank	Blank amplifier of first decoupler
dec2blank	Blank amplifier of second decoupler
dec3blank	Blank amplifier of third decoupler
dec4blank	Blank amplifier of fourth decoupler
decoff	Turn off first decoupler
dec2off	Turn off second decoupler
dec3off	Turn off third decoupler
dec4off	Turn off fourth decoupler
decoffset	Set frequency offset of first decoupler
dec2offset	Set frequency offset of second decoupler
dec3offset	Set frequency offset of third decoupler
dec4offset	Set frequency offset of fourth decoupler
decon	Turn on first decoupler
dec2on	Turn on second decoupler
dec3on	Turn on third decoupler
dec4on	Turn on fourth decoupler
decphase	Set quadrature phase of first decoupler
dec2phase	Set quadrature phase of second decoupler
dec3phase	Set quadrature phase of third decoupler
dec4phase	Set quadrature phase of fourth decoupler
decpower	Set power level of first decoupler
dec2power	Set power level of second decoupler
dec3power	Set power level of third decoupler
dec4power	Set power level of fourth decoupler
decprgoff	End waveform decoupling on first decoupler
dec2prgoff	End waveform decoupling on second decoupler
dec3prgoff	End waveform decoupling on third decoupler
dec4prgoff	End waveform decoupling on fourth decoupler
decprgon	Start waveform decoupling on first decoupler
dec2prgon	Start waveform decoupling on second decoupler
dec3prgon	Start waveform decoupling on third decoupler
dec4prgon	Start waveform decoupling on fourth decoupler
decprgonOffset	Start waveform decoupling on first decoupler with offset
dec2prgonOffset	Start waveform decoupling on second decoupler with offset
dec3prgonOffset	Start waveform decoupling on third decoupler with offset
dec4prgonOffset	Start waveform decoupling on fourth decoupler with offset
decpulse	Perform pulse with assigned values on first decoupler
decpwrf	Set fine power level of first decoupler
dec2pwrf	Set fine power level of second decoupler

dec3pwrfl	Set fine power level of third decoupler
dec4pwrfl	Set fine power level of fourth decoupler
decr	Decrement real-time integer value
decrpulse	Perform pulse on first decoupler
dec2rgpulse	Perform pulse on second decoupler
dec3rgpulse	Perform pulse on third decoupler
dec4rgpulse	Perform pulse on fourth decoupler
decshaped_pulse	Perform shaped pulse on first decoupler
dec2shaped_pulse	Perform shaped pulse on second decoupler
dec3shaped_pulse	Perform shaped pulse on third decoupler
dec4shaped_pulse	Perform shaped pulse on fourth decoupler
decspinlock	Perform waveform spinlock on first decoupler
dec2spinlock	Perform waveform spinlock on second decoupler
dec3spinlock	Perform waveform spinlock on third decoupler
dec4spinlock	Perform waveform spinlock on fourth decoupler
decstepsize	Set small-angle phase stepsize for first decoupler
dec2stepsize	Set small-angle phase stepsize for second decoupler
dec3stepsize	Set small-angle phase stepsize for third decoupler
dec4stepsize	Set small-angle phase stepsize for fourth decoupler
decunblank	Unblank amplifier of first decoupler
dec2unblank	Unblank amplifier of second decoupler
dec3unblank	Unblank amplifier of third decoupler
dec4unblank	Unblank amplifier of fourth decoupler
delay	Execute time delay
divn	Divide real-time integer values
dps_off	Turn off graphical display of statements
dps_on	Turn on graphical display of statements
dps_show	Display pulse and delay icons in graphical display
dps_skip	Skip graphical display of next statement
elsenz	Execute real-time else statement
endhardloop	End hardware loop
endacq	End explicit acquisition
endacq_obs	End explicit acquisition in parallel section
endacq_rcvr	End explicit acquisition in parallel section
endif	Execute real-time endif statement
endloop	End real-time loop
endmsloop	End switchable multislice loop
endpeloop	End switchable phase-encode loop
exe_grad_rotation	Set oblique gradient-coordinate rotation angles in real-time
F_initval	Always assign real-time integer value using numeric value
getarray	Obtain all values from arrayed parameter
getelem	Assign real-time integer using table element
getgradpowerintegral	get gradient power integral for X, Y and Z axes
getorientation	Read image-plane orientation
getstr	Obtain value of string parameter
getval	Obtain value of numeric parameter
hlv	Assign half the value of real-time integer
hsdelay	Execute time delay with optional homospoil pulse
ifrtGE	Execute real-time greater-than-or-equal
ifrtGT	Execute real-time greater-than
ifrtLE	Execute real-time less-than-or-equal
ifrtLT	Execute real-time less-than
ifrtEQ	Execute real-time equal
ifrtNEQ	Execute real-time not-equal
ifzero	Execute real-time equal-zero
incr	Increment real-time integer value

### 3 Pulse Sequence Statement Reference

<code>init_decpattern</code>	Create pattern file for waveform decoupling
<code>init_gradpattern</code>	Create pattern file for gradient shape
<code>init_rfpattern</code>	Create pattern file for shaped pulse
<code>initval</code>	Assign real-time integer value using numeric value
<code>kzendlloop</code>	End real-time loop with fixed duration
<code>kzloop</code>	Start real-time loop with fixed duration
<code>lk_hold</code>	Set lock correction circuitry to hold correction
<code>lk_sample</code>	Set lock correction circuitry to sample lock signal
<code>loadtable</code>	Assign table elements from table text file
<code>loop</code>	Start real-time loop
<code>loop_check</code>	Check number of FIDs for compressed acquisition.
<code>magradient</code>	Set three-axis gradient at magic angle
<code>magradpulse</code>	Perform three-axis gradient pulse at magic angle
<code>mashapedgradient</code>	Perform three-axis gradient shape at magic angle
<code>mashapedgradpulse</code>	Perform three-axis shaped gradient pulse at magic angle
<code>mod2</code>	Assign real-time integer value modulo 2
<code>mod4</code>	Assign real-time integer value modulo 4
<code>modn</code>	Assign real-time integer value modulo n
<code>msloop</code>	Start switchable multislice loop
<code>mult</code>	Multiply real-time integer values
<code>obl_gradient</code>	Set three-axis oblique gradient
<code>obl_shapedgradient</code>	Perform three-axis oblique gradient shape, single pattern
<code>obl_shaped3gradient</code>	Perform three-axis oblique gradient shape, three patterns
<code>obsblank</code>	Blank amplifier of observe channel
<code>obsoffset</code>	Set frequency offset of observe channel
<code>obspower</code>	Set power level of observe channel
<code>obsprgoff</code>	End waveform decoupling on observe channel
<code>obsprgon</code>	Start waveform decoupling on observe channel
<code>obspulse</code>	Perform pulse with assigned values on observe channel
<code>obsprwf</code>	Set fine power level of observe channel
<code>obsstepsize</code>	Set small-angle phase stepsize of observe channel
<code>obsunblank</code>	Unblank amplifier of observe channel
<code>offset</code>	Set frequency offset of any channel
<code>offsetglist</code>	Create offset array from position and gradient-amplitude array
<code>offsetlist</code>	Create offset array from position and gradient-amplitude
<code>parallelacquire_obs</code>	Acquire data explicitly in parallel section
<code>parallelacquire_rcvr</code>	Acquire data explicitly in parallel section
<code>parallelend</code>	End parallel section of pulse sequence
<code>parallelstart</code>	Start parallel section of pulse sequence
<code>parallelsync</code>	Position parallel synchronization delays
<code>pe_gradient</code>	Set oblique gradient, phase encode one axis
<code>pe2_gradient</code>	Set oblique gradient, phase encode two axes
<code>pe3_gradient</code>	Set oblique gradient, phase encode three axes
<code>pe_shapedgradient</code>	Perform oblique gradient shape, one pattern, phase encode one axis
<code>pe_shaped3gradient</code>	Perform oblique gradient shape, three patterns, phase encode one axis
<code>pe2_shapedgradient</code>	Perform oblique gradient shape, one pattern, phase encode two axes
<code>pe2_shaped3gradient</code>	Perform oblique gradient shape, three patterns, phase encode two axes
<code>pe3_shapedgradient</code>	Perform oblique gradient shape, one pattern, phase encode three axes
<code>pe3_shaped3gradient</code>	Perform oblique gradient shape, three patterns, phase encode three axes
<code>peloop</code>	Start switchable phase-encode loop
<code>phase_encode3_gradient</code>	Set general oblique gradient, phase encode three axes
<code>phase_encode3_gradpulse</code>	Perform general oblique gradient pulse, phase encode three axes
<code>phase_encode3_oblshapedgradient</code>	Perform general oblique shaped gradient, phase encode three axes
<code>nt</code>	
<code>poffset</code>	Return frequency offset based on position
<code>poffset_list</code>	Create offset array from position array
<code>position_offset</code>	Return frequency offset based on position

position_offset_list	Create offset array from position array
psg_abort	Abort PSG process at run-time
pulse	Perform pulse with assigned values on observe channel
putarray	Set all values of arrayed parameter from pulse sequence
putCmd	Send command to VnmrJ from pulse sequence
putstring	Set string parameter from pulse sequence
putvalue	Set numeric value from pulse sequence
rcvloff	Turn off receiver and unblank observe amplifier
rcvron	Turn on receiver and blank observe amplifier
readMRIUserByte	Read MRI user byte on MRI User Panel
recoff	Turn off receiver gate
recon	Turn on receiver gate
rcvrphase	Set small-angle phase of receivers
rcvrstepsize	Set small-angle phase stepsize of receivers
rgpulse	Perform pulse on observe channel
rgradient	Set DAC level of any one gradient axis
rlloop	Start real-time loop with fixed count
rlpower	Set power level of any channel
rlpwrfl	Set fine power level of any channel
rot_angle	Set user-defined oblique gradient-coordinate rotation angles
rot_angle_list	Set oblique gradient-coordinate rotation angles from list
rotate	Set standard oblique gradient-coordinate rotation angles
rotorperiod	Obtain period of external tachometer signal
rotorsync	Execute time delay based on external tachometer signal
sample	Acquire data explicitly during time delay
setacqmode	Set windowed acquisition for explicit sampling
set_angle_list	Select angle from 1D real-time list
setMRIUserGates	Set all three gates of MRI User Panel
setreceiver	Set quadrature receiver phase from table
setstatus	Set decoupler status of any channel
settable	Assign integer array to table
shapedpulse	Perform shaped pulse on observe channel
shaped_pulse	Perform shaped pulse on observe channel
shapedgradient	Perform shaped gradient pulse on any one axis
shapelist	Create pulse-shape list from base pattern and offset array
shapedpulselist	Perform shaped pulses from shape list on observe channel
shapedpulseoffset	Perform shaped pulse on observe channel with offset
shapelistpw	Return exact pulse width of shaped-pulse pattern
simpulse	Perform simultaneous pulses, observe, and first decoupler
sim3pulse	Perform simultaneous pulses, observe, first, and second decouplers
sim4pulse	Perform simultaneous pulses, observe, first, second, and third decouplers
simshaped_pulse	Perform simultaneous shaped pulses, observe, and first decoupler
sim3shaped_pulse	Perform simultaneous shaped pulses, observe, first, and second decouplers
sim4shaped_pulse	Perform simultaneous shaped pulses, observe, first, second, and third decouplers
sp#off	Turn off spare line (#=1,2, or 3)
sp#on	Turn on spare line (#=1,2, or 3)
spinlock	Perform waveform spinlock on observe channel
starthardloop	Start hardware loop
startacq	Initialize explicit acquisition
startacq_obs	Initialize explicit acquisition in parallel section
startacq_rcvr	Initialize explicit acquisition in parallel section
status	Set status of decoupler and homospoil
statusdelay	Execute status statement within time delay
stepsize	Set small-angle phase stepsize of any channel
sub	Subtract real-time integer values

### 3 Pulse Sequence Statement Reference

swift_acquire	Execute SWIFT rf pulses and gated acquire pulse train
text_error	Send error message to VnmrJ
text_message	Send message to VnmrJ
triggerSelect	Select trigger input on MRI User Panel
tsadd	Add integer to table elements
tsdiv	Divide integer into table elements
tsmult	Multiply integer with table elements
tssub	Subtract integer from table elements
ttadd	Add table to second table
ttdiv	Divide table into second table
ttmult	Multiply table by second table
ttsub	Subtract table from second table
txphase	Set quadrature phase of observe channel
userDECshape	Create waveform pattern directly, with interpolation
var_active	Check if parameter is used
vdecpwrfl	Set fine power level of first decoupler in real-time
vdecpwrflstepsize	Set step size for real-time fine power of first decoupler
vdec2pwrfl	Set fine power level of second decoupler in real-time
vdec3pwrflstepsize	Set step size for real-time fine power of second decoupler
vdec3pwrfl	Set fine power level of third decoupler in real-time
vdec4pwrflstepsize	Set step size for real-time fine power of third decoupler
vdec4pwrfl	Set fine power level of fourth decoupler in real-time
vdec4pwrflstepsize	Set step size for real-time fine power of fourth decoupler
vdelay	Execute time delay with fixed timebase and real-time count
vdelay_list	Get delay value from delay list with real-time index
vobspwrfl	Set fine power level of observe channel in real-time
vobspwrflstepsize	Set step size for real-time fine power of observe channel
warn_message	Send warning message to VnmrJ
writeMRIUserByte	Set user byte on MRI User Panel
xgate	Gate pulse sequence from external tachometer signal
xmtroff	Turn off observe channel
xmtron	Turn on observe channel
xmtrphase	Set small-angle phase of observe channel
zero_all_gradients	Zero gradient DAC level for all axes
zgradpulse	Perform gradient pulse on z axis



## A

<code>abort_message</code>	Abort PSG at run-time and send message to VnmrJ
<code>acquire</code>	Acquire data explicitly
<code>acquire_obs</code>	Acquire data explicitly in parallel section
<code>acquire_rcvr</code>	Acquire data explicitly in parallel section
<code>add</code>	Add real-time integer values
<code>assign</code>	Assign real-time integer value using real-time integer

**abort\_message****Abort PSG at run-time and send message to VnmrJ**

**Syntax:** `abort_message(message, varnames)`  
`char *message` /\* a formatted string  
containing the message \*/  
`varnames` /\* char, int, or doubles,  
used in message \*/

**Description:** Abort the PSG process at run-time, send a formatted warning message to VnmrJ, and cause a beep. The formatting is similar to the C printf statement.

**Related:** `psg_abort` Abort PSG Process  
`text_error` Send error message to VnmrJ  
`text_message` Send message to VnmrJ  
`warn_message` Send warning message to VnmrJ

**acquire****Acquire data explicitly**

**Syntax:** `acquire(points, dwell)`  
`double points;` /\* number of points to acquire \*/  
`double dwell;` /\* dwell time, seconds \*/

**Description:** The acquire statement is an alias of the sample statement for VNMRS. The construction `acquire(points,dwell)` executes the statement `sample(time)` in which `time= points*dwell/2.0`. The individual arguments have no independent significance other than to form the product time. For VNMRS, the sample statement acquires data points using the digital receiver with a 12.5 ns dwell time (80 MHz rate) for a duration of `time`, in seconds. The data points are automatically downsampled with calculation of digital filters to produce a set of complex points with a dwell time of 1.0/sw. For Unity-series systems, the acquire statement explicitly controls the digitizer of the receiver. The value `points` is the number of sample points, real and imaginary, to obtain and `dwell` is the time between sample points.

The construction `acquire(np,1.0/sw)` is identical to `sample(np/(2.0*sw))` and is compatible with both VNMRS and Unity-series systems.

Arguments: `points*dwel` is the duration, in seconds, of the sampling interval. Note that `points` is a *double*. For example, the construction `acquire(2,0.2e-6)` will cause an error; it should be `acquire(2.0,0.2e-6)`.

Related: `endacq` End explicit acquisition  
`rcvloff` Turn on receiver and unblank observe amplifier  
`rcvron` Turn off receiver and blank observe amplifier  
`sample` Acquire data explicitly during time delay  
`startacq` Initialize explicit acquisition

**acquire\_obs**

**Acquire data explicitly in parallel section**

Syntax: `acquire_obs(points,dwell)`  
`double points; /* number of points  
(complex pairs) to acquire */`  
`double dwell; /* dwell time, seconds */`

Description: Analogous to the `acquire` statement but to be used in "obs" parallel sections of a pulse sequence.

Related: `acquire_rcvr` Acquire data explicitly in parallel section  
`startacq_obs` Initialize explicit acquisition in parallel section  
`startacq_rcvr` Initialize explicit acquisition in parallel section  
`endacq_obs` End explicit acquisition in parallel section  
`endacq_rcvr` End explicit acquisition in parallel section  
`parallelacquire_obs` Acquire data explicitly in parallel section  
`parallelacquire_rcvr` Acquire data explicitly in parallel section

**acquire\_rcvr**

**Acquire data explicitly in parallel section**

Syntax: `acquire_rcvr(points,dwell)`  
`double points; /* number of points  
(complex pairs) to acquire */`  
`double dwell; /* dwell time, seconds */`

**Description:** Analogous to the acquire statement but to be used in "rcvr" parallel sections of a pulse sequence.

**add****Add real-time integer values**

<b>Related:</b>	acquire_obs	Acquire data explicitly in parallel section
	startacq_obs	Initialize explicit acquisition in parallel section
	startacq_rcvr	Initialize explicit acquisition in parallel section
	endacq_obs	End explicit acquisition in parallel section
	endacq_rcvr	End explicit acquisition in parallel section
	parallelacquire_obs	Acquire data explicitly in parallel section
	parallelacquire_rcvr	Acquire data explicitly in parallel section

**Syntax:**

```
add(vi,vj,vk)
codeint vi;      /* real-time variable for
addend */
codeint vj;      /* real-time variable for
addend */
codeint vk;      /* real-time variable for sum
*/
```

**Description:** Set the value of vk equal to the sum of integer values of vi and vj.

**Arguments:** vi contains the integer value of the addend, vj contains the integer value of the addend, and vk contains the resulting sum. Each argument must be a real-time variable (v1 to v42, oph, etc).

**Examples:** add(v1,v2,v3);

<b>Related:</b>	assign	Assign real-time integer value using real-time integerL
	dbl	Double real-time integer value
	decr	Decrement real-time integer value
	divn	Divide real-time integer values
	hlv	Assign half the value of real-time integer
	incr	Increment real-time integer value
	initval	Assign real-time integer value using numeric value
	mod2	Assign real-time integer value modulo 2
	mod4	Assign real-time integer value modulo 4

modn	Assign real-time integer value modulo n
mult	Multiply real-time integer values
sub	Subtract real-time integer values

**assign**

**Assign real-time integer using real-time integer**

**Syntax:** `Syntax:assign(vi,vj)`  
`codeint vi; /* real-time variable for input */`  
`codeint vj; /* real-time variable to be assigned */`

**Description:** Set the value of vj equal to the integer value of vi.

**Arguments:** vi contains the value to be assigned and vj contains the resulting assignment. Each argument must be a real-time variable (v1 to v42, oph, etc).

**Examples:** `assign(v3,v2);`

**Related:**

add	Add real-time integer values
dbl	Double real-time integer value
decr	Decrement real-time integer value
divn	Divide real-time integer values
hlv	Assign half the value of real-time integer
incr	Increment real-time integer value
initval	Assign real-time integer value using numeric value
mod2	Assign real-time integer value modulo 2
mod4	Assign real-time integer value modulo 4
modn	Assign real-time integer value modulo n
mult	Multiply real-time integer values
sub	Subtract real-time integer values

## C

clearapdatatable	Zero all data in digital receiver memory
create_angle_list	Create real-time list of gradient-coordinate rotation angles
create delay list	Create table of delays
create offset list	Create table of frequency offsets
create_rotation_list	Create list of gradient-coordinate rotation angles

**clearapdatatable****Zero all data in digital receiver memory**

Syntax: clearapdatatable()

Description: Zero the acquired data table. The data table is automatically zeroed at the start of the execution of a pulse sequence.

**create\_angle\_list****Create 1D real-time list of angles**

Syntax: listId=create\_rotation\_list(name,  
angle\_array, num\_angles);  
char name; /\* name of the rotation list \*/  
double \*angle\_array[3]; /\* 1D angle-list  
array \*/  
int num\_angles; /\* number of angles in array  
\*/  
int listId; /\* integer label of list \*/

Description: Create a 1D list of angles, identified by the integer listId, to set one of the values psi, theta, and phi for oblique gradients. The gradient rotation angles rotate from the logical gradient axes read, phase, and slice to the gradient axes X, Y and Z. The input argument angle\_array is a one-dimensional array of doubles, to be used as angles, with a dimension of num\_angles.

A list created by create\_angle\_list is used by set\_angle\_list to set one of the three oblique rotation angles in either real-time, *compressed* mode, or at run-time for use in an array. In contrast, a list created by create\_rotation\_list sets all three oblique rotation angles, but can only be used at run-time.

Arguments: name is a string variable that sets the name of the list of angles

angle\_array is a 1D array to hold one of the Euler rotation angles psi, theta, or phi, and has as many rows as the number of angles in the list. The declaration angle\_array[128] creates an array to hold 128 values for one angle.

num\_angles is an integer specifying the number of angles in the array.

The return listId is an integer identifying the rotation list. The first list is identified as 0, the

second 1, *etc.* listId is an argument to the set\_angle\_list statement.

Related:	exe_grad_rotation	Set oblique gradient-coordinate rotation angles in real-time
	create_angle_list	Create 1D real-time list of angles
	create_rotation_list	Create list of oblique gradient-coordinate rotation angles
	rot_angle	Set user-defined oblique gradient-coordinate rotation angles
	rot_angle_list	Set oblique gradient-coordinate rotation angles from a list
	rotate	Set standard oblique gradient-coordinate rotation angles
	set_angle_list	Select angle from a 1D real-time list

**create\_delay\_list**

**Create table of delays**

Applicability: Direct Drive systems.

Syntax: 

```
list_number =
create_delay_list(list,nvals)
double *list;      /* pointer to list of
delays */
int nvals;         /* number of values in
list */
int list_number;  /* integer for list
number return value */
```

Description: Creates a global list of delays that can be accessed with a real-time variable or table element for dynamic setting in pulse sequences. There can be a maximum of 256 lists, depending on the size of the lists. The lists are stored in data memory and compete for space with the acquisition data for each array element. If a list is created, the return value is the number of the list (0 to 255); if an error occurs, the return value is negative. create\_delay\_list creates what is called a global list. Global lists are different from AP tables in that the lists are sent down to the acquisition console when the experiment starts up and are accessible until the experiment completes. In working with arrayed experiments, a list is created for every array

element (including implicit 2D cases using the `ni` parameter) unless care is taken to call the function only for the first array element (see example below). To read in an array of values, use the `getarray` statement. To ensure that the list is only created once, check the global array counter variable `ix`, and only call `create_delay_list` to create the list when it equals 1 (as shown in the example).

**Arguments:** `list_number` is an integer return value identifying the newly created list.  
`nvals` is the number of values in the list.  
`list` a C array of delays

**Examples:**

```
pulsesequence()? {?
    /* Declare static to save between calls
    */?
    static int list1;?
    int i, nvals;?
    double delay1[1024];? ?
    nvals = 1024;?
    if (ix == 1) {?
        for (i=0; i<nvals; i++) {?
            ... /* Initialize delay1 array
        */?
        }?
        list1 =
        create_delay_list(delay1,nvals); ?
    }? ...
    vdelay_list(delay1,v5); /* v5 is the
    real time index into the delay1 list */
}
```

### Related Statements

<code>create_freq_list</code>	Create table of frequencies
<code>create_offset_list</code>	Create table of frequency offsets
<code>delay</code>	Delay for a specified time
<code>getarray</code>	Retrieves all values of an arrayed parameter
<code>vdelay</code>	Real-time delay
<code>vdelay_list</code>	Select delay from table based on real-time index

### `create_offset_list`

### Create table of frequency offsets

**Applicability:** UNITY<sup>INNOVA</sup> systems.

**Syntax:** `create_freq_list(list,nvals,device,list_number)`

```

create_offset_list(list,nvals,device,list_number)

double *list;      /* pointer to list of
frequency offsets */

int nvals;         /* number of values in list
*/

int device;        /* OBSch, DECch, DEC2ch, or
DEC3ch */

int list_number;  /* number 0-255 for each
list */

```

**Description:** Stores global lists of frequencies that can be accessed with a real-time variable or table element for dynamic setting of frequency offsets. Offset lists define lists of frequency offsets in Hz (such as from *tof*, *dof*). Imaging pulse sequences typically use offset lists, not frequency lists. The lists need to be created in order starting from 0 using the *list\_number* argument, or by setting the *list\_number* argument to -1, which makes the software allocate and create the next free list and give the list number as a return value. Each list must have a unique and sequential *list\_number*. There can be a maximum of 256 lists depending on the size of the lists. The lists are stored in data memory and compete for space with the acquisition data for each array element. If a list is created, the return value is the number of the list (0 to 255); if an error occurs, the return value is negative.

*create\_offset\_list* creates what is called a global list. Global lists are different from AP tables in that the lists are sent down to the acquisition console when the experiment starts up and are accessible until the experiment completes. In working with arrayed experiments, be careful when using a -1 in the *list\_number* argument because a list will be created for *each* array element. In this case, a list parameter can be created as an arrayed parameter with protection bit 8 (256) set. To read in the values of this type of parameter, use the *getarray* statement. To ensure that the list is only created once, check the global array counter variable *ix*, and only call *create\_offset\_list* to create the list when it equals 1. An example is shown in the entry for the *create\_delay\_list* statement.

**Arguments:** *list* is a pointer to a list of frequency offsets.  
*nvals* is the number of values in the list.  
*device* is OBSch (observe transmitter), DECch (first decoupler), DEC2ch (second decoupler), or DEC3ch (third decoupler).



`list_number` is -1 or a unique number from 0 to 255 for each list created.

Examples: See the example for the `create_delay_list` statement.

### Related statements

<code>create_delay_list</code>	Create table of delays
<code>create_freq_list</code>	Create table of frequencies
<code>getarray</code>	Retrieves all values of an arrayed parameter
<code>delay</code>	Delay for a specified time
<code>voffset</code>	Select frequency offset from table

## `create_rotation_list`

### Create list of gradient-coordinate rotation angles

Syntax: `listId=create_rotation_list(name, angle_array, num_angles);`  
`char name; /* name of the rotation list */`  
`double *angle_array[3]; /* 2D angle-list array with 3 columns */`  
`int num_angles; /* number of angle sets in array */`  
`int listId; /* return value is an integer, the ID of the list created */`

Description: Create a list of oblique gradient rotation angles, identified by the integer `listId`, to set the values of `psi`, `theta`, and `phi` for oblique gradients. The gradient rotation angles rotate from the logical gradient axes `read`, `phase`, and `slice` to the gradient axes X, Y and Z. The input argument `angle_array` is a two-dimensional array of *doubles* whose three columns are `psi`, `theta`, and `phi`, respectively and whose rows are the number of rotation-angle sets.

Arguments: `name` is a string variable that sets the name of the list of angles

`angle_array` is a 2D array with three columns to hold the Euler rotation angle `psi`, `theta`, and `phi`, and as many rows as the number of angles in the list. The declaration `angle_array[128][3]` creates an array to hold 128 sets of three angles.

`num_angles` is an integer specifying the number of sets of rotation angle in the array.

The return `listId` is an integer identifying the rotation list. The first list is identified as 0, the second 1, *etc.* `listId` is an argument to the `rot_angle_list` statement.

Related:	<code>rotate</code>	Set standard oblique gradient-coordinate rotation angles
	<code>rot_angle</code>	Set user-defined oblique gradient-coordinate rotation angles
	<code>rot_angle_list</code>	Set gradient-coordinate rotation angles from a list

## D

dbl	Double real-time integer value
dcplrphase	Set small-angle phase of first decoupler
dcplr2phase	Set small-angle phase of second decoupler
dcplr3phase	Set small-angle phase of third decoupler
dcplr4phase	Set small-angle phase of fourth decoupler
decblank	Blank amplifier of first decoupler
dec2blank	Blank amplifier of second decoupler
dec3blank	Blank amplifier of third decoupler
dec4blank	Blank amplifier of fourth decoupler
decoff	Turn off first decoupler
dec2off	Turn off second decoupler
dec3off	Turn off third decoupler
dec4off	Turn off fourth decoupler
decoffset	Set frequency offset of first decoupler
dec2offset	Set frequency offset of second decoupler
dec3offset	Set frequency offset of third decoupler
dec4offset	Set frequency offset of fourth decoupler
decon	Turn on first decoupler
dec2on	Turn on second decoupler
dec3on	Turn on third decoupler
dec4on	Turn on fourth decoupler
decphase	Set quadrature phase of first decoupler
dec2phase	Set quadrature phase of second decoupler
dec3phase	Set quadrature phase of third decoupler
dec4phase	Set quadrature phase of fourth decoupler
decpower	Set power level of first decoupler
dec2power	Set power level of second decoupler
dec3power	Set power level of third decoupler
dec4power	Set power level of fourth decoupler
decprgoff	End waveform decoupling on first decoupler
dec2prgoff	End waveform decoupling on second decoupler
dec3prgoff	End waveform decoupling on third decoupler
dec4prgoff	End waveform decoupling on fourth decoupler
decprgon	Start waveform decoupling on first decoupler
dec2prgon	Start waveform decoupling on second decoupler
dec3prgon	Start waveform decoupling on third decoupler
dec4prgon	Start waveform decoupling on fourth decoupler
decprgonOffset	Start waveform decoupling on first decoupler with offset
dec2prgonOffset	Start waveform decoupling on second decoupler with offset
dec3prgonOffset	Start waveform decoupling on third decoupler with offset
dec4prgonOffset	Start waveform decoupling on fourth decoupler with offset
decpulse	Perform pulse with assigned values on first decoupler
decpwrf	Set fine power level of first decoupler
dec2pwrf	Set fine power level of second decoupler
dec3pwrf	Set fine power level of thrd decoupler
dec4pwrf	Set fine power level of fourth decoupler
decr	Decrement real-time integer value
decrpulse	Perform pulse on first decoupler
dec2rgpulse	Perform pulse on second decoupler
dec3rgpulse	Perform pulse on third decoupler
dec4rgpulse	Perform pulse on fourth decoupler
decshaped_pulse	Perform shaped pulse on first decoupler
dec2shaped_pulse	Perform shaped pulse on second decoupler
dec3shaped_pulse	Perform shaped pulse on third decoupler
dec4shaped_pulse	Perform shaped pulse on fourth decoupler
decspinlock	Perform waveform spinlock on first decoupler
dec2spinlock	Perform waveform spinlock on second decoupler
dec3spinlock	Perform waveform spinlock on third decoupler

dec4spinlock	Perform waveform spinlock on fourth decoupler
decstepsize	Set small-angle phase stepsize for first decoupler
dec2stepsize	Set small-angle phase stepsize for second decoupler
dec3stepsize	Set small-angle phase stepsize for third decoupler
dec4stepsize	Set small-angle phase stepsize for fourth decoupler
decunblank	Unblank amplifier of first decoupler
dec2unblank	Unblank amplifier of second decoupler
dec3unblank	Unblank amplifier of third decoupler
dec4unblank	Unblank amplifier of fourth decoupler
delay	Execute a time delay
divn	Divide real-time integer values
dps_off	Turn off graphical display of statements
dps_on	Turn on graphical display of statements
dps_show	Display pulse and delay icons in graphical display
dps_skip	Skip graphical display of next statement

**dbl**

**Double real-time integer value**

Syntax: `dbl(vi,vj)`  
`codeint vi; /* real-time variable for`  
`input */`  
`codeint vj; /* real-time variable for`  
`output */`

Description: Set the value of *vj* equal to twice the integer value of *vi*.

Arguments: *vi* contains the value to be doubled and *vj* contains the result. Each argument must be a real-time variable (*v1* to *v42*, *oph*, *etc*).

Examples: `dbl(v1,v2);`

Related:	<code>add</code>	Add real-time integer values
	<code>assign</code>	Assign real-time integer value using real-time integer
	<code>decr</code>	Decrement real-time integer value
	<code>divn</code>	Divide real-time integer values
	<code>hlv</code>	Assign half the value of real-time integer
	<code>incr</code>	Increment real-time integer value
	<code>initval</code>	Assign real-time integer value using numeric value
	<code>mod2</code>	Assign real-time integer value modulo 2
	<code>mod4</code>	Assign real-time integer value modulo 4
	<code>modn</code>	Assign real-time integer value modulo n
	<code>mult</code>	Multiply real-time integer values
	<code>sub</code>	Subtract real-time integer values

**dcplrphase****Set small-angle phase of first decoupler**

**Syntax:** `dcplrphase(multiplier)`  
`codeint multiplier; /* real-time`  
`phase-step multiplier */`

**Description:** Set the phase as a product of multiplier and a phase stepsize, which is set in degrees by the decstepsize statement. If decstepsize has not been used, the default stepsize is 90°.

The `dcplrphase` statement sets the total phase as a sum of a quadrature part, a multiple of 90°, and a small-angle part, 0° to 90°. The small-angle phase of a DD2 MR system may have a phase resolution of 360.0/65536 (~0.0055°). VNMR5 systems may have a phase resolution of 360.0/8192 (~0.044°) depending on the available transmitter, and the small-angle phase is set to the nearest step. Consult the configuration file to determine which transmitter is present.

The `dcplrphase` statement overrides the quadrature part of the phase, set by any previous `decphase` statement. The `decphase` statement overrides only the quadrature part of the phase. Use `dcplrphase(zero)` to remove small-angle phase and return to only quadrature phases.

You should be aware that `decrpulse` can set only the quadrature phase. Set a small-angle phase cycle with `dcplrphase` before the pulse and use zero as the quadrature-phase multiplier for the pulse.

**Arguments:** `multiplier` is a small-angle phaseshift multiplier for the first decoupler. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*) or a real-time table (`t1` to `t60`).

**Examples:** `dcplrphase(t1);`  
`dcplrphase(zero);`

<b>Related:</b>	<code>dcplr2phase</code>	Set small-angle phase of second decoupler
	<code>dcplr3phase</code>	Set small-angle phase of third decoupler
	<code>dcplr4phase</code>	Set small-angle phase of fourth decoupler
	<code>decphase</code>	Set quadrature phase of first decoupler
	<code>decstepsize</code>	Set small-angle phase stepsize of first decoupler
	<code>obsstepsize</code>	Set small-angle phase stepsize of observe channel
	<code>txphase</code>	Set quadrature phase of observe channel

`xmtrphase` Set small-angle phase of observe channel

**dcplr2phase****Set small-angle phase of second decoupler**

**Syntax:** `dcplr2phase(multiplier)`  
`codeint multiplier; /* real-time`  
`phase-step multiplier */`

**Description:** Set the phase as a product of multiplier and a phase stepsize, which is set in degrees by the `dec2stepsize` statement. If `dec2stepsize` has not been used, the default stepsize is 90°.

The `dcplr2phase` statement sets the total phase as a sum of a quadrature part, a multiple of 90°, and a small-angle part, 0° to 90°. The small-angle phase of a DD2 MR system may have a phase resolution of 360.0/65536 (~0.0055°). VNMR5 systems may have a phase resolution of 360.0/8192 (~0.044°) depending on the available transmitter, and the small-angle phase is set to the nearest step. Consult the configuration file to determine which transmitter is present.

The `dcplr2phase` statement overrides the quadrature part of the phase, set by any previous `decphase` statement. The `dec2phase` statement overrides only the quadrature part of the phase. Use `dcplr2phase(zero)` to remove small-angle phase and return to only quadrature phases.

You should be aware that `dec2rgpulse` can set only the quadrature phase. Set a small-angle phase cycle with `dcplr2phase` before the pulse and use `zero` as the quadrature-phase multiplier for the pulse.

**Arguments:** `multiplier` is a small-angle phaseshift multiplier for the first decoupler. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*) or a real-time table (`t1` to `t60`).

**Examples:** `dcplr2phase(t1);`  
`dcplr2phase(zero);`

**Related:** `dcplrphase` Set small-angle phase of first decoupler  
`dcplr3phase` Set small-angle phase of third decoupler  
`dcplr4phase` Set small-angle phase of fourth decoupler  
`dec2phase` Set quadrature phase of second decoupler  
`dec2stepsize` Set small-angle phase stepsize of second decoupler

obsstepsize	Set small-angle phase stepsize of observe channel
txphase	Set quadrature phase of observe channel
xmtrphase	Set small-angle phase of observe channel

**dcplr3phase****Set small-angle phase of third decoupler**

**Syntax:** `dcplr3phase(multiplier)`  
`codeint multiplier; /* real-time`  
`phase-step multiplier */`

**Description:** Set the phase as a product of multiplier and a phase stepsize, which is set in degrees by the `dec3stepsize` statement. If `dec3stepsize` has not been used, the default stepsize is 90°.

The `dcplr3phase` statement sets the total phase as a sum of a quadrature part, a multiple of 90°, and a small-angle part, 0° to 90°. The small-angle phase of the DD2 MR system has a phase resolution of 360.0/65536 (~0.0055°) and the VNMRS system has a phase resolution of 360.0/8192 (~0.044°) depending on the available transmitter, and the small-angle phase is set to the nearest step. Consult the configuration file to determine which transmitter is present.

The `dcplr3phase` statement overrides the quadrature part of the phase, set by any previous `dec3phase` statement. The `dec3phase` statement overrides only the quadrature part of the phase. Use `dcplr3phase(zero)` to remove small-angle phase and return to only quadrature phases.

You should be aware that `dec3rgpulse` can set only the quadrature phase. Set a small-angle phase cycle with `dcplr3phase` before the pulse and use `zero` as the quadrature-phase multiplier for the pulse.

**Arguments:** `multiplier` is a small-angle phaseshift multiplier for the first decoupler. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*) or a real-time table (`t1` to `t60`).

**Examples:** `dcplr3phase(t1);`  
`dcplr3phase(zero);`

<b>Related:</b>	<code>dcplrphase</code>	Set small-angle phase of first decoupler
	<code>dcplr2phase</code>	Set small-angle phase of second decoupler
	<code>dcplr4phase</code>	Set small-angle phase of fourth decoupler

dec3phase	Set quadrature phase of third decoupler
dec3stepsize	Set small-angle phase stepsize of third decoupler
obsstepsize	Set small-angle phase stepsize of observe channel
txphase	Set quadrature phase of observe channel
xmtrphase	Set small-angle phase of observe channel

**dcplr4phase****Set small-angle phase of fourth decoupler**

**Syntax:** `dcplr4phase(multiplier)`  
`codeint multiplier; /* real-time`  
`phase-step multiplier */`

**Description:** Set the phase as a product of multiplier and a phase stepsize, which is set in degrees by the `dec4stepsize` statement. If `dec4stepsize` has not been used, the default stepsize is 90°.

The `dcplr4phase` statement sets the total phase as a sum of a quadrature part, a multiple of 90°, and a small-angle part, 0° to 90°. The small-angle phase of the DD2 MR system has a phase resolution of 360.0/65536 (~0.0055°) and the VNMR5 system has a phase resolution of 360.0/8192 (~0.044°) depending on the available transmitter, and the small-angle phase is set to the nearest step. Consult the configuration file to determine which transmitter is present.

The `dcplr4phase` statement overrides the quadrature part of the phase, set by any previous `dec4phase` statement. The `dec4phase` statement overrides only the quadrature part of the phase. Use `dcplr4phase(zero)` to remove small-angle phase and return to only quadrature phases.

You should be aware that `dec4rgpulse` can set only the quadrature phase. Set a small-angle phase cycle with `dcplr4phase` before the pulse and use `zero` as the quadrature-phase multiplier for the pulse.

**Arguments:** `multiplier` is a small-angle phaseshift multiplier for the first decoupler. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*) or a real-time table (`t1` to `t60`).

**Examples:** `dcplrphase(t1);`  
`dcplr4phase(zero);`

**Related:** `dcplrphase` Set small-angle phase of first decoupler

`dcplr2phase` Set small-angle phase of second decoupler



dcplr3phase	Set small-angle phase of third decoupler
dec4phase	Set quadrature phase of fourth decoupler
dec4stepsize	Set small-angle phase stepsize of fourth decoupler
obsstepsize	Set small-angle phase stepsize of observe channel
txphase	Set quadrature phase of observe channel
xmtrphase	Set small-angle phase of observe channel

**decblank****Blank amplifier of first decoupler**

Syntax: `decblank()`

Disable the amplifier for the first decoupler if the amplifier is in pulse mode. `decblank` has no effect if the amplifier is in continuous mode. The first decoupler is in continuous mode by default unless it has an associated receiver or is in the frequency band of another observe channel, for which it is in pulse mode.

`decblank` is usually used before acquisition or between pulses and decoupling periods to suppress amplifier noise. It must be used before acquisition if the first decoupler channel has an associated receiver.

To place the amplifier of the first decoupler in pulse mode, create the parameter `ampmode` and set the appropriate character to 'p'.

Related:	<code>dec2blank</code>	Blank amplifier of second decoupler
	<code>dec3blank</code>	Blank amplifier of third decoupler
	<code>dec4blank</code>	Blank amplifier of fourth decoupler
	<code>decunblank</code>	Unblank amplifier of first decoupler
	<code>obsblank</code>	Blank amplifier of observe channel
	<code>obsunblank</code>	Unblank amplifier of observe channel

**dec2blank****Blank amplifier of second decoupler**

Syntax: `dec2blank()`

Description: Disables the amplifier for the second decoupler if the amplifier is in pulse mode. `dec2blank` has no effect if the amplifier is in continuous mode. The second decoupler is in continuous mode by default

unless it has an associated receiver or is in the frequency band of another observe channel, for which it is in pulse mode.

`dec2blank` is generally used before acquisition or between pulses and decoupling periods to suppress amplifier noise. It must be used before acquisition if the second decoupler channel has an associated receiver.

To place the amplifier of the second decoupler in pulse mode, create the parameter `ampmode` and set the appropriate character to 'p'.

Related:	<code>decblank</code>	Blank amplifier of first decoupler
	<code>dec3blank</code>	Blank amplifier of third decoupler
	<code>dec4blank</code>	Blank amplifier of fourth decoupler
	<code>dec2unblank</code>	Unblank amplifier of second decoupler
	<code>obsblank</code>	Blank amplifier of observe channel
	<code>obsunblank</code>	Unblank amplifier of observe channel

**dec3blank**

**Blank amplifier of third decoupler**

Syntax: `dec3blank()`

Description: Disables the amplifier for the third decoupler if the amplifier is in pulse mode. `dec3blank` has no effect if the amplifier is in continuous mode. The third decoupler is in continuous mode by default unless it has an associated receiver or is in the frequency band of another observe channel, for which it is in pulse mode.

`dec3blank` is generally used before acquisition or between pulses and decoupling periods to suppress amplifier noise. It must be used before acquisition if the third decoupler channel has an associated receiver.

To place the amplifier of the third decoupler in pulse mode, create the parameter `ampmode` and set the appropriate character to 'p'.

Related:	<code>decblank</code>	Blank amplifier of first decoupler
	<code>dec2blank</code>	Blank amplifier of second decoupler
	<code>dec4blank</code>	Blank amplifier of fourth decoupler

dec3unblank	Unblank amplifier of third decoupler
obsblank	Blank amplifier of observe channel
obsunblank	Unblank amplifier of observe channel

**dec4blank****Blank amplifier of fourth decoupler**

Syntax: `dec4blank()`

Description: Disables the amplifier for the fourth decoupler if the amplifier is in pulse mode. `dec4blank` has no effect if the amplifier is in continuous mode. The fourth decoupler is in continuous mode by default unless it has an associated receiver or is in the frequency band of another observe channel, for which it is in pulse mode.

`dec4blank` is generally used before acquisition or between pulses and decoupling periods to suppress amplifier noise. It must be used before acquisition if the fourth decoupler channel has an associated receiver.

To place the amplifier of the fourth decoupler in pulse mode, create the parameter `ampmode` and set the appropriate character to 'p'.

Related:	<code>decblank</code>	Blank amplifier of first decoupler
	<code>dec2blank</code>	Blank amplifier of second decoupler
	<code>dec4blank</code>	Blank amplifier of fourth decoupler
	<code>dec3unblank</code>	Unblank amplifier of third decoupler
	<code>obsblank</code>	Blank amplifier of observe channel
	<code>obsunblank</code>	Unblank amplifier of observe channel

**decoff****Turn off first decoupler**

Syntax: `decoff()`

Description: Explicitly gate off the first decoupler in the pulse sequence. Amplifier blanking state is unchanged.

Related:	<code>dec2off</code>	Turn off second decoupler
	<code>dec3off</code>	Turn off third decoupler
	<code>dec4off</code>	Turn off fourth decoupler
	<code>decon</code>	Turn on first decoupler
	<code>xmtroff</code>	Turn off observe channel
	<code>xmtron</code>	Turn on observe channel

#### **dec2off**

##### **Turn off second decoupler**

Syntax: `dec2off()`

Description: Explicitly gate off the second decoupler in the pulse sequence. Amplifier blanking state is unchanged.

Related:

<code>decoff</code>	Turn off first decoupler
<code>dec3off</code>	Turn off third decoupler
<code>dec4off</code>	Turn off fourth decoupler
<code>dec2on</code>	Turn on second decoupler
<code>xmtroff</code>	Turn off observe channel
<code>xmtron</code>	Turn on observe channel

#### **dec3off**

##### **Turn off third decoupler**

Syntax: `dec3off()`

Description: Explicitly gate off the third decoupler in the pulse sequence. Amplifier blanking state is unchanged.

Related:

<code>decoff</code>	Turn off first decoupler
<code>dec2off</code>	Turn off second decoupler
<code>dec4off</code>	Turn off fourth decoupler
<code>dec3on</code>	Turn on third decoupler
<code>xmtroff</code>	Turn off observe channel
<code>xmtron</code>	Turn on observe channel

#### **dec4off**

##### **Turn off fourth decoupler**

Syntax: `dec4off()`

Description: Explicitly gate off the fourth decoupler in the pulse sequence. Amplifier blanking state is unchanged.

Related:

<code>decoff</code>	Turn off first decoupler
<code>dec2off</code>	Turn off second decoupler
<code>dec3off</code>	Turn off third decoupler
<code>dec4on</code>	Turn on fourth decoupler
<code>xmtroff</code>	Turn off observe channel
<code>xmtron</code>	Turn on observe channel

#### **decoffset**

##### **Set frequency offset of first decoupler**

Syntax: `decoffset(frequency)`  
`double frequency; /* frequency offset, Hz */`

Description: Set the frequency offset of the first decoupler. The offset of the first decoupler is initialized to dof before the first `decoffset` statement is applied. You

must explicitly use `decoffset(dof)` to return the offset to `dof`.

The `decoffset` statement sets the VNMRS synthesizer frequency, which simultaneously determines the base frequency of pulses and the center frequency of the receiver if it is present. You should use caution in resetting the frequency offset. Incorrect use of `decoffset` can influence the phase coherence of the receiver and pulses scan-to-scan.

The `decoffset` statement inserts a 50 ns delay into the pulse sequence. The VNMRS synthesizer can take several microseconds to set. It may be necessary to compensate for these delays in the pulse sequence.

**Arguments:** `frequency` is the desired frequency offset, in Hz.

**Examples:** `decoffset(newoffset);`  
`decoffset(dof);`

<b>Related:</b>	<code>dec2offset</code>	Set frequency offset of second decoupler
	<code>dec3offset</code>	Set frequency offset of third decoupler
	<code>dec4offset</code>	Set frequency offset of fourth decoupler
	<code>obsoffset</code>	Set frequency offset of observe channel
	<code>offset</code>	Set frequency offset of any channel

## **dec2offset**

### **Set frequency offset of second decoupler**

**Syntax:** `dec2offset(frequency)`  
`double frequency; /* frequency offset, Hz */`

**Description:** Set the frequency offset of the second decoupler. The offset of the second decoupler is initialized to `dof2` before the first `dec2offset` statement is applied. You must explicitly use `dec2offset(dof2)` to return the offset to `dof2`.

The `dec2offset` statement sets the VNMRS synthesizer frequency, which simultaneously determines the base frequency of pulses and the center frequency of the receiver if it is present. You should use caution in resetting the frequency offset. Incorrect use of `dec2offset` can influence the phase coherence of the receiver and pulses scan-to-scan.

The `dec2offset` statement inserts a 50 ns delay into the pulse sequence. The VNMRS synthesizer can take several microseconds to set. It may be

necessary to compensate for these delays in the pulse sequence.

Arguments: frequency is the desired frequency offset, in Hz.

Examples: `dec2offset(newoffset);`  
`dec2offset(dof2);`

Related:	<code>decoffset</code>	Set frequency offset of first decoupler
	<code>dec3offset</code>	Set frequency offset of third decoupler
	<code>dec4offset</code>	Set frequency offset of fourth decoupler
	<code>obsoffset</code>	Set frequency offset of observe channel
	<code>offset</code>	Set frequency offset of any channel

**dec3offset**

**Set frequency offset of third decoupler**

Syntax: `dec3offset(frequency)`  
`double frequency; /* frequency offset in Hz */`

Description: Set the frequency offset of the third decoupler. The offset of the third decoupler is initialized to `dof3` before the first `dec3offset` statement is applied. You must explicitly use `dec3offset(dof3)` to return the offset to `dof3`.

The `dec3offset` statement sets the VNMR5 synthesizer frequency, which simultaneously determines the base frequency of pulses and the center frequency of the receiver if it is present. You should use caution in resetting the frequency offset. Incorrect use of `dec3offset` can influence the phase coherence of the receiver and pulses scan-to-scan.

The `dec3offset` statement inserts a 50 ns delay into the pulse sequence. The VNMR5 synthesizer can take several microseconds to set. It may be necessary to compensate for these delays in the pulse sequence.

Arguments: frequency is the desired frequency offset, in Hz.

Examples: `dec3offset(newoffset);`  
`dec3offset(dof3);`

Related:	<code>decoffset</code>	Set frequency offset of first decoupler
	<code>dec2offset</code>	Set frequency offset of second decoupler
	<code>dec4offset</code>	Set frequency offset of fourth decoupler

obsoffset	Set frequency offset of observe channel
offset	Set frequency offset of any channel

**dec4offset****Set frequency offset of fourth decoupler**

**Syntax:** `dec4offset(frequency)`  
`double frequency; /* frequency offset, Hz */`

**Description:** Set the frequency offset of the fourth decoupler. The offset of the fourth decoupler is initialized to `dof4` before the first `dec4offset` statement is applied. You must explicitly use `dec4offset(dof4)` to return the offset to `dof4`.

The `dec4offset` statement sets the VNMR5 synthesizer frequency, which simultaneously determines the base frequency of pulses and the center frequency of the receiver if it is present. You should use caution in resetting the frequency offset. Incorrect use of `dec4offset` can influence the phase coherence of the receiver and pulses scan-to-scan.

The `dec4offset` statement inserts a 50 ns delay into the pulse sequence. The VNMR5 synthesizer can take several microseconds to set. It may be necessary to compensate for these delays in the pulse sequence.

**Arguments:** `frequency` is the desired frequency offset, in Hz.

**Examples:** `dec4offset(newoffset);`  
`dec4offset(dof4);`

<b>Related:</b>	<code>decoffset</code>	Set frequency offset of first decoupler
	<code>dec2offset</code>	Set frequency offset of second decoupler
	<code>dec3offset</code>	Set frequency offset of third decoupler
	<code>obsoffset</code>	Set frequency offset of observe channel
	<code>offset</code>	Set frequency offset of any channel

**decon****Turn on first decoupler**

**Syntax:** `decon()`

**Description:** Explicitly gate on the first decoupler in the pulse sequence outside of a pulse. Amplifier blanking state is unchanged. The associated amplifier must be previously unblanked with `decunblank`, at least

2.0  $\mu$ s before decon. Follow decon with a delay, decoff, and an optional decblank.

Related:    decoff      Turn off first decoupler  
             dec2on      Turn on second decoupler  
             dec3on      Turn on third decoupler  
             dec4on      Turn on fourth decoupler  
             xmtroff     Turn off observe channel  
             xmtron      Turn on observe channel

#### **dec2on**

#### **Turn on second decoupler**

Syntax:    dec2on()

Description: Explicitly gate on the second decoupler in the pulse sequence outside of a pulse. Amplifier blanking state is unchanged. The associated amplifier must be previously unblanked with dec2unblank, at least 2.0  $\mu$ s before dec2on. Follow dec2on with a delay, dec2off, and an optional dec2blank.

Related:    dec2off     Turn off second decoupler  
             decon        Turn on first decoupler  
             dec3on      Turn on third decoupler  
             dec4on      Turn on fourth decoupler  
             xmtroff     Turn off observe channel  
             xmtron      Turn on observe channel

#### **dec3on**

#### **Turn on third decoupler**

Syntax:    dec3on()

Description: Explicitly gate on the third decoupler in the pulse sequence outside of a pulse. Amplifier blanking state is unchanged. The associated amplifier must be previously unblanked with dec3unblank, at least 2.0  $\mu$ s before dec3on. Follow dec3on with a delay, dec3off, and an optional dec3blank.

Related:    dec3off     Turn off third decoupler  
             decon        Turn on first decoupler  
             dec2on      Turn on second decoupler  
             dec4on      Turn on fourth decoupler  
             xmtroff     Turn off observe channel  
             xmtron      Turn on observe channel



**dec4on****Turn on fourth decoupler**

Syntax: `dec4on()`

Description: Explicitly gate on the first decoupler in the pulse sequence outside of a pulse. Amplifier blanking state is unchanged. The associated amplifier must be previously unblanked with `dec4unblank`, at least 2.0  $\mu$ s before `dec4on`. Follow `dec4on` with a `delay`, `dec4off`, and an optional `dec4blank`.

Related:	<code>dec4off</code>	Turn off fourth decoupler
	<code>decon</code>	Turn on first decoupler
	<code>dec2on</code>	Turn on second decoupler
	<code>dec3on</code>	Turn on third decoupler
	<code>xmtroff</code>	Turn off observe channel
	<code>xmtron</code>	Turn on observe channel

**decphase****Set quadrature phase of first decoupler**

Syntax: `decphase(multiplier)`  
`codeint phase; /*real-time quadrature-phase multiplier*/`

Description: Explicitly set quadrature phase (multiple of 90°) for the first decoupler outside of a pulse. If a small-angle phase has been set previously with `dcplrphase`, `decphase` adjusts only the portion of the phase that is a multiple of 90°. Use `dcplrphase(zero)` to clear any small-angle phase if required.

Arguments: `multiplier` is a 90° multiplier for the first-decoupler phase. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*), or a real-time table (`t1` to `t60`).

Examples: `decphase(v4);`  
`decphase(t2);`

Related:	<code>dcplrphase</code>	Set small-angle phase of first decoupler
	<code>dec2phase</code>	Set quadrature phase of second decoupler
	<code>dec3phase</code>	Set quadrature phase of third decoupler
	<code>dec4phase</code>	Set quadrature phase of fourth decoupler
	<code>txphase</code>	Set quadrature phase of observe channel

**dec2phase****Set quadrature phase of second decoupler**

Syntax: `dec2phase(multiplier)`  
`codeint phase; /*real-time quadrature-phase multiplier */`

Description: Explicitly set quadrature phase (multiple of 90°) for the first decoupler outside of a pulse. If a small-angle phase has been set previously with `dcplr2phase`, `dec2phase` adjusts only the portion of the phase that is a multiple of 90°. Use `dcplr2phase(zero)` to clear any small-angle phase if required.

Arguments: `multiplier` is a 90° multiplier for the first-decoupler phase. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*), or a real-time table (`t1` to `t60`).

Examples: `dec2phase(v4);`  
`dec2phase(t2);`

Related:	<code>dcplr2phase</code>	Set small-angle phase of second decoupler
	<code>decphase</code>	Set quadrature phase of first decoupler
	<code>dec3phase</code>	Set quadrature phase of third decoupler
	<code>dec4phase</code>	Set quadrature phase of fourth decoupler
	<code>txphase</code>	Set quadrature phase of observe channel

**dec3phase****Set quadrature phase of third decoupler**

Syntax: `dec3phase(multiplier)`  
`codeint phase; /*real-time quadrature-phase multiplier */`

Description: Explicitly set quadrature phase (multiple of 90°) for the first decoupler outside of a pulse. If a small-angle phase has been set previously with `dcplr3phase`, `dec3phase` adjusts only the portion of the phase that is a multiple of 90°. Use `dcplr3phase(zero)` to clear any small-angle phase if desired.

Arguments: `multiplier` is a 90° multiplier for the first-decoupler phase. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*), or a real-time table (`t1` to `t60`).

Examples: `dec3phase(v4);`  
`dec3phase(t2);`

Related:	dcplr3phase	Set small-angle phase of third decoupler
	decphase	Set quadrature phase of first decoupler
	dec2phase	Set quadrature phase of third decoupler
	dec4phase	Set quadrature phase of fourth decoupler
	txphase	Set quadrature phase of observe channel

**dec4phase****Set quadrature phase of fourth decoupler**

Syntax: `dec4phase(multiplier)`  
`codeint phase; /*real-time quadrature-phase multiplier*/`

Description: Explicitly set quadrature phase (multiple of 90°) for the first decoupler outside of a pulse. If a small-angle phase has been set previously with `dcplr4phase`, `dec4phase` adjusts only the portion of the phase that is a multiple of 90°. Use `dcplr4phase(zero)` to clear any small-angle phase if desired.

Arguments: `multiplier` is a 90° multiplier for the first-decoupler phase. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*), or a real-time table (`t1` to `t60`).

Examples: `dec4phase(v4);dec4phase(t2);`

Related:	dcplr4phase	Set small-angle phase of fourth decoupler
	decphase	Set quadrature phase of first decoupler
	dec2phase	Set quadrature phase of third decoupler
	dec3phase	Set quadrature phase of third decoupler
	txphase	Set quadrature phase of observe channel

**decpower****Set power level of first decoupler**

Syntax: `decpower(power)`  
`double power; /* power-level value, dB */`

Description: Set the power level of the first decoupler using the coarse attenuator. The power level of the first decoupler is initialized to `dpwr` before the first

`decpower` statement is applied. You must explicitly use `decpower(dpwr)` to return the power level to `dpwr`.

The `decpower` statement inserts a delay of 50 ns into the pulse sequence and you should allow 3 μs for the power to reach the new level during the next delay. The coarse attenuator can introduce a transient when it changes and it is good practice to execute `decpower` only when the transmitter is blanked and gated off.

**Arguments:** `power` sets the coarse attenuator in 0.5 dB steps from a maximum power of 63 to a minimum of -37 if the 100 db attenuator is present. The stepsize is truncated to 1.0 dB and the minimum is -16 dB if the 79 dB attenuator is present. Consult the configuration file to determine the available attenuator.

<b>Related:</b>	<code>dec2power</code>	Set power level of second decoupler
	<code>dec3power</code>	Set power level of third decoupler
	<code>dec4power</code>	Set power level of fourth decoupler
	<code>obspower</code>	Set power level of observe channel
	<code>rlpower</code>	Set the power level of any channel

**dec2power**

**Set power level of second decoupler**

**Syntax:** `dec2power(power)`  
`double power; /* power-level value, dB */`

**Description:** Set the power level of the first decoupler using the coarse attenuator. The power level of the first decoupler is initialized to `dpwr2` before the first `dec2power` statement is applied. You must explicitly use `dec2power(dpwr2)` to return the power level to `dpwr2`.

The `dec2power` statement inserts a delay of 50 ns into the pulse sequence and you should allow 3 μs for the power to reach the new level during the next delay. The coarse attenuator can introduce a transient when it changes and it is good practice to execute `dec2power` only when the transmitter is blanked and gated off.

**Arguments:** `power` sets the coarse attenuator in 0.5 dB steps from a maximum power of 63 to a minimum of -37 if the 100 db attenuator is present. The stepsize is truncated to 1.0 dB and the minimum is -16 dB if the 79 dB attenuator is present. Consult the configuration file to determine the available attenuator.

Related:	<code>decpower</code>	Set power level of first decoupler
	<code>dec3power</code>	Set power level of third decoupler
	<code>dec4power</code>	Set power level of fourth decoupler
	<code>obspower</code>	Set power level of observe channel
	<code>rlpower</code>	Set the power level of any channel

**dec3power****Set power level of third decoupler**

Syntax: `decpower(power)`  
`double power; /* power-level value, dB*/`

Description: Set the power level of the third decoupler using the coarse attenuator. The power level of the third decoupler is initialized to `dpwr3` before the first `dec3power` statement is applied. You must explicitly use `dec3power(dpwr3)` to return the power level to `dpwr3`.

The `dec3power` statement inserts 50 ns into the pulse sequence and you should allow 3  $\mu$ s for the power to reach the new level during the next delay. The coarse attenuator can introduce a transient when it changes and it is good practice to execute `dec3power` only when the transmitter is blanked and gated off.

Arguments: `power` sets the coarse attenuator in 0.5 dB steps from a maximum power of 63 to a minimum of -37 if the 100 dB attenuator is present. The stepsize is truncated to 1.0 dB and the minimum is -16 dB if the 79 dB attenuator is present. Consult the configuration file to determine the available attenuator.

Related:	<code>decpower</code>	Set power level of first decoupler
	<code>dec2power</code>	Set power level of second decoupler
	<code>dec4power</code>	Set power level of fourth decoupler
	<code>obspower</code>	Set power level of observe channel
	<code>rlpower</code>	Set the power level of any channel

**dec4power****Set power level of fourth decoupler**

Syntax: `dec4power(power)`  
`double power; /* power-level value, dB*/`

Description: Set the power level of the first decoupler using the coarse attenuator. The power level of the first decoupler is initialized to `dpwr4` before the first

dec4power statement is applied. You must explicitly use dec4power(dpwr4) to return the power level to dpwr4.

The dec4power statement inserts 50 ns into the pulse sequence and you should allow 3 s for the power to reach the new level during the next delay. The coarse attenuator can introduce a transient when it changes and it is good practice to execute dec4power only when the transmitter is blanked and gated off.

**Arguments:** power sets the coarse attenuator in 0.5 dB steps from a maximum power of 63 to a minimum of -37 if the 100 db attenuator is present. The stepsize is truncated to 1.0 dB and the minimum is -16 dB if the 79 dB attenuator is present. Consult the configuration file to determine the available attenuator.

<b>Related:</b>	decpower	Set power level of first decoupler
	dec2power	Set power level of second decoupler
	dec3power	Set power level of third decoupler
	obspower	Set power level of observe channel
	rlpower	Set the power level of any channel

**decprgoff**

**End waveform decoupling on first decoupler**

**Syntax:** decprgoff()

**Description:** Terminate programmable waveform decoupling on the first decoupler, gate the first decoupler off, and blank the associated amplifier if it is in pulse mode.

<b>Related:</b>	decblank	Blank amplifier of first decoupler
	dec2prgoff	End waveform decoupling on second decoupler
	dec3prgoff	End waveform decoupling on third decoupler
	dec4prgoff	End waveform decoupling on fourth decoupler
	decprgon	Start waveform decoupling on first decoupler
	decprgonOffset	Start waveform decoupling on first decoupler with offset
	obsprgoff	End waveform decoupling on observe channel

obsprgon	Start waveform decoupling on observe channel
obsprgonOffset	Start waveform decoupling on observe channel with offset

**dec2prgoff****End waveform decoupling on second decoupler**

Syntax: `dec2prgoff()`

Description: Terminate programmable waveform decoupling on the second decoupler, gate the second decoupler off, and blank the associated amplifier if it is in pulse mode.

Related: <code>dec2blank</code>	Blank amplifier of second decoupler
<code>Decprgoff</code>	End waveform decoupling on first decoupler
<code>dec3prgoff</code>	End waveform decoupling on third decoupler
<code>dec4prgoff</code>	End waveform decoupling on fourth decoupler
<code>dec2prgon</code>	Start waveform decoupling on second decoupler
<code>dec2prgonOffset</code>	Start waveform decoupling on second decoupler with offset
<code>obsprogoff</code>	End waveform decoupling on observe channel
<code>obsprgon</code>	Start waveform decoupling on observe channel
<code>obsprgonOffset</code>	Start waveform decoupling on observe channel with offset

**dec3prgoff****End waveform decoupling on third decoupler**

Syntax: `dec3prgoff()`

Description: Terminate programmable waveform decoupling on the third decoupler, gate the third decoupler off, and blank the associated amplifier if it is in pulse mode.

Related: <code>dec3blank</code>	Blank amplifier of third decoupler
<code>decprgoff</code>	End waveform decoupling on first decoupler

dec2prgoff	End waveform decoupling on second decoupler
dec4prgoff	End waveform decoupling on fourth decoupler
dec3prgon	Start waveform decoupling on third decoupler
dec3prgonOffset	Start waveform decoupling on third decoupler with offset
obsprogoff	End waveform decoupling on observe channel
obsprgon	Start waveform decoupling on observe channel
obsprgonOffset	Start waveform decoupling on observe channel with offset

**dec4prgoff**

**End waveform decoupling on fourth decoupler**

Syntax: dec4prgoff()

Description: Terminate programmable waveform decoupling on the fourth decoupler, gate the fourth decoupler off, and blank the associated amplifier if it is in pulse mode.

Related: dec4blank	Blank amplifier of fourth decoupler
decprgoff	End waveform decoupling on first decoupler
dec2prgoff	End waveform decoupling on second decoupler
dec3prgoff	End waveform decoupling on third decoupler
dec4prgon	Start waveform decoupling on fourth decoupler
dec4prgonOffset	Start waveform decoupling on fourth decoupler with offset
obsprgoff	End waveform decoupling on observe channel
obsprgon	Start waveform decoupling on observe channel
obsprgonOffset	Start waveform decoupling on observe channel with offset

**decprgon**

**Start waveform decoupling on first decoupler**

Syntax: decprgon(pattern, 90\_pulselength, tipangle\_resoln)  
char \*pattern; /\* name of .DEC file \*/  
double 90\_pulselength; /\* 90-degree pulse length, seconds \*/  
double tipangle\_resoln; /\* tip-angle resolution \*/



```
return int ticks          /* 12.5 ns ticks,
one cycle */
```

**Description:** Execute programmable decoupling on the first decoupler under waveform control, unblank the associated amplifier, and gate the first decoupler on for patterns without an explicit gate column. `decprgon` returns an integer with the number of 12.5-ns ticks in one cycle of the decoupling pattern. It is a good practice to unblank the associated amplifier with `decunblank`, at least 2.0  $\mu$ s before `decprgon`.

**Arguments:** `pattern` is the root name of a text file in the `shapelib` directory with a `.DEC` extension to store the decoupling pattern.

`90_pulselength` is the pulse time-duration, in seconds, for an element, a block of elements, or an integer divisor of all elements in the pattern, which is labeled with a tip-angle duration equal to 90°. Often `90_pulselength` is set equal  $1/\text{dmf}$  in which `dmf` is a step rate.

`tipangle_resoln` is the smallest common divisor of the tip-angle durations of all the elements of the pattern.

For many patterns, `90_pulselength` is the actual 90° pulse length and the elements are labeled with tip-angle durations that are multiples of 90. In this case, `tipangle_resoln` is 90.0.

For many other patterns, `90_pulselength` is set to be the duration of the minimum stepsize of which all elements are multiples. In these cases, `tipangle_resoln` is nominally 90 and elements in the pattern have tip-angle durations that are multiples of 90.

For some patterns, `90_pulselength` is the actual 90° pulse length but elements of the pattern have arbitrary flip angles that are multiples of a `tipangle_resoln`, that is less than 90° (*c.f.* 1.0°). In this case `tipangle_resoln` is the divisor (*c.f.* 1.0) and for good practice it is also a divisor of 90.0.

**Examples:**

```
decprgon("garpl",1/dmf, 1.0);
decprgon(modtype,pwx90,dres);
ticks = decprgon("waltz16",1/dmf,90.0);
```

<b>Related:</b>	<code>decprgoff</code>	End waveform decoupling on first decoupler
	<code>decprgon</code>	Start waveform decoupling on second decoupler with offset
	<code>dec2prgon</code>	Start waveform decoupling on second decoupler

dec3prgon	Start waveform decoupling on third decoupler
dec4prgon	Start waveform decoupling on fourth decoupler
decunblank	Unblank the amplifier of first decoupler
obsprogoff	End waveform decoupling on observe channel
obsprgon	Start waveform decoupling on observe channel
obsprgonOffset	Start waveform decoupling on observe channel with offset

**dec2prgon**

**Start waveform decoupling on second decoupler**

Syntax: `dec2prgon(pattern, 90_pulselength, tipangle_resoln)`  
`char *pattern; /* name of .DEC file */`  
`double 90_pulselength; /* 90-degree pulse length, seconds */`  
`double tipangle_resoln; /* tip-angle resolution */`  
`return int ticks /* 12.5 ns ticks, one cycle */`

Description: Execute programmable decoupling on the second decoupler under waveform control, unblank the associated amplifier, and gate the second decoupler on for patterns without an explicit gate column. `decprgon` returns an integer with the number of 12.5-ns ticks in one cycle of the decoupling pattern. It is a good practice to unblank the associated amplifier with `dec2unblank`, at least 2.0  $\mu$ s before `decprgon`.

Arguments: `pattern` is the root name of a text file in the `shapelib` directory with a `.DEC` extension to store the decoupling pattern.

`90_pulselength` is the pulse time-duration, in seconds, for an element, a block of elements, or an integer divisor of all elements in the pattern, which is labeled with a tip-angle duration equal to 90°. Often `90_pulselength` is set equal `1/dmf` in which `dmf` is a step rate.

`tipangle_resoln` is the smallest common divisor of the tip-angle durations of all the elements of the pattern.

For many patterns `90_pulselength` is the actual 90° pulse length and the elements are labeled with tip-angle durations that are multiples of 90. In this case `tipangle_resoln` is 90.0.

For many other patterns, `90_pulselength` is set to be the duration of the minimum stepsize of which all elements are multiples. In these cases,

`tipangle_resoln` is nominally 90 and elements in the pattern have tip-angle durations that are multiples of 90.

For some patterns, `90_pulselength` is the actual 90o pulse length but elements of the pattern have arbitrary flip angles that are multiples of a `tipangle_resoln`, that is less than 90.0° (*c.f.* 1.0°). In this case `tipangle_resoln` is the divisor (*c.f.* 1.0) and for good practice it is also a divisor of 90.0.

Examples: 

```
dec2prgon("garp1",1/dmf, 1.0);
dec2prgon(modtype,pwx90,dres);
ticks = dec2prgon("waltz16",1/dmf,90.0);
```

Related:	<code>dec2prgoff</code>	End waveform decoupling on second decoupler
	<code>decprgon</code>	Start waveform decoupling on first decoupler
	<code>dec2prgonOffset</code>	Start waveform decoupling on second decoupler
	<code>dec3prgon</code>	Start waveform decoupling on third decoupler
	<code>dec4prgon</code>	Start waveform decoupling on fourth decoupler
	<code>dec2unblank</code>	Unblank amplifier of first decoupler
	<code>obsprgoff</code>	End waveform decoupling on observe channel
	<code>obsprgon</code>	Start waveform decoupling on observe channel
	<code>obsprgonOffset</code>	Start waveform decoupling on observe channel with offset

### **dec3prgon**

#### **Start waveform decoupling on third decoupler**

Syntax: 

```
dec3prgon(pattern,90_pulselength,tipangle_resoln)
char *pattern;          /* name of .DEC file */
double 90_pulselength; /* 90-degree pulse
length, seconds */
double tipangle_resoln; /* tip-angle
resolution */
return int ticks       /* 12.5 ns ticks,
one cycle */
```

Description: Execute programmable decoupling on the first decoupler under waveform control, unblank the associated amplifier, and gate the decoupler on for patterns without an explicit gate column. `dec3prgon` returns an integer with the number of 12.5-ns ticks in one cycle of the decoupling pattern. It is a good practice to unblank the associated amplifier with `dec3unblank`, at least 2.0 μs before `dec3prgon`.

**Arguments:** `pattern` is the root name of a text file in the `shapelib` directory with a `.DEC` extension to store the decoupling pattern.

`90_pulselength` is the pulse time-duration, in seconds, for an element, a block of elements, or an integer divisor of all elements in the pattern, which is labeled with a tip-angle duration equal to 90. Often `90_pulselength` is set equal  $1/\text{dmf}$  in which `dmf` is a step rate.

`tipangle_resoln` is the smallest common divisor of the tip-angle durations of all the elements of the pattern.

For many patterns, `90_pulselength` is the actual 90° pulse length and the elements are labeled with tip-angle durations that are multiples of 90. In this case, `tipangle_resoln` is 90.0.

For many other patterns, `90_pulselength` is set to be the duration of the minimum stepsize of which all elements are multiples. In these cases, `tipangle_resoln` is nominally 90 and elements in the pattern have tip-angle durations that are multiples of 90.

For some patterns, `90_pulselength` is the actual 90° pulse length but elements of the pattern have arbitrary flip angles that are multiples of a `tipangle_resoln`, that is less than 90.0° (*c.f.* 1.0°). In this case `tipangle_resoln` is the divisor (*c.f.* 1.0) and for good practice it is also a divisor of 90.0.

**Examples:**

```
dec3prgon("garp1",1/dmf, 1.0);
dec3prgon(modtype,pwx90,dres);
ticks = dec3prgon("waltz16",1/dmf,90.0);
```

**Examples:**

<b>Related:</b>	<code>dec3prgoff</code>	End waveform decoupling on third decoupler
	<code>decprgon</code>	Start waveform decoupling on first decoupler
	<code>dec2prgon</code>	Start waveform decoupling on second decoupler
	<code>dec3prgonOffset</code>	Start waveform decoupling on third decoupler with offset
	<code>dec4prgon</code>	Start waveform decoupling on fourth decoupler
	<code>dec3unblank</code>	Unblank amplifier of third decoupler
	<code>obsprogoff</code>	End waveform decoupling on observe channel
	<code>obsprgon</code>	Start waveform decoupling on observe channel

obsprgonOffset      Start waveform decoupling  
on observe channel with  
offset

**dec4prgon****Start waveform decoupling on fourth decoupler**

**Syntax:** `dec4prgon(pattern,90_pulselength,tipangle_resoln)`  
`char *pattern;            /* name of .DEC file */`  
`double 90_pulselength;    /* 90-degree pulse`  
`length, seconds */`  
`double tipangle_resoln;   /* tip-angle`  
`resolution */`  
`return int ticks         /* 12.5 ns ticks,`  
`one cycle */`

**Description:** Execute programmable decoupling on the first decoupler under waveform control, unblank the associated amplifier, and gate the decoupler on for patterns without an explicit gate column. `dec4prgon` returns an integer with the number of 12.5-ns ticks in one cycle of the decoupling pattern. It is a good practice to unblank the associated amplifier with `dec4unblank`, at least 2.0  $\mu$ s before `dec4prgon`.

**Arguments:** `pattern` is the root name of a text file in the `shapelib` directory with a `.DEC` extension to store the decoupling pattern.

`90_pulselength` is the pulse time-duration, in seconds, for an element, a block of elements, or an integer divisor of all elements in the pattern, which is labeled with a tip-angle duration equal to 90°. Often `90_pulselength` is set equal `1/dmf` in which `dmf` is a step rate.

`tipangle_resoln` is the smallest common divisor of the tip-angle durations of all the elements of the pattern.

For many patterns, `90_pulselength` is the actual 90° pulse length and the elements are labeled with tip-angle durations that are multiples of 90. In this case, `tipangle_resoln` is 90.0.

For many other patterns, `90_pulselength` is set to be the duration of the minimum stepsize of which all elements are multiples. In these cases, `tipangle_resoln` is nominally 90 and elements in the pattern have tip-angle durations that are multiples of 90.

For some patterns, `90_pulselength` is the actual 90° pulse length but elements of the pattern have arbitrary flip angles that are multiples of a `tipangle_resoln`, that is less than 90.0° (*c.f.* 1.0°). In this case, `tipangle_resoln` is the divisor (*c.f.* 1.0) and for good practice it is also a divisor of 90.0.

**Examples:**

```
dec4prgon("garp1",1/dmf, 1.0);
dec4prgon(modtype,pwx90,dres);
ticks = dec4prgon("waltz16",1/dmf,90.0);
```

<b>Related:</b>	dec4prgoff	End waveform decoupling on fourth decoupler
	decprgon	Start waveform decoupling on first decoupler
	dec2prgon	Start waveform decoupling on second decoupler
	dec3prgon	Start waveform decoupling on third decoupler
	dec4prgonOffset	Start waveform decoupling on fourth decoupler with offset
	dec4unblank	Unblank amplifier of fourth decoupler
	obsprgoff	End waveform decoupling on observe channel
	obsprgon	Start waveform decoupling on observe channel
	obsprgonOffset	Start waveform decoupling on observe channel with offset

#### decprgonOffset

#### Start waveform decoupling on first decoupler with offset

**Syntax:**

```
decprgon(pattern,90_pulselength,tipangle_resoln,
offset)
char *pattern;          /* name of .DEC file */
double 90_pulselength; /* 90-degree pulse
length, seconds */
double tipangle_resoln; /* tip-angle
resolution */
double offset; /* frequency offset, Hz */
return int ticks /* 12.5 ns ticks, one cycle */
```

**Description:** Execute programmable decoupling on the first decoupler under waveform control with a frequency offset that is applied automatically. The statement `decprgonOffset` unblanks the associated amplifier and gates the first decoupler on for patterns without an explicit gate column. `decprgonOffset` returns an integer with the number of 12.5-ns ticks in one cycle of the decoupling pattern. It is a good practice to unblank the associated amplifier with `decunblank`, at least 2.0  $\mu$ s before `decprgonOffset`.

The frequency offset is applied by phase modulation of the base pattern in the rf controller. The phase modulation is achieved by expanding the pattern to include linear phase steps as well as by interpolation in real-time. Use of `decprgonOffset` with a base pattern achieves about a ten-fold compression of steps relative to an equivalent pattern supplied in the .DEC file.

**Arguments:** `pattern` is the root name of a text file in the `shapelib` directory with a `.DEC` extension to store the decoupling pattern.

`90_pulselength` is the pulse time-duration, in seconds, for an element, a block of elements, or an integer divisor of all elements in the pattern, which is labeled with a tip-angle duration equal to  $90^\circ$ . Often `90_pulselength` is set equal  $1/\text{dmf}$  in which `dmf` is a step rate. `decprgonOffset`

`tipangle_resoln` is the smallest common divisor of the tip-angle durations of all the elements of the pattern.

For many patterns, `90_pulselength` is the actual  $90^\circ$  pulse length and the elements are labeled with tip-angle durations that are multiples of  $90^\circ$ . In this case, `tipangle_resoln` is `90.0`.

For many other patterns, `90_pulselength` is set to be the duration of the minimum stepsize of which all elements are multiples. In these cases, `tipangle_resoln` is nominally `90` and elements in the pattern have tip-angle durations that are multiples of  $90^\circ$ .

For some patterns, `90_pulselength` is the actual  $90^\circ$  pulse length but elements of the pattern have arbitrary flip angles that are multiples of a `tipangle_resoln`, that is less than  $90^\circ$  (*c.f.* `1.0`). In this case, `tipangle_resoln` is the divisor (*c.f.* `1.0`) and for good practice it is also a divisor of `90.0`.

`offset` is a frequency offset relative to the synthesizer frequency, in Hz.

**Examples:**

```
decprgonOffset("garp1",1/dmf, 1.0,1000.0);
decprgonOffset(modtype,pwx90,dres,1000.0);
ticks =
decprgonOffset("waltz16",1/dmf,90.0,1000.0);
```

<b>Related:</b>	<code>decprgoff</code>	End waveform decoupling on first decoupler
	<code>decprgon</code>	Start waveform decoupling on first decoupler
	<code>dec2prgonOffset</code>	Start waveform decoupling on second decoupler with offset
	<code>dec3prgonOffset</code>	Start waveform decoupling on third decoupler with offset
	<code>dec4prgonOffset</code>	Start waveform decoupling on fourth decoupler with offset
	<code>decunblank</code>	Unblank the amplifier of first decoupler

obsprogoff	End waveform decoupling on observe channel
obsprgon	Start waveform decoupling on observe channel
obsprgonOffset	Start waveform decoupling on observe channel with offset

**dec2prgonOffset****Start waveform decoupling on second decoupler with offset**

**Syntax:** `dec2prgonOffset(pattern,90_pulselength,tipangle_resoln,offset)`  
`char *pattern; /* name of .DEC file */`  
`double 90_pulselength; /* 90-degree pulse length, seconds */`  
`double tipangle_resoln; /* tip-angle resolution */`  
`double offset; /* frequency offset, Hz */`  
`return int ticks /* 12.5 ns ticks, one cycle */`

**Description:** Execute programmable decoupling on the first decoupler under waveform control with a frequency offset that is applied automatically. The statement `dec2prgonOffset` unblanks the associated amplifier and gates the first decoupler on for patterns without an explicit gate column. `dec2prgonOffset` returns an integer with the number of 12.5-ns ticks in one cycle of the decoupling pattern. It is a good practice to unblank the associated amplifier with `dec2unblank`, at least 2.0  $\mu$ s before `dec2prgonOffset`.

The frequency offset is applied by phase modulation of the base pattern in the rf controller. The phase modulation is achieved by expanding the pattern to include linear phase steps as well as by interpolation in real-time. Use of `dec2prgonOffset` with a base pattern achieves about a ten-fold compression of steps relative to an equivalent pattern supplied in the .DEC file.

**Arguments:** `pattern` is the root name of a text file in the `shapelib` directory with a .DEC extension to store the decoupling pattern.

`90_pulselength` is the pulse time-duration, in seconds, for an element, a block of elements, or an integer divisor of all elements in the pattern, which is labeled with a tip-angle duration equal to 90°. Often `90_pulselength` is set equal  $1/dmf$  where `dmf` is a step rate. `dec2prgonOffset`

`tipangle_resoln` is the smallest common divisor of the tip-angle durations of all the elements of the pattern.



For many patterns, `90_pulselength` is the actual 90° pulse length and the elements are labeled with tip-angle durations that are multiples of 90. In this case, `tipangle_resoln` is 90.0.

For many other patterns, `90_pulselength` is set to be the duration of the minimum stepsize of which all elements are multiples. In these cases, `tipangle_resoln` is nominally 90 and elements in the pattern have tip-angle durations that are multiples of 90.

For some patterns, `90_pulselength` is the actual 90° pulse length but elements of the pattern have arbitrary flip angles that are multiples of a `tipangle_resoln`, that is less than 90° (*c.f.* 1.0°). In this case, `tipangle_resoln` is the divisor (*c.f.* 1.0) and for good practice it is also a divisor of 90.0.

`offset` is a frequency offset relative to the synthesizer frequency, in Hz.

Examples: 

```
dec2prgonOffset("garp1",1/dmf, 1.0,1000.0);
dec2prgonOffset(modtype,pwx90,dres,1000.0);
ticks =
dec2prgonOffset("waltz16",1/dmf,90.0,1000.0)
;
```

Related:	<code>dec2prgoff</code>	End waveform decoupling on second decoupler
	<code>decprgonOffset</code>	Start waveform decoupling on first decoupler with offset
	<code>dec2prgon</code>	Start waveform decoupling on second decoupler
	<code>dec3prgonOffset</code>	Start waveform decoupling on third decoupler with offset
	<code>dec4prgonOffset</code>	Start waveform decoupling on fourth decoupler with offset
	<code>dec2unblank</code>	Unblank the amplifier of second decoupler
	<code>obsprgoff</code>	End waveform decoupling on observe channel
	<code>obsprgon</code>	Start waveform decoupling on observe channel
	<code>obsprgonOffset</code>	Start waveform decoupling on observe channel with offset

### `dec3prgonOffset`

### Start waveform decoupling on third decoupler with offset

Syntax: 

```
dec3prgonOffset(pattern,90_pulselength,tipangle_resoln,offset)
char *pattern;          /* name of .DEC file */
```

```

double 90_pulselength;    /* 90-degree pulse
length, seconds */
double tipangle_resoln;   /* tip-angle
resolution */
double offset;           /* frequency offset,
Hz */
return int ticks         /* 12.5 ns ticks,
one cycle */

```

**Description:** Execute programmable decoupling on the first decoupler under waveform control with a frequency offset that is applied automatically. The statement `dec3prgonOffset` unblanks the associated amplifier and gates the first decoupler on for patterns without an explicit gate column. `dec3prgonOffset` returns an integer with the number of 12.5-ns ticks in one cycle of the decoupling pattern. It is a good practice to unblank the associated amplifier with `dec3unblank`, at least 2.0  $\mu$ s before `dec3prgonOffset`.

The frequency offset is applied by phase modulation of the base pattern in the rf controller. The phase modulation is achieved by expanding the pattern to include linear phase steps as well as by interpolation in real-time. Use of `dec3prgonOffset` with a base pattern achieves about a ten-fold compression of steps relative to an equivalent pattern supplied in the .DEC file.

**Arguments:** `pattern` is the root name of a text file in the `shapelib` directory with a .DEC extension to store the decoupling pattern.

`90_pulselength` is the pulse time-duration, in seconds, for an element, a block of elements, or an integer divisor of all elements in the pattern, which is labeled with a tip-angle duration equal to 90°. Often `90_pulselength` is set equal  $1/\text{dmf}$  where `dmf` is a step rate.

`tipangle_resoln` is the smallest common divisor of the tip-angle durations of all the elements of the pattern.

For many patterns, `90_pulselength` is the actual 90° pulse length and the elements are labeled with tip-angle durations that are multiples of 90. In this case, `tipangle_resoln` is 90.0.

For many other patterns, `90_pulselength` is set to be the duration of the minimum stepsize of which all elements are multiples. In these cases, `tipangle_resoln` is nominally 90 and elements in the pattern have tip-angle durations that are multiples of 90.

For some patterns, `90_pulselength` is the actual 90° pulse length but elements of the pattern have arbitrary flip angles that are multiples of a

tipangle\_resoln, that is less than 90° (*c.f.* 1.0°). In this case, tipangle\_resoln is the divisor (*c.f.* 1.0) and for good practice it is also a divisor of 90.0.

offset is a frequency offset relative to the synthesizer frequency, in Hz.

Examples: 

```
dec3prgonOffset("garp1",1/dmf, 1.0,1000.0);
dec3prgonOffset(modtype,pwx90,dres,1000.0);
ticks =
dec3prgonOffset("waltz16",1/dmf,90.0,1000.0)
;
```

Related:	dec3prgoff	End waveform decoupling on third decoupler
	decprgonOffset	Start waveform decoupling on first decoupler with offset
	dec2prgonOffset	Start waveform decoupling on second decoupler with offset
	dec3prgon	Start waveform decoupling on third decoupler
	dec4prgonOffset	Start waveform decoupling on fourth decoupler with offset
	dec3unblank	Unblank the amplifier of third decoupler
	obsprgoff	End waveform decoupling on observe channel
	obsprgon	Start waveform decoupling on observe channel
	obsprgonOffset	Start waveform decoupling on observe channel with offset

### dec4prgonOffset

#### Start waveform decoupling on first decoupler with offset

Syntax: 

```
dec4prgonOffset(pattern,90_pulselength,tipangle_resoln,offset)
char *pattern; /* name of .DEC file */
double 90_pulselength; /* 90-degree pulse length, seconds */
double tipangle_resoln; /* tip-angle resolution */
double offset; /* frequency offset, Hz */
return int ticks /* 12.5 ns ticks, one cycle */
```

Description: Execute programmable decoupling on the fourth decoupler under waveform control with a frequency offset that is applied automatically. The statement dec4prgonOffset unblanks the associated amplifier

and gates the first decoupler on for patterns without an explicit gate column. `dec4prgonOffset` returns an integer with the number of 12.5-ns ticks in one cycle of the decoupling pattern. It is a good practice to unblank the associated amplifier with `dec4unblank`, at least 2.0  $\mu$ s before `dec4prgonOffset`.

The frequency offset is applied by phase modulation of the base pattern in the rf controller. The phase modulation is achieved by expanding the pattern to include linear phase steps as well as by interpolation in real-time. Use of `dec4prgonOffset` with a base pattern achieves about a ten-fold compression of steps relative to an equivalent pattern supplied in the `.DEC` file.

**Arguments:** `pattern` is the root name of a text file in the `shapelib` directory with a `.DEC` extension to store the decoupling pattern.

`90_pulselength` is the pulse time-duration, in seconds, for an element, a block of elements, or an integer divisor of all elements in the pattern, which is labeled with a tip-angle duration equal to  $90^\circ$ . Often `90_pulselength` is set equal  $1/dmf$  where `dmf` is a step rate.

`tipangle_resoln` is the smallest common divisor of the tip-angle durations of all the elements of the pattern.

For many patterns, `90_pulselength` is the actual  $90^\circ$  pulse length and the elements are labeled with tip-angle durations that are multiples of 90. In this case, `tipangle_resoln` is 90.0.

For many other patterns, `90_pulselength` is set to be the duration of the minimum stepsize of which all elements are multiples. In these cases, `tipangle_resoln` is nominally 90 and elements in the pattern have tip-angle durations that are multiples of 90.

For some patterns, `90_pulselength` is the actual  $90^\circ$  pulse length but elements of the pattern have arbitrary flip angles that are multiples of a `tipangle_resoln`, that is less than  $90^\circ$  (*c.f.* 1.0 $^\circ$ ). In this case, `tipangle_resoln` is the divisor (*c.f.* 1.0) and for good practice it is also a divisor of 90.0.

`offset` is a frequency offset relative to the synthesizer frequency, in Hz.

**Examples:**

```
dec4prgonOffset("garp1",1/dmf, 1.0,1000.0);
dec4prgonOffset(modtype,pwx90,dres,1000.0);
ticks =
dec4prgonOffset("waltz16",1/dmf,90.0,1000.0)
;
```

<b>Related:</b>	<code>dec4prgoff</code>	End waveform decoupling on fourth decoupler
	<code>decprgonOffset</code>	Start waveform decoupling on first decoupler with offset
	<code>dec2prgonOffset</code>	Start waveform decoupling on second decoupler with offset
	<code>dec3prgonOffset</code>	Start waveform decoupling on third decoupler with offset
	<code>dec4prgon</code>	Start waveform decoupling on fourth decoupler
	<code>dec4unblank</code>	Unblank the amplifier of fourth decoupler
	<code>obsprogoff</code>	End waveform decoupling on observe channel
	<code>obsprgon</code>	Start waveform decoupling on observe channel
	<code>obsprgonOffset</code>	Start waveform decoupling on observe channel with offset

**decpulse****Perform pulse with assigned values on first decoupler**

**Syntax:** `decpulse(width,phase)`  
`double width; /* duration of pulse, seconds */`  
`codeint phase; /* real-time quadrature-phase multiplier for pulse */`

**Description:** Set the quadrature phase and gates the first decoupler on and off at the current power level with amplifier unblanking and blanking and no predelay or postdelay. It is a good practice to unblank the associated amplifier with `decunblank`, at least 2.0  $\mu$ s before `decpulse` if the associated amplifier is in pulse mode.

**Arguments:** `width` is the duration of the pulse, in seconds.  
`phase` is a 90° multiplier for the first-decoupler phase. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*), or a real-time table (`t1` to `t60`).

**Examples:** `decpulse(pp,v3);`  
`decpulse(2.0*pp,zero);`

<b>Related:</b>	<code>decrpulse</code>	Perform pulse on first decoupler
	<code>obspulse</code>	Perform pulse with assigned values on observe channel

pulse	Perform pulse with assigned values on observe channel
rgpulse	Perform pulse on observe channel

**decpwr**

**Set fine power level of first decoupler**

**Syntax:** `decpwr(amplitude)`  
`double amplitude; /* fine-power value, */`

**Description:** Set the fine-power level of the first decoupler using the linear modulator. The fine-power level of the first decoupler is initialized to `dpwr` before the first `decpwr` statement is applied. You must explicitly use `decpwr(dpwr)` to return the power level to `dpwr`. Fine power units are linear in voltage.

**Arguments:** `amplitude` sets the fine power in units of 1.0 from a maximum of 4095.0 to a minimum of 0.0. If the 12-bit linear modulator is available the stepsize is 1.0. If the 16-bit linear modulator is present decimal values (3 significant figures) are allowed. Consult the configuration file to determine the available modulator. The fine power and coarse power can be used interchangeably where 6 dB units of coarse power correspond to a x2 change of the fine power.

**Examples:** `decpwr(3500);`  
`decpwr(3500.324);`

<b>Related:</b>	<code>dec2pwr</code>	Set fine power level of second decoupler
	<code>dec3pwr</code>	Set fine power level of third decoupler
	<code>dec4pwr</code>	Set fine power level of fourth decoupler
	<code>obspwr</code>	Set fine power level of observe channel
	<code>rlpwr</code>	Set fine power level of any channel

**dec2pwr**

**Set fine power level of second decoupler**

**Syntax:** `dec2pwr(amplitude)`  
`double amplitude; /* fine-power value*/`

**Description:** Set the fine-power level of the second decoupler using the linear modulator. The fine-power level of the second decoupler is initialized to `dpwr2` before the first `decpwr` statement is applied. You must explicitly use `dec2pwr(dpwr2)` to return the

power level to `dpwrf2`. Fine power units are linear in voltage.

**Arguments:** `amplitude` sets the fine power in units of 1.0 from a maximum of 4095.0 to a minimum of 0.0. If the 12-bit linear modulator is available the stepsize is 1.0. If the 16-bit linear modulator is present decimal values (3 significant figures) are allowed. Consult the configuration file to determine the available modulator. The fine power and coarse power can be used interchangeably where 6 dB units of coarse power correspond to a x2 change of the fine power.

**Examples:** `dec2pwrf(3500);`  
`dec2pwrf(3500.324);`

<b>Related:</b>	<code>decpwrf</code>	Set fine power level of first decoupler
	<code>dec3pwrf</code>	Set fine power level of third decoupler
	<code>dec4pwrf</code>	Set fine power level of fourth decoupler
	<code>obspwrf</code>	Set fine power level of observe channel
	<code>rlpwrf</code>	Set fine power level of any channel

### **dec3pwrf**

#### **Set fine power level of third decoupler**

**Syntax:** `dec3pwrf(amplitude)`  
`double amplitude; /* fine-power value */`

**Description:** Set the fine-power level of the third decoupler using the linear modulator. The fine-power level of the third decoupler is initialized to `dpwrf3` before the first `dec3pwrf` statement is applied. You must explicitly use `dec3pwrf(dpwrf)` to return the power level to `dpwrf3`. Fine power units are linear in voltage.

**Arguments:** `amplitude` sets the fine power in units of 1.0 from a maximum of 4095.0 to a minimum of 0.0. If the 12-bit linear modulator is available the stepsize is 1.0. If the 16-bit linear modulator is present decimal values (3 significant figures) are allowed. Consult the configuration file to determine the available modulator. The fine power and coarse power can be used interchangeably where 6 dB units of coarse power correspond to a x2 change of the fine power.

**Examples:** `dec3pwrf(3500);`  
`dec3pwrf(3500.324);`

Related:	<code>decpwrif</code>	Set fine power level of first decoupler
	<code>dec2pwrif</code>	Set fine power level of second decoupler
	<code>dec4pwrif</code>	Set fine power level of fourth decoupler
	<code>obspwrif</code>	Set fine power level of observe channel
	<code>rlpwrif</code>	Set fine power level of any channel

**dec4pwrif**

**Set fine power level of fourth decoupler**

Syntax: `dec4pwrif(amplitude)`  
`double amplitude; /* fine-power value*/`

Description: Set the fine-power level of the fourth decoupler using the linear modulator. The fine-power level of the fourth decoupler is initialized to `dpwrif4` before the first `dec4pwrif` statement is applied. You must explicitly use `dec4pwrif(dpwrif)` to return the power level to `dpwrif4`. Fine power units are linear in voltage.

Arguments: `amplitude` sets the fine power in units of 1.0 from a maximum of 4095.0 to a minimum of 0.0. If the 12-bit linear modulator is available the stepsize is 1.0. If the 16-bit linear modulator is present decimal values (3 significant figures) are allowed. Consult the configuration file to determine the available modulator. The fine power and coarse power can be used interchangeably where 6 dB units of coarse power correspond to a x2 change of the fine power.

Examples: `dec4pwrif(3500);`  
`dec4pwrif(3500.324);`

Related:	<code>decpwrif</code>	Set fine power level of first decoupler
	<code>dec2pwrif</code>	Set fine power level of second decoupler
	<code>dec3pwrif</code>	Set fine power level of third decoupler
	<code>obspwrif</code>	Set fine power level of observe channel
	<code>rlpwrif</code>	Set fine power level of any channel

**decr**

**Decrement real-time integer value**

Syntax: `decr(vi)`  
`codeint vi; /* real-time variable to be decremented */`



**Description:** Decrement the integer value of  $v_i$  by 1.

**Arguments:**  $v_i$  is the integer to be decremented. It must be a real-time variable ( $cc$  to  $v42$ ,  $oph$ , *etc*).

**Examples:** `decr(v5);`

<b>Related:</b>	<code>add</code>	Add real-time integer values
	<code>assign</code>	Assign real-time integer value using real-time integer
	<code>dbl</code>	Double real-time integer value
	<code>divn</code>	Divide real-time integer values
	<code>hlv</code>	Assign half the value of real-time integer
	<code>incr</code>	Increment real-time integer value
	<code>initval</code>	Assign real-time integer value using numeric value
	<code>mod2</code>	Assign real-time integer value modulo 2
	<code>mod4</code>	Assign real-time integer value modulo 4
	<code>modn</code>	Assign real-time integer value modulo $n$
	<code>mult</code>	Multiply real-time integer values
	<code>sub</code>	Subtract real-time integer values

**decrpulse****Perform pulse on first decoupler**

**Syntax:**

```
decrpulse(width,phase,RG1,RG2)
double width;      /* duration of pulse, seconds
*/
codeint phase;    /* real-time
quadrature-phase multiplier for pulse */
double RG1;       /* duration of predelay,
seconds */
double RG2;       /* duration of postdelay,
seconds */
```

**Description:** Set the quadrature phase and gates of the first decoupler on and off at the current power level with amplifier unblanking and blanking. `decrpulse` is preceded by a predelay and followed by a postdelay. The associated amplifier is unblanked if it is in pulse mode and the phase is set, both at the beginning of the predelay. The associated amplifier is blanked at the end of the postdelay if it is in pulse mode.

**Arguments:**  $width$  is the duration of the pulse, in seconds.

$phase$  is a  $90^\circ$  multiplier for the first-decoupler phase. The value must be a real-time variable ( $v_1$  to  $v42$ ,  $oph$ , *etc*), a real-time constant (*zero*, *one*, *etc*), or a real-time table ( $t_1$  to  $t60$ ).

$RG1$  is the duration of the predelay, in seconds.

Examples: `RG2` is the duration of the postdelay, in seconds.  
`decrgpulse(pp,v3,rof1,rof2);`  
`decrgpulse(pp,zero,1.0e-6,0.2e-6);`

Related:	<code>decpulse</code>	Perform pulse with assigned values on first decoupler
	<code>dec2rgpulse</code>	Perform pulse on second decoupler
	<code>dec3rgpulse</code>	Perform pulse on third decoupler
	<code>dec4rgpulse</code>	Perform pulse on fourth decoupler
	<code>decshaped_pulse</code>	Perform shaped pulse on first decoupler
	<code>obspulse</code>	Perform pulse with assigned values on observe channel
	<code>pulse</code>	Perform pulse with assigned values on observe channel
	<code>rgpulse</code>	Perform pulse on observe channel
	<code>shaped_pulse</code>	Perform shaped pulse on observe channel
	<code>simshaped_pulse</code>	Perform simultaneous shaped pulses, observe and first decoupler
	<code>simpulse</code>	Perform simultaneous pulses, observe and first decoupler

**dec2rgpulse**

**Perform pulse on second decoupler**

Syntax: `dec2rgpulse(width,phase,RG1,RG2)`  
`double width; /* duration of pulse, seconds */`  
`codeint phase; /* real-time`  
`quadrature-phase multiplier for pulse */`  
`double RG1; /* duration of predelay, seconds`  
`*/`  
`double RG2; /* duration of postdelay, seconds`  
`*/`

Description: Set the quadrature phase and gates of the second decoupler on and off at the current power level with amplifier unblanking and blanking. `dec2rgpulse` is preceded by a predelay and followed by a postdelay. The associated amplifier is unblanked if it is in pulse mode and the phase is set, both at the beginning of the predelay. The associated amplifier is blanked at the end of the postdelay if it is in pulse mode.

Arguments: `width` is the duration of the pulse, in seconds.

phase is a  $90^\circ$  multiplier for the first-decoupler phase. The value must be a real-time variable (*v1* to *v42*, *oph*, *etc*), a real-time constant (*zero*, *one*, *etc*), or a real-time table (*t1* to *t60*).

RG1 is the duration of the predelay, in seconds.

RG2 is the duration of the postdelay, in seconds.

Examples: `dec2rgpulse(pp,v3,rof1,rof2);`  
`dec2rgpulse(pp,zero,1.0e6,0.2e6);`

Related:	<code>decpulse</code>	Perform pulse with assigned values on first decoupler
	<code>decrgpulse</code>	Perform pulse on first decoupler
	<code>dec3rgpulse</code>	Perform pulse on third decoupler
	<code>dec4rgpulse</code>	Perform pulse on fourth decoupler
	<code>dec2shaped_pulse</code>	Perform shaped pulse on second decoupler
	<code>obspulse</code>	Perform pulse with assigned values on observe channel
	<code>pulse</code>	Perform pulse with assigned values on observe channel
	<code>rgpulse</code>	Perform pulse on observe channel
	<code>shaped_pulse</code>	Perform shaped pulse on observe channel
	<code>sim3shaped_pulse</code>	Perform simultaneous shaped pulses, observe, first and second decoupler
	<code>sim3pulse</code>	Perform simultaneous pulses, observe, first and second decoupler

### **dec3rgpulse**

#### **Perform pulse on third decoupler**

Syntax: `dec3rgpulse(width,phase,RG1,RG2)`  
`double width; /* duration of pulse, seconds */`  
`codeint phase; /* real-time`  
`quadrature-phase multiplier for pulse */`  
`double RG1; /* duration of predelay,`  
`seconds */`  
`double RG2; /* duration of postdelay,`  
`seconds */`

Description: Set the quadrature phase and gates of the third decoupler on and off at the current power level with amplifier unblanking and blanking. `dec3rgpulse` is preceded by a predelay and

followed by a postdelay. The associated amplifier is unblanked if it is in pulse mode and the phase is set, both at the beginning of the predelay. The associated amplifier is blanked at the end of the postdelay if it is in pulse mode.

**Arguments:** width is the duration of the pulse, in seconds.  
 phase is a 90° multiplier for the first-decoupler phase. The value must be a real-time variable (v1 to v42, oph, etc), a real-time constant (zero, one, etc), or a real-time table (t1 to t60).  
 RG1 is the duration of the predelay, in seconds.  
 RG2 is the duration of the predelay, in seconds.

**Examples:** dec3rgpulse(pp,v3,rof1,rof2);  
 dec3rgpulse(pp,zero,1.0e6,0.2e6);

<b>Related:</b>	decpulse	Perform pulse with assigned values on first decoupler
	decrpulse	Perform pulse on first decoupler
	dec3rgpulse	Perform pulse on third decoupler
	dec4rgpulse	Perform pulse on fourth decoupler
	dec3shaped_pulse	Perform shaped pulse on third decoupler
	obspulse	Perform pulse with assigned values on observe channel
	pulse	Perform pulse with assigned values on observe channel
	rgpulse	Perform pulse on observe channel
	shaped_pulse	Perform shaped pulse on observe channel
	sim4shaped_pulse	Perform simultaneous shaped pulses, observe, first second and third decouplers
	sim4pulse	Perform simultaneous pulses, observe, first second and third decouplers

**dec4rgpulse**

**Perform pulse on fourth decoupler**

**Syntax:** dec4rgpulse(width,phase,RG1,RG2)  
 double width; /\* duration of pulse, seconds \*/  
 codeint phase; /\* real-time quadrature-phase multiplier for pulse \*/  
 double RG1; /\* duration of predelay, seconds \*/  
 double RG2; /\* duration of postdelay, seconds \*/

**Description:** Set the quadrature phase and gates of the fourth decoupler on and off at the current power level with amplifier unblanking and blanking. `dec4rgpulse` is preceded by a predelay and followed by a postdelay. The associated amplifier is unblanked if it is in pulse mode and the phase is set, both at the beginning of the predelay. The associated amplifier is blanked at the end of the postdelay if it is in pulse mode.

**Arguments:** `width` is the duration of the pulse, in seconds.  
`phase` is a  $90^\circ$  multiplier for the first-decoupler phase. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*), or a real-time table (`t1` to `t60`).  
`RG1` is the duration of the predelay, in seconds.  
`RG2` is the duration of the postdelay, in seconds.

**Examples:** `dec4rgpulse(pp,v3,rof1,rof2);`  
`dec4rgpulse(pp,zero,1.0e6,0.2e6);`

<b>Related:</b>	<code>decpulse</code>	Perform pulse with assigned values on first decoupler
	<code>decrgpulse</code>	Perform pulse on first decoupler
	<code>dec2rgpulse</code>	Perform pulse on second decoupler
	<code>dec3rgpulse</code>	Perform pulse on third decoupler
	<code>dec4shaped_pulse</code>	Perform shaped pulse on fourth decoupler
	<code>obspulse</code>	Perform pulse with assigned values on observe channel
	<code>pulse</code>	Perform pulse with assigned values on observe channel
	<code>rgpulse</code>	Perform pulse on observe channel
	<code>shaped_pulse</code>	Perform shaped pulse on observe channel
	<code>simshaped_pulse</code>	Perform simultaneous shaped pulses, observe and first decoupler
	<code>simpulse</code>	Perform simultaneous pulses, observe and first decoupler

**decshaped\_pulse****Perform shaped pulse on first decoupler**

**Syntax:** `decshaped_pulse(pattern,width,phase,RG1,RG2)`  
`pattern;` /\* name of .RF text file \*/  
`double width;` /\* duration of pulse, seconds

```

*/
codeint phase; /* real-time
quadrature-phase multiplier for pulse */
double RG1; /* duration of predelay,
seconds */
double RG2; /* duration of postdelay,
seconds */

```

**Description:** Set the quadrature phase, gate the first decoupler on and off at the current power level with amplifier unblanking and blanking, and apply a shaped-pulse pattern. `decshaped_pulse` is preceded by a predelay and followed by a postdelay. The associated amplifier is unblanked if it is in pulse mode and the phase is set, both at the beginning of the predelay. The associated amplifier is blanked at the end of the postdelay if it is in pulse mode.

**Arguments:** `pattern` is the root name of a text file in the `shapelib` directory with a `.RF` extension to store the decoupling pattern.

`width` is the duration of the pulse, in seconds.

`phase` is a  $90^\circ$  multiplier for the first-decoupler phase. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*), or a real-time table (`t1` to `t60`).

`RG1` is the duration of the predelay, in seconds.

`RG2` is the duration of the postdelay, in seconds.

**Examples:** `decshaped_pulse("sinc",p1,v5,rof1,rof2);`

<b>Related:</b>	<code>decrpulse</code>	Perform pulse on first decoupler
	<code>dec2shaped_pulse</code>	Perform shaped pulse on second decoupler
	<code>dec3shaped_pulse</code>	Perform shaped pulse on third decoupler
	<code>dec4shaped_pulse</code>	Perform shaped pulse on fourth decoupler
	<code>rgpulse</code>	Perform pulse on observe channel
	<code>shaped_pulse</code>	Perform shaped pulse on observe channel
	<code>simshaped_pulse</code>	Perform simultaneous shaped pulses, observe and first decoupler
	<code>simpulse</code>	Perform simultaneous pulses, observe and first decoupler

**dec2shaped\_pulse****Perform shaped pulse on second decoupler**

**Syntax:** `dec2shaped_pulse(pattern,width,phase,RG1,RG2)`  
`pattern;` /\* name of .RF text file \*/  
`double width;` /\* duration of pulse, seconds \*/  
`codeint phase;` /\* real-time  
quadrature-phase multiplier for pulse \*/  
`double RG1;` /\* duration of predelay,  
seconds \*/  
`double RG2;` /\* duration of postdelay,  
seconds \*/

**Description:** Set the quadrature phase, gate the second decoupler on and off at the current power level with amplifier unblanking and blanking, and apply a shaped-pulse pattern. `dec2shaped_pulse` is preceded by a predelay and followed by a postdelay. The associated amplifier is unblanked if it is in pulse mode and the phase is set, both at the beginning of the predelay. The associated amplifier is blanked at the end of the postdelay if it is in pulse mode.

**Arguments:** `pattern` is the root name of a text file in the `shapelib` directory with a `.RF` extension to store the decoupling pattern.

`width` is the duration of the pulse, in seconds.

`phase` is a  $90^\circ$  multiplier for the first-decoupler phase. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*), or a real-time table (`t1` to `t60`).

`RG1` is the duration of the predelay, in seconds.

`RG2` is the duration of the postdelay, in seconds.

**Examples:** `decs2haped_pulse("sinc",p1,v5,rof1,rof2);`

<b>Related:</b>	<code>dec2rgpulse</code>	Perform pulse on second decoupler
	<code>decshaped_pulse</code>	Perform shaped pulse on first decoupler
	<code>dec3shaped_pulse</code>	Perform shaped pulse on third decoupler
	<code>dec4shaped_pulse</code>	Perform shaped pulse on fourth decoupler
	<code>rgpulse</code>	Perform pulse on observe channel
	<code>shaped_pulse</code>	Perform shaped pulse on observe channel
	<code>sim3shaped_pulse</code>	Perform simultaneous shaped pulses, observe, first and second decouplers
	<code>sim3pulse</code>	Perform simultaneous pulses, observe, first and second decouplers

**dec3shaped\_pulse**

**Perform shaped pulse on third decoupler**

**Syntax:** `dec3shaped_pulse(pattern,width,phase,RG1,RG2)
pattern; /* name of .RF text file */
double width; /* duration of pulse in sec
*/
codeint phase; /* real-time quadrature
phase multiplier for pulse */
double RG1; /* duration of predelay in
sec */
double RG2; /* duration of postdelay in
sec */`

**Description:** Set the quadrature phase, gate the third decoupler on and off at the current power level with amplifier unblanking and blanking, and apply a shaped-pulse pattern. `dec3shaped_pulse` is preceded by a predelay and followed by a postdelay. The associated amplifier is unblanked if it is in pulse mode and the phase is set, both at the beginning of the predelay. The associated amplifier is blanked at the end of the postdelay if it is in pulse mode.

**Arguments:** `pattern` is the root name of a text file in the `shapelib` directory with a `.RF` extension to store the decoupling pattern.

`width` is the duration of the pulse, in seconds.

`phase` is a 90° multiplier for the first-decoupler phase. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*), or a real-time table (`t1` to `t60`).

`RG1` is the duration of the predelay, in seconds.

`RG2` is the duration of the postdelay, in seconds.

**Examples:** `dec3shaped_pulse("sinc",p1,v5,rof1,rof2);`

<b>Related:</b>	<code>dec3rgpulse</code>	Perform pulse on third decoupler
	<code>decshaped_pulse</code>	Perform shaped pulse on first decoupler
	<code>dec2shaped_pulse</code>	Perform shaped pulse on second decoupler
	<code>dec4shaped_pulse</code>	Perform shaped pulse on fourth decoupler
	<code>rgpulse</code>	Perform pulse on observe channel
	<code>shaped_pulse</code>	Perform shaped pulse on observe channel
	<code>sim4shaped_pulse</code>	Perform simultaneous shaped pulses, observe, first, second and third decouplers



sim4pulse                      Perform simultaneous pulses, observe, first, second and third decouplers

**dec4shaped\_pulse****Perform shaped pulse on fourth decoupler**

**Syntax:**    `dec4shaped_pulse(pattern,width,phase,RG1,RG2)`  
`pattern;`                    `/* name of .RF text file */`  
`double width;`                `/* duration of pulse, seconds`  
`*/`  
`codeint phase;`                `/* real-time`  
`quadrature-phase multiplier for pulse */`  
`double RG1;`                    `/* duration of predelay in`  
`sec */`  
`double RG2;`                    `/* duration of postdelay in`  
`sec */`

**Description:**    Set the quadrature phase, gate the first decoupler on and off at the current power level with amplifier unblanking and blanking, and apply a shaped-pulse pattern. `dec4shaped_pulse` is preceded by a predelay and followed by a postdelay. The associated amplifier is unblanked if it is in pulse mode and the phase is set, both at the beginning of the predelay. The associated amplifier is blanked at the end of the postdelay if it is in pulse mode.

**Arguments:**    `pattern` is the root name of a text file in the `shapelib` directory with a `.RF` extension to store the decoupling pattern.

`width` is the duration of the pulse, in seconds.

`phase` is a  $90^\circ$  multiplier for the first-decoupler phase. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, `cc`), or a real-time table (`t1` to `t60`).

`RG1` is the duration of the predelay, in seconds.

**Examples:**    `dec4shaped_pulse("sinc",p1,v5,rof1,rof2);`

<b>Related:</b>	<code>dec4rgpulse</code>	Perform pulse on fourth decoupler
	<code>decshaped_pulse</code>	Perform shaped pulse on first decoupler
	<code>dec2shaped_pulse</code>	Perform shaped pulse on second decoupler
	<code>dec3shaped_pulse</code>	Perform shaped pulse on third decoupler
	<code>rgpulse</code>	Perform pulse on observe channel
	<code>shaped_pulse</code>	Perform shaped pulse on observe channel

simshaped_pulse	Perform simultaneous shaped pulses, observe and first decoupler
simpulse	Perform simultaneous pulses, observe and first decoupler

**decspinlock**

**Perform waveform spinlock on first decoupler**

Syntax: `decspinlock(pattern,90_pulselength,tipangle_resoln, phase,ncycles)`  
`char *pattern; /* name of .DEC text file */`  
`double 90_pulselength; /* 90-deg pulse length in sec */`  
`double tipangle_resoln; /* resolution of tip angle */`  
`codeint phase; /* real-time quadrature-phase multiplier*/`  
`int ncylices; /* number of cycles to execute */`

Description: Execute a spinlock with an integer number of cycles of programmable decoupling on the first decoupler under waveform control. `decspinlock` unblanks and blanks the associated amplifier and gates the first decoupler on and off for patterns without an explicit gate column. It is a good practice to unblank the associated amplifier with `decunblank`, at least 2.0  $\mu$ s before `decspinlock`.

Arguments: `pattern` is the root name of a text file in the `shapelib` directory with a `.DEC` extension to store the decoupling pattern.

`90_pulselength` is the pulse time-duration, in seconds, for an element, a block of elements, or an integer divisor of all elements in the pattern, which is labeled with a tip-angle duration equal to 90°. Often `90_pulselength` is set equal  $1/dmf$  where `dmf` is a step rate.

`tipangle_resoln` is the smallest common divisor of the tip-angle durations of all the elements of the pattern.

For many patterns, `90_pulselength` is the actual 90° pulse length and the elements are labeled with tip-angle durations that are multiples of 90. In this case, `tipangle_resoln` is 90.0.

For many other patterns, `90_pulselength` is set to be the duration of the minimum stepsize of which all elements are multiples. In these cases, `tipangle_resoln` is nominally 90 and elements in the pattern have tip-angle durations that are multiples of 90.

For some patterns, `90_pulselength` is the actual 90° pulse length but elements of the pattern have

arbitrary flip angles that are multiples of a `tipangle_resoln`, that is less than  $90^\circ$  (*c.f.* 1.0°). In this case, `tipangle_resoln` is the divisor (*c.f.* 1.0) and for good practice it is also a divisor of 90.0. `ncycles` is the number of times that the spinlock pattern is to be executed.

Examples: `decspinlock("mlev16", p190, dres, v1, 30);`  
`decspinlock(spinlk, pp90, dres, v1, cycles);`

Related:	<code>dec2spinlock</code>	Perform waveform spinlock on second decoupler
	<code>dec3spinlock</code>	Perform waveform spinlock on third decoupler
	<code>dec4spinlock</code>	Perform waveform spinlock on fourth decoupler
	<code>spinlock</code>	Perform waveform spinlock on observe channel

## `dec2spinlock`

### Perform waveform spinlock on second decoupler

Syntax: `dec2spinlock(pattern, 90_pulselength, tipangle_resoln, phase, ncycles)`  
`char *pattern; /* name of .DEC text file */`  
`double 90_pulselength; /* 90-deg pulse length in sec */`  
`double tipangle_resoln; /* resolution of tip angle */`  
`codeint phase; /* real-time quadrature-phase multiplier*/`  
`int ncycles; /* number of cycles to execute */`

Description: Execute a spinlock with an integer number of cycles of programmable decoupling on the second decoupler under waveform control. `dec2spinlock` unblanks and blanks the associated amplifier and gates the first decoupler on and off for patterns without an explicit gate column. It is a good practice to unblank the associated amplifier with `dec2unblank`, at least 2.0  $\mu\text{s}$  before `dec2spinlock`.

Arguments: `pattern` is the root name of a text file in the `shapelib` directory with a `.DEC` extension to store the decoupling pattern.

`90_pulselength` is the pulse time-duration, in seconds, for an element, a block of elements, or an integer divisor of all elements in the pattern, which is labeled with a tip-angle duration equal to  $90^\circ$ . Often `90_pulselength` is set equal  $1/\text{dmf}$  where `dmf` is a step rate.

`tipangle_resoln` is the smallest common divisor of the tip-angle durations of all the elements of the pattern.

For many patterns, `90_pulselength` is the actual 90° pulse length and the elements are labeled with tip-angle durations that are multiples of 90. In this case, `tipangle_resoln` is 90.0.

For many other patterns, `90_pulselength` is set to be the duration of the minimum stepsize of which all elements are multiples. In these cases, `tipangle_resoln` is nominally 90 and elements in the pattern have tip-angle durations that are multiples of 90.

For some patterns, `90_pulselength` is the actual 90° pulse length but elements of the pattern have arbitrary flip angles that are multiples of a `tipangle_resoln`, that is less than 90° (*c.f.* 1.0°). In this case `tipangle_resoln` is the divisor (*c.f.* 1.0) and for good practice it is also a divisor of 90.0.

`ncycles` is the number of times that the spinlock pattern is to be executed.

Examples: `dec2spinlock("mlev16",p190,dres,v1,30);`  
`dec2spinlock(spinlk,pp90,dres,v1,cycles);`

Related:	<code>decspinlock</code>	Perform waveform spinlock on first decoupler
	<code>dec3spinlock</code>	Perform waveform spinlock on third decoupler
	<code>dec4spinlock</code>	Perform waveform spinlock on fourth decoupler
	<code>spinlock</code>	Perform waveform spinlock on observe channel

### `dec3spinlock`

#### Perform waveform spinlock on third decoupler

Syntax: `dec3spinlock(pattern,90_pulselength,tipangle_resoln, phase,ncycles)`  
`char *pattern; /* name of .DEC text file */`  
`double 90_pulselength; /* 90-deg pulse length in sec */`  
`double tipangle_resoln; /* resolution of tip angle */`  
`codeint phase; /* real-time quadrature-phase multiplier */`  
`int ncylces; /* number of cycles to execute */`

Description: Execute a spinlock with an integer number of cycles of programmable decoupling on the third decoupler under waveform control. `dec3spinlock` unblanks and blanks the associated amplifier and gates the first decoupler on and off for patterns without an

explicit gate column. It is a good practice to unblank the associated amplifier with `dec3unblank`, at least 2.0  $\mu$ s before `dec3spinlock`.

**Arguments:** `pattern` is the root name of a text file in the `shapelib` directory with a `.DEC` extension to store the decoupling pattern.

`90_pulselength` is the pulse time-duration, in seconds, for an element, a block of elements, or an integer divisor of all elements in the pattern, which is labeled with a tip-angle duration equal to 90°. Often `90_pulselength` is set equal  $1/\text{dmf}$  where `dmf` is a step rate.

`tipangle_resoln` is the smallest common divisor of the tip-angle durations of all the elements of the pattern.

For many patterns, `90_pulselength` is the actual 90° pulse length and the elements are labeled with tip-angle durations that are multiples of 90. In this case, `tipangle_resoln` is 90.0.

For many other patterns, `90_pulselength` is set to be the duration of the minimum stepsize of which all elements are multiples. In these cases, `tipangle_resoln` is nominally 90 and elements in the pattern have tip-angle durations that are multiples of 90.

For some patterns, `90_pulselength` is the actual 90° pulse length but elements of the pattern have arbitrary flip angles that are multiples of a `tipangle_resoln`, that is less than 90.0° (*c.f.* 1.0°). In this case, `tipangle_resoln` is the divisor (*c.f.* 1.0) and for good practice it is also a divisor of 90.0.

`ncycles` is the number of times that the spinlock pattern is to be executed.

**Examples:**

```
dec3spinlock("mlev16",p190,dres,v1,30);
dec3spinlock(spinlk,pp90,dres,v1,cycles);
```

<b>Related:</b>	<code>decspinlock</code>	Perform waveform spinlock on first decoupler
	<code>dec2spinlock</code>	Perform waveform spinlock on second decoupler
	<code>dec4spinlock</code>	Perform waveform spinlock on fourth decoupler
	<code>spinlock</code>	Perform waveform spinlock on observe channel

### **dec4spinlock**

### **Perform waveform spinlock on fourth decoupler**

**Syntax:**

```
dec4spinlock(pattern,90_pulselength,tipangle_resoln,
             phase,ncycles)
char *pattern;           /* name of .DEC text
file */
```

```

double 90_pulselength; /* 90-deg pulse
length in sec */
double tipangle_resoln; /* resolution of tip
angle */
codeint phase; /* real-time
quadrature-phase multiplier */
int ncycles; /* number of cycles
to execute */

```

**Description:** Execute a spinlock with an integer number of cycles of programmable decoupling on the first decoupler under waveform control. `dec4spinlock` unblanks and blanks the associated amplifier and gates the first decoupler on and off for patterns without an explicit gate column. It is a good practice to unblank the associated amplifier with `dec4unblank`, at least 2.0  $\mu$ s before `dec4spinlock`.

**Arguments:** `pattern` is the root name of a text file in the `shapelib` directory with a `.DEC` extension to store the decoupling pattern.

`90_pulselength` is the pulse time-duration, in seconds, for an element, a block of elements, or an integer divisor of all elements in the pattern, which is labeled with a tip-angle duration equal to 90°. Often `90_pulselength` is set equal  $1/\text{dmf}$  where `dmf` is a step rate.

`tipangle_resoln` is the smallest common divisor of the tip-angle durations of all the elements of the pattern.

For many patterns, `90_pulselength` is the actual 90° pulse length and the elements are labeled with tip-angle durations that are multiples of 90. In this case, `tipangle_resoln` is 90.0.

For many other patterns, `90_pulselength` is set to be the duration of the minimum stepsize of which all elements are multiples. In these cases, `tipangle_resoln` is nominally 90 and elements in the pattern have tip-angle durations that are multiples of 90.

For some patterns `'cc` is the actual 90° pulse length but elements of the pattern have arbitrary flip angles that are multiples of a `tipangle_resoln`, that is less than 90.0° (*c.f.* 1.0°). In this case, `tipangle_resoln` is the divisor (*c.f.* 1.0) and for good practice it is also a divisor of 90.0.

`ncycles` is the number of times that the spinlock pattern is to be executed.

**Examples:** `dec4spinlock("mlev16",p190,dres,v1,30);`  
`dec4spinlock(spinlk,pp90,dres,v1,cycles);`

**Related:** `decspinlock` Perform waveform spinlock on first decoupler

dec2spinlock	Perform waveform spinlock on second decoupler
dec3spinlock	Perform waveform spinlock on third decoupler
spinlock	Perform waveform spinlock on observe channel

**decstepsize****Set small-angle phase stepsize of first decoupler**

Syntax: `decstepsize(step_size)`  
`double step_size; /* small-angle phase stepsize */`

Description: Set the small-angle phase stepsize of the first decoupler. The `dcplrphase` statement sets the small-angle phase as a product of the current `step_size` and a real-time multiplier. If `decstepsize` is not used, the default `step_size` is 90°. The `decstepsize` statement can be used multiple times in a sequence.

The small-angle phase of the DD2 MR system has a phase resolution of 360.0/65536 (~0.0055°). The VNMRS has a phase resolution of 360.0/8192 (~0.044°) and the small-angle phase is set to the nearest step. Consult the configuration file to determine which transmitter is present.

Arguments: `step_size` is a value, in degrees, corresponding to one unit of the real-time phase multiplier.

Examples: `decstepsize(30.0);`

Related:	<code>dcplrphase</code>	Set small-angle phase of first decoupler
	<code>dec2stepsize</code>	Set small-angle phase stepsize of second decoupler
	<code>dec3stepsize</code>	Set small-angle phase stepsize of third decoupler
	<code>dec4stepsize</code>	Set small-angle phase stepsize of fourth decoupler
	<code>obsstepsize</code>	Set small-angle phase stepsize of observe channel
	<code>stepsize</code>	Set small-angle phase stepsize of any channel

**dec2stepsize****Set small-angle phase stepsize of second decoupler**

Syntax: `dec2stepsize(step_size)`  
`double step_size; /* small angle phase stepsize */`

Description: Set the small-angle phase stepsize of the second decoupler. The `dcplr2phase` statement sets the small-angle phase as a product of `step_size` and a real-time multiplier. If `dec2stepsize` is not used,

the default `step_size` is 90°. The `dec2stepsize` statement can be used multiple times in a sequence.

The small-angle phase of the DD2 MR system has a phase resolution of 360.0/65536 (~0.0055°). The VNMRS has a phase resolution of 360.0/8192 (~0.044°) and the small-angle phase is set to the nearest step. Consult the configuration file to determine which transmitter is present.

**Arguments:** `step_size` is a value, in degrees, corresponding to one unit of the real-time phase multiplier. .

**Examples:** `dec2stepsize(30.0);`

<b>Related:</b>	<code>dcplr2phase</code>	Set small-angle phase of second decoupler
	<code>decstepsize</code>	Set small-angle phase stepsize of first decoupler
	<code>dec3stepsize</code>	Set small-angle phase stepsize of third decoupler
	<code>dec4stepsize</code>	Set small-angle phase stepsize of fourth decoupler
	<code>obsstepsize</code>	Set small-angle phase stepsize of observe channel
	<code>stepsize</code>	Set small-angle phase stepsize of any channel

**dec3stepsize**

**Set small-angle phase stepsize of third decoupler**

**Syntax:** `dec3stepsize(step_size)`  
`double step_size; /* small-angle phase stepsize */`

**Description:** Set the small angle phase stepsize of the third decoupler. The `dcplr3phase` statement sets the small-angle phase as a product of `step_size` and a real-time multiplier. If `dec3stepsize` is not used, the default `step_size` is 90°. The `dec3stepsize` statement can be used multiple times in a sequence.

The small-angle phase of the DD2 MR system has a phase resolution of 360.0/65536 (~0.0055°). The VNMRS has a phase resolution of 360.0/8192 (~0.044°) and the small-angle phase is set to the nearest step. Consult the configuration file to determine which transmitter is present.

**Arguments:** `step_size` is a value, in degrees, corresponding to one unit of the real-time phase multiplier.

**Examples:** `dec3stepsize(30.0);`



Related:	dcplr3phase	Set small-angle phase of third decoupler
	decstepsize	Set small-angle phase stepsize of first decoupler
	dec2stepsize	Set small-angle phase stepsize of second decoupler
	dec4stepsize	Set small-angle phase stepsize of fourth decoupler
	obsstepsize	Set small-angle phase stepsize of observe channel
	stepsize	Set small-angle phase stepsize of any channel

**dec4stepsize****Set small-angle phase stepsize of fourth decoupler**

Syntax: `dec4stepsize(step_size)`  
`double step_size; /* small-angle phase stepsize */`

Description: Sets the small-angle phase stepsize of the fourth decoupler. The `dcplr4phase` statement sets the small-angle phase as a product of `step_size` and a real-time multiplier. If `dec4stepsize` is not used, the default `step_size` is 90°. The `dec4stepsize` statement can be used multiple times in a sequence. The small-angle phase of VNMR5 may have a phase resolution of 360.0/65536 (~0.0055°) or 360.0/8192 (~0.044°) depending on the available transmitter, and the small-angle phase is set to the nearest step. Consult the configuration file to determine which transmitter is present.

Arguments: `step_size` is a value, in degrees, corresponding to one unit of the real-time phase multiplier.

Examples: `dec4stepsize(30.0);`

Related:	dcplr4phase	Set small-angle phase of fourth decoupler
	decstepsize	Set small-angle phase stepsize of first decoupler
	dec2stepsize	Set small-angle phase stepsize of second decoupler
	dec3stepsize	Set small-angle phase stepsize of third decoupler
	obsstepsize	Set small-angle phase stepsize of observe channel
	stepsize	Set small-angle phase stepsize of any channel

**decunblank Unblank amplifier of first decoupler**

Syntax: `decunblank()`

**Description:** Enables the amplifier for the first decoupler if the amplifier is in pulse mode. For pulse mode, `decunblank` is required at least 2.0  $\mu$ s before pulses, spinlocks, and decoupling periods.

For continuous mode, `decunblank` is the default and the statement is not needed. The first decoupler is in continuous mode by default unless it has an associated receiver or is in the frequency band of another observe channel, for which it is in pulse mode.

To place the first decoupler in pulse mode, create the parameter `ampmode` and set the appropriate character to 'p'.

**Related:**

<code>decblank</code>	Blank amplifier of first decoupler
<code>dec2unblank</code>	Unblank amplifier of second decoupler
<code>dec3unblank</code>	Unblank amplifier of third decoupler
<code>dec4unblank</code>	Unblank amplifier of fourth decoupler
<code>obsunblank</code>	Unblank amplifier of observe channel

**dec2unblank Unblank amplifier of second decoupler**

**Syntax:** `dec2unblank()`

**Description:** Enable the amplifier for the second decoupler if the amplifier is in pulse mode. For pulse mode, `dec2unblank` is required at least 2.0  $\mu$ s before pulses, spinlocks, and decoupling periods.

For continuous mode, `dec2unblank` is the default and the statement is not needed. The first decoupler is in continuous mode by default unless it has an associated receiver or is in the frequency band of another observe channel, for which it is in pulse mode.

To place the first decoupler in pulse mode, create the parameter `ampmode` and set the appropriate character to 'p'.

**Related:**

<code>dec2blank</code>	Blank amplifier of second decoupler
<code>decunblank</code>	Unblank amplifier of first decoupler
<code>dec3unblank</code>	Unblank amplifier of third decoupler
<code>dec4unblank</code>	Unblank amplifier of fourth decoupler

obsunblank      Unblank amplifier of observe channel

**dec3unblank****Unblank amplifier of third decoupler**

Syntax:      dec3unblank()

Description:      Enable the amplifier for the second decoupler if the amplifier is in pulse mode. For pulse mode, dec3unblank is required at least 2.0  $\mu$ s before pulses, spinlocks, and decoupling periods.

For continuous mode, dec3unblank is the default and the statement is not needed. The second decoupler is in continuous mode by default unless it has an associated receiver or is in the frequency band of another observe channel, for which it is in pulse mode.

To place the first decoupler in pulse mode, create the parameter `ampmode` and set the appropriate character to 'p'.

Related:	dec3blank	Blank amplifier of third decoupler
	decunblank	Unblank amplifier of first decoupler
	dec2unblank	Unblank amplifier of second decoupler
	dec4unblank	Unblank amplifier of fourth decoupler
	obsunblank	Unblank amplifier of observe channel

**dec4unblank****Unblank amplifier of fourth decoupler**

Syntax:      dec4unblank()

Description:      Enable the amplifier for the first decoupler if the amplifier is in pulse mode. For pulse mode, dec4unblank is required at least 2.0  $\mu$ s before pulses, spinlocks, and decoupling periods.

For continuous mode, dec4unblank is the default and the statement is not needed. The first decoupler is in continuous mode by default unless it has an associated receiver or is in the frequency band of another observe channel, for which it is in pulse mode.

To place the first decoupler in pulse mode, create the parameter `ampmode` and set the appropriate character to 'p'.

Related:      dec4blank      Blank amplifier of fourth decoupler

decunblank	Unblank amplifier of first decoupler
dec2unblank	Unblank amplifier of second decoupler
dec3unblank	Unblank amplifier of third decoupler
obsunblank	Unblank amplifier of observe channel

**delay**

**Execute time delay**

Syntax: `delay(time)`  
`double time; /* duration of delay, seconds */`

Description: Execute a delay for a specified number of seconds.

Arguments: `time` specifies the duration of the delay, in seconds. The DD2 MR system transmitter has a 25 ns minimum step and 12.5 ns resolution. The VNMRS system will have a 50 ns step and 12.5 ns resolution. Consult the configuration file to determine which transmitter is present.

Examples: `delay(d1);`  
`delay(d2/2.0);`

Related: `delay` Execute time delay  
`hsdelay` Execute time delay with optional homospoil pulse  
`vdelay` Execute time delay with fixed timebase and real-time count

**divn**

**Divide real-time integer values**

Syntax: `divn(vi,vj,vk)`  
`codeint vi; /* real-time variable for dividend */`  
`codeint vj; /* real-time variable for divisor */`  
`codeint vk; /* real-time variable for quotient */`

Description: Set the integer value of `vk` equal to the value of `vi` divided by the value of `vj`. The remainder of the division is truncated.

Arguments: `vi` contains the dividend, `vj` contains the divisor, and `vk` contains the resulting quotient. All three are real-time variables (`v1` to `v42`, `oph`, etc).

Examples: `divn(v2,v3,v4);`

Related: `add` Add real-time integer values  
`assign` Assign real-time integer value using real-time integer

dbl	Double real-time integer value
decr	Decrement real-time integer value
hlv	Assign half the value of real-time integer
incr	Increment real-time integer value
initval	Assign real-time integer value using numeric value
mod2	Assign real-time integer value modulo 2
mod4	Assign real-time integer value modulo 4
modn	Assign real-time integer value modulo n
mult	Multiply real-time integer values
sub	Subtract real-time integer values

**dps\_off****Turn off graphical display of statements**

Syntax: `dps_off()`

Description: Turn off `dps` display of statements. Pulse statements following `dps_off` are not shown in the graphical display.

Related: `dps_on` Turn on graphical display of statements  
`dps_show` Display pulse and delay icons in graphical display  
`dps_skip` Skip graphical display of next statement

**dps\_on****Turn on graphical display of statements**

Syntax: `dps_on()`

Description: Turn on `dps` display of statements. Pulse statements following `dps_on` are shown in the graphical display.

Related: `dps_off` Turn off graphical display of statements  
`dps_show` Display pulse and delay icons in graphical display  
`dps_skip` Skip graphical display of next statement

**dps\_show****Display pulse and delay icons in graphical display**

Syntax: (1)  
`dps_show("delay",time)`  
double time; /\* delay, seconds \*/

Syntax: (2) `dps_show("pulse",channel,label,width)`  
char \*channel; /\* "obs", "dec",  
"dec2",or "dec3" \*/  
char \*label; /\* text label selected  
by user \*/

```
double width;          /* pulse length, seconds
*/
```

Syntax:

```
(3)
dps_show("shaped_pulse",channel,label,width)
char *channel;        /* "obs", "dec",
"dec2",or "dec3" */
char *label;          /* text label selected
by user */
double width;         /* pulse length, seconds
*/
```

Syntax:

```
(4)
dps_show("simpulse",label_of_obs,width_of_obs,
char *label_of_obs;   /* text label
selected by user */
double width_of_obs;  /* pulse length,
seconds */
char *label_of_dec;   /* text label selected
by user */
double width_of_dec;  /* pulse length,
seconds */
```

Syntax:

```
(5)
dps_show("simshaped_pulse",label_of_obs,
width_of_obs,label_of_dec,width_of_dec)
char *label_of_obs;   /* text label selected
by user */
double width_of_obs;  /* pulse length,
seconds */
char *label_of_dec;   /* text label selected
by user */
double width_of_dec;  /* pulse length,
seconds */
```

Syntax:

```
(6)
dps_show("sim3pulse",label_of_obs,width_of_obs,
label_of_dec,width_of_dec,label_of_dec2,width_of_dec2)
char *label_of_obs;   /* text label selected
by user */
double width_of_obs;  /* pulse length,
seconds */
char *label_of_dec;   /* text label selected
by user */
double width_of_dec;  /* pulse length,
seconds */
char *label_of_dec2;  /* text label selected
by user */
double width_of_dec2; /* pulse length,
seconds */
```

Syntax:

```
(7)
dps_show("sim3shaped_pulse",label_of_obs,width_of_obs,
label_of_dec,width_of_dec,label_of_dec2,width_of_dec2)
char *label_of_obs;   /* text label selected
by user */
```

```

double width_of_obs; /* pulse length,
seconds */
char *label_of_dec; /* text label selected
by user */
double width_of_dec; /* pulse length,
seconds */
char *label_of_dec2; /* text label selected
by user */
double width_of_dec2; /* pulse length,
seconds */

```

Syntax: (8)

```

dps_show("zgradpulse",value,delay)
double value; /* gradient amplitude, z
channel, DAC units */
double delay; /* gradient-pulse length,
seconds */

```

Syntax: (9)

```

dps_show("rgradient",channel,value)
char channel; /* 'X', 'x', 'Y', 'y', 'Z',
or 'z' */
double value; /* gradient amplitude, DAC
units */

```

Syntax: (10)

```

dps_show("shapedgradient",pattern,width,amp,
channel,loops, wait)
char *pattern; /* pattern name */
double width; /* gradient-pulse length,
seconds */
double amp; /* gradient amplitude, DAC
units */
char channel; /* gradient axis 'x', 'y',
or 'z' */
int loops; /* number of loops */
int wait; /* WAIT or NOWAIT */

```

Description: Draw icons for pulses, gradient pulses, and delays in the dps graphical display. The statement name is always the first argument, followed by values for labels associated with the arguments of the statement. Use the `dps_off` statement before and the `dps_on` statement after `dps_show`.

Syntax 1 draws a line to represent a delay.

Syntax 2 draws a pulse icon.

Syntax 3 draws a shaped pulse icon.

Syntax 4 draws a pulse icon for the observe and first decoupler.

Syntax 5 draws a shaped pulse icon for observe and first decoupler.

Syntax 6 draws a pulse icon for observe, first and second decouplers.

Syntax 7 draws a shaped pulse icon for observe, first and second decouplers.

Syntax 8 draws a gradient pulse icon on the z channel.

Syntax 9 draws a gradient pulse icon on the specified channel.

Syntax 10 draws a shaped gradient pulse icon on a specified channel.

Examples:

```
dps_show("delay",d1);
dps_show("pulse","obs","obspulse",p1);
dps_show("pulse","dec","pw",pw);
dps_show("shaped_pulse","obs","shaped",p1*2)
;
dps_show("shaped_pulse","dec2","gauss",pw);
dps_show("simpulse","obs_pulse",p1,"dec_pulse",p2);
dps_show("simshaped_pulse","gauss",p1,"gauss",p2);
dps_show("sim3pulse","p1",p1,"p2",p2,"p1*2",p1*2);
dps_show("zgradpulse",123.0,d1);
dps_show("rgradient","x",1234.0);
dps_show("shapedgradient","sinc",1000.0,3000.0,'y',1,NOWAIT);
```

Related:

delay	Execute a time delay
dps_off	Turn off graphical display of statements
dps_on	Turn on graphical display of statements
dps_skip	Skip graphical display of next statement
pulse	Perform pulse with assigned values on observe channel
rgradient	Set DAC level of any one gradient axis
shaped_pulse	Perform shaped pulse on observe channel
shapedgradient	Perform shaped gradient pulse on any one axis
simpulse	Perform simultaneous pulses, observe and first decoupler
sim3pulse	Perform simultaneous pulses, observe, first and second decouplers
simshaped_pulse	Perform simultaneous shaped pulses, observe and first decoupler



<code>sim3shaped_pulse</code>	Perform simultaneous shaped pulses, observe, first and second decouplers
<code>zgradpulse</code>	Perform gradient pulse on the z channel

**dps\_skip****Skip graphical display of next statement**

Syntax: `dps_skip()`

Description: Skip the `dps` display of the next statement. The statement following `dps_skip` is not shown in the graphical display.

Related:	<code>dps_off</code>	Turn off graphical display of statements
	<code>Dps_on</code>	Turn on graphical display of statements
	<code>dps_show</code>	Display pulse and delay icons in graphical display

E, F

<code>elsenz</code>	Execute a real-time else statement
<code>endhardloop</code>	End hardware loop
<code>endacq</code>	End explicit acquisition
<code>endacq_obs</code>	End explicit acquisition in parallel section
<code>endacq_rcvr</code>	End explicit acquisition in parallel section
<code>endif</code>	Execute a real-time endif statement
<code>endloop</code>	End real-time loop
<code>endmsloop</code>	End switchable multislice loop
<code>endpeloop</code>	End switchable phase-encode loop
<code>exe_grad_rotation</code>	Set oblique gradient-coordinate rotation angles in real-time
<code>F_initval</code>	Always assign real-time variable using numeric value

**elsenz**

**Execute a real-time else statement**

Syntax: `elsenz (vi)`  
`codeint vi; /* real-time variable tested as 0 or not */`

Description: Placed between the `ifzero` and `endif` statements to execute succeeding statements if `vi` is non-zero. The `elsenz` statement can be omitted if it is not required. It is also not necessary for any statements to appear between the `ifzero` and the `elsenz`, or between the `elsenz` and the `endif` statements.

Arguments: `vi` is a real-time variable (`v1` to `v42`, `oph`, *etc*) tested for either being zero or non-zero.

Examples: `elsenz (v2);`

Related: `endif` Execute a real-time endif statement  
`ifzero` Execute a real-time equal-zero

**endacq**

**End explicit acquisition**

Syntax: `endacq()`

Description: Finalize the digital receiver after the explicit use of one or more `sample` statements to provide *explicit acquisition* of `np` points. The `endacq` statement takes no time itself but a 200 ns housekeeping delay must pass before the next scan or before the next execution of `startacq`. Usually the housekeeping delay occupies the `d1` period of the next scan and sets a minimum value for `d1`.

If an `endacq` statement is omitted, it will be inserted automatically at run-time. However, it is a good practice to use `endacq` as there may be some ambiguity as to where it is inserted automatically. For a single use of `sample (np / (2.0 * sw))`, the `endacq`

statement should immediately follow `sample`. For *windowed, explicit acquisition*, `endacq` is placed immediately after the `endloop` statement that has produced `np` points. For *compressed acquisition*, `startacq-endacq` pairs should surround each instance of `sample(np/2.0*sw)`.

Examples: `endacq()`;

Related:	<code>acquire</code>	Acquire data explicitly
	<code>rcvroff</code>	Turn off receiver and unblank observe amplifier
	<code>rcvron</code>	Turn on receiver and blank observe amplifier
	<code>sample</code>	Acquire data explicitly during time delay
	<code>startacq</code>	Initialize explicit acquisition

### `endacq_obs`

#### End explicit acquisition in parallel section

Syntax: `endacq_obs()`

Description: Analogous to the `endacq` statement but to be used in "obs" parallel sections of a pulse sequence.

Related:	<code>acquire_obs</code>	Acquire data explicitly in parallel section
	<code>acquire_rcvr</code>	Acquire data explicitly in parallel section
	<code>startacq_obs</code>	Initialize explicit acquisition in parallel section
	<code>startacq_rcvr</code>	Initialize explicit acquisition in parallel section
	<code>endacq_rcvr</code>	End explicit acquisition in parallel section
	<code>parallelacquire_obs</code>	Acquire data explicitly in parallel section
	<code>parallelacquire_rcvr</code>	Acquire data explicitly in parallel section

### `endacq_rcvr`

#### End explicit acquisition in parallel section

Syntax: `endacq_rcvr()`

Description: Analogous to the `endacq` statement but to be used in "rcvr" parallel sections of a pulse sequence.

Related:	<code>acquire_obs</code>	Acquire data explicitly in parallel section
	<code>acquire_rcvr</code>	Acquire data explicitly in parallel section
	<code>startacq_obs</code>	Initialize explicit acquisition in parallel section
	<code>startacq_rcvr</code>	Initialize explicit acquisition in parallel section
	<code>endacq_obs</code>	End explicit acquisition in parallel section
	<code>parallelacquire_obs</code>	Acquire data explicitly in parallel section
	<code>parallelacquire_rcvr</code>	Acquire data explicitly in parallel section

**endhardloop**

**End hardware loop (obsolete)**

Syntax: `endhardloop(count)`  
`int count; /* real-time variable with loop count */`

Description: End a real-time loop with `count` repetitions that begin with the `starthardloop` statement. This statement is identical to the `endloop` statement, except that the loop index is not accessible. The `endhardloop` statement is obsolete. Use the `loop - endloop` statements instead.

Arguments: `count` is the number of loop repetitions. It must be a real-time variable (`v1` to `v42`) or an index (`id2`, `id3` *etc*). The index of `endhardloop` is not accessible.

Related:	<code>endloop</code>	End real-time loop
	<code>loop</code>	Start real-time loop
	<code>starthardloop</code>	Start hardware loop

**endif**

**Execute a real-time endif statement**

Syntax: `endif(vi)`  
`codeint vi; /* real-time variable to test as 0 or not */`

Description: End conditional execution started by the `ifzero` and `elsenz` statements.

Arguments: `vi` is a real-time variable (`v1` to `v42`, `oph`, *etc*) that is tested for either being zero or non-zero.

Examples: `endif(v4);`

Related:	<code>elsenz</code>	Execute a real-time else statement
	<code>ifzero</code>	Execute a real-time equal-zero

**endloop****End real-time loop**

Syntax: `endloop(index)`  
`codeint index; /* real-time variable index */`

Description: End a loop that was started by a `loop` statement.

Arguments: `index` is a real-time variable used as an index to keep track of the number of times through the loop. It must be the same variable as that used in `loop` and its value should not be altered by any statements within the loop.

Examples: `endloop(v2);`

Related:	<code>endhardloop</code>	End hardware loop
	<code>loop</code>	Start real-time loop
	<code>starthardloop</code>	Start hardware loop

**endmsloop****End switchable multislice loop**

Syntax: `endmsloop(state,apv2)`

Syntax: `char state; /* compressed or standard */`

Syntax: `codeint apv2; /* current counter value */`

Description: End a loop that was started by a `msloop` statement.

Arguments: `state` is either 'c' to designate the compressed mode, or 's' to designate the standard arrayed mode. It should be the same value that was in the `state` argument in the `msloop` statement that it is ending.

`apv2` is a real-time variable used as an index to keep track of the number of times through the loop. It must be the same as that in `msloop` and not be altered by any statements within the loop.

Examples: `endmsloop(seqcon[1],v12);`

Related:	<code>msloop</code>	Start switchable multislice loop
	<code>endloop</code>	End real-time loop
	<code>endpeloop</code>	End switchable phase-encode loop
	<code>loop</code>	Start real-time loop

loopcheck	Check number of FIDS for compressed acquisition
peloop	Start switchable phase-encode loop

**endpeloop**

**End switchable phase-encode loop**

Syntax: `endpeloop(state,apv2)`  
`char state; /* compressed or standard */`  
`codeint apv2; /* real-time variable index */`

Description: End a loop that was started by a `peloop` statement.

Arguments: `state` is either 'c' to designate the compressed mode, or 's' to designate the standard arrayed mode. It should be the same value that was in the `state` argument in the `peloop` statement that it is ending.

`apv2` is a real-time variable used as an index to keep track of the number of times through the loop. It must be the same as that in `peloop` and not be altered by any statements within the loop.

Examples: `endpeloop(seqcon[1],v12);`

Related:	<code>msloop</code>	Start switchable multislice loop
	<code>endloop</code>	End real-time loop
	<code>endmsloop</code>	End switchable multislice loop
	<code>loop</code>	Start real-time loop
	<code>loopcheck</code>	Check number of FIDS for compressed acquisition
	<code>peloop</code>	Start switchable phase-encode loop

**exe\_grad\_rotation**

**Set oblique gradient-coordinate rotation angles in real-time**

Syntax: `exe_grad_rotation()`

Description: Set user-defined, oblique Euler rotation angles from a set of values defined by the statement `create_angle_list` and selected by `set_angle_list`. The angles selected by `set_angle_list` can have been indexed in real-time or run-time mode. All three angles, `psi`, `phi` and `theta` must be selected separately before `exe_grad_rotation` is executed. Otherwise a default angle is the last value set or 0.0.

Examples: `exe_grad_rotation();`

Related:	<code>exe_grad_rotation</code>	Set oblique gradient-coordinate rotation angles in real-time
	<code>create_angle_list</code>	Create 1D real-time list of angles
	<code>create_rotation_list</code>	Create list of oblique gradient-coordinate rotation angles
	<code>rot_angle</code>	Set user-defined oblique gradient-coordinate rotation angles
	<code>rot_angle_list</code>	Set oblique gradient-coordinate rotation angles from a list
	<code>rotate</code>	Set standard oblique gradient-coordinate rotation angles
	<code>set_angle_list</code>	Select angle from a 1D real-time list

**F\_initval****Always assign real-time variable using numeric value**

Syntax: `F_initval(number,vi)`  

```
double number;          /* numeric value used
for initialization */
codeint vi;             /* real-time variable
to be initialized */
```

Description: Initialize a real-time variable using a numeric value. The numeric input `number` is rounded to an integer and assigned to the value of `vi`. `F_initval` (unlike `initval`) is executed every time it is encountered in the sequence.

`number` is the real number, used to assign the value of the real-time variable.

`vi` is the real-time variable (`v1` to `v42`, *etc*) to be initialized

Examples: `F_initval(nt,v8);`

Related: `initval` Assign real-time variable using numeric value

## G

getarray	Obtain all values from arrayed parameter
getelem	Assign real-time integer using table element
getgradpowerintegral	Get gradient-power integrals for X, Y and Z axes
getorientation	Read image plane orientation
getstr	Obtain value of string parameter
getval	Obtain value of numeric parameter

**getarray****Obtain all values from arrayed parameter**

**Syntax:** `number=getarray(parname,values)`  
`char *parname; /* name of arrayed`  
`parameter */`  
`double array[]; /* array to contain`  
`values */`  
`return N /* number of elements`  
`in arrayed parameter */`

**Description:** Retrieve all values of a current VnmrJ arrayed parameter `parname`. `getarray` performs a `sizeof` on the array and returns the value as the integer `N`. This statement is very useful when obtaining parameter values for the creation of global lists, for example, statements such as `poffset_list` and `position_offset_list`.

The `getarray` statement is executed with every increment of a multidimensional or arrayed experiment. If `getarray` is to be used to obtain values for a global list, you should set protection bit 8 (256) of the arrayed parameter to limit its execution to the first increment. An example is the `pss` parameter, which is used with a compressed slice-select acquisition. Use the statements `create(pss,real); setprotect(pss,on,256)` to create the protected `pss` parameter.

**Arguments:** `parname` is the name of a numeric parameter in VnmrJ, from which to obtain the array of values.  
`values` is an arrayed variable in the pulse sequence to hold the values.

`N` is an integer return that holds the number of elements of the arrayed parameter `parname`.

**Examples:** `double upss[256]; /* declare array`  
`upss */ int uns;`  
`uns = getarray(upss,upss); /* get values from`  
`upss */`  
`poffset_list(upss,gss,uns,v12);`

**Related:** `getval` Obtain value of numeric parameter  
`getstr` Obtain value of string parameter



<code>offsetglist</code>	Create offset array from position and gradient-amplitude array
<code>offsetlist</code>	Create offset array from position and gradient amplitude
<code>poffset_list</code>	Create offset array from position array
<code>position_off set_list</code>	Create offset array from position array
<code>putarray</code>	Set all values of arrayed parameter from pulse sequence
<code>putstring</code>	Set string parameter from pulse sequence
<code>putvalue</code>	Set numeric parameter from pulse sequence

**getelem****Assign real-time integer using table element**

**Syntax:** `getelem(table,index,dest)`  
`codeint table; /* table variable  
containing element */`  
`codeint index; /* real-time index to  
element */`  
`codeint dest; /* real-time variable to  
be assigned */`

**Description:** Get an element from `table` determined by the value of `index` and assign it to a real-time variable `dest`. By default, tables area accessed sequentially, once per scan.

The `getelem` statement is necessary to access a table mutiple times within a single scan, for example, within a pair of real-time `loop-endloop` statements, or in a compressed 2D acquisition. The `autoincrement` statement, which provided this capability in earlier software, is now obsolete.

**Arguments:** `table` specifies the name of the table (t1 to t60).  
`index` is a real-time variable (`v1` to `v42`, `oph`, `ct`, `bsctr`, or `ssctr`) that contains the index of the desired table element. The first element of a table has an index of 0.  
`dest` is a real-time variable (`v1` to `v42`, `oph`, *etc*) to which the retrieved table element is assigned.

**Examples:** `getelem(t25,ct,v1);`

**Related:** `loadtable` Assign table elements from table text file  
`setreceiver` Set quadrature receiver phase cycle from table  
`settable` Assign integer array to table

**getgradpowerintegral**

**Get gradient-power integrals for X, Y and Z axes**

Syntax: `getgradpowerintegral(power_array)`  
`double power_array[3]; /* gradient power for X, Y and Z axes */`

Description: Get the cumulative gradient power integrals (total gradient energy) for 3 axes into a three-member array of *double* whose elements 0, 1 and 2 contain power for the X, Y and Z axes respectively. If a gradient axis is not configured, its corresponding array element is set to zero. The

Arguments: `power_array` must be defined as a three-element C-array of *double* to hold the output energy values.

Examples: `double gradenergy(3);`  
`.`  
`getgradpowerintegral(gradenergy);`  
`printf("X grad energy = %g\n", gradenergy[0]);`  
`printf("Y grad energy = %g\n", gradenergy[1]);`  
`printf("Z grad energy = %g\n", gradenergy[2]);`

Related: `showpowerintegral` Show gradient-power integrals for X, Y and Z axes

**getorientation**

**Read image plane orientation**

Syntax: `<error_return =>`  
`getorientation(&char1, &char2, &char3, search_string)`  
`char *char1, *char2, *char3; /* program variable pointers */`  
`char *search_string; /* pointer to search string */`

Description: Read in and processes the value of a string parameter, used typically for control of magnetic field gradients. The source of the string value is typically a user-created parameter available in the current parameters of the experiment used to initiate acquisition.

Arguments: `error_return` can contain the following values:  
`error_return` is set to zero if `getorientation` was successful in finding the parameter given in `cc` and reading in the value of that parameter.  
`cc` is set to -1 if `search_string` was not empty but it did not contain the correct characters.  
`error_return` is set to a value greater than zero if the procedure failed or if the string value is made up of characters other than n, x, y, and z.  
`char1`, `char2`, and `char3` are user-created program variables of type `char` (single characters). The address operator (`&`) is used with these arguments

to pass the address, rather than the values of these variables, to `getorientation`.

`search_string` is a literal string that `getorientation` will search for in the VnmrJ parameter set, that is, the parameter name. For example, if `search_string="orient"`, the value of parameter `orient` will be accessed. The value of the parameter should not exceed three characters and should only be made up of characters from the set `n, x, y, and z`.

The message `can't find variable in tree aborts getorientation`. This means that either there is no string associated with `search_string` or the parameter name cannot be found.

Examples:

```
(1) pulsesequence()
{
...
char phase,read,slice;
...
getorientation(&read,&phase,&slice,"orient")
;
...
}
(2) pulsesequence()
{
...
char rd, ph, sl;
int error;
...
error=getorientation(&rd,&ph,&sl,"ort");
...
}
```

Related: `rgradient` Set DAC level of any one gradient axis

### **getstr** Obtain value of string parameter

Syntax: `getstr(paname,string)`  
`char *paname; /* name of string parameter */`  
`char *string; /* variable to contain the string */`

Description: Obtain the value of the current string parameter `paname` and set a variable `string` in the pulse sequence. If `paname` is not found in the current parameter table, `string` is set to the null string and the PSG produces a warning message

Arguments: `paname` is the name of a string parameter in VnmrJ.  
`string` is a string variable in the pulse sequence to hold the string.

Examples: `char xpol[MAXSTR]; getstr("xpol",xpol);`

Related:	<code>getarray</code>	Obtain all values from arrayed parameter
	<code>getval</code>	Obtain value of numeric parameter
	<code>putarray</code>	Set all values of arrayed parameter from pulse sequence
	<code>putstring</code>	Set string parameter from pulse sequence
	<code>putvalue</code>	Set numeric parameter from pulse sequence

**getval****Obtain value of numeric parameter**

**Syntax:** `value = getval(parname)`  
`char *parameter_name; /* name of numeric parameter */`

**Description:** Obtain the value of the current numeric parameter `parname` and return its value to a variable `value` in the pulse sequence. If `parname` is not found in the current parameter table, `value` is set to zero and the PSG produces a warning message.

The `getval` statement can be used as an argument of another statement in the pulse sequence, to avoid the need to explicitly define the variable `value`. The `dps` graphical display labels the icon for the statement with the string value `parname`.

**Arguments:** `parname` is the name of a numeric parameter in VnmrJ.

`value` is a *double* variable in the pulse sequence to hold the value.

**Examples:** `J=getval("J");`  
`acqtime=getval("at");`  
`delay(getval("mix"));`

Related:	<code>getarray</code>	Obtain all values from arrayed parameter
	<code>getstr</code>	Obtain value of string parameter
	<code>putarray</code>	Set all values of arrayed parameter from pulse sequence
	<code>putstring</code>	Set string parameter from pulse sequence
	<code>putvalue</code>	Set numeric parameter from pulse sequence

## H

hlv                   Assign half the value of real-time integer  
 hsdelay             Execute time delay with optional homospoil pulse

## hlv

**Assign half the value of real-time integer**

Syntax:    hlv(vi, vj)  
           codeint vi;                   /\* real-time variable for  
           input \*/  
           codeint vj;                  /\* real-time variable for  
           output \*/

Description: Set the value of **vj** equal to the integer part of one-half of the value of **vi**. The remainder of the division is truncated.

Arguments: **vi** contains the value to be halved and **vj** contains the result. Each argument must be a real-time variable (**v1** to **v42**, **oph**, etc).

Examples:    hlv(v2, v5);

Related:    add           Add real-time integer values  
           assign         Assign real-time integer value using  
                           real-time integer  
           dbl            Double real-time integer value  
           decr          Decrement real-time integer value  
           divn          Divide real-time integer values  
           incr          Increment real-time integer value  
           initval       Assign real-time integer value using  
                           numeric value  
           mod2          Assign real-time integer value modulo 2  
           mod4          Assign real-time integer value modulo 4  
           modn          Assign real-time integer value modulo n  
           mult          Multiply real-time integer values  
           sub           Subtract real-time integer values

## hsdelay

**Execute time delay with optional homospoil pulse**

Syntax:    hsdelay(time)  
           double time;                 /\* delay in sec \*/

Description: Execute a time delay for a specified number of seconds that may contain an optional homospoil pulse on the Z shim gradient. To execute a homospoil pulse, the `hsdelay` statement must follow a `status` statement and the `hs` parameter must be set with 'y' for the character corresponding to the status period. If the parameter `hs` is set correctly, `hsdelay` inserts a homospoil pulse of length `hst` seconds at the beginning of `hsdelay`.

The *global* parameter `gradtype='mnh'` configures the Z shim as a gradient channel and provides an alternative to *status* and *hsdelay*. If `gradtype` is set correctly, the statement `hsdelay(hst)` can be replaced with the statement `zgradpulse(hst,gzlvl)`, or the homospoil Z shim can be turned on and off with `rgradient`. A value of `gzlvl > 0` turns on the homospoil shim.

**Arguments:** `time` specifies the duration of the delay, in seconds. The DD2 MR system transmitter has a 25 ns minimum step and 12.5 ns resolution. Older systems will have a 50 ns step and 12.5 ns resolution. Consult the configuration file to determine which transmitter is present.

**Examples:** `hsdelay(d1);`  
`hsdelay(1.5e-3);`

<b>Related:</b>	<code>delay</code>	Execute a time delay
	<code>setstatus</code>	Set decoupler status of any channel
	<code>status</code>	Set status of decoupler or homospoil
	<code>statusdelay</code>	Execute status statement within a time delay
	<code>vdelay</code>	Execute a time delay with fixed timebase and real-time count

ifrtGE	Execute real-time greater-than-or-equal
ifrtGT	Execute real-time greater-than
ifrtLE	Execute real-time less-than-or-equal
ifrtLT	Execute real-time less-than
ifrtEQ	Execute real-time equal
ifrtNEQ	Execute real-time not-equal
ifzero	Execute real-time equal-zero
incr	Increment real-time integer value
init_decpattern	Create pattern file for waveform decoupling
init_gradpattern	Create pattern file for gradient shape
init_rfpattern	Create pattern file for shaped pulse
initval	Assign real-time integer value using numeric value

**ifrtGE****Execute real-time greater-than-or-equal**

**Syntax:** `ifrtGE(rtarg1,rtarg2,rtarg3);`  
`codeint rtarg1; /* first real-time`  
`variable for comparison */`  
`codeint rtarg2; /* second real-time`  
`variable for comparison */`  
`codeint rtarg3; /* real-time variable to`  
`be set as 0 or not */`

**Description:** Execute succeeding statements if  $rtarg1 \geq rtarg2$  and then execute the code until an `endif(rtarg3)` or an `elsenz(rtarg3)` is encountered. A matching `endif(rtarg3)` is required. An optional `elsenz(rtarg3)` may be used.

**Arguments:** `rtarg1`, `rtarg2`, and `rtarg3` are real-time variables.

**Examples:**

```
ifrtGE(v1,v2,v3);
pulse(pw,v2);
delay(d3);

elsenz(v3);
pulse(2.0*pw,v2);
delay(d3/2.0);

endif(v3);
```

**Related:**

<code>elsenz</code>	Execute real-time else statement
<code>endif</code>	Execute real-time endif statement
<code>ifrtEQ</code>	Execute real-time equal
<code>ifrtGT</code>	Execute real-time greater-than
<code>ifrtLE</code>	Execute real-time less-than-or-equal
<code>ifrtLT</code>	Execute real-time less-than
<code>iftrtNEQ</code>	Execute real-time not-equal

Related: `elsenz` Execute real-time else statement  
`ifzero` Execute real-time equal-zero

**ifrtGT**

**Execute real-time greater-than**

Syntax: `firtGt(rtarg1,rtarg2,rtarg3)`  
`codeint rtarg1; /* first real-time variable for comparison */`  
`codeint rtarg2; /* second real-time variable for comparison */`  
`codeint rtarg3; /* real-time variable to be set as 0 or not */`

Description: Execute succeeding statements. If `rtarg1 > rtarg2`, then execute the code until an `endif(rtarg3)` or an `elsenz(rtarg3)` is encountered. A matching `endif(rtarg3)` is required. An optional `elsenz(rtarg3)` may be used.

Arguments: `rtarg1`, `rtarg2`, and `rtarg3` are real-time variables.

Examples: `ifrtGT(v1,v2,v3);`  
`pulse(pw,v2);`  
`delay(d3);`  
`elsenz(v3);`  
`pulse(2.0*pw,v2);`  
`delay(d3/2.0);`  
`endif(v3);`

Related: `elsenz` Execute real-time else statement  
`endif` Execute real-time endif statement  
`ifrtEQ` Execute real-time equal  
`ifrtGE` Execute real-time greater-than-or-equal  
`ifrtLE` Execute real-time less-than-or-equal  
`ifrtLT` Execute real-time less-than  
`iftrtNEQ` Execute real-time not-equal  
`ifzero` Execute real-time equal-zero

**ifrtLE**

**Execute real-time less-than-or-equal**

Syntax: `ifrtLE(rtarg1,rtarg2,rtarg3);`  
`codeint rtarg1; /* first real-time variable for comparison */`  
`codeint rtarg2; /* second real-time variable for comparison */`  
`codeint rtarg3; /* real-time variable to be set as 0 or not */`

Description: Execute succeeding statements. If `rtarg1 ≥ rtarg2`, then execute the code until an `endif(rtarg3)` or an `elsenz(rtarg3)` is



encountered. A matching `endif(rtarg3)` is required. An optional `elsenz(rtarg3)` may be used.

**Arguments:** `rtarg1`, `rtarg2`, and `rtarg3` are real-time variables.

**Examples:**

```
ifrtLE(v1,v2,v3);
pulse(pw,v2);
delay(d3);
```

```
elsenz(v3);
pulse(2.0*pw,v2);
delay(d3/2.0);
```

```
endif(v3);
```

<b>Related:</b>	<code>elsenz</code>	Execute real-time else statement
	<code>endif</code>	Execute real-time endif statement
	<code>ifrtEQ</code>	Execute real-time equal
	<code>ifrtGE</code>	Execute real-time greater-than-or-equal
	<code>ifrtGT</code>	Execute real-time greater-than
	<code>ifrtLT</code>	Execute real-time less-than
	<code>iftrtNEQ</code>	Execute real-time not-equal
	<code>ifzero</code>	Execute real-time equal-zero

## **ifrtLT**

### **Execute real-time less-than**

**Syntax:**

```
ifrtLT(rtarg1,rtarg2,rtarg3);
codeint rtarg1; /* first real-time
variable for comparison */
codeint rtarg2; /* second real-time
variable for comparison */
codeint rtarg3; /* real-time variable to
be set as 0 or not */
```

**Description:** Execute succeeding statements. If `rtarg1 < rtarg2`, then execute the code until an `endif(rtarg3)` or an `elsenz(rtarg3)` is encountered. A matching `endif(rtarg3)` is required. An optional `elsenz(rtarg3)` may be used.

**Arguments:** `rtarg1`, `rtarg2`, and `rtarg3` are real-time variables.

**Examples:**

```
ifrtLT(v1,v2,v3);
pulse(pw,v2);
delay(d3);
```

```
elsenz(v3);
pulse(2.0*pw,v2);
delay(d3/2.0);
```

```
endif(v3);
```

Related: `elsenz` Execute real-time else statement  
`endif` Execute real-time endif statement  
`ifrtEQ` Execute real-time equal  
`ifrtGE` Execute real-time greater-than-or-equal  
`ifrtGT` Execute real-time greater-than  
`ifrtLE` Execute real-time less-than-or-equal  
`iftrtNEQ` Execute real-time not-equal  
`ifzero` Execute real-time equal-zero

**ifrtEQ**

**Execute real-time equal**

Syntax: `ifrtEQ(rtarg1,rtarg2,rtarg3);`  
`codeint rtarg1; /* first real-time`  
`variable for comparison */`  
`codeint rtarg2; /* second real-time`  
`variable for comparison */`  
`codeint rtarg3; /* real-time variable to`  
`be set as 0 or not */`

Description: Execute succeeding statements. If `rtarg1`  $\neq$  `rtarg2`, then execute the code until an `endif(rtarg3)` or an `elsenz(rtarg3)` is encountered. A matching `endif(rtarg3)` is required. An optional `elsenz(rtarg3)` may be used.

Arguments: `rtarg1`, `rtarg2`, and `rtarg3` are real-time variables.

Examples: `ifrtEQ(v1,v2,v3);`  
`pulse(pw,v2);`  
`delay(d3);`  
  
`elsenz(v3);`  
`pulse(2.0*pw,v2);`  
`delay(d3/2.0);`  
  
`endif(v3);`

Related: `elsenz` Execute real-time else statement  
`endif` Execute real-time endif statement  
`ifrtGE` Execute real-time greater-than-or-equal  
`ifrtGT` Execute real-time greater-than  
`ifrtLE` Execute real-time less-than-or-equal  
`ifrtLT` Execute real-time less-than  
`iftrtNEQ` Execute real-time not-equal  
`ifzero` Execute real-time equal-zero

**ifrtNEQ****Execute real-time not-equal**

**Syntax:** `ifrtNEQ(rtarg1,rtarg2,rtarg3);`  
`codeint rtarg1; /* first real-time`  
`variable for comparison */`  
`codeint rtarg2; /* second real-time`  
`variable for comparison */`  
`codeint rtarg3; /* real-time variable to`  
`be set as 0 or not */`

**Description:** Execute succeeding statements. If `rtarg1 = rtarg2`, then execute the code until an `endif(rtarg3)` or an `elsenz(rtarg3)` is encountered. A matching `endif(rtarg3)` is required. An optional `elsenz(rtarg3)` may be used.

**Arguments:** `rtarg1`, `rtarg2`, and `rtarg3` are real-time variables.

**Examples:** `ifrtEQ(v1,v2,v3);`  
`pulse(pw,v2);`  
`delay(d3);`  
  
`elsenz(v3);`  
`pulse(2.0*pw,v2);`  
`delay(d3/2.0);`  
  
`endif(v3);`

**Related:** `elsenz` Execute real-time else statement  
`endif` Execute real-time endif statement  
`ifrtEQ` Execute real-time equal  
`ifrtGE` Execute real-time greater-than-or-equal  
`ifrtGT` Execute real-time greater-than  
`ifrtLE` Execute real-time less-than-or-equal  
`ifrtLT` Execute real-time less-than  
`ifzero` Execute real-time equal-zero

**ifzero****Execute real-time equal-zero**

**Syntax:** `fzero(vi)`  
`codeint vi; /* real-time variable to`  
`test as 0 or not */`

**Description:** Execute succeeding statements if `vi` is zero. If `vi` is non-zero and an `elsenz` statement exits before the next `endif` statement, execution moves to the `elsenz` statement. Conditional execution ends when the `endif` statement is reached. It is not necessary for any statements to appear between the `ifzero` and the `elsenz` or between the `elsenz` and the `endif` statements.

**Arguments:** `vi` is a real-time variable (`v1` to `v42`, `oph`, *etc*) that is tested for a zero or non-zero value.

**Examples:**

```

mod2(ct,v1);          /* v1=010101... */
ifzero(v1);          /* test if v1 is zero */
pulse(pw,v2);        /* execute if v1 is zero */
delay(d3);           /* execute if v1 is zero */
elsenz(v1);          /* test if v1 is non-zero */
/*
pulse(2.0*pw,v2);    /* execute if v1 is non-zero */
delay(d3/2.0);       /* execute if v1 is non-zero */
endif(v1);           /* end conditional
execution */

```

**Related:**

elsenz	Execute real-time else statement
endif	Execute real-time endif statement
ifrtEQ	Execute real-time equal
ifrtGE	Execute real-time greater-than-or-equal
ifrtGT	Execute real-time greater-than
ifrtLE	Execute real-time less-than-or-equal
ifrtLT	Execute real-time less-than
iftrtNEQ	Execute real-time not-equal

#### incr

#### Increment real-time integer value

**Syntax:**

```

incr(vi)
codeint vi;          /* real-time variable to be
incremented */

```

**Description:** Increment the integer value of *vi* by 1.

**Arguments:** *vi* is the integer to be incremented, It must be a real-time variable (*v1* to *v42*, *oph*, *etc*).

**Examples:**

```

incr(v4);

```

**Related:**

add	Add real-time integer values
assign	Assign real-time integer value using real-time integer
dbl	Double real-time integer value
decr	Decrement real-time integer value
divn	Divide real-time integer values
hlv	Assign half the value of real-time integer
initval	Assign real-time integer value using numeric value
mod2	Assign real-time integer value modulo 2
mod4	Assign real-time integer value modulo 4
modn	Assign real-time integer value modulo n

mult      Multiply real-time integer values  
 sub       Subtract real-time integer values

**init\_rfpattern****Create pattern file for shaped pulse**

**Syntax:** `init_rfpattern(pattern,rfpat_struct,nsteps)`  
`char *pattern;                    /* name of .RF`  
`text file */`  
`RFpattern *rfpat_struct;        /* pointer to`  
`struct RFpattern */`  
`int nsteps;                      /* number of steps`  
`in pattern */`  
`typedef struct _RFpattern {`  
`double phase;                   /* phase of pattern`  
`step */`  
`double amp;                     /* amplitude of pattern`  
`step */`  
`double time;                    /* relative time of`  
`pattern step */`  
`double gate;                    /* gate 1.0 for on and`  
`0.0 for off */`  
`} RFpattern`

**Description:** Create and define a .RF pattern in shapelib for shaped pulses with the root name pattern based on information in the RFpattern structure rfpattern. The pulse sequence should contain code to populate the structure with nsteps entries of phase, amp, time, and gate.

**Arguments:** pattern is the root name of a text file in the shapelib directory with a .DEC extension to store the decoupling pattern.

rfpat\_struct is the a structure with values to create the pattern.

nsteps is the number of steps in the pattern.

**Examples:**

```
#include "standard.h"
pulsesequance()
{
int nsteps;
RFpattern pulse1[512], pulse2[512];
Gpattern gshape[512];
...
nsteps = 0;
for (j=0; j<256; j++) {
pulse1[j].phase = (double)j*0.5;
pulse1[j].amp = (double)j*2;
pulse1[j].time = 1.0;
nsteps = nsteps +1;
}
init_rfpattern(plpat,pulse1,nsteps);
nsteps = 512;
for (j=0; j<nsteps; j++) {
gshape[j].amp = 32767.0*sin((double)j/50.0);
gshape[j].time = 1.0;
```

```

}
init_gradpattern("gpat", gshape, nsteps);
...
shaped_pulse(plpat, p1, v1, rof1, rof1);
...
shapedgradient("gpat", .01, 16000.0, 'z', 1,
WAIT);
...
}

```

Related: `init_decpattern` Create pattern file for waveform decoupling  
`init_gradpattern` Create pattern file for gradient shape

**init\_decpattern****Create pattern file for waveform decoupling**

Syntax: `init_decpattern`  
`(pattern, decpat_struct, nsteps)`  
`char *pattern; /* name of .DEC`  
`text file */`  
`DECpattern *rfpat_struct; /* pointer to`  
`struct DECpattern */`  
`int nsteps; /* number of steps`  
`in pattern */`  

```

typedef struct _DECpattern {
double phase; /* phase of pattern step */
double amp; /* amplitude of pattern step */
double duration; /* tip-angle duration */
double gate; /* gate 1.0 for on and
0.0 for off*/
} DECpattern

```

Description: Create and define a .DEC pattern in shapelib for programmable waveform decoupling with the root name `pattern` based on information in the `DECpattern` structure `decpat_structure`. The pulse sequence should contain code to populate the structure with `nsteps` entries of phase, amp, duration, and gate.

Arguments: `pattern` is the root name of a text file in the shapelib directory with a .DEC extension to store the decoupling pattern.

`decpat_struct` is a structure with values to create the pattern.

`nsteps` is the number of steps in the pattern.

Related: `init_gradpattern` Create pattern file for gradient shape  
`init_rfpattern` Create pattern file for shaped pulse

**init\_gradpattern****Create pattern file for gradient shape**

Syntax: `init_gradpattern(pattern_name, gradpat_struct, nsteps)`

```
char *pattern; /* name of .GID pattern file */
Gpattern *gradpat_struct; /* pointer to
struct Gpattern */
int nsteps; /* number of steps in pattern */
typedef struct _Gpattern {
double amp; /* amplitude of pattern step */
double time; /* pattern step length in sec */
} Gpattern
```

Description: Create and define a .GRD pattern in shapelib for a gradient shape with the root name `pattern` based on information in the `Gpattern` structure `gradpat_structure`. The pulse sequence should contain code to populate the structure with `nsteps` entries of `amp` and `time` within a pulse sequence.

Arguments: `pattern` is the root name of a text file in the shapelib directory with a .GRD extension to store the gradient shape..

`gradpat_struct` is a structure with values to create the pattern.

`nsteps` is the number of steps in the pattern.

Related: `init_decpattern` Create pattern file for waveform decoupling  
`init_rfpattern` Create pattern file for shaped pulse

**initval****Assign real-time variable using a numeric value**

Syntax: `initval(number, vi)`  

```
double number; /* numeric value used
for initialization */
codeint vi; /* real-time variable
to be initialized */
```

Description: Initialize a real-time variable using a numeric value. The numeric input `number` is rounded to an integer and assigned to the value of `vi`. `initval` is executed *once and only once* at the start of a non-arrayed 1D experiment or at the start of each increment in an *n*-dimensional or an arrayed experiment, not at the start of each transient.

`number` is the real number, used to assign the value of the real-time variable.

`vi` is the real-time variable (`v1` to `v42`, etc) to be initialized

Examples: (1) `initval(nt, v8);`  
(2) `ifzero(ct);`  
`assign(v8, v7);`

```
elsenz(ct);  
decr(v7);  
endif(ct);
```

<b>Related:</b>	<code>add</code>	Add real-time integer values
	<code>assign</code>	Assign real-time integer value using real-time integer
	<code>dbl</code>	Double real-time integer value
	<code>decr</code>	Decrement real-time integer value
	<code>divn</code>	Divide real-time integer values
	<code>hlv</code>	Assign half the value of real-time integer
	<code>incr</code>	Increment real-time integer value
	<code>mod2</code>	Assign real-time integer value modulo 2
	<code>mod4</code>	Assign real-time integer value modulo 4
	<code>modn</code>	Assign real-time integer value modulo n
	<code>mult</code>	Multiply real-time integer values
	<code>sub</code>	Subtract real-time integer values



## K

kzloop	End real-time loop with fixed duration
kzloop	Start real-time loop with fixed duration

**kzloop****End real-time loop with fixed duration**

**Syntax:** `void kzloop(index)`  
`codeint index /* real-time variable index */`

**Description:** End a loop that was started by a `kzloop` statement.

**Arguments:** `index` is a real-time variable used as an index to keep track of the number of times through the loop. It must be the same variable used in `kzloop` and its value should not be altered by any real-time math statements.

**Examples:** `kzloop(v10);`

<b>Related:</b>	<code>endloop</code>	End real-time loop
	<code>kzloop</code>	Start real-time loop with fixed duration
	<code>loop</code>	Start real-time loop
	<code>parallelend</code>	End parallel section of pulse sequence
	<code>parallelstart</code>	Start parallel section of pulse sequence
	<code>rlendloop</code>	End real-time loop with fixed count
	<code>rlloop</code>	Start real-time loop with fixed count

**kzloop****Start real-time loop with fixed duration**

**Syntax:** `int kzloop(time, vcount, index)`  
`double time /* duration of the loop, seconds */`  
`codeint vcount /* number of times to loop */`  
`codeint index /* real-time index to steps of loop */`  
`int return /* number of times to loop */`

**Description:** Start a loop to execute statements in real-time with a fixed number of repetitions. The `kzloop` statement ends the loop begun by `kzloop`. This statement should be used to replace `loop` in parallel sections of a pulse sequence, created by `parallelstart-parallelend`. Unlike `loop` the total duration of `kzloop` is known at run time.

**Arguments:** `time` is a duration that contains the loop. The number of times through the loop is set to complete the loop within the duration time. A delay is added after the loop, if necessary, to complete the duration time.

`vcount` is a real-time variable containing the number of times through the loop. It is calculated and initialized based on the duration time. The value of `vcount` should not be changed by any real-time math statements.

`index` is a real-time variable used as an index to keep track of the number of times through the loop. It must be the same variable as that used in `kzendloop` and its value should not be altered by any real-time math statements.

**Examples:**

In this example, the loop count is calculated as  $\text{trunc}(14.0/3.0) = 4$  repetitions of 3 seconds. The remaining duration,  $14.0 - 4*3.0 = 2.0$  seconds will be added as a delay after the repetitions.

```
kzloop(14.0, v2, v11);  
    delay(3.0);  
kzendloop(v11);
```

<b>Related:</b>	<code>endloop</code>	End real-time loop
	<code>kzendloop</code>	End real-time loop with fixed duration
	<code>loop</code>	Start real-time loop
	<code>parallelend</code>	End parallel section of pulse sequence
	<code>parallelstart</code>	Start parallel section of pulse sequence
	<code>rlendloop</code>	End real-time loop with fixed count
	<code>rlloop</code>	Start real-time loop with fixed count

## L

<code>lk_hold</code>	Set lock correction circuitry to hold correction
<code>lk_sample</code>	Set lock correction circuitry to sample lock signal
<code>loadtable</code>	Assign table elements from table text file
<code>loop</code>	Start real-time loop
<code>loop_check</code>	Check number of FIDs for compressed acquisition

**lk\_hold****Set lock correction circuitry to hold correction**

Syntax: `lk_hold()`

Description: Make the lock correction circuitry hold the correction to the `z0` constant, thereby ignoring any influence on the lock signal such as a gradient or pulses at  $^2\text{H}$  frequency. The correction remains in effect until the statement `lk_sample` is called or until the end of an experiment. If an acquisition is aborted, the lock correction circuitry will be reset to sample the lock signal.

Related: `lk_sample` Set lock correction circuitry to hold correction

**lk\_sample****Set lock correction circuitry to sample lock signal**

Syntax: `lk_sample()`

Description: Make the lock correction circuitry continuously sample the lock signal and correct `z0` with the time constant as set by the parameter `lockacqtc`. The correction remains in effect until the statement `lk_hold` is called.

Related: `lk_hold` Set lock correction circuitry to hold correction

**loadtable****Assign table elements from table text file**

Syntax: `loadtable(file)`  
`char *file; /* name of table file */`

Description: Load table elements from a text table file in `tablib`. `loadtable` can be called multiple times within a pulse sequence but the same table name may not be used more than once within all the table files accessed by the sequence.

`file` is the name of a table file in a user's private `tablib` or in the system `tablib`.

Examples: `loadtable("tabletest");`

Related: `getelem` Assign real-time integer using table element  
`setreceiver` Set quadrature receiver phase cycle from table  
`settable` Assign integer array to table

**loop**

**Start real-time loop**

Syntax: `loop(count, index)`  
`codeint count /* number of times to loop */`  
`codeint index /* real-time index to steps of loop */`

Description: Start a loop to execute statements in real-time within one scan of a pulse sequence. The loop ends with the `endloop` statement.

Arguments: `count` is a real-time variable used to specify the number of times through the loop. `count` can be any positive number, including zero.

Arguments: `index` is a real-time variable used as an index to keep track of the number of times through the loop. It must be the same variable as that used in `endloop` and its value should not be altered by any statements within the loop.

Examples: `initval(5.0, v1); /* set first loop count */`  
`loop(v1, v10);`  
`dbl(ct, v2); /* set second loop count */`  
`loop(v2, v9);`  
`rgpulse(p1, v1, 0.0, 0.0);`  
`endloop(v9);`  
`delay(d2);`  
`endloop(v10);`

Related: `endhardloop` End hardware loop  
`loop` Start real-time loop  
`starthardloop` Start hardware loop

**loop\_check**

**Check number of FIDs for compressed acquisition**

Syntax: `loop_check()`

Description: Check that the number of FIDs in a compressed acquisition `nf` is consistent with the number of slices `ns`, number of echoes `ne`, number of phase encoding steps in the various dimensions (`nv`, `nv2`, `nv3`), and `seqcon`.

<b>Related:</b>	<code>msloop</code>	Start switchable multislice loop
	<code>endloop</code>	End real-time loop
	<code>endmsloop</code>	End switchable multislice loop
	<code>endpeloop</code>	End switchable phase-encode loop
	<code>loop</code>	Start real-time loop
	<code>peloop</code>	Start switchable phase-encode loop

## M

magradient	Set three-axis gradient at magic angle
magradpulse	Perform three-axis gradient pulse at magic angle
mashapedgradient	Perform three-axis gradient shape at magic angle
mashapedgradpulse	Perform three-axis shaped gradient pulse at magic angle
mod2	Assign real-time integer value modulo 2
mod4	Assign real-time integer value modulo 4
modn	Assign real-time integer value modulo n
msloop	Start switchable multislice loop
mult	Multiply real-time integer values

**magradient**      **Set three-axis gradient at the magic angle**

Syntax: `magradient(gradlvl)`  
`double gradlvl;      /* magic-angle gradient`  
`amplitude, Gauss/cm */`

Description: Apply static gradients simultaneously to the *x*, *y*, and *z* axes at the magic angle, 54.7° to  $B_0$ . Information from a gradient table is used to convert Gauss/cm into DAC settings for the three axes. The gradients remain set until they are turned off. To turn off the gradients, add another `magradient` statement with `gradlvl` set to 0.0 or insert the statement `zero_all_gradients`.

Use `magradient(gradlvl); delay(width); magradient(0.0)` to apply a magic-angle gradient pulse with duration `width`. Use the `magradient` construction rather than `magradpulse` if additional statements are required before the end of the gradient pulse.

Arguments: `gradlvl` is the gradient amplitude at the magic angle, in Gauss/cm.

Examples: `magradient(3.0);`  
`pulse(pw, oph);`  
`delay(0.001 - pw);`  
`magradient(0.0);`

Related:	<code>magradpulse</code>	Perform three-axis gradient pulse at magic angle
	<code>mashapedgradient</code>	Perform three-axis shaped gradient at magic angle
	<code>mashapedgradpulse</code>	Perform three-axis shaped gradient pulse at magic angle
	<code>zero_all_gradients</code>	Zero gradient DAC level for all axes

**magradpulse Perform three-axis gradient pulse at magic angle**

Syntax: `magradpulse(gradlvl,width)`  
`double gradlvl; /* magic-angle gradient`  
`amplitude, Gauss/cm */`  
`double width; /* duration of the gradient`  
`pulse, seconds */`

Description: Apply a static gradient pulse simultaneously to the  $x$ ,  $y$ , and  $z$  axes at the magic angle,  $54.7^\circ$  to  $B_0$ . Information from the gradient table is used to convert Gauss/cm into DAC settings for the three axes. The gradient DACS are set to 0.0 at the completion of the pulse.

Arguments: `gradlvl` is the gradient-pulse amplitude at the magic angle, in Gauss/cm.  
`width` is the duration of the gradient pulse, in seconds.

Examples: `magradpulse(3.0,0.001);`

Related:	<code>magradient</code>	Set three-axis gradient at magic angle
	<code>mashapedgradient</code>	Perform three-axis shaped gradient at magic angle
	<code>mashapedgradpulse</code>	Perform three-axis shaped gradient pulse at magic angle
	<code>zero_all_gradients</code>	Zero gradient DAC level for all axes

**mashapedgradient****Perform three-axis gradient shape at magic angle**

Syntax: `mashapedgradient(pattern,gradlvl,width,loops,wait)`  
`char *pattern; /* names of a .GRD file */`  
`double gradlvl; /* magic angle gradient`  
`amplitude, Gauss/cm */`  
`double width; /* duration of gradient shape,`  
`seconds */`  
`int loops; /* not implemented */`  
`int wait; /* WAIT or NOWAIT */`

Description: Apply a static, gradient shape with duration `width` simultaneously to the  $x$ ,  $y$ , and  $z$  axes at the magic angle,  $54.7^\circ$  to  $B_0$ . Information from the gradient table is used to convert Gauss/cm into DAC settings for the three axes. When `mashapedgradient` completes, the gradients remain set at their last value until they are turned off. To turn off the gradients, add the statement `magradient` with `gradlvl` set to 0.0 or insert the statement `zero_all_gradients`.

Use `mashapedgradient` with the `NOWAIT` option rather than `mashapedgradpulse` if additional statements are required before the end of the gradient pulse.

**Arguments:** `pattern` is the root name of a text file in the `shapelib` directory with a `.GRD` extension to store the gradient shape.

`width` is the duration of the gradient shape, in seconds.

`loops` is not implemented. The shape is executed once. Set 0 for clarity.

`wait` is a keyword with values of `WAIT` or `NOWAIT`, which determines the delay until the execution of next statement. If the value is `WAIT`, the delay is `width` seconds. If the value is `NOWAIT`, the delay is 0.0 seconds.

**Examples:**

```
mashapedgradient("ramp_hold",3.0,trise,0,NOWAIT);
pulse(pw,oph);
delay(0.001-pw-2*trise);
mashapedgradient("ramp_down",3.0,trise,0,NOWAIT);
```

<b>Related:</b>	<code>magradient</code>	Set three-axis gradient at magic angle
	<code>magradpulse</code>	Perform three-axis gradient pulse at magic angle
	<code>mashapedgradpulse</code>	Perform three-axis shaped gradient pulse at magic angle
	<code>zero_all_gradients</code>	Zero gradient DAC level for all axes

### **mashapedgradpulse**

### **Perform three-axis shaped gradient pulse at magic angle**

**Syntax:**

```
mashapedgradpulse(pattern,gradlvl,width)
char *pattern;      /* name of a .GRD file */
double gradlvl;    /* gradient amplitude,
Gauss/cm */
double width;      /* duration of gradient
pulse, seconds */
int loops; /* number of waveform loops */
int wait;      /* WAIT or NOWAIT */
```

**Description:** Apply a static, shaped gradient pulse with duration `width` simultaneously to the  $x$ ,  $y$ , and  $z$  axes at the magic angle,  $54.7^\circ$  to  $B_0$ . Information from the gradient table is used to convert Gauss/cm into DAC settings for the three axes. The gradient DACs are set to 0.0 at the completion of the pulse.

**Arguments:** `pattern` is the root name of a text file in the `shapelib` directory with a `.GRD` extension to store the gradient shape.



`gradlvl` is the gradient-pulse amplitude at the magic angle, in Gauss/cm.

`width` is the duration of the gradient pulse, in seconds.

Examples: `mashapedgradpulse("hsine",3.0, 0.001);`

Related:	<code>magradient</code>	Set three-axis gradient at magic angle
	<code>magradpulse</code>	Perform three-axis gradient pulse at magic angle
	<code>mashapedgradient</code>	Perform three-axis shaped gradient at magic angle
	<code>zero_all_gradients</code>	Zero gradient DAC level for all axes

## **mod2**

### **Assign real-time integer value modulo 2**

Syntax: `mod2(vi,vj)`  
`codeint vi; /* real-time variable for input */`  
`codeint vj; /* real-time variable for output */`

Description: Set the value of `vj` equal to the integer value of `vi` modulo 2 (the remainder after the value of `vi` is divided by 2).

Arguments: `vi` contains the input value and `vj` contains the result of the value of `vi` modulo 2. Both arguments must be real-time variables (`v1` to `v42`, *etc*).

Examples: `mod2(v3,v5);`

Related:	<code>add</code>	Add real-time integer values
	<code>assign</code>	Assign real-time integer value using real-time integer
	<code>dbl</code>	Double real-time integer value
	<code>decr</code>	Decrement real-time integer value
	<code>divn</code>	Divide real-time integer values
	<code>hlv</code>	Assign half the value of real-time integer
	<code>incr</code>	Increment real-time integer value
	<code>initval</code>	Assign real-time integer value using numeric value
	<code>mod4</code>	Assign real-time integer value modulo 4
	<code>modn</code>	Assign real-time integer value modulo n
	<code>mult</code>	Multiply real-time integer values
	<code>sub</code>	Subtract real-time integer values

**mod4**

**Assign real-time integer value modulo 4**

Syntax: `mod4(vi,vj)`  
`codeint vi; /* real-time variable for`  
`input */`  
`codeint vj; /* real-time variable for`  
`output */`

Description: Set the value of `vj` equal to the integer value of `vi` modulo 4 (the remainder after the value of `vi` is divided by 4)..

Arguments: `vi` contains the input value and `vj` contains the result of the value of `vi` modulo 4. Both arguments must be real-time variables (`v1` to `v42`, *etc*).

Examples: `mod4(v3,v5);`

Related:	<code>add</code>	Add real-time integer values
	<code>assign</code>	Assign real-time integer value using real-time integer
	<code>dbl</code>	Double real-time integer value
	<code>decr</code>	Decrement real-time integer value
	<code>divn</code>	Divide real-time integer values
	<code>hlv</code>	Assign half the value of real-time integer
	<code>incr</code>	Increment real-time integer value
	<code>initval</code>	Assign real-time integer value using numeric value
	<code>mod2</code>	Assign real-time integer value modulo 2
	<code>modn</code>	Assign real-time integer value modulo n
	<code>mult</code>	Multiply real-time integer values
	<code>sub</code>	Subtract real-time integer values

**modn**

**Assign real-time integer value modulo n**

Syntax: `modn(vi,vj,vk)`  
`codeint vi; /* real-time variable for input */`  
`codeint vj; /* real-time variable for`  
`modulo number */`  
`codeint vk; /* real-time variable for output`  
`*/`

Description: Set the value of `vk` equal to the integer value of `vi` modulo the value of `vj` (the remainder after the value of `vi` is divided by the value of `vj`).

Arguments: `vi` contains the input value, `vj` contains the modulo value `n`, and `vk` contains the result of the value of `vi` modulo `n`. All arguments must be real-time variables (`v1` to `v42`, *etc*).

Examples: `modn(v3,v5,v4);`

Related:	add	Add real-time integer values
	assign	Assign real-time integer value using real-time integer
	dbl	Double real-time integer value
	decr	Decrement real-time integer value
	divn	Divide real-time integer values
	hlv	Assign half the value of real-time integer
	incr	Increment real-time integer value
	initval	Assign real-time integer value using numeric value
	mod2	Assign real-time integer value modulo 2
	mod4	Assign real-time integer value modulo 4
	mult	Multiply real-time integer values
	sub	Subtract real-time integer values

**msloop****Start switchable multislice loop**

Syntax: `msloop(state,max_count,apv1,apv2)`

```
char state;          /* compressed or standard
*/
double max_count;   /* numeric value to
initialize apv1 */
codeint apv1;       /* real-time variable
with maximum count */
codeint apv2;       /* real-time counter */
```

Description: Execute a sequence-switchable loop that can use real-time variables in what is known as a compressed loop or can use the standard arrayed features of PSG. In imaging sequences, `msloop` uses the second character of the `seqcon` string parameter (`seqcon[1]`) for the state argument. `msloop` is used in conjunction with `endmsloop`.

Arguments: `state` is either 'c' to designate the compressed mode, or 's' to designate the standard arrayed mode.

`max_count` initializes `apv1`. If `state` is 'c', this value should equal the number of slices. If `state` is 's', this value should be 1.0.

`apv1` is a real-time variable that holds the maximum count.

`apv2` is a real-time variable that holds the current counter value. If `state` is 'c', `apv2` counts from 0 to `max_count-1`. If `state` is 's', `apv2` is set to zero.

Examples: `msloop(seqcon[1],ns,v11,v12);`  
`...`  
`poffset_list(pss,gss,ns,v12);`  
`...`

```
acquire(np,1.0/sw);
...
endmsloop(seqcon[1],v12);
```

Related: `endloop` End real-time loop  
`endmsloop` End switchable multislice loop  
`endpeloop` End switchable phase-encode loop  
`loop` Start real-time loop  
`loopcheck` Check number of FIDS for compressed acquisition  
`peloop` Start switchable phase-encode loop

**mult**

**Multiply real-time integer values**

Syntax: `mult(vi,vj,vk)`  
`codeint vi; /* real-time variable for first factor */`  
`codeint vj; /* real-time variable for second factor */`  
`codeint vk; /* real-time variable for product */`

Description: Set the value of `vk` equal to the product of the integer values of `vi` and `vj`.

Arguments: `vi` contains an integer factor, `vj` contains second integer factor, and `vk` contains the product. All arguments are real-time variables (`v1` to `v42`, *etc*).

Examples: `mult(v3,v5,v4);`

Related: `add` Add real-time integer values  
`assign` Assign real-time integer value using real-time integer  
`dbl` Double real-time integer value  
`decr` Decrement real-time integer value  
`divn` Divide real-time integer values  
`hlv` Assign half the value of real-time integer  
`incr` Increment real-time integer value  
`initval` Assign real-time integer value using numeric value  
`mod2` Assign real-time integer value modulo 2  
`mod4` Assign real-time integer value modulo 4  
`modn` Assign real-time integer value modulo n  
`sub` Subtract real-time integer values

## 0

<code>obl_gradient</code>	Set three-axis oblique gradient
<code>obl_shapedgradient</code>	Perform three-axis oblique gradient shape, single pattern
<code>obl_shaped3gradient</code>	Perform three-axis oblique gradient shape, three patterns
<code>obsblank</code>	Blank amplifier of observe channel
<code>obsoffset</code>	Set frequency offset of observe channel
<code>obspower</code>	Set power level of observe channel
<code>obsprgoff</code>	End waveform decoupling on observe channel
<code>obsprgon</code>	Start waveform decoupling on observe channel
<code>obspulse</code>	Perform pulse with assigned values on observe channel
<code>obspwr</code>	Set fine power level of observe channel
<code>obsstepsize</code>	Set small-angle phase stepsize for observe channel
<code>obsunblank</code>	Unblank amplifier of observe channel
<code>offset</code>	Change frequency offset of any channel
<code>offsetglist</code>	Create offset array from position and gradient amplitude arrays
<code>offsetlist</code>	Create offset array from position and gradient amplitude

**obl\_gradient****Set three-axis oblique gradient**

**Syntax:** `obl_gradient(gradlv11,gradlv12,gradlv13)`  
`double gradlv11,gradlv12,gradlv13; /* static`  
`gradient amplitudes,`  
`/* Gauss/cm */`

**Description:** Apply a static, oblique gradient using the angles `psi`, `theta`, and `psi`, obtained from the parameters of those names.

**Arguments:** `gradlv11`, `gradlv12`, `gradlv13` are the three amplitudes of the static gradient, in Gauss/cm, along the three logical axes, `read`, `phase`, and `slice`, respectively.

**Examples:** `obl_gradient(0.0,0.0,gss);`  
`obl_gradient(gro,0.0,0.0);`

<b>Related:</b>	<code>mashapedgradient</code>	Perform three-axis shaped gradient at magic angle
	<code>obl_shapedgradient</code>	Perform three-axis oblique shaped gradient, one pattern
	<code>obl_shaped3gradient</code>	Perform three-axis oblique shaped gradient, three patterns
	<code>rot_angle</code>	Set user-defined oblique gradient-coordinate rotation axes
	<code>pe3_shaped3gradient</code>	Perform oblique shaped gradient, three patterns, phase encode three axes

**obl\_shapedgradient****Perform three-axis oblique gradient shape, single pattern**

**Syntax:** `obl_shapedgradient(pattern,width,gradlv11,gradlv12,gradlv13)`  
`char *pattern; /* name of .GRD file */`  
`double width; /* duration of gradient pulse, seconds */`  
`double gradlv11,gradlv12,gradlv13;`  
`/* static gradient amplitudes, Gauss/cm */`  
`int wait; /* WAIT or NOWAIT */`

**Description:** Apply a static, oblique, gradient shape with a single pattern for all three axes, using the angles `psi`, `theta`, and `phi`, obtained from the parameters of those names, in either `WAIT` or `NOWAIT` mode.

**Arguments:** `pattern` is the root name of a text file in the `shapelib` directory with a `.GRD` extension to store the gradient shape.

`width` is the duration of the shaped gradient, in seconds.

`gradlv11`, `gradlv12`, `gradlv13` are the three components of the static portion of the gradient, in Gauss/cm, along the three logical axes, `read`, `phase`, and `slice`, respectively.

`wait` is a keyword with values of `WAIT` or `NOWAIT`, that determines the delay until the execution of next statement. If the value is `WAIT`, the delay is `width` seconds. If the value is `NOWAIT`, the delay is 0.0 seconds.

**Examples:** `obl_shapedgradient("ramp_hold",trise,gro,0.0,0.0,NOWAIT);`

<b>Related:</b>	<code>mashapedgradient</code>	Perform three-axis shaped gradient at magic angle
	<code>obl_gradient</code>	Perform three-axis oblique gradient
	<code>obl_shaped3gradient</code>	Perform three-axis oblique shaped gradient, three patterns
	<code>rot_angle</code>	Set user-defined oblique gradient-coordinate rotation axes
	<code>pe3_shaped3gradient</code>	Perform oblique shaped gradient, three patterns, phase encode three axes

**obl\_shaped3gradient****Perform three-axis oblique gradient shape, three patterns**

**Syntax:** `obl_shaped3gradient(pat1,pat2,pat3,width,gradlv11,gradlv12,gradlv13,wait)`  
`char *pat1,*pat2,*pat3; /* names of .GRD files */`  
`double width; /* duration of gradient pulse, seconds */`  
`double gradlv11,gradlv12,gradlv13;`  
`/* static gradient amplitudes, Gauss/cm */`  
`int wait; /* WAIT or NOWAIT */`

**Description:** Apply a static, oblique, gradient shape with a separate pattern for each axis, using the angles `psi`, `theta`, and `psi`, obtained from the parameters of those names, in either `WAIT` or `NOWAIT` mode.

**Arguments:** `pat1,pat2,pat3` are the root names of text files in the `shapelib` directory with a `.GRD` extension to store the gradient shapes.

`width` is the duration of the shaped gradient, in seconds.

`gradlv11, gradlv12, gradlv13` are the three components of the static portion of the gradient, in Gauss/cm, along the three logical axes, `read`, `phase`, and `slice`, respectively.

`wait` is a keyword with values of `WAIT` or `NOWAIT`, that determines the delay until the execution of next statement. If the value is `WAIT`, the delay is `width` seconds. If the value is `NOWAIT`, the delay is 0.0 seconds.

**Examples:** `obl_shaped3gradient("ramp_hold","",trise,gro,0.0,0.0,NOWAIT);`

<b>Related:</b>	<code>mashapedgradient</code>	Perform three-axis shaped gradient at magic angle
	<code>obl_gradient</code>	Perform three-axis oblique gradient
	<code>obl_shapedgradient</code>	Perform three-axis oblique shaped gradient, one pattern
	<code>rot_angle</code>	Set user-defined oblique gradient-coordinate rotation axes
	<code>pe3_shaped3gradient</code>	Perform oblique shaped gradient, three patterns, phase encode three axes

**obsblank****Blank amplifier of observe channel**

**Syntax:** `obsblank()`

**Description:** Disable the amplifier for the observe channel if the amplifier is in pulse mode. `obsblank` has no effect if the amplifier is in continuous mode. The observe

channel is in pulse mode by default because it is always associated with a receiver.

`obsblank` must be used before acquisition and it is often used between pulses and decoupling periods to suppress amplifier noise.

To place the observe channel in continuous mode, create the parameter `ampmode` and set the appropriate character to 'c'. For continuous mode, `obsblank` has no effect and the statement is not needed. Placing an observe channel in continuous mode will usually lower the signal-to-noise of the acquisition.

Related:	<code>dec2blank</code>	Blank amplifier of second decoupler
	<code>dec3blank</code>	Blank amplifier of third decoupler
	<code>dec4blank</code>	Blank amplifier of fourth decoupler
	<code>obsblank</code>	Blank amplifier of observe channel
	<code>obsunblank</code>	Unblank amplifier of observe channel

## **obsoffset**

### **Set frequency offset of observe channel**

Syntax: `obsoffset(frequency)`  
`double frequency; /* frequency offset, Hz */`

Description: Set the frequency offset of the observe channel. The offset of the observe channel is initialized to `tof` before the first `obsoffset` statement is applied. You must explicitly use `obsoffset(tof)` to return the offset to `tof`.

The `obsoffset` statement sets the VNMR5 synthesizer frequency that simultaneously determines the base frequency of pulses and the center frequency of the receiver. You should use caution in resetting the frequency offset. Incorrect use of `obsoffset` can influence the phase coherence of the receiver and pulses scan-to-scan.

The `obsoffset` statement inserts a 50 ns delay into the pulse sequence. The VNMR5 synthesizer can take several microseconds to set. It may be necessary to compensate for these delays in the pulse sequence.

Arguments: `frequency` is the desired frequency offset, in Hz.

Examples: `obsoffset(newoffset);`  
`obsoffset(dof4);`

Related:	<code>dec2offset</code>	Set frequency offset of second decoupler
	<code>dec3offset</code>	Set frequency offset of third decoupler



<code>dec4offset</code>	Set frequency offset of fourth decoupler
<code>offset</code>	Set frequency offset of any channel

**obspower****Set power level of observe channel**

Syntax: `obspower(power)`  
`double power; /* power-level value, dB */`

Description: Set the power level of the observe channel using the coarse attenuator. The power level of the observe channel is initialized to `tpwr` before the first `obspower` statement is applied. You must explicitly use `obspower(tpwr)` to return the power level to `tpwr`. `obspower` is functionally the same as `rlpower(value, DODEV)`.

The `obspower` statement inserts a delay of 50 ns into the pulse sequence and you should allow 3  $\mu$ s for the power to reach the new level during the next delay. The coarse attenuator can introduce a transient when it changes and it is a good practice to execute `obspower` only when the transmitter is blanked and gated off.

Arguments: `power` sets the coarse attenuator in 0.5 dB steps from a maximum power of 63 to a minimum of -37 if the 100 dB attenuator is present. The stepsize is truncated to 1.0 dB and the minimum is -16 dB if the 79 dB attenuator is present. Consult the configuration file to determine the available attenuator.

Related: `dec2power` Set power level of second decoupler  
`dec3power` Set power level of third decoupler  
`dec4power` Set power level of fourth decoupler  
`rlpower` Set the power level of any channel

**obsprgoff****End waveform decoupling on observe channel**

Syntax: `obsprgoff()`

Description: Terminates programmable waveform decoupling on the observe channel, gates the observe channel off, and blanks the associated amplifier if it is in pulse mode.

Related: `decprgoff` End waveform decoupling on first decoupler  
`dec2prgoff` End waveform decoupling on second decoupler

dec3prgoff	End waveform decoupling on third decoupler
dec4prgoff	End waveform decoupling on fourth decoupler
obsblank	Blank amplifier of observe channel
obsprgon	Start waveform decoupling on observe channel
obsprgonOffset	Start waveform decoupling on observe channel with offset

**obsprgon****Start waveform decoupling on observe channel**

**Syntax:** `obsprgon(pattern,90_pulselength,tipangle_resoln)`  
`char *pattern; /* name of .DEC file */`  
`double 90_pulselength; /* 90-degree pulse length in sec */`  
`double tipangle_resoln; /* tip-angle resolution */`  
`return int ticks /* 12.5 ns ticks, one cycle */`

**Description:** Execute programmable decoupling on the observe channel under waveform control, unblank the associated amplifier, and gate the observe channel on for patterns without an explicit gate column. `obsprgon` returns an integer with the number of 12.5-ns ticks in one cycle of the decoupling pattern. It is a good practice to unblank the associated amplifier with `obsunblank`, at least 2.0  $\mu$ s before `obsprgon`.

**Arguments:** `pattern` is the root name of a text file in the `shapelib` directory with a `.DEC` extension to store the decoupling pattern.

`90_pulselength` is the pulse time-duration, in seconds, for an element, a block of elements, or an integer divisor of all elements in the pattern, which is labeled with a tip-angle duration equal to 90°. Often `90_pulselength` is set equal  $1/dmf$  where `dmf` is a step rate.

`tipangle_resoln` is the smallest common divisor of the tip-angle durations of all the elements of the pattern.

For many patterns, `90_pulselength` is the actual 90° pulse length and the elements are labeled with tip-angle durations that are multiples of 90. In this case, `tipangle_resoln` is 90.0.

For many other patterns, `90_pulselength` is set to be the duration of the minimum stepsize of which all elements are multiples. In these cases, `tipangle_resoln` is nominally 90 and elements in

the pattern have tip-angle durations that are multiples of 90.

For some patterns, 90\_pulselength is the actual 90° pulse length but elements of the pattern have arbitrary flip angles that are multiples of a tipangle\_resoln, that is less than 90° (*c.f.* 1.0°). In this case, tipangle\_resoln is the divisor (*c.f.* 1.0) and for good practice it is also a divisor of 90.0.

**Examples:**

```
obsprgon("garp1",1/dmf, 1.0);
obsprgon(modtype,pwx90,dres);
ticks = obsprgon("waltz16",1/dmf,90.0);
```

<b>Related:</b>	decprgon	Start waveform decoupling on first decoupler
	dec2prgon	Start waveform decoupling on second decoupler
	dec3prgon	Start waveform decoupling on third decoupler
	dec4prgon	Start waveform decoupling on fourth decoupler
	obsprgoff	End waveform decoupling on observe channel
	obsprgonOffset	Start waveform decoupling on observe channel with offset
	obsunblank	Unblank amplifier of observe channel

### obsprgonOffset

#### Start waveform decoupling on observe channel with offset

**Syntax:**

```
obsprgonOffset(pattern,90_pulselength,
tipangle_resoln,offset)
char *pattern; /* name of .DEC file */
double 90_pulselength; /* 90-degree pulse
length in sec */
double tipangle_resoln; /* tip-angle
resolution */
double offset; /* frequency offset, in Hz */
return int ticks /* 12.5 ns ticks, one cycle */
```

**Description:** Execute programmable decoupling on the first decoupler under waveform control with a frequency offset that is applied automatically. The statement obsprgonOffset unblanks the associated amplifier and gates the first decoupler on for patterns without an explicit gate column. obsprgonOffset returns an integer with the number of 12.5-ns ticks in one cycle of the decoupling pattern. It is a good practice to unblank the associated amplifier with obsunblank, at least 2.0 μs before obsprgonOffset.

The frequency offset is applied by phase modulation of the base pattern in the rf controller. The phase modulation is achieved by expanding the pattern to include linear phase steps as well as by

interpolation in real-time. Use of `obsprgonOffset` with a base pattern achieves about a ten-fold compression of steps relative to an equivalent pattern supplied in the `.DEC` file.

**Arguments:** `pattern` is the root name of a text file in the `shapelib` directory with a `.DEC` extension to store the decoupling pattern.

`90_pulselength` is the pulse time-duration, in seconds, for an element, a block of elements, or an integer divisor of all elements in the pattern, which is labeled with a tip-angle duration equal to  $90^\circ$ . Often `90_pulselength` is set equal  $1/\text{dmf}$  where `dmf` is a step rate.

`tipangle_resoln` is the smallest common divisor of the tip-angle durations of all the elements of the pattern.

For many patterns, `90_pulselength` is the actual  $90^\circ$  pulse length and the elements are labeled with tip-angle durations that are multiples of 90. In this case, `tipangle_resoln` is 90.0.

For many other patterns, `90_pulselength` is set to be the duration of the minimum stepsize of which all elements are multiples. In these cases, `tipangle_resoln` is nominally 90 and elements in the pattern have tip-angle durations that are multiples of 90.

For some patterns, `90_pulselength` is the actual  $90^\circ$  pulse length but elements of the pattern have arbitrary flip angles that are multiples of a `tipangle_resoln`, that is less than  $90^\circ$  (*c.f.* 1.0 $^\circ$ ). In this case, `tipangle_resoln` is the divisor (*c.f.* 1.0) and for good practice it is also a divisor of 90.0.

`offset` is a frequency offset relative to the synthesizer frequency, in Hz.

**Examples:**

```
obsprgonOffset("garp1",1/dmf, 1.0,1000.0);
obsprgonOffset(modtype,pwx90,dres,1000.0);
ticks =
obsprgonOffset("waltz16",1/dmf,90.0,1000.0);
```

<b>Related:</b>	<code>decprgonOffset</code>	Start waveform decoupling on first decoupler with offset
	<code>dec2prgonOffset</code>	Start waveform decoupling on second decoupler with offset
	<code>dec3prgonOffset</code>	Start waveform decoupling on third decoupler with offset
	<code>dec4prgonOffset</code>	Start waveform decoupling on fourth decoupler with offset
	<code>obsprgoff</code>	End waveform decoupling on observe channel
	<code>obsprgon</code>	Start waveform decoupling on observe channel

obsunblank            Unblank the amplifier of  
observe channel

**obspulse****Perform pulse with assigned values on observe channel**

Syntax:    obspulse()

Description:    Set the quadrature phase and gates the observe channel on and off at the current power level with amplifier unblanking and blanking, using a predelay of `rof1` and a postdelay of `rof2`. The pulse width is automatically set to `pw` and the phase is automatically set to `oph`. The associated amplifier is unblanked if it is in pulse mode and the phase is set, both at the beginning of the predelay. The associated amplifier is blanked at the end of the postdelay if it is in pulse mode.

Related:    `decpulse`    Perform pulse with assigned values on first decoupler  
             `decrpulse`    Perform pulse on first decoupler  
             `pulse`            Perform pulse with assigned values on observe channel  
             `rgpulse`        Perform pulse on observe channel

**obspwrf****Set fine power level of observe channel**

Syntax:    `obspwrf(amplitude)`  
             `double power;            /* fine-power value */`

Description:    Set the fine-power value of the observe channel using the linear modulator. The fine power level of the observe channel is initialized to `tpwrf` before the first `obspwrf` statement is applied. You must explicitly use `obspwrf(tpwrf)` to return the power level to `tpwrf`. The statement `obspwrf` is functionally similar to `rlpwrf(amplitude,DODEV)`. Fine power units are linear in voltage.

Arguments:    `amplitude` sets the fine power in units of 1.0 from a maximum of 4095.0 to a minimum of 0.0. If the 12-bit linear modulator is available the stepsize is 1.0. If the 16-bit linear modulator is present decimal values (3 significant figures) are allowed. Consult the configuration file to determine the available modulator. The fine power and coarse power can be used interchangeably where 6 dB units of coarse power correspond to a x2 change of the fine power.

Examples:    `obspwrf(3500);obspwrf(3500.324);`

Related: `decpwrf` Set fine power level of first decoupler  
`dec2pwrf` Set fine power level of second decoupler  
`dec3pwrf` Set fine power level of third decoupler  
`dec4pwrf` Set fine power level of fourth decoupler  
`rlpwrf` Set fine power level of any channel

**obsstepsize**

**Set small-angle phase stepsize of observe channel**

Syntax: `obsstepsize(stepsize)`  
`double stepsize; /* small-angle phase stepsize */`

Description: Set the small-angle phase `stepsize` of the observe channel. The `xmtrphase` statement sets the small-angle phase as a product of `step_size` and a real-time multiplier. `obsstepsize` is not used, the default `step_size` is 90°. The `obsstepsize` statement can be used multiple times in a sequence.

The small-angle phase of the DD2 MR system has a phase resolution of 360.0/65536 (~0.0055°). The VNMRS has a phase resolution of 360.0/8192 (~0.044°) depending on the available transmitter, and the small-angle phase is set to the nearest step. Consult the configuration file to determine which transmitter is present.

Arguments: `stepsize` is a value, in degrees, corresponding to one unit of the real-time phase multiplier.

Examples: `obsstepsize(30.0);`

Related: `decstepsize` Set small-angle phase stepsize of first decoupler  
`dec2stepsize` Set small-angle phase stepsize of second decoupler  
`dec3stepsize` Set small-angle phase stepsize of third decoupler  
`dec4stepsize` Set small-angle phase stepsize of fourth decoupler  
`obsstepsize` Set small-angle phase stepsize of observe channel  
`stepsize` Set small-angle phase stepsize of any channel  
`xmtrphase` Set small-angle phase of observe channel

**obsunblank**

**Unblank amplifier of observe channel**

Syntax: `obsunblank()`

**Description:** Enable the amplifier for the observe channel if the amplifier is in pulsed mode. For pulsed mode, `obsunblank` is required at least 2.0  $\mu$ s before pulses, spinlocks, and decoupling periods.

The observe channel is in pulsed mode by default because it is always associated with a receiver.

To place the observe channel in continuous mode, create the parameter `ampmode` and set the appropriate character to 'c'. For continuous mode, `obsunblank` is the default and the statement is not needed. Placing an observe channel in continuous mode will usually lower the signal-to-noise of the acquisition.

**Related:**

<code>decunblank</code>	Unblank amplifier of first decoupler
<code>dec2unblank</code>	Unblank amplifier of second decoupler
<code>dec3unblank</code>	Unblank amplifier of third decoupler
<code>dec4unblank</code>	Unblank amplifier of fourth decoupler
<code>obsunblank</code>	Unblank amplifier of observe channel

## **offset**

### **Change frequency offset of any channel**

**Arguments:** Use `obsoffset`, `decoffset`, `dec2offset`, or `dec3offset`, as appropriate.

**Related:**

<code>decoffset</code>	Set frequency offset of first decoupler
<code>dec2offset</code>	Set frequency offset of second decoupler
<code>dec3offset</code>	Set frequency offset of third decoupler
<code>dec4offset</code>	Set frequency offset of fourth decoupler
<code>obsoffset</code>	Set frequency offset of observe channel

## **offsetglist**

### **Create offset array from position and gradient-amplitude array**

**Syntax:**

```
offsetlist(posarray, gradarray, resfrq,
offsetarray, ns, state)
double posarray[];      /* input positions, in
cm */
double gradarray;      /* input gradient
amplitudes in cm */
double resfrq;         /* optional
```

```

chemical-shift offset, in Hz */
double offsetarray[]; /*returned array of
calculated offsets */
doublem ns;           /* number of positions */
char state;           /* the mode, cromptessed,
strandard or indexed */

```

**Description:** Calculate a C *double* array of rf frequency offsets `offsetarray` from a position array `posarray` and a gradient-amplitude array `gradarray` in Gauss/cm. This array can then be used in the `shapelist` command.

**Arguments:** `posarray` is an input array containing slice positions, in cm.

`gradarray` is an input array containing gradient amplitudes, in Gauss/cm.

`resfrq` is an optional chemical-shift frequency offset, in Hz, to be added to the computed offsets.

`offsetarray` is returned containing a set of frequency offsets, in Hz, from the calculation. This array should be declared as an array of *doubles* with a size greater than or equal to the number of offsets.

`ns` is the number of positions in the array.

`state` is either 'c' to designate the compressed mode, 's' to designate the standard arrayed mode, or 'i' to designate an integer indexed mode to be used in a C loop.

<b>Related:</b>	<code>getarray</code>	Obtain all values from arraye parameter
	<code>offsetlist</code>	Create offset array from position and gradient amplitude
	<code>poffset</code>	Return frequency offset based on position
	<code>poffset_list</code>	Create offset array from position array
	<code>position_offset</code>	Return frequency offset based on position
	<code>position_offset_list</code>	Create offset array from position array

**offsetlist****Create offset array from position and gradient-amplitude**

**Syntax:** `offsetlist(posarray, gradlvl, resfrq, offsetarray, ns, state)`

```

double posarray[]; /* input positions, in
cm */
double gradlvl;    /* input gradient
amplitudes in cm */
double resfrq;    /* optional

```



```

chemical-shift offset, in Hz */
double offsetarray[]; /* returned array of
calculated offsets */
doublem ns;           /* number of positions */
char state;           /* the mode, crompressed,
strandard or indexed */

```

**Description:** Calculate a C *double* array of rf frequency offsets `offsetarray` from a position array `posarray` and a gradient-amplitude array `gradarray` in Gauss/cm. This array can then be used in the `shapelist` command.

**Arguments:** `posarray` is an input array containing slice positions, in cm.

`gradlvl` is a gradient amplitudes, in Gauss/cm.

`resfrq` is an optional chemical-shift frequency offset, in Hz, to be added to the computed offsets.

`offsetarray` is returned containing a set of frequency offsets, in Hz, from the calculation. This array should be declared as an array of *doubles* with a size greater than or equal to the number of offsets.

`ns` is the number of positions in the array.

`state` is either 'c' to designate the compressed mode, 's' to designate the standard arrayed mode, or 'i' to designate an integer indexed mode to be used in a C loop.

<b>Related:</b>	<code>getarray</code>	Obtain all values from arraye parameter
	<code>offsetglist</code>	Create offset array from position and gradient-amplitude array
	<code>poffset</code>	Return frequency offset based on position
	<code>poffset_list</code>	Create offset array from position array
	<code>position_offset</code>	Return frequency offset based on position
	<code>position_offset_list</code>	Create offset array from position array

**P**

parallelacquire_obs	Acquire data explicitly in parallel section
parallelacquire_rcvr	Acquire data explicitly in parallel section
parallelend	End parallel section of pulse sequence
parallelstart	Start parallel section of pulse sequence
parallelsync	Position parallel synchronization delays
pe_gradient	Set oblique gradient, phase encode one axis
pe2_gradient	Set oblique gradient, phase encode two axes
pe3_gradient	Set oblique gradient, phase encode three axes
pe_shapedgradient	Perform oblique gradient shape, single pattern, phase encode one axis
pe_shaped3gradient	Perform oblique gradient shape, three patterns, phase encode one axis
pe2_shapedgradient	Perform oblique gradient shape, single pattern, phase encode two axes
pe2_shaped3gradient	Perform oblique gradient shape, three patterns, phase encode two axes
pe3_shapedgradient	Perform oblique gradient shape, single pattern, phase encode three axes
pe3_shaped3gradient	Perform oblique gradient shape, three patterns, phase encode three axes
peloop	Start switchable phase-encode loop
phase_encode3_gradient	Set general oblique gradient, phase encode three axes
phase-encode3_gradpulse	Perform general oblique gradient pulse, phase encode three axes
phase_encode3_oblshapedgradient	Perform general oblique shaped gradient, three patterns, phase encode three axes
poffset	Return frequency offset based on position
poffset_list	Create offset array from position array
position_offset	Set frequency based on position
position_offset_list	Create offset array from position array
psg_abort	Abort PSG process at run-time
pulse	Perform pulse with assigned values on observe channel
putarray	Set all values of arrayed parameter from pulse sequence
putCmd	Send command to VnmrJ from pulse sequence
putstring	Set string parameter from pulse sequence
putvalue	Set numeric parameter from pulse sequence

**parallelacquire\_obs****Acquire data explicitly in parallel section**

Syntax: `parallelacquire_obs(delay,points,dwell)`

```
double delay; /* time duration before
the next event */

double points; /* number of points
(complex pairs) to acquire */

double dwell; /* dwell time, seconds */
```

Description: Combination of the `startacq_obs`, `acquire_obs`, and `endacq_obs` statements to be used in "obs" parallel sections of a pulse sequence.

Arguments: The first argument (`delay`) corresponds to the argument used by `startacq`.  
The next two arguments correspond to the arguments used by `acquire`.

Related:	<code>acquire_obs</code>	Acquire data explicitly in parallel section
	<code>acquire_rcvr</code>	Acquire data explicitly in parallel section
	<code>startacq_obs</code>	Initialize explicit acquisition in parallel section
	<code>startacq_rcvr</code>	Initialize explicit acquisition in parallel section
	<code>endacq_obs</code>	End explicit acquisition in parallel section
	<code>endacq_rcvr</code>	End explicit acquisition in parallel section
	<code>parallelacquire_rcvr</code>	Acquire data explicitly in parallel section

**parallelacquire\_rcvr****Acquire data explicitly in parallel section**

Syntax: `parallelacquire_rcvr(delay,points,dwell)`

```
double delay; /* time duration before
the next event */

double points; /* number of points
(complex pairs) to acquire */

double dwell; /* dwell time, seconds */
```

Description: Combination of the `startacq_rcvr`, `acquire_rcvr`, and `endacq_rcvr` statements to be used in "rcvr" parallel sections of a pulse sequence.

Arguments: The first argument (`delay`) corresponds to the argument used by `startacq`.

The next two arguments correspond to the arguments used by `acquire`.

Related:	<code>acquire_obs</code>	Acquire data explicitly in parallel section
	<code>acquire_rcvr</code>	Acquire data explicitly in parallel section
	<code>startacq_obs</code>	Initialize explicit acquisition in parallel section
	<code>startacq_rcvr</code>	Initialize explicit acquisition in parallel section
	<code>endacq_obs</code>	End explicit acquisition in parallel section
	<code>endacq_rcvr</code>	End explicit acquisition in parallel section
	<code>parallelacquire_obs</code>	Acquire data explicitly in parallel section

**parallelend**

**End parallel section of pulse sequence**

Syntax: `double parallelend()`  
`double return /* duration of longest parallel section, seconds */`

Description: End a parallel section of a pulse sequence. The `parallelend` statement must follow one or more `parallelstart` statements. It determines the duration of the longest parallel section and sets this as the total duration, synchronizes each of the durations of the other parallel sections with a synchronization delay and applies a synchronous delay to the other channels equal to the total delay. `parallelend` returns the total delay in seconds.

Arguments: None.

Examples: `double duration;`  
`parallelstart("obs");`  
`delay(d3/2.0);`  
`pulse(pw, oph);`  
`delay(d3/2.0);`  
`parallelstart("dec");`  
`delay(d2);`  
`decpulse(pw, oph);`  
`duration = parallelend();`

Related: `parallelstart` Start parallel section of pulse sequence

parallelsync                      Position parallel  
synchronization delays

**parallelstart****Start parallel section of pulse sequence**

Syntax:    void parallelstart(chnl)  
          char \*chnl   /\* The designated parallel  
                      channel name \*/

Description:    Start a parallel section of a pulse sequence. Following a parallelstart statement until the next parallelstart or parallelend statement, pulse-sequence statements are executed solely on the channel designated by chnl. Synchronous delays are not applied to other channels, as is done outside of parallelstart-parallelend. Multiple parallel sections begun by parallelstart, before parallelend start executing at the same time in parallel. The parallelend statement applies a synchronization delay at the end of each parallel section so that all sections have the duration of the longest section, the total delay. A set of parallel sections must include every channel for which there is activity during the parallel period. The total delay is applied synchronously with the parallel sections to any channels that are not included. If a parallelsync statement is included in a parallel section the synchronization delay is applied at the location of the parallelsync statement rather than at the end of the parallel section.

Only statements for the designated channel are allowed in a parallel section, or those statements for calculation that do not designate a channel. The delay statement is interpreted to be for only the designated channel. Real-time statements that create delays such as loop-endloop, ifzero-elsenz-endif and vdelay are not allowed because they can invalidate the synchronization created by parallelend. Special fixed loops rllloop-rlendloop and kzloop-kzendloop are provided for use in a parallel section. Real-time statements or misplaced pulse-sequence statements in a parallel section will cause the pulse sequence to fail to operate or operate in an unpredictable manner. Statements that reference more than one channel such as simpulse are not allowed in a parallel section.

The xgate and rotorsync statements are allowed in a parallel section with the understanding that xgate halts and resumes all channels synchronously. One must be aware of the effect of

xgate and rotorsync on the other parallel sections.

**Arguments:** chnl is a string keyword enclosed in double quotes designating the channel of the parallel section. The possible keywords are obs, dec, dec2, dec3, dec4, grad, or rcvr.

**Examples:** This example places a pulse on the obs channel in the middle of the delay d2\_max. As d2 varies from 0.0 to d2\_max the pulse on dec moves from the beginning of the d2\_max delay to the end.

```
parallelstart("obs");
    delay(d2_max/2.0);
    pulse(pw, oph);
    delay(d3/2.0);
parallelstart("dec");
    delay(d2);
    decpulse(pw, oph);
parallelend();
```

<b>Related:</b>	endloop	End real-time loop
	kzendloop	End real-time loop with fixed duration
	kzloop	Start real-time with fixed duration
	loop	Start real-time loop
	parallelend	End parallel section of pulse sequence
	parallelsync	Position parallel synchronization delays
	rlendloop	End real-time loop with fixed count
	rlloop	Start real-time with fixed count

**parallelsync**

**Position parallel synchronization delays**

**Syntax:** void parallelsync()

**Description:** The parallelsync statement applies the synchronization delay for a parallel section, calculated by parallelend as a delay at the position of the parallelsync statement. If the parallelsync statement is not present in a section the synchronization delay is applied at the end of the section.

**Arguments:** None.

**Examples:** parallelstart("obs");

```

delay(d3/2.0);
pulse(pw, oph);
delay(d3/2.0);
parallelstart("dec");
parallelsync();
decpulse(pw, oph);
delay(d2);
parallelend();

```

Related: `parallelend` End parallel section of pulse sequence  
`parallelstart` Start parallel section of pulse sequence

**pe\_gradient****Set oblique gradient, phase encode one axis**

Syntax: `pe_gradient(stat1,stat2,stat3,step2,vmult2)`  
double stat1,stat2,stat3; /\* static gradient amplitudes, Gauss/cm \*/  
double step2; /\* gradient-amplitude stepsize, Gauss/cm \*/  
codeint vmult2; /\* real-time index of the steps \*/

Description: Apply a static, oblique gradient with a variable phase-encode gradient along the second logical axis (phase axis).

The pulse sequence aborts at run-time if the DACs on a particular gradient are overrun, after the angles and amplitude have been resolved.

Arguments: `stat1, stat2, stat3` are the three amplitudes of the static portion of the gradient, in Gauss/cm, along the three logical axes, read, phase, and slice, respectively.

`step2` is the amplitude stepsize, in Gauss/cm, for the variable gradient along the phase axis.

`vmult2` is a real-time index to the steps of the variable gradient. The argument `vmult2` can be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*), or a real-time table (`t1` to `t60`).

Examples: `pe_gradient(0.0, -sgpe*nv/2.0, gss, sgpe, v6);`

Related: `mashapedgradient` Perform three-axis shaped gradient at magic angle  
`obl_shaped3gradient` Perform three-axis oblique shaped gradient, three patterns

pe2_gradient	Perform oblique gradient, phase encode two axes
pe3_gradient	Perform oblique gradient, phase encode three axes
pe_shapedgradient	Perform oblique gradient, one pattern, phase encode one axis
pe_shaped3gradient	Perform oblique gradient, three patterns, phase encode one axis
pe2_shapedgradient	Perform oblique gradient, one pattern, phase encode two axes
pe2_shaped3gradient	Perform oblique gradient, three patterns, phase encode two axes
pe3_shapedgradient	Perform oblique gradient, one pattern, phase encode three axes
pe3_shaped3gradient	Perform oblique gradient, three patterns, phase encode three axes
phase_encode3oblshapedgradient	Perform general oblique shaped gradient, phase encode three axes
rot_angle	Set user-defined oblique gradient-coordinate rotation axes

**pe2\_gradient**

**Set oblique gradient, phase encode two axes**

Syntax: `pe2_gradient (stat1,stat2,stat3,step2,step3,vmult2,vmult3)`  
`double stat1,stat2,stat3; /* static gradient amplitudes, Gauss/cm */`  
`double step2,step3; /* gradient-amplitude stepsizes, Gauss/cm */`  
`codeint vmult2,vmult3 /* real-time indexes of the steps */`

Description: Apply a static, oblique gradient with variable phase-encode gradients along the second (phase) and third (slice) logical axes.

The pulse sequence aborts at run-time if the DACs on a particular gradient are overrun, after the angles and amplitude have been resolved.

Arguments: stat1, stat2, stat3 are the three amplitudes of the static portion of the gradient, in Gauss/cm, along the three logical axes, read, phase, and slice, respectively.



step2, step3 are the amplitude stepsizes, in Gauss/cm, for the variable gradients along the phase and slice axes.

vmult2, vmult3 are real-time indexes to the steps of the variable gradients. These arguments can be a real-time variables (v1 to v42, oph, etc), real-time constants (zero, one, etc), or real-time tables (t1 to t60).

**Examples:** `pe2_gradient(gro,sgpe*nv/2.0,sgpe2*nv2/2.0,sgpe,sgpe2,v6,v8);`

<b>Related:</b>	<code>mashapedgradient</code>	Perform three-axis shaped gradient at magic angle
	<code>obl_shaped3gradient</code>	Perform three-axis oblique shaped gradient, three patterns
	<code>pe_gradient</code>	Perform oblique gradient, phase encode one axis
	<code>pe3_gradient</code>	Perform oblique gradient, phase encode three axes
	<code>pe_shapedgradient</code>	Perform oblique gradient, one pattern, phase encode one axis
	<code>pe_shaped3gradient</code>	Perform oblique gradient, three patterns, phase encode one axis
	<code>pe2_shapedgradient</code>	Perform oblique gradient, one pattern, phase encode two axes
	<code>pe2_shaped3gradient</code>	Perform oblique gradient, three patterns, phase encode two axes
	<code>pe3_shapedgradient</code>	Perform oblique gradient, one pattern, phase encode three axes
	<code>pe3_shaped3gradient</code>	Perform oblique gradient, three patterns, phase encode three axes
	<code>phase_encode3oblshapedgradient</code>	Perform general oblique shaped gradient, phase encode three axes
	<code>rot_angle</code>	Set user-defined oblique gradient-coordinate rotation axes

### **pe3\_gradient**

#### **Set oblique gradient, phase encode three axes**

**Syntax:** `pe3_gradient(stat1,stat2,stat3,step1,step2,step3,vmult1,t1,vmult2,vmult3)`  
`double stat1,stat2,stat3; /* static gradient amplitudes, Gauss/cm */`  
`double step1,step2,step3;`  
`/*gradient-amplitude stepsizes, Gauss/cm */`

```
codeint vmult1,vmult2,vmult3;      /*
real-time indexes of the steps */
```

**Description:** Apply a static, oblique gradient with variable phase-encode gradients along the first (*read*), second (*phase*), and third (*slice*) logical axes. The pulse sequence aborts at run-time if the DACs on a particular gradient are overrun, after the angles and amplitude have been resolved.

**Arguments:** *stat1*, *stat2*, *stat3* are the three amplitudes of the static portion of the gradient, in Gauss/cm, along the three logical axes, *read*, *phase*, and *slice*, respectively.

*step1*, *step2*, *step3* are the amplitude stepsizes, in Gauss/cm, for the variable gradients along the *read*, *phase*, and *slice* axes.

*vmult1*, *vmult2*, *vmult3* are real-time indexes to the steps of the variable gradients. These arguments can be a real-time variables (*v1* to *v42*, *oph*, etc), real-time constants (*zero*, *one*, etc), or real-time tables (*t1* to *t60*).

**Examples:** `pe3_gradient (gro,sgpe*nv/2.0,sgpe2*nv2/2.0,0.0,sgpe,sgpe2,zero,v6,v8);`

**Related:**

<code>mashapedgradient</code>	Perform three-axis shaped gradient at magic angle
<code>obl_shaped3gradient</code>	Perform three-axis oblique shaped gradient, three patterns
<code>pe_gradient</code>	Perform oblique gradient, phase encode one axis
<code>pe2_gradient</code>	Perform oblique gradient, phase encode two axes
<code>pe_shapedgradient</code>	Perform oblique gradient, one pattern, phase encode one axis
<code>pe_shaped3gradient</code>	Perform oblique gradient, three patterns, phase encode one axis
<code>pe2_shapedgradient</code>	Perform oblique gradient, one pattern, phase encode two axes
<code>pe2_shaped3gradient</code>	Perform oblique gradient, three patterns, phase encode two axes
<code>pe3_shapedgradient</code>	Perform oblique gradient, one pattern, phase encode three axes
<code>pe3_shaped3gradient</code>	Perform oblique gradient, three patterns, phase encode three axes

phase_encode3oblsha pedgradient	Perform general oblique shaped gradient, phase encode three axes
rot_angle	Set user-defined oblique gradient-coordinate rotation axes

**pe\_shapedgradient****Perform oblique gradient shape, one pattern, phase encode  
one axis**

Syntax: `pe_shapedgradient (pattern,width,stat1,stat2,stat3,step2,vmult2,wait)`

```

char *pattern;           /* name of .GRD
file */
double width;           /* duration of
gradient shape, seconds */
double stat1,stat2,stat3; /* static gradient
amplitudes, Gauss/cm */
double step2;           /* gradient-amplitude
stepsize, Gauss/cm */
codeint vmult2;         /* real-time index
of steps */
int wait;               /* WAIT or NOWAIT */

```

Description: Apply a static, oblique, shaped gradient with a single pattern for all three axes and a variable phase-encode gradient along the second (phase) logical axis in either WAIT or NOWAIT mode.

The pulse sequence aborts at run-time if the DACs on a particular gradient are overrun, after the angles and amplitude have been resolved.

Arguments: `pattern` is the root name of a text file in the `shapelib` directory with a `.GRD` extension to store the gradient shape.

`width` is the duration of the gradient shape, in seconds.

`stat1`, `stat2`, `stat3` are the three components of the static portion of the gradient, in Gauss/cm, along the three logical axes, `read`, `phase` and `slice`, respectively.

`step2` is the stepsize, in Gauss/cm, for the variable gradient along the `phase` axis.

`vmult2` is a real-time index to the steps of the variable gradient. This argument can be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, `oph`, *etc*), or a real-time table (`t1` to `t60`).

`wait` is a keyword with values of `WAIT` or `NOWAIT`, that determines the delay until the execution of next statement. If the value is `WAIT`, the delay is `width` seconds. If the value is `NOWAIT`, the delay is 0.0 seconds.

Related:	mashapedgradient	Perform three-axis shaped gradient at magic angle
	obl_shaped3gradient	Perform three-axis oblique shaped gradient, three patterns
	pe_gradient	Perform oblique gradient, phase encode one axis
	pe2_gradient	Perform oblique gradient, phase encode two axes
	pe3_gradient	Perform oblique gradient, phase encode three axes
	pe_shaped3gradient	Perform oblique gradient, three patterns, phase encode one axis
	pe2_shapedgradient	Perform oblique gradient, one pattern, phase encode two axes
	pe2_shaped3gradient	Perform oblique gradient, three patterns, phase encode two axes
	pe3_shapedgradient	Perform oblique gradient, one pattern, phase encode three axes
	pe3_shaped3gradient	Perform oblique gradient, three patterns, phase encode three axes
	phase_encode3oblshapedgradient	Perform general oblique shaped gradient, phase encode three axes
	rot_angle	Set user-defined oblique gradient-coordinate rotation axes

**pe\_shaped3gradient**

**Perform oblique gradient shape, three patterns, phase encode one axis**

Syntax: `pe_shapedgradient (pat1,pat2,pat3,width,stat1,stat2,stat3,step2,vmult2,wait)`  
`char *pat1,*pat2,*pat3; /* names of .GRD files */`  
`double width; /* duration of gradient shape, seconds */`  
`double stat1,stat2,stat3; /* static gradient amplitudes, Gauss/cm */`  
`double step2; /* gradient-amplitude stepsize, Gauss/cm */`  
`codeint vmult2; /* real-time index of steps */`  
`int wait; /* WAIT or NOWAIT */`

Description: Apply a static, oblique, shaped gradient with a separate pattern for each axis and a variable

phase-encode gradient along the second (*phase*) logical axis in either `WAIT` or `NOWAIT` mode.

The pulse sequence aborts at run-time if the DACs on a particular gradient are overrun, after the angles and amplitude have been resolved.

**Arguments:** *pat1*, *pat2*, *pat3* are the root names of text files in the `shapelib` directory with a `.GRD` extension to store the gradient shapes.

*width* is the duration of the gradient shape, in seconds.

*stat1*, *stat2*, *stat3* are the three components of the static portion of the gradient, in Gauss/cm, along the three logical axes, `read`, `phase`, and `slice`, respectively.

*step2* is the stepsize, in Gauss/cm, for the variable gradient along the `phase` axis.

*vmult2* is a real-time index to the steps of the variable gradient. This argument can be a real-time variable (*v1* to *v42*, *oph*, *etc*), a real-time constant (*zero*, *one*, *etc*), or a real-time table (*t1* to *t60*).

*wait* is a keyword with values of `WAIT` or `NOWAIT`, that determines the delay until the execution of next statement. If the value is `WAIT`, the delay is *width* seconds. If the value is `NOWAIT`, the delay is 0.0 seconds.

<b>Related:</b>	<code>mashapedgradient</code>	Perform three-axis shaped gradient at magic angle
	<code>obl_shaped3gradient</code>	Perform three-axis oblique shaped gradient, three patterns
	<code>pe_gradient</code>	Perform oblique gradient, phase encode one axis
	<code>pe2_gradient</code>	Perform oblique gradient, phase encode two axes
	<code>pe3_gradient</code>	Perform oblique gradient, phase encode three axes
	<code>pe_shapedgradient</code>	Perform oblique gradient, one pattern, phase encode one axis
	<code>pe2_shapedgradient</code>	Perform oblique gradient, one pattern, phase encode two axes
	<code>pe2_shaped3gradient</code>	Perform oblique gradient, three patterns, phase encode two axes
	<code>pe3_shapedgradient</code>	Perform oblique gradient, one pattern, phase encode three axes
	<code>pe3_shaped3gradient</code>	Perform oblique gradient, three patterns, phase encode three axes

phase_encode3 pedgradient	Perform general oblique shaped gradient, phase encode three axes
rot_angle	Set user-defined oblique gradient-coordinate rotation axes

**pe2\_shapedgradient****Perform oblique gradient shape, one pattern, phase encode two axes**

**Syntax:**

```
pe2_shapedgradient(pattern,width,stat1,stat2,stat3,step2,step3,vmult2,vmult3,wait)
char *pattern;          /* name of .GRD file */
double width;           /* duration of gradient shape, seconds */
double stat1,stat2,stat3; /* static gradient amplitudes, Gauss/cm */
double step2,step3;     /* gradient-amplitude stepsizes, Gauss/cm */
codeint vmult2,vmult3;  /* real-time index of steps */
int wait;               /* WAIT or NOWAIT */
```

**Description:** Apply a static, oblique, shaped gradient with a single pattern for all three axes and a variable phase-encode gradient along the second (phase), and third (slice) logical axes in either WAIT or NOWAIT mode.

The pulse sequence aborts at run-time if the DACs on a particular gradient are overrun, after the angles and amplitude have been resolved.

**Arguments:** pattern is the root name of a text file in the shapelib directory with a .GRD extension to store the gradient shape.

width is the duration of the gradient shape, in seconds.

stat1, stat2, stat3 are the three components of the static portion of the gradient, in Gauss/cm, along the three logical axes, read, phase, and slice, respectively.

step2, step3 are the stepsizes, in Gauss/cm, for the variable gradients along the phase and slice axes.

vmult2, vmult3 are real-time indexes to the steps of the variable gradients. These arguments can be real-time variables (v1 to v42, oph, etc), real-time constants (zero, one, etc), or real-time tables (t1 to t60).

wait is a keyword with values of WAIT or NOWAIT, that determines the delay until the execution of next statement. If the value is WAIT, the delay is width seconds. If the value is NOWAIT, the delay is 0.0 seconds.

Related:	<code>mashapedgradient</code>	Perform three-axis shaped gradient at magic angle
	<code>obl_shaped3gradient</code>	Perform three-axis oblique shaped gradient, three patterns
	<code>pe_gradient</code>	Perform oblique gradient, phase encode one axis
	<code>pe2_gradient</code>	Perform oblique gradient, phase encode two axes
	<code>pe3_gradient</code>	Perform oblique gradient, phase encode three axes
	<code>pe_shapedgradient</code>	Perform oblique gradient, one pattern, phase encode one axis
	<code>pe_shaped3gradient</code>	Perform oblique gradient, three patterns, phase encode one axis
	<code>pe2_shaped3gradient</code>	Perform oblique gradient, three patterns, phase encode two axes
	<code>pe3_shapedgradient</code>	Perform oblique gradient, one pattern, phase encode three axes
	<code>pe3_shaped3gradient</code>	Perform oblique gradient, three patterns, phase encode three axes
	<code>phase_encode3oblshapedgradient</code>	Perform general oblique shaped gradient, phase encode three axes
	<code>rot_angle</code>	Set user-defined oblique gradient-coordinate rotation axes

**pe2\_shaped3gradient****Perform oblique gradient shape, three patterns, phase encode two axes**

**Syntax:**

```
pe2_shaped3gradient(pat1,pat2,pat3,width,stat1,stat2,stat3,step2,step3,vmult2,vmult3,wait)
char *pat1,*pat2,*pat3; /* names of .GRD files */
double width; /* duration of gradient shape, seconds */
double stat1,stat2,stat3; /* static gradient amplitudes, Gauss/cm */
double step2,step3; /* gradient-amplitude stepsizes, Gauss/cm */
codeint vmult2,vmult3; /* real-time index of steps */
int wait; /* WAIT or NOWAIT */
```

**Description:** Apply a static, oblique, shaped gradient with a separate pattern for each axis and variable phase-encode gradients along the second (phase)

and third (*slice*) logical axes in either `WAIT` or `NOWAIT` mode.

The pulse sequence aborts at run-time if the DACs on a particular gradient are overrun, after the angles and amplitude have been resolved.

Arguments:

`pat1`, `pat2`, `pat3` are the root names of text files in the `shapelib` directory with a `.GRD` extension to store the gradient shapes.

`width` is the duration of the gradient shape, in seconds.

`stat1`, `stat2`, `stat3` are the three components of the static portion of the gradient, in Gauss/cm, along the three logical axes, `read`, `phase`, and `slice`, respectively.

`step2`, `step3` are the stepsizes, in Gauss/cm, for the variable gradients along the `phase` and `slice` axes.

`vmult2`, `vmult3` are real-time indexes to the steps of the variable gradients. These arguments can be real-time variables (`v1` to `v42`, `oph`, *etc*), real-time constants (`zero`, `one`, `oph`, *etc*), or real-time tables (`t1` to `t60`).

`wait` is a keyword with values of `WAIT` or `NOWAIT`, that determines the delay until the execution of next statement. If the value is `WAIT`, the delay is `width` seconds. If the value is `NOWAIT`, the delay is 0.0 seconds.

Related:

<code>mashapedgradient</code>	Perform three-axis shaped gradient at magic angle
<code>obl_shaped3gradient</code>	Perform three-axis oblique shaped gradient, three patterns
<code>pe_gradient</code>	Perform oblique gradient, phase encode one axis
<code>pe2_gradient</code>	Perform oblique gradient, phase encode two axes
<code>pe3_gradient</code>	Perform oblique gradient, phase encode three axes
<code>pe_shapedgradient</code>	Perform oblique gradient, one pattern, phase encode one axis
<code>pe_shaped3gradient</code>	Perform oblique gradient, three patterns, phase encode one axis
<code>pe2_shapedgradient</code>	Perform oblique gradient, one pattern, phase encode two axes
<code>pe3_shapedgradient</code>	Perform oblique gradient, one pattern, phase encode three axes



pe3_shaped3gradient	Perform oblique gradient, three patterns, phase encode three axes
phase_encode3oblshapedgradient	Perform general oblique shaped gradient, phase encode three axes
rot_angle	Set user-defined oblique gradient-coordinate rotation axes

**pe3\_shapedgradient****Perform oblique gradient shape, one pattern, phase encode three axes**

**Syntax:** `pe3_shapedgradient (pattern,width,stat1,stat2,stat3,step1,step2,step3,vmult2,wait)`  
`char *pattern; /* name of .GRD file */`  
`double width; /* duration of gradient shape, seconds */`  
`double stat1,stat2,stat3; /* static gradient amplitudes, Gauss/cm */`  
`double step1,step2,step3; /*gradient-amplitude stepsizes, Gauss/cm */`  
`codeint vmult2; /* real-time index of steps */`  
`int wait; /* WAIT or NOWAIT */`

**Description:** Apply a static, oblique, shaped gradient with a single pattern for all three axes and a variable phase-encode gradient along the first (*read*), second (*phase*), and third (*slice*) logical axes in either *WAIT* or *NOWAIT* mode.

The pulse sequence aborts at run-time if the DACs on a particular gradient are overrun, after the angles and amplitude have been resolved.

**Arguments:** *pattern* is the root name of a text file in the *shapelib* directory with a *.GRD* extension to store the gradient shape.

*width* is the duration of the gradient shape, in seconds.

*stat1*, *stat2*, *stat3* are the three components of the static portion of the gradient, in Gauss/cm, along the three logical axes, *read*, *phase*, and *slice*, respectively.

*step1*, *step2*, *step3* are the stepsizes, in Gauss/cm, for the variable gradients along the *read*, *phase*, and *slice* axes.

*vmult1*, *vmult2*, *vmult3* are real-time indexes to the steps of the variable gradients. These arguments can be real-time variables (*v1* to *v42*, *oph*, *etc*), real-time constants (*zero*, *one*, *etc*), or real-time tables (*t1* to *t60*).

*wait* is a keyword with values of *WAIT* or *NOWAIT*, that determines the delay until the execution of

next statement. If the value is `WAIT`, the delay is width seconds. If the value is `NOWAIT`, the delay is 0.0 seconds.

Related:	<code>mashapedgradient</code>	Perform three-axis shaped gradient at magic angle
	<code>obl_shaped3gradient</code>	Perform three-axis oblique shaped gradient, three patterns
	<code>pe_gradient</code>	Perform oblique gradient, phase encode one axis
	<code>pe2_gradient</code>	Perform oblique gradient, phase encode two axes
	<code>pe3_gradient</code>	Perform oblique gradient, phase encode three axes
	<code>pe_shapedgradient</code>	Perform oblique gradient, one pattern, phase encode one axis
	<code>pe_shaped3gradient</code>	Perform oblique gradient, three patterns, phase encode one axis
	<code>pe2_shapedgradient</code>	Perform oblique gradient, one pattern, phase encode two axes
	<code>pe2_shaped3gradient</code>	Perform oblique gradient, three patterns, phase encode two axes
	<code>pe3_shaped3gradient</code>	Perform oblique gradient, three patterns, phase encode three axes
	<code>phase_encode3oblshapedgradient</code>	Perform general oblique shaped gradient, phase encode three axes
	<code>rot_angle</code>	Set user-defined oblique gradient-coordinate rotation axes

**pe3\_shaped3gradient**

**Perform oblique gradient shape, three patterns, phase encode three axes**

Syntax: `pe3_shaped3gradient(pat1,pat2,pat3,width,stat1,stat2,stat3,step1,step2,step3,vmult2,wait)`  
`char *pat1,*pat2,*pat3; /* names of .GRD files */`  
`double width; /* duration of gradient shape, seconds */`  
`double stat1,stat2,stat3; /* static gradient amplitudes, Gauss/cm */`  
`double step1,step2,step3; /*gradient-amplitude stepsizes, Gauss/cm */`  
`codeint vmult2; /* real-time index of steps */`  
`int wait; /* WAIT or NOWAIT */`

**Description:** Apply a static, oblique, shaped gradient with a separate pattern for each axis and variable phase-encode gradients along the first (*read*), second (*phase*), and third (*slice*) logical axes in either *WAIT* or *NOWAIT* mode.

The pulse sequence aborts at run-time if the DACs on a particular gradient are overrun, after the angles and amplitude have been resolved.

**Arguments:** *pat1*, *pat2*, *pat3* are the root names of text files in the *shapelib* directory with a *.GRD* extension to store the gradient shapes.

*width* is the duration of the gradient shape, in seconds.

*stat1*, *stat2*, *stat3* are the three components of the static portion of the gradient, in Gauss/cm, along the three logical axes, *read*, *phase*, and *slice*, respectively.

*step1*, *step2*, *step3* are the stepsizes, in Gauss/cm, for the variable gradients along the *read*, *phase*, and *slice* axes.

*vmult1*, *vmult2*, *vmult3* are real-time indexes to the steps of the variable gradients. These arguments can be real-time variable (*v1* to *v42*, *oph*, *etc*), real-time constants (*zero*, *one*, *etc*), or real-time tables (*t1* to *t60*).

*wait* is a keyword with values of *WAIT* or *NOWAIT*, that determines the delay until the execution of next statement. If the value is *WAIT*, the delay is *width* seconds. If the value is *NOWAIT*, the delay is 0.0 seconds.

<b>Related:</b>	<i>mashapedgradient</i>	Perform three-axis shaped gradient at magic angle
	<i>obl_shaped3gradient</i>	Perform three-axis oblique shaped gradient, three patterns
	<i>pe_gradient</i>	Perform oblique gradient, phase encode one axis
	<i>pe2_gradient</i>	Perform oblique gradient, phase encode two axes
	<i>pe3_gradient</i>	Perform oblique gradient, phase encode three axes
	<i>pe_shapedgradient</i>	Perform oblique gradient, one pattern, phase encode one axis
	<i>pe_shaped3gradient</i>	Perform oblique gradient, three patterns, phase encode one axis
	<i>pe2_shapedgradient</i>	Perform oblique gradient, one pattern, phase encode two axes

pe2_shaped3gradient	Perform oblique gradient, three patterns, phase encode two axes
pe3_shapedgradient	Perform oblique gradient, one pattern, phase encode three axes
phase_encode3oblshapedgradient	Perform general oblique shaped gradient, phase encode three axes
rot_angle	Set user-defined oblique gradient-coordinate rotation axes

**peloop**

**Start switchable phase-encode loop**

Syntax: `peloop(state,max_count,apv1,apv2)`  
`char state; /* compressed or standard */`  
`double max_count; /* numeric value to initialize apv1 */`  
`codeint apv1; /* real-time variable with maximum count */`  
`codeint apv2; /* real-time index of steps */`

Description: Execute a sequence-switchable loop that can use real-time variables in what is known as a compressed loop, or can use the standard arrayed features of PSG. In the imaging sequences, it uses the third character of the `seqcon` string parameter `seqcon[2]` for the state argument. The statement is used in conjunction with the `endpeloop` statement.

`peloop` differs from `msloop` in how it sets the `apv2` variable in standard arrayed mode (state is 's'). In standard arrayed mode, `apv2` is set to `nth2D-1` if `max_count` is greater than zero. `nth2D` is a PSG internal counting variable for the second dimension. When in the compressed mode, `apv2` counts from zero to `max_count-1`.

Arguments: `state` is either 'c' to designate the compressed mode, or 's' to designate the standard arrayed mode.  
`apv1` is a real-time variable that holds the maximum count.  
`apv2` is a real-time variable that holds the current count

Related: `msloop` Start switchable multislice loop  
`endloop` End real-time loop  
`endmsloop` End switchable multislice loop  
`endpeloop` End switchable phase-encode loop  
`loop` Start real-time loop  
`loopcheck` Check number of FIDS for compressed acquisition

**phase\_encode3\_gradient** Set general oblique gradient, phase encode three axes

**Syntax:** `phase_encode3_gradient (width,stat1,stat2,stat3,step1,step2,step3,vmult1,vmult2,vmult3,lim1,lim2,lim3)`

```
double width;                /*duration of
gradient shape, seconds */
double stat1,stat2,stat3;    /*static gradient
amplitudes, Gauss/cm */
double step1,step2,step3;
/*gradient-amplitude stepsizes, Gauss/cm */
codeint vmult1,vmult2,vmult3; /* real-time
indexes of steps */
double lim1,lim2,lim3;      /* maximum
gradient-amplitudes */
```

**Description:** Apply a static, oblique, gradient with variable phase-encode gradients along the first (*read*), second (*phase*), and third (*slice*) logical axes.

This statement accepts limits for the three stepsizes to avoid overrunning the DACs, after the angles and amplitude have been resolved.

**Arguments:** *stat1*, *stat2*, *stat3* are the three components of the static portion of the gradient, in Gauss/cm, along the three logical axes, *read*, *phase*, and *slice*, respectively.

*step1*, *step2*, *step3* are the stepsizes, in Gauss/cm, for the variable gradients along the *read*, *phase*, and *slice* axes.

*vmult1*, *vmult2*, *vmult3* are a real-time indexes to the steps of the variable gradients. These arguments can be real-time variables (*v1* to *v42*, *oph*, *etc*), real-time constants (*zero*, *one*, *etc*), or real-time tables (*t1* to *t60*).

*lim1*,*lim2*,*lim3* are maximum stepsizes in Gauss/cm.

<b>Related:</b>	<code>mashedgradient</code>	Perform three-axis shaped gradient at magic angle
	<code>obl_shaped3gradient</code>	Perform three-axis oblique shaped gradient, three patterns
	<code>pe3_shaped3gradient</code>	Perform oblique gradient, three patterns, phase encode three axes
	<code>phase_encode3_gradpulse</code>	Perform general oblique shaped gradient pulse, phase encode three axes
	<code>phase_encode3oblshapedgradient</code>	Perform general oblique shaped gradient, phase encode three axes

rot_angle	Set user-defined oblique gradient-coordinate rotation axes
-----------	--

**phase\_encode3\_gradpulse Perform general oblique gradient pulse, phase encode three axes**

**Syntax:** phase\_encode3\_gradient (width,stat1,stat2,stat3,step1,step2,step3,vmult1,vmult2,vmult3,lim1,lim2,lim3)

**Syntax:** double width; /\*duration of gradient pulse, seconds \*/  
double stat1,stat2,stat3; /\* static gradient amplitudes, Gauss/cm \*/  
double step1,step2,step3;  
/\*gradient-amplitude stepsizes, Gauss/cm \*/  
codeint vmult1,vmult2,vmult3; /\* real-time indexes of steps \*/  
double lim1,lim2,lim3; /\* maximum gradient-amplitude steps \*/

**Description:** Apply a static, oblique, gradient pulse with variable phase-encode gradients along the first (*read*), second (*phase*), and third (*slice*) logical axes.

This statement accepts limits for the three stepsizes to avoid overrunning the DACs, after the angles and amplitude have been resolved.

**Arguments:** width is the duration of the gradient pulse, in seconds.

stat1, stat2, stat3 are the three components of the static portion of the gradient, in Gauss/cm, along the three logical axes, *read*, *phase*, and *slice*, respectively.

step1, step2, step3 are the stepsizes, in Gauss/cm, for the variable gradients along the *read*, *phase*, and *slice* axes.

vmult1, vmult2, vmult3 are a real-time indexes to the steps of the variable gradients. These arguments can be real-time variables (*v1* to *v42*, *oph*, *etc*), real-time constants (*zero*, *one*, *etc*), or real-time tables (*t1* to *t60*).

lim1,lim2,lim3 are maximum stepsizes in Gauss/cm.

<b>Related:</b>	mashedgradient	Perform three-axis shaped gradient at magic angle
	obl_shaped3gradient	Perform three-axis oblique shaped gradient, three patterns
	pe3_shaped3gradient	Perform oblique gradient, three patterns, phase encode three axes

phase_encode3_gradient	Perform general oblique gradient, phase encode three axes
phase_encode3oblshapedgradient	Perform general oblique shaped gradient, phase encode three axes
rot_angle	Set user-defined oblique gradient-coordinate rotation axes

**phase\_encode3\_oblshapedgradient** Perform general oblique gradient shape, three patterns, phase encode three axes

**Syntax:**

```
phase_encode3_oblshapedgradient(pat1,pat2,pat3,width
,stat1,stat2,stat3,step1,step2,step3,vmult1,
vmult2,vmult3,lim1,lim2,lim3,loops,wait,tag)
char *pat1,*pat2,*pat3; /* names of .GRD
files */
double width; /* duration of
gradient shape in seconds */
double stat1,stat2,stat3; /* static gradient
amplitudes, Gauss/cm */
double step1,step2,step3;
/*gradient-amplitude stepsizes, Gauss/cm */
codeint vmult1,vmult2,vmult3; /* real-time
indexes of steps */
double lim1,lim2,lim3; /* maximum
gradient-amplitudes */
int loops; /* number of times
to loop */
int wait; /* WAIT or NOWAIT */
int tag; /*function to be
determined*/
```

**Description:** Apply a static, oblique, shaped gradient with a separate pattern for each axis and variable phase-encode gradients along the first (*read*), second (*phase*), and third (*slice*) logical axes in either *WAIT* or *NOWAIT* mode with looping.

This statement accepts limits for the three stepsizes to avoid overrunning the DACs, after the angles and amplitude have been resolved.

**Arguments:** *pat1*, *pat2*, *pat3* are the root names of text files in the *shapelib* directory with a *.GRD* extension to store the gradient shapes.

*width* is the duration of the gradient shape, in seconds.

*stat1*, *stat2*, *stat3* are the three components of the static portion of the gradient, in Gauss/cm, along the three logical axes, *read*, *phase*, and *slice*, respectively.

*step1*, *step2*, *step3* are the stepsizes, in Gauss/cm, for the variable gradients along the *read*, *phase*, and *slice* axes.

vmult1, vmult2, vmult3 are real-time indexes to the steps of the variable gradients. These arguments can be a real-time variables (v1 to v42, oph, etc), real-time constants (zero, one, etc), or real-time tables (t1 to t60).

lim1, lim2, lim3 are maximum stepsizes in Gauss/cm.

wait is a keyword with values of WAIT or NOWAIT, that determines the delay until the execution of next statement. If the value is WAIT, the delay is width seconds. If the value is NOWAIT, the delay is 0.0 seconds.

tag is a keyword to be determined.

Related:	mashapedgradient	Perform three-axis shaped gradient at magic angle
	obl_shaped3gradient	Perform three-axis oblique shaped gradient, three patterns
	pe3_shaped3gradient	Perform oblique gradient, three patterns, phase encode three axes
	phase_encode3_gradient	Perform general oblique gradient, phase encode three axes
	phase_encode3_gradpulse	Perform general oblique shaped gradient pulse, phase encode three axes
	rot_angle	Set user-defined oblique gradient-coordinate rotation axes

**poffset**

**Return frequency offset based on position**

Syntax: `offset = poffset(pos, gradlv1);`  
`double pos; /* slice position, cm */`  
`double grad; /* gradient amplitude, Gauss/cm */`

Description: Returns a frequency offset value (in Hz) from position and conjugate gradient values. Does not set the frequency generator device.

Arguments: pos is the slice position, in cm.  
gradlv1 is the gradient amplitude in Gauss/cm, used in the slice selection process.



Related:	<code>offsetglist</code>	Create offset array from position and gradient-amplitude array
	<code>offsetlist</code>	Create offset array from position and gradient amplitude
	<code>poffset_list</code>	Create offset array from position array
	<code>position_offset</code>	Return frequency offset based on position
	<code>position_offset_list</code>	Create offset array from position array

### **poffset\_list** Create offset array from position array

Syntax: `poffset_list(posarray,gradlvl,ns,apv1)`  

```
double posarray[]; /* position values, cm */
double gradlvl;    /* gradient amplitude, Gauss/cm */
double ns;         /* number of slices */
codeint apv1;     /* real-time index for slices */
```

Description: Set the rf frequency from a position list, conjugate gradient value and a real time index. `poffset_list` is functionally the same as `position_offset_list` except that `poffset_list` takes the value of `resfrq` from the `resto` parameter, assumes the device is the observe channel `OBSch`, and assumes that the list number is zero.

Arguments: `posarray` is an array of position values, in cm.  
`gradlvl` is the gradient amplitude in Gauss/cm, used in the slice selection process.  
`nslices` is the number of slices or position values.  
`apv1` is an index for the slices or position values. It can be real-time variable (v1 to v42) or table (t1 to t60).

Examples: `poffset_list(pss,gss,ns,v8);`

Related:	<code>getarray</code>	Obtain all values from arrayed parameter
	<code>offsetglist</code>	Create offset array from position and gradient-amplitude array
	<code>offsetlist</code>	Create offset array from position and gradient amplitude
	<code>poffset</code>	Return frequency offset based on position

position_offset	Retrun frequency offset based on position
position_offset_list	Create offset array from position array

**position\_offset**

**Return frequency offset based on position**

Syntax: `position_offset(pos,gradlv1,resfrq,device)`  
`double pos; /* slice position, cm */`  
`double gradlv1; /* gradient amplitude, Gauss/cm */`  
`double resfrq; /* resonance offset, Hz */`  
`int device; /* OBSch, DECch, DEC2ch, or DEC3ch */`

Description: Set the rf frequency from position and conjugate gradient values.

Arguments: pos is the slice position, in cm.  
gradlv1 is the gradient amplitude in Gauss/cm, used in the slice selection process.  
resfrq is the resonance offset value in Hz for the nucleus of interest.  
device is one of the *global* integer constants OBSch (observe channel), DECch (first decoupler), DEC2ch (second decoupler), DEC3ch (third decoupler), or DEC4ch (fourth decoupler).

Examples: `position_offset(pos1,gvox1,resto,OBSch);`

Related:	getarray	Obtain all values from arraye parameter
	offsetglist	Create offset array from position and gradient-amplitude array
	offsetlist	Create offset array from position and gradient amplitude
	poffset	Return frequency offset based on position
	poffset_list	Create offset array from position array
	position_offset_list	Create offset array from position array

**position\_offset\_list**

**Create offset array from position array**

Syntax: `position_offset_list(posarray,gradlv1,ns,resfrq,device,listId,apv1)`  
`double posarray[]; /* slice position, cm */`  
`double gradlv1; /* gradient amplitude,`

```

Gauss/cm */
double ns;           /* number of slices */
double resfrq;      /* resonance offset, Hz
*/
int device;         /* OBSch, DECch, DEC2ch,
or DEC3ch */
int listId;         /* integer label of
input list */
codeint apv1;       /* real-time index for
slices */

```

**Description:** Set the rf frequency from a position list, conjugate gradient value and a real-time index. The real-time index selects the slice offset value in the array. The arrays provided in this statement must count zero up; that is, array[0] must have the first slice position and array[ns-1] the last.

**Arguments:** posarray is a list of position values, in cm.  
gradlvl is the gradient amplitude, in Gauss/cm, used in the slice selection process.  
ns is the number of slices or position values.  
resfrq is the resonance offset, in Hz, for the nucleus of interest.  
device is one of the *global* integer constants OBSch (observe channel), DECch (first decoupler), DEC2ch (second decoupler), DEC3ch (third decoupler), or DEC4ch (fourth decoupler).  
listId is a value for identifying a *global* list. The first *global* list to be created is assigned 0 and each subsequent list must be incremented by one.  
apv1 is an index for the slices or position values. It can be real-time variable (v1 to v42) or table (t1 to t60).

<b>Related:</b>	getarray	Obtain all values from arraye parameter
	offsetglist	Create offset array from position and gradient-amplitude array
	offsetlist	Create offset array from position and gradient amplitude
	poffset	Return frequency offset based on position
	poffset_list	Create offset array from position array
	position_offset	Return frequency offset based on position

#### psg\_abort

#### Abort the PSG process at run-time

**Syntax:** `psg_abort(int_error)`  
`int_error; /* error argument, must be`  
`1 */`

**Description:** Abort the PSG process at run-time. The acquisition will not start. The `int_error` argument must be 1.

**Arguments:** `int_error` is the error argument, set to 1.

**Related:**

<code>abort_message</code>	Abort PSG process at run-time and send a message to VnmrJ
<code>text_error</code>	Send error message to VnmrJ
<code>text_message</code>	Send message to VnmrJ
<code>warn_message</code>	Send warning message to VnmrJ

#### pulse

#### Perform pulse with assigned values on observe channel

**Syntax:** `pulse(width,phase)`  
`double width; /* duration of pulse, seconds`  
`*/`  
`codeint phase; /* real-time quadrature-phase`  
`multiplier for pulse */`

**Description:** Set the quadrature phase and gate the observe channel on and off at the current power level with amplifier unblanking and blanking and set the predelay to `rof1` and the postdelay to `rof2`.

**Arguments:** `width` is the duration of the pulse, in seconds.  
`phase` is a 90° multiplier for the first-decoupler phase. The value must be a real-time variable (`v1` to `v42`, `oph`, etc), a real-time constant (`zero`, `one`, etc), or a real-time table (`t1` to `t60`).

**Examples:** `pulse(pw,v3);`  
`pulse(2.0*pp,zero);`

**Related:**

<code>decpulse</code>	Perform pulse with assigned values on first decoupler
<code>decrpulse</code>	Perform pulse on first decoupler
<code>obspulse</code>	Perform pulse with assigned values on observe channel
<code>pulse</code>	Perform pulse with assigned values on observe channel
<code>rgpulse</code>	Perform pulse on observe channel

#### putarray

#### Set all values of arrayed parameter from pulse sequence

**Syntax:** `putarray(paname,values,N);`  
`char *paname; /* name of arrayed parameter`  
`*/`  
`double values[]; /* array containing values`

```

*/
int N;          /* number of values */

```

**Description:** Set the values of a VnmrJ arrayed parameter `parname`, if it exists, for both the current and the processed trees, using the first `N` elements of an arrayed variable `values` in the pulse sequence.

**Arguments:** `parname` is the name of a numeric parameter in VnmrJ, to hold the array of values.

`values` is an arrayed variable containing numeric values to return.

`N` is the number of values to return.

**Examples:** `putarray("bvalue",bvalue,6);`

**Related:**

<code>getarray</code>	Obtain all values from arrayed parameter
<code>getval</code>	Obtain value of numeric parameter
<code>getstr</code>	Obtain value of string parameter
<code>offsetglist</code>	Create offset array from position and gradient-amplitude array
<code>offsetlist</code>	Create offset array from position and gradient amplitude
<code>poffset_list</code>	Create offset array from position array
<code>position_offset_list</code>	Create offset array from position array
<code>putCmd</code>	Send command to VnmrJ from pulse sequence
<code>putstring</code>	Set string parameter from pulse sequence
<code>putvalue</code>	Set numeric parameter from pulse sequence

**putCmd****Send command to VnmrJ from pulse sequence**

**Syntax:** `putCmd(command, varnames)`

```

char *command;      /* formatted command-line
expression */
varnames;           /*char, int and double
variables to be formatted */

```

**Description:** Allow the execution of any expression on the VnmrJ commandline from a pulse sequence. The argument `command` is a formatted string containing the command-line expression, using values of the following variables, `varnames`. Formatting is similar to the C `printf` statement.

The `putCmd` statement is often used to update parameter values in the workspace. For example:

```
putCmd("setvalue('d1',%g,'processed')
setvalue('d1',%g,'current')",d1,d1);
```

updates d1 for both the current and processed trees. It is important to explicitly convert from seconds to microseconds when setting parameters of the pulse subtype. For example:

```
putCmd("pw=%g", pw*1e6)
```

The go('check') command will execute the pulse sequence and any putCmd statements without starting an acquisition.

The integer checkflag has a value 1 if go('check') was called. Use checkflag as a conditional if putCmd should be run only with go('check'). For example:

```
if (checkflag) putCmd("d1=%g",d1);
```

The putCmd function is only active for the first increment of an arrayed or multidimensional experiment.

Related	getarray	Obtain all values from arrayed parameter
	getval	Obtain value of numeric parameter
	getstr	Obtain value of string parameter
	putarray	Set all values of arrayed parameter from pulse sequence
	putstring	Set string parameter from pulse sequence
	putvalue	Set numeric parameter from pulse sequence

**putstring**

**Set string parameter from pulse sequence**

Syntax: 

```
putstring(parname,string);
char *parname; /* name of string parameter */
char *string /* variable containing the string */
```

Description: Set a string parameter parname , if it exists, for both the current and the processed trees, using a variable string in the pulse sequence.

Arguments: parname is the name of a numeric parameter in VnmrJ.  
string is a variable in the pulse sequence with a string value to return.

Examples: 

```
putvalue("petable",petable_str);
```

Related	getarray	Obtain all values from arrayed parameter
	getval	Obtain value of numeric parameter

getstr	Obtain value of string parameter
putarray	Set all values of arrayed parameter from pulse sequence
putCmd	Send command to VnmrJ from pulse sequence
putstring	Set string parameter from pulse sequence
putvalue	Set numeric parameter from pulse sequence

**putvalue:****Set numeric parameter from pulse sequence**

**Syntax:** `putvalue(parname,value);`  
`char *parname; /* name of string parameter */`  
`double value /* variable containing the string */`

**Description:** Set a numeric parameter `parname`, if it exists, for both the current and the processed trees, using a variable `value` in the pulse sequence.

**Arguments:** `parname` is the name of a numeric parameter in VnmrJ.

`value` is a variable in the pulse sequence with a numeric value to return. .

**Examples:** `putvalue("te",minte);`

Related	getarray	Obtain all values from arrayed parameter
	getval	Obtain value of numeric parameter
	getstr	Obtain value of string parameter
	putarray	Set all values of arrayed parameter from pulse sequence
	putCmd	Send command to VnmrJ from pulse sequence
	putstring	Set string parameter from pulse sequence
	putvalue	Set numeric parameter from pulse sequence

## R

<code>rcvloff</code>	Turn off receiver and unblank observe amplifier
<code>rcvron</code>	Turn on receiver and blank observe amplifier
<code>rcvrphase</code>	Set small-angle phase of receivers
<code>rcvrstepsize</code>	Set small-angle phase stepsize of receivers
<code>readMRIUserByte</code>	Read MRI user byte on MRI User Panel
<code>recoff</code>	Turn off receiver gate
<code>rcvrphase</code>	Set small-angle phase of the receiver
<code>recon</code>	Turn on receiver gate
<code>rlloop</code>	Start real-time loop with fixed count
<code>rgradient</code>	Set DAC level of any one gradient axis
<code>rlpower</code>	Set power level of any channel
<code>rlpwrf</code>	Set fine power level of any channel
<code>rotate</code>	Set standard oblique gradient-coordinate rotation axes
<code>rot_angle</code>	Set user-defined oblique gradient-coordinate rotation axes
<code>rot_angle_list</code>	Set oblique gradient-coordinate rotation axes from list
<code>rotorperiod</code>	Obtain rotor period of external tachometer signal
<code>rotorsync</code>	Execute time delay based on external tachometer signal

**`rcvloff`****Turn off receiver and unblank observe amplifier**

Syntax: `rcvloff()`

Description: Gate all receivers off, put the respective T/R switches in transmit mode, and unblank the amplifiers associated with all observe channels. All gates are changed simultaneously and the `rcvloff` statement adds no time.

The `rcvloff` statement also sets an internal gate to disable the automatic blanking at the end of `rgpulse`.

It is usually not necessary to use `rcvloff` explicitly in a sequence to control the receiver. The VNMR5 receiver is off by default, except during acquisition. The `startacq` statement (either implicit or explicit) executes `rcvron` and `endacq` executes `rcvloff`. Amplifier blanking should be controlled explicitly by `obsblank` and `obsunblank`.

The `rcvloff` statement is sometimes used instead of `obsunblank` before the first pulse only to unblank the observe amplifier. This use of `rcvloff` suppresses the automatic blanking associated with the end of `rgpulse`. There is no effect on the receiver because it is already off by default.

For windowed, explicit acquisition, a pair of statements `rcvron` and `rcvloff` are used around the `sample` statement to toggle the receiver and amplifier blanking, so as to allow pulses between the `sample` periods.

Examples: `rcvloff();`

Related: `obsblank` Blank amplifier of observe channel



obsunblank	Unblank amplifier of observe channel
rcvroff	Turn off receiver and unblank observe amplifier
rcvron	Turn on receiver gate and blank observe amplifier
recon	Turn on receiver gate
rgpulse	Perform pulse on observe channel

**rcvron****Turn on receiver and blank observe amplifier**

Syntax: `rcvron(time)`

Description: Gate all receivers on, put the respective T/R switches in receive mode, and blank the amplifiers associated with all observe channels. Execute a delay of `time`, in seconds.

The value of `time` is often set equal to `alfa` or `ad`, parameters that define a minimum time for receiver turn-on before sampling. Usually `time` is 4.0 to 6.0 s, but for windowed sampling of protons, it can be as little as 0.5  $\mu$ s.

The `rcvron` statement is executed by the `startacq` statement. The `rcvron` and `startacq` statements are usually preceded by a delay with a duration of `rof2` or `rd`, parameters that define a time for probe ringdown before the receiver is turned on.

The `rcvron` statement executes a minimum time delay of `rof3` if this parameter exists. If `rof3` does not exist, `rcvron` executes a minimum delay of 2.0  $\mu$ s. The T/R switch is put in receiver mode and the observe amplifier is blanked immediately. The receiver is gated on after the delay of `rof3`. A final delay of `time - rof3` follows. The receiver and T/R switch typically take 0.5 to 2.0  $\mu$ s to turn on.

The delay of `rof3` is present to ensure that the receiver is not gated on immediately after a pulse. The value of `rof3` can be set to 0.0 if one can ensure that there is another delay between the pulse and `rcvron`. The value `rof3 = 0.0` is often used when `rcvron` is contained in a windowed, explicit acquisition loop.

For windowed, explicit acquisition, a pair of statements `rcvron` and `rcvroff` are used around the `sample` statement to toggle the receiver and amplifier blanking, so as to allow pulses between the `sample` periods.

Arguments: `time` is the duration of the `rcvron` statement, in seconds.

Examples: `rcvron(alfa); rcvron(ad);`

Related: `obsblank` Blank amplifier of observe channel

<code>obsunblank</code>	Unblank amplifier of observe channel
<code>rcvroff</code>	Turn off receiver and unblank observe amplifier
<code>rcvron</code>	Turn on receiver gate and blank observe amplifier
<code>recon</code>	Turn on receiver gate

**rcvrphase****Set small-angle phase of receivers.**

Syntax: `rcvrphase(multiplier)`  
`codeint multiplier; /* real-time`  
`receiver-phase multiplier */`

Description: Designate the phase of all receivers to be the product of a small-angle receiver `stepsize` and the value of real-time variable, `multiplier`. The small-angle receiver `stepsize` is set in degrees by the `rcvrstepsize(stepsize)` statement. The receiver phase is determined at acquisition as the sum of a quadrature component, determined by the real-time variable `oph`, and the small angle component determined by the value of `multiplier` and `stepsize`.

The phase of the acquired data is determined by a calculation in the DSP processors of the digital receiver and does not affect the phase coherence of the RF of the frequency synthesizer. Unlike `txphase` and `xmtrphase`, which change the transmitter phase when executed, `setreceiver` and `rcvrphase` can be used in either order with the same result.

Typically `oph` is set from a quadrature receiver phase table, using the `setreceiver` statement and `rcvrphase` is used to supply a constant small-angle phase offset. In this case, use `rcvrstepsize` to set the phase offset and fix `multiplier` with a value of 1.0.

The need for a phase offset might arise in a two-dimensional experiment if pulses or a spinlock just before acquisition have a frequency offset relative to the base frequency, set by `tof`. If the offset pulses change increment to increment, `rcvrphase` can be used with a changing `stepsize` to remove the resulting phase modulation from the F1 spectra.

The `rcvrphase` statement can be used in place of `setreceiver` to implement a small-angle phase table that varies scan-to-scan. The following statements would replace `setreceiver` and implement a phase table whose elements are multiples of 30°.

```
rcvrstepsize(30.0);
settable(t1,ct,rcvrtable);
rcvrphase(v1);
```

```
getelem(t1,ct,v1);
assign(oph,zero);
```

In this example, `rcvrtable` contains integers that multiply a 30° stepsize. The `getelem` statement is needed because `rcvrphase` does not accept a table argument. The `assign` statement guarantees that the automatic values of the `oph` phase table, set by the parameter `cp`, do not affect the acquisition.

During explicit acquisition, a receiver phase is recalculated at the beginning of each `startacq` statement. The `getelem` statement of the example above might also be used in a real-time loop to apply the phase table to subsequent blocks of a compressed acquisition, using `nf`.

The `rcvrphase` statement itself is executed in real-time. Multiple `rcvrphase` statements can also be used to change multiplier for different blocks of acquisition. The value of stepsize is a PSG *global* and so has a fixed value in each increment.

When multiple receivers are present, the receiver phase calculation can also depend on the values of the *global* variables `rcvrp[i]` and `rcvrp1[i]`, if they exist. The full equation for the receiver phase is:

```
phase[i] = rcvrp[i] + oph*90 +
multiplier*(stepsize + rcvrp1[i])
```

in which *i* designates the individual receiver, 0 = 1 to *n*-1, in which *n* is the number of receivers. The value `rcvrp[i]` is an overall zero-order phase offset and `rcvrp1[i]` is a first order phase offset to stepsize, and both can be set individually for each receiver.

**Arguments:** `multiplier` is a real-time multiplier of `stepsize` to determine the small-angle receiver phase. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), or a real-time constant (`zero`, `one`, *etc*). It cannot be a real-time table (`t1` to `t60`).

<b>Related:</b>	<code>rcvrphase</code>	Set small-angle phase of receiver
	<code>rcvrstepsize</code>	Set small-angle phase stepsize of receiver
	<code>setreceiver</code>	Set quadrature receiver phase from table
	<code>startacq</code>	Initialize explicit acquisition

### **rcvrstepsize**

#### **Set small-angle phase stepsize of receivers.**

**Syntax:** `rcvrstepsize(stepsize)`  

```
double stepsize;      /* receiver-phase
stepsize, in degrees */
```

**Description:** Set the small-angle phase `stepsize` in degrees for all receivers. The value `stepsize` is used along with a real-time `multiplier` to set the phase of the receiver at the beginning of each acquisition. The `multiplier` is set by the `rcvrphase` statement.

The argument `stepsize` is a PSG *global*. For explicit acquisition, a receiver phase is calculated at run-time using the current value of `stepsize` and fixed individually for each `startacq` statement.

The value of `stepsize` can be changed between blocks of a compressed acquisition, using `nf`, to provide a different value of `stepsize` for each block. Because `stepsize` is *global*, the blocks of acquisition must be programmed explicitly or placed in a C for loop. Only `multiplier`, which is a real-time variable, can be changed in a real-time loop.

When multiple receivers are present, the receiver phase calculation can also depend on the values of the global variables `rcvrp[i]` and `rcvrp1[i]`, if they exist. The full equation for the receiver phase is:

$$\text{phase}[i] = \text{rcvrp}[i] + \text{oph} * 90 + \text{multiplier} * (\text{stepsize} + \text{rcvrp1}[i])$$

in which `i` designates the individual receiver, `0 = 1` to `n-1`, in which `n` is the number of receivers. The value `rcvrp[i]` is an overall zero-order phase offset and `rcvrp1[i]` is a first order phase offset to `stepsize`, and both can be set individually for each receiver.

**Arguments:** `stepsize` sets a psg global variable at run-time, in degrees.

<b>Related:</b>	<code>rcvrphase</code>	Set small-angle phase of receiver
	<code>rcvrstepsize</code>	Set small-angle phase stepsize of receiver
	<code>setreceiver</code>	Set quadrature receiver phase from table
	<code>startacq</code>	Initialize explicit acquisition

**readMRIUserByte**

**Read MRI user byte on MRI User Panel**

**Syntax:** `readMRIUserByte(a,delay)`  
`codeint a; /* a real-time variable to contain the user byte */`  
`double delay;`

**Description:** Read the eight input lines from the 15-pin D-connector, User In, on the MRI User Panel on the back of the console, into the real-time variable `a` in a duration `delay`. A minimum delay of 50 ms is needed to complete the read. The read is performed

synchronously with the experiment. The value of `delay` can be used to assure that the user byte has been read before the next statement.

**Arguments:** `a` is a real-time variable in which the byte is stored.  
`delay` is a duration in seconds before the next statement.

**Examples:** `readMRIUserByte(v1, 50e-3);`

**recoff****Turn off receiver**

**Syntax:** `recoff()`

**Description:** Gate all receivers off and put the respective T/R switches in transmit mode.

This statement is identical to `rcvroff`, except that the transmitter must be unblanked explicitly.

**Related:**

<code>rcvroff</code>	Turn off receiver and unblank observe amplifier
<code>rcvron</code>	Turn on receiver and blank observe amplifier
<code>recon</code>	Turn on receiver

**recon****Turn on receiver**

**Syntax:** `recon()`

**Description:** Gate all receivers on and put the respective T/R switches in receive mode.

This statement is identical to `rcvron`, except that the transmitter must be blanked explicitly. The `recon` statement also does not affect the blanking state of `rgpulse`.

**Related:**

<code>rcvroff</code>	Turn off receiver and unblank observe amplifier
<code>rcvron</code>	Turn on receiver and blank observe amplifier
<code>recoff</code>	Turn off receiver

**rgpulse****Perform pulse on observe channel**

**Syntax:** `rgpulse(width, phase, RG1, RG2)`  
`double width; /* duration of pulse, seconds */`  
`codeint phase; /* real-time variable for phase */`  
`double RG1; /* duration of predelay in sec */`  
`double RG2; /* duration of postdelay in sec */`

**Description:** Set the quadrature phase and gates of the observe channel on and off at the current power level with amplifier unblanking and blanking. `rgpulse` is preceded by a predelay `RG1` and followed by a postdelay `RG2`. The associated amplifier is unblanked if it is in pulse mode and the phase is set, both at the beginning of the predelay. The associated amplifier is blanked at the end of the postdelay if it is in pulse mode.

Blanking at the end of the postdelay is suppressed if a `rcvroff` statement has been executed previously.

**Arguments:** `width` is the duration of the pulse, in seconds.  
`phase` is a 90° multiplier for the first-decoupler phase. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*), or a real-time table (`t1` to `t60`).  
`RG1` is the duration of the predelay, in seconds.  
`RG2` is the duration of the predelay, in seconds.  
`rcvroff`

**Examples:** `rgpulse(pw,v3,rof1,rof2);`  
`rgpulse(pw,zero,1.0e-6,0.2e-6);`

<b>Related:</b>	<code>decpulse</code>	Perform pulse with assigned values on first decoupler
	<code>dec2rgpulse</code>	Perform pulse on second decoupler
	<code>dec3rgpulse</code>	Perform pulse on third decoupler
	<code>dec4rgpulse</code>	Perform pulse on fourth decoupler
	<code>decshaped_pulse</code>	Perform shaped pulse on first decoupler
	<code>obspulse</code>	Perform pulse with assigned values on observe channel
	<code>pulse</code>	Perform pulse with assigned values on observe channel
	<code>rcvroff</code>	Turn off receiver and unblank observe amplifier
	<code>shaped_pulse</code>	Perform shaped pulse on observe channel
	<code>simshaped_pulse</code>	Perform simultaneous shaped pulses, observe and first decoupler
	<code>simpulse</code>	Perform simultaneous pulses, observe and first decoupler

**rgradient**

**Set DAC level of any one gradient axis**

**Syntax:** `rgradient(axis,daclvl)`  
`char axis; /* gradient 'x', 'X', 'y',`

```
'Y', 'z' or 'Z' */
double daclvl; /* gradient amplitude, DAC
units */
```

**Description:** Set the gradient amplifier to specified value in DAC units. The `gradalt` parameter can be used to multiply the amplitude on alternative scans.

**Arguments:** `axis` specifies the gradient channel. It can be one of the characters 'X', 'x', 'Y', 'y', 'Z', or 'z'. For imaging, `axis` can be 'gread', 'gphase', or 'gslice'.

`daclvl` specifies the gradient amplitude using a real number from -4096.0 to 4095.0 for the Performa I PFG module, and from -32768.0 to 32767.0 for the Performa II PFG module and imaging. *oph*, *etc*.

**Examples:** `rgradient('z',1327.0);`

<b>Related:</b>	<code>getorientation</code>	Read image-plane orientation
	<code>shapedgradient</code>	Generate shaped gradient
	<code>zgradpulse</code>	Create a gradient pulse on the z channel

## **rlendloop**

### **End real-time loop with fixed count**

**Syntax:** `void rlendloop(index)`  
`codeint index /* real-time variable index */`

**Description:** End a loop that was started by an `rlloop` statement.

**Arguments:** `index` is a real-time variable used as an index to keep track of the number of times through the loop. It must be the same variable used in `rlloop` and its value should not be altered by any real-time math statements.

**Examples:** `rlendloop(v10);`

<b>Related:</b>	<code>endloop</code>	End real-time loop
	<code>kzendloop</code>	End real-time loop with fixed duration
	<code>kzloop</code>	Start real-time loop with fixed duration
	<code>loop</code>	Start real-time loop
	<code>parallelend</code>	End parallel section of pulse sequence
	<code>parallelstart</code>	Start parallel section of pulse sequence
	<code>rlloop</code>	Start real-time loop with fixed count

**rlloop**

**Start real-time loop with fixed count**

```
Syntax:  int rlloop(count, vcount, index)
          int count                                /*
          duration of the loop, seconds */         /*
          codeint vcount                          /* number of times to
          loop */                                  /*
          codeint index                            /*
          real-time index to steps of loop */      /*
          int return                               /* number of times to loop */
```

**Description:** Start a loop to execute statements in real-time with a fixed number of repetitions. The `rlendloop` statement ends the loop begun by `rlloop`. This statement should be used to replace `loop` in parallel sections of a pulse sequence, created by `parallelstart-parallelend`. Unlike `loop` the total duration of `rlloop` is known at run time. `kzloop` returns the number of repetitions as an integer.

**Arguments:** `count` is an integer containing the number of times through the loop.

`vcount` is a real-time variable containing the number of times through the loop. Its value is initialized from `count`. The value of `vcount` should not be changed by any real-time math statements.

`index` is a real-time variable used as an index to keep track of the number of times through the loop. It must be the same variable as that used in `rlendloop` and its value should not be altered by any real-time math statements.

**Examples:**

```
int count;
count = (int) getval("loopcount");
rlloop(count,v1,v10);
    delay(3.0);
    rgpulse(p1,v1,0.0,0.0);
rlendloop(v10);
```

<b>Related:</b>	<code>endloop</code>	End real-time loop
	<code>kzendloop</code>	End real-time loop with fixed duration
	<code>kzloop</code>	Start real-time loop with fixed duration
	<code>loop</code>	Start real-time loop
	<code>parallelend</code>	End parallel section of pulse sequence
	<code>parallelstart</code>	Start parallel section of pulse sequence



`rlendloop` End real-time loop with fixed count

**rlpower****Set power level of any channel**

**Syntax:** `rlpower(power, device)`  
`double power; /* power level value, dB */`  
`int device; /* OBSch, DECch, DEC2ch,`  
`DEC3ch or DEC4ch */`

**Description:** Set the power level of any channel using the coarse attenuator. The second integer argument `device` selects the channel. The statement `rlpwrf(amplitude, OBSch)` is identical to `obspwrf(amplitude)`.

The `rlpower` statement inserts a delay of 50 ns into the pulse sequence and you should allow 3  $\mu$ s for the power to reach the new level during the next delay. The coarse attenuator can introduce a transient when it changes and it is a good practice to execute `obspower` only when the transmitter is blanked and gated off.

**Arguments:** `power` sets the coarse attenuator in 1 dB steps from a maximum power of 63 to a minimum of -16.

`device` is one of the *global* integer constants `OBSch` (observe channel), `DECch` (first decoupler), `DEC2ch` (second decoupler), `DEC3ch` (third decoupler), or `DEC4ch` (fourth decoupler).

**Examples:** `rlpower(63, OBSch);`

**Related:**

<code>dec2power</code>	Set power level of second decoupler
<code>dec3power</code>	Set power level of third decoupler
<code>dec4power</code>	Set power level of fourth decoupler
<code>rlpower</code>	Set the power level of any channel

**rlpwrf****Set fine power level of any channel**

**Syntax:** `rlpwrf(amplitude)`  
`double power; /* fine-power value */`  
`int device; /* OBSch, DECch, DEC2ch,`  
`DEC3ch or DEC4ch */`

**Description:** Set the fine-power value of the any channel using the linear modulator. The second integer argument `device` selects the channel. The statement `rlpwrf(amplitude, OBSch)` is identical to `obspwrf(amplitude)`.

**Arguments:** `amplitude` sets the fine power in units of 1 from a maximum of 4095 to a minimum of 0. The fine power and the coarse power can be used

interchangeably when 6 dB units of coarse power correspond to a  $\times 2$  change of the fine power.

device is one of the *global* integer constants OBSch (observe channel), DECch (first decoupler), DEC2ch (second decoupler), DEC3ch (third decoupler), or DEC4ch (fourth decoupler).

Examples: `rlpwr(3500, OBSch);`

Related: `decpwr` Set fine power level of first decoupler  
`dec2pwr` Set fine power level of second decoupler  
`dec3pwr` Set fine power level of third decoupler  
`dec4pwr` Set fine power level of fourth decoupler  
`rlpwr` Set fine power level of any channel

**rotate**

**Set standard oblique gradient-coordinate rotation angles**

Syntax: `rotate( )`

Description: Apply the standard oblique Euler rotation angles, defined by the *global* variables `phi`, `theta`, and `psi` and their respective parameters, for rotation between the logical axes (read, phase, and slice) and the physical axes X, Y, and Z.

Examples: `rotate( );`

Related: `create_rotation_list` Create list of gradient-coordinate rotation angles  
`rot_angle` Set user-defined oblique gradient-coordinate rotation angles  
`rot_angle_list` Set gradient-coordinate rotation angles from a list

**rot\_angle**

**Set user-defined oblique gradient-coordinate rotation angles**

Syntax: `rot_angle(phi, theta, psi)`  
`double phi, theta, psi; /* user defined rotation angles*/`

Description: Sets user-defined, oblique Euler rotation angles `phi`, `theta`, and `psi` for the gradient rotation between the logical axes (read, phase, and slice) and the physical axes X, Y, and Z.

Arguments: `phi`, `theta`, and `psi` are the user defined oblique Euler angles in degrees.

Examples: `rot_angle(phi, theta, psi);`

Related:	<code>create_rotation_list</code>	Create list of gradient-coordinate rotation angles
	<code>rotate</code>	Set standard oblique gradient-coordinate rotation angles
	<code>rot_angle_list</code>	Set gradient-coordinate rotation angles from list

**rot\_angle\_list****Set gradient-coordinate rotation angles from list**

Syntax: `rot_angle_list(listId, state, vindex)`  
`int listId; /* ID of the list */`  
`char mode; /* indexing mode */`  
`codeint vindex; /* real-time v-variable */`

Description: Set user-defined, oblique Euler rotation angles from a previously defined list, created using the `create_rotation_list` statement. The list is referenced by an integer `listId`. The angles in the list can be accessed in real-time loops with a real-time index `vindex` or in a C-loop using an integer index, depending on the value of `state`.

Arguments: `listId` is an integer that identifies the rotation-angle list, created earlier using the `create_rotation_list` statement.

`state` is a character indicating the indexing mode. It can be set to 'c' for compressed mode, 's' for standard mode, and 'i' for indexed mode. For index mode, `vindex` should be an integer index of a C for statement.

`vindex` is a real-time variable for compressed and standard modes. For index mode, `vindex` should be an integer index of a C for statement. The value should range from 1 to the number of rotation angle sets in the specified list.

Related:	<code>create_rotation_list</code>	Create list of gradient-coordinate rotation angles
	<code>rotate</code>	Set standard oblique gradient-coordinate rotation angles
	<code>rot_angle</code>	Set user-defined oblique gradient-coordinate rotation angles

**rotorperiod****Obtain period of external tachometer signal**

Syntax: `rotorperiod(period)`  
`codeint period; /* real-time variable for one tachometer period */`

**Description:** Assign the current tachometer period from the master controller to a real-time variable designating period in units of 100 ns. Execution of this statement requires the connection of a square-wave signal source as described for the statement `rotorsync`.

**Arguments:** `period` should be a real-time variable (v1 to v42) whose value will be assigned as an integer corresponding to the tachometer period in units of 100 ns. For example, if one tachometer period is 170 s, the statement `rotorperiod(v4)` will assign a value of 1700 to v4. The corresponding tachometer frequency (rotor speed) would be  $1E + 7 / 1700 = 5882$  Hz.

**Examples:** `rotorperiod(v4);`

**Related:**

<code>rotorsync</code>	Execute time delay based on external tachometer signal
<code>xgate</code>	Gate pulse sequence from external tachometer signal

**rotorsync**

**Execute time delay based on external tachometer signal**

**Syntax:** `rotorsync( periods )`  
`codeint periods; /* number of tachometer periods to wait */`

**Description:** Halt all channels and resume execution after a duration of `periods` tachometer signal cycles. The tachometer is a square-wave signal that is usually generated from a MAS Speed Controller or from a Pneumatics and Tachometer Box. These devices that supply the signal should be connected to the Tachometer input of the Pneumatics Router and tee'd off to the external trigger port of the Acquisition Computer.

The `rotorsync` statement operates by continuous measurement of the tachometer period in the master controller and by separately acknowledging the rise-time of the square wave at the external trigger. The `rotorsync` statement measures the duration to the first rise-time event. It then counts subsequent events until `periods-1` events have passed and finally supplies the remainder of one tachometer period to complete `periods` full cycles.

**Arguments:** `periods` is a real-time variable that specifies the number of tachometer cycles to pass before restarting the pulse sequence.

**Examples:** `rotorsync(v6);`

Related: rotorperiod Obtain period of external  
tachometer signal  
xgate Gate pulse sequence from external  
tachometer signal

## S

sample	Acquire data explicitly during time delay
setacqmode	Set windowed acquisition for explicit sampling
set_angle_list	Select angle from 1D real-time list
setMRIUserGates	Set all three gates of MRI User panel
setreceiver	Set quadrature receiver phase from table
setstatus	Set decoupler status of any channel
settable	Assign integer array to table
shapedpulse	Perform shaped pulse on observe channel
shaped_pulse	Perform shaped pulse on observe channel
shapedgradient	Perform shaped gradient pulse on any one axis
shapelist	Create pulse-shape list from base pattern and offset array
shapedpulselist	Perform shaped pulses from shape list on observe channel
shapedpulseoffset	Perform shaped pulse on observe channel with offset
shapelistpw	Return exact pulse width of shaped-pulse pattern
simpulse	Simultaneous pulses, observe and first decoupler
sim3pulse	Simultaneous pulses, observe, first and second decouplers
sim4pulse	Simultaneous pulses, observe, first, second and third decouplers
simshaped_pulse	Simultaneous shaped pulses, observe and first decoupler
sim3shaped_pulse	Simultaneous shaped pulses, observe, first and second decouplers
sim4shaped_pulse	Simultaneous shaped pulses, observe, first, second and third decouplers
sp#off	Turn off spare line (#=1,2, or 3)
sp#on	Turn on spare line (#=1,2, or 3)
spinlock	Perform waveform spinlock on observe channel
starthardloop	Start hardware loop
startacq	Initialize explicit acquisition
startacq_obs	Initialize explicit acquisition in parallel section
startacq_rcvr	Initialize explicit acquisition in parallel section
status	Set status of decoupler and homospoil
statusdelay	Execute status statement within time delay
stepsize	Set small-angle phase stepsize of any channel
sub	Subtract real-time integer values
swift-acquire	Execute SWIFT RF pulses and gated acquire pulse train

**sample****Acquire data explicitly during time delay**

Syntax: `sample(time)`  
double time /\* time duration of sample  
in sec \*/

Description: Acquire data points using the VNMR5 digital receiver with a 12.5 ns dwell time (80 MHz rate) for a duration of `time`, in seconds. The data points are automatically downsampled with calculation of digital filters to produce a set of complex points with a dwell time of  $1.0/sw$ .

For acquisition of one FID with a single sample statement, set `time = at`. The resulting FID will consist of  $np/2.0$  complex points with a dwell time of  $1.0/sw$ , where  $at = np/(2.0*sw)$ . The latter equality is usually set automatically by `VnmrJ` when any of the three parameters `at`, `np`, and `sw` are set, but this situation is not certain. If `time` is not equal

to  $np / (2.0 * sw)$ , the number of points produced will be erroneous. Use `sample(np / (2.0 * sw))` to avoid `np` errors.

The statements `acquire(np, 1.0 / sw)` and `sample(np / (2.0 * sw))` have exactly the same function on VNMRS. The `sample` statement is not back-compatible with Unity-series spectrometers. For VNMRS, the parameters `at`, `np`, and `sw` designate the number of downsampled points and the nominal dwell time to be calculated from the 80 MHz data stream of the digital receiver and they have no direct effect on the timing of the sequence.

The actual sampling period is determined by the arrangement and timing of all the `sample` statements in the pulse sequence. You should take care to program the sequence so that the designated sampling period in the sequence does not exceed `at`. If you do not do so, unpredictable behavior and/or a program crash can result.

If a single `sample` or `acquire` statement is used explicitly in a pulse sequence, the statements `startacq` and `endacq` will be inserted at run time, before and after. It is good practice to also explicitly include these statements. If a pulse sequence does not explicitly contain `sample` or `acquire`, `acquire(np, 1.0 / sw)` will be inserted at run-time, bracketed by `startacq` and `endacq`. This construction is used commonly and is called *implicit acquisition*. In this case, no statements may be executed after acquisition. Decouplers set by `status` will be turned off automatically after an implicit acquisition.

If one or more `sample` or `acquire` statements are present, the construction is called *explicit acquisition*. In this case, you can control the timing of acquisition in the sequence relative to other statements and insert multiple `sample` statements.

You may insert multiple statements

`sample(np / (2.0 * sw))` (bracketed by `startacq` and `endacq`) to collect multiple FIDs in a single scan, each with `np` points. This construction is called *compressed acquisition*. When compressed acquisition is used, you must create the parameter `nf` with a value equal to the number of FIDs. Compressed acquisition is used commonly in imaging experiments to obtain phase-encoded two-dimensional data in a single scan. In this case, the `sample` statements should be preceded by a real-time loop or the switchable loops `msloop` and/or `peloop` and be followed by an `endloop` statement.

A compressed acquisition does not require the use of a real-time or switchable loop. Instances of

`sample` may be arranged in the sequence or placed in a `C for` loop to be arranged at run-time.

You may insert multiple `sample` statements to collect a single FID with `np` points, for which the time between `sample` statements may contain pulses or decoupling waveforms. This construction is known as *windowed acquisition*. For windowed acquisition the `sample` statements are usually inserted between `loop` and `endloop`. A `startacq` statement must precede the `loop` statement and an `endacq` statements must follow the `endloop` statement. These statements should not be present in the loop. The parameter `nf` is not needed for windowed acquisition, or it can be set to 1.0.

A windowed acquisition may be performed in *constant sampling* mode. In this mode, sampling of data begins with the first instance of `sample` and proceeds continuously until the end of the last `sample` statement. The digital receiver outputs data points with a value of zero during the time between `sample` statements. All points are downsampled without regard to whether they contain data or zeros. Constant sampling mode is the default for windowed acquisition loops bracketed by `startacq` and `endacq`. It is used primarily for multipulse experiments for solid-state NMR.

A windowed experiment may be performed in *explicit sampling* mode. In this mode, sampling of data occurs only during the `sample` statements and all the blocks of sampling are joined before downsampling. You can designate explicit sampling by including the statement `setacqmode(WACQ|NZ)` in the sequence before the `startacq` that begins the first acquisition. Also, you may designate explicit sampling by failing to include the `startacq` and `endacq` statements around the acquisition loop. Explicit sampling is also the default for compressed acquisition, where `startacq` and `endacq` statements are inside the acquisition loop.

For all pulse sequences with windowed acquisition, you must be sure that the parameter `at` and the related parameters, `np` and `sw`, designate a time that is greater than or equal to the sampling time of the sequence. If `at` is less than the sampling time, an error will result, possibly with a crash of the pulse sequence. If `at` is greater than the sampling time, the data will continue sampling zeros so that  $np = at / (2.0 * sw)$ .

For any single acquisition of `np` downsampled points, for either a single or compressed acquisition, by default, the digital receiver will sample for a required period after the last point for a period of about  $1.0 / sw$ . This period is present to sample enough data to accurately construct the last point. The actual sampling period in the sequence



will always exceed the acquisition time `at`. The duration of the actual sampling period will be placed in the parameter `acqtm`, if it exists. To force `acqtm = at`, create `acqtm` in the parameter set and set `acqtm = 'n'`. In this case, the extra sampling will be suppressed, though the last point will be inaccurate. For many experiments, the last point is noise, and you can set `acqtm = 'n'`. If truncated data is expected, you should not set `acqtm='n'`.

An overhead period of about 100 to 200  $\mu$ s is initiated by the `endacq` statement at the end of any acquisition of `np` downsampled points. This period is placed in the delay following the `endacq` statement. An error will result if the succeeding delay does not accommodate this period. For most sequences, the delay following `endacq` is the recycle delay `d1` and the overhead period sets the minimum value of `d1`. For other sequences and for sequences using compressed acquisition, you must be sure a delay is present to accommodate the overhead period.

**Arguments:** `time` is the duration, in seconds, of the sampling interval. For VNMR5, the two numbers have no individual meaning. For compatibility with the Unity series spectrometers, `number_points/2.0` is the nominal number of complex points to be sampled and `sampling_interval` is the nominal time between complex points.

<b>Related:</b>	<code>endacq</code>	End explicit acquisition
	<code>rcvloff</code>	Turn off receiver and unblank observe amplifier
	<code>rcvron</code>	Turn on receiver and blank observe amplifier
	<code>sample</code>	Acquire data explicitly during time delay
	<code>setacqmode</code>	Set acquisition for explicit sampling
	<code>startacq</code>	Initialize explicit acquisition

### **setacqmode**

#### **Set acquisition for explicit sampling**

**Syntax:** `setacqmode(WACQ|NZ)`

**Description:** Set a windowed acquisition to sample data only during the `sample` statement and to exclude sampling during periods between `sample` statements. Blocks of data are concatenated before downsampling. This acquisition mode is called *explicit sampling*. The statement `setacqmode(WACQ|NZ)` should be present in the pulse sequence somewhere before the `startacq` statement that begins acquisition. Alternatively, you

may designate explicit sampling by failing to include the `startacq` and `endacq` statements around the acquisition loop. In this case, `setacqmode(WACQ|NZ)` is not needed. Explicit sampling is also the default for compressed acquisition, where `startacq` and `endacq` statements are inside the acquisition loop.

The alternative to explicit sampling is *constant sampling*. In this mode, sampling of data begins with the first instance of `sample` and proceeds continuously until the end of the last `sample` statement. The digital receiver outputs data points with a value of zero during the time between `sample` statements. All points are downsampled without regard to whether they contain data or zeros. Constant sampling mode is the default for windowed acquisition loops bracketed by `startacq` and `endacq`. It is used primarily for mutlipulse experiments for solid-state NMR.

**Arguments:** The values of the arguments must be entered precisely as `WACQ|NZ`. There are no other public uses of the `setacqmode` statement.

**Examples:** `setacqmode(WACQ|NZ);`

<b>Related:</b>	<code>endacq</code>	End explicit acquisition
	<code>rcvloff</code>	Turn on receiver and blank observe amplifier
	<code>rcvron</code>	Turn off receiver and unblank observe amplifier
	<code>sample</code>	Acquire data explicitly during time delay
	<code>setacqmode</code>	Set acquisition for explicit sampling
	<code>startacq</code>	Initialize explicit acquisition

### `set_angle_list`

#### Set angle from 1D real-time list

**Syntax:** `set_angle_list(listId,angle_name,state,vindex)`  
`int listId; /* ID of the list */`  
`char *angle_name; /* angle to be set, psi,`  
`theta or phi */`  
`char state; /* indexing mode */`  
`codeint vindex; /* real-time v-variable */`

**Description:** Set one of three user-defined, oblique Euler rotation angles from a previously defined list, created using the `create_angle_list` statement. The list is referenced by an integer `listId`. The angles in the list must be accessed in real-time loops with the real-time index `vindex` and state must be set to 'c'.

`listId` is an integer that identifies the rotation-angle list, created earlier using the `create_angle_list` statement.

`angle_name` is a string specifying the angle as one of "psi", "theta" or "phi".

`state` is a character indicating the indexing mode. It must be set to 'c' for compressed mode. Standard 's' and indexed mode 'i' are not implemented.

`vindex` is a real-time variable used for compressed mode.

Examples: 

```
set_angle_list(Id1, "psi", 'c',v2));
set_angle_list(Id1, "theta", 'c',v2));
set_angle_list(Id1, "phi", 'c',v2));
```

Related:	<code>exe_grad_rotation</code>	Set oblique gradient-coordinate rotation angles in real-time
	<code>create_angle_list</code>	Create 1D real-time list of angles
	<code>create_rotation_list</code>	Create list of oblique gradient-coordinate rotation angles
	<code>rot_angle</code>	Set user-defined oblique gradient-coordinate rotation angles
	<code>rot_angle_list</code>	Set oblique gradient-coordinate rotation angles from a list
	<code>rotate</code>	Set standard oblique gradient-coordinate rotation angles
	<code>set_angle_list</code>	Select angle from a 1D real-time list

### **setMRIUserGates**

#### **Set all three gates of MRI User Panel**

Syntax: 

```
setMRIUserGates(int a)
```

Description: Set the state of the three BNCs (GATE1, GATE2 and GATE3) on the MRI User Panel on the back of the console from the binary representation of the real-time variable `a`.

Arguments: The argument is the real-time variable with value 0 to 7 that contains the binary states of the gates.

Examples: 

```
setMRIUserGates(v3);
```

### **setreceiver**

#### **Set quadrature receiver phase from table**

Syntax: 

```
setreceiver(table)
codeint table; /* real-time table */
```

Description: Assign the  $c^{\text{th}}$  element of a table to the receiver variable `oph`. Multiple `setreceiver` statements may be used or the value of `oph` can be changed by

real-time math statements; but this practice is not recommended. The last value of `oph` prior to the acquisition of data determines the value of the receiver phase.

**Arguments:** `table` specifies the name of the table (t1 to t60).

**Examples:** `setreceiver(t18);`

<b>Related:</b>	<code>getelem</code>	Assign real-time integer using table element
	<code>loadtable</code>	Assign table elements from a table text file
	<code>setreceiver</code>	Set quadrature receiver phase cycle from table
	<code>settable</code>	Assign integer array to table

## setstatus

### Set decoupler status of any channel

**Syntax:**

```
setstatus(channel,on,mode,sync,mod_freq)
int channel;          /* OBSch, DECch, DEC2ch,
DEC3ch, DEC4ch */
int on; /* TRUE (=on) or FALSE (=off) */
char mode;           /* 'c', 'w', 'g', etc */
int sync;            /* TRUE (=synchronous) or
FALSE (=asynchronous) */
double mod_freq;    /* modulation frequency */
```

**Description:** Set the decoupling status of any channel in the pulse sequence and override the status parameters such as `dm` and `dmm`. Unlike `status(A)`, the `setstatus` statement is never executed with the `su` command.

If `sync = FALSE`, the pattern is executed, thereby starting scan-to-scan at a pseudo-random position in the pattern in order to average decoupling sidebands. If `sync = TRUE`, the pattern is always executed starting at the beginning.

**Arguments:** `channel` has the values `OBSch`, `DECch`, `DEC2ch`, `DEC3ch` or `DEC4ch` to set the channel for decoupling.

`on` is `TRUE` (turn on decoupler) or `FALSE` (turn off decoupler).

`mode` is one of the following values for a decoupler mode (for further information on decoupler modes, refer to the description of the `dmm` parameter in the manual *Command and Parameter Reference*:

'c' sets continuous wave (CW) modulation.

'f' sets fm-fm modulation (swept-square wave).

'g' sets GARP modulation.

'm' sets MLEV-16 modulation.

'p' sets programmable waveform modulation.

'r' sets square wave modulation.

'u' sets square wave modulation.

'w' sets WALTZ-16 modulation.

'x' sets XY32 modulation.

sync is TRUE (decoupler is synchronous) or FALSE (decoupler is asynchronous).

mod\_freq is the modulation frequency.

Examples:

```
setstatus(DECch,TRUE,'w',FALSE,dmf);
setstatus(DEC2ch,FALSE,'c',FALSE,dmf2);
```

Related:	hsdelay	Execute time delay with optional homospoil pulse
	status	Set status of decoupler and homospoil
	statusdelay	Execute status statement with time delay

## settable

### Assign integer array to table

Syntax: `settable(tablename,numelements,intarray)`  

```
codeint tablename; /* real-time table
variable */
int numelements; /* number in array */
int *intarray; /* pointer to array of
elements */
```

Description: Store an integer array in a real-time table.

Arguments: table is the name of the table (t1 to t60).

number\_elements is the size of the table.

intarray is a C array that contains the table elements. Before calling settable, the integer array must be predefined and predimensioned in the pulse sequence using C statements.

Examples: `settable(t1,8,int_array);`

Related:	getelem	Assign real-time integer using table element
	loadtable	Assign table elements from table text file
	setreceiver	Set quadrature receiver phase cycle from table

## shapelist

### Create pulse-shape list from base pattern and offset array

Syntax: `listId=shapelist(pattern,width,offsetarray,ns,state)`  

```
char *pattern; /* name of the base shape */
double width; /* duration of pulse, seconds */
doubleoffsetarray[]; /* double array
containing offsets */
```

```

doublens; /* number of shapes */
char state; /* compression 'c','s' or 'i' */
return listId: /* integer with the
list ID */

```

**Description:** Generate an internal list of RF shape patterns from a base shape and an array of offsets. The shapes are phase modulated to shift the frequency of pulses. The shapes are stored directly in the RF controller and are not written into user's *shapelib* directory.

Frequency offsets are input from *offsetarray*, a C *double array*. This array is typically calculated using the *offsetlist* command, but can also be computed directly by the user. The *shapelist* statement returns an integer value *listId* to identify the list. The mode of usage for the resulting shape patterns is determined by *state*, which can be 'c' for compressed, 's' for standard, or 'i' for indexed mode. The usage of *shapelist* inside C code for loops should be avoided, as all the shape calculations are done in a single call for mode = 'c' or 'i'.

**Arguments:** *pattern* is the root name of a *.RF* file in *shapelib* containing the base shape.

*width* is the duration of the pulse, in seconds.

*offsetarray* is a C array containing the offset frequencies in Hz.

*ns* is the the number of offset shapes in the list.

*state* is a character, where 'c' is compressed mode, 's' is standard mode and 'i' is indexed mode.

The return is *listid*, an integer index that identifies the list of shapes.

<b>Related:</b>	<i>getarray</i>	Obtain all values from arrayed parameter
	<i>offsetglist</i>	Create offset array from position and gradient-amplitude array
	<i>offsetlist</i>	Create offset array from position and gradient amplitude
	<i>poffset</i>	Return frequency offset based on position
	<i>poffset_list</i>	Create offset array from position array
	<i>position_offset</i>	Return frequency offset based on position
	<i>position_offset_list</i>	Create offset array from position array

shapelist	Create pulse-shape list from base pattern and offset array
shapedpulselist	Perform shaped pulses from shape list on observe channel
shapedpulseoffset	Perform shaped pulse on observe channel with offset

**shapedpulselist****Perform shaped pulses from shape list on observe channel**

**Syntax:**

```
shapepulselist(listid, width, phase, RG1,
RG2, state, index)
int listid; ; /* integer with the list ID
*/
double width; /* duration of the pulse,
seconds */
int phase; /* real-time
quadrature-phase multiplier */
double RG1; /* duration of predelay,
seconds */
double RG2; /* duration of postdelay,
seconds */
char state; /* compression mode
'c','s'or 'i'*/
int index; /* index of individual shape
*/
```

**Description:** Apply a shaped pulse with a width in seconds, phase, predelay RG1, and postdelay RG2 on the observe channel that is obtained from a pre-computed list of shapes. The pre-computed list contains shapes created by the shapelist statement, which phase-modulates a base shape to produce a group of frequency offset shaped pulses. The shapelist command returns an integer listid, which is used as an argument of shapedpulselist. The list of shapes is indexed using the state and index arguments. The state argument specifies compressed, standard, or indexed modes. Its value must be the same as that used in the related shapelist and peloop or msloop statements. For imaging sequences, the value of state is supplied by the variable seqcon[n]. If the state is compressed, index should be a real-time variable.

**Arguments:** listid is an integer identifying the list of shapes.  
width is the duration of the pulse, in seconds.  
phase is a real-time multiplier to set the quadrature phase.  
RG1 is the duration of the predelay, in seconds.  
RG2 is the duration of the postdelay, in seconds.

state is a character, where 'c' is compressed mode, 's' is standard mode and 'i' is indexed mode.

index is an index identifying an individual shape. For compressed mode, index should be a real-time variable. For standard mode, index is a C integer set to 1. For indexed mode, index is a C integer identifying the individual shape.

Related:	getarray	Obtain all values from arrayed parameter
	offsetglist	Create offset array from position and gradient-amplitude array
	offsetlist	Create offset array from position and gradient amplitude
	poffset	Return frequency offset based on position
	poffset_list	Create offset array from position array
	position_offset	Return frequency offset based on position
	position_offset_list	Create offset array from position array
	shapelist	Create pulse-shape list from base pattern and offset array
	shapedpulselist	Perform shaped pulses from shape list on observe channel
	shapedpulseoffset	Perform shaped pulse on observe channel with offsey

### shapedpulseoffset

### Perform shaped pulse on observe channel with offset

Syntax: `shapedpulseoffset(pattern, width, phase, RG1, RG2, offset)`  

```
char *pattern; /* name of the base shape */
double width; /* duration of pulse, seconds */
int phase; /* real-time quadrature-phase multiplier */
double RG1; /* duration of predelay, seconds */
double RG2; /* duration of postdelay, seconds */
double offset; /* frequency offset in Hz */
```

Description: Applies a shaped pulse on the observe channel with the root name pattern and a frequency offset offset, with a width in seconds, phase, predelay RG1 and postdelay RG2. The offset is created by



phase modulation, which is applied automatically in the RF controller.

**Arguments:** `pattern` is the root name of a `.RF` file in `shapelib` containing the base shape.  
`width` is the duration of the pulse, in seconds.  
`phase` is a real-time multiplier to set the quadrature phase.  
`RG1` is the duration of the predelay, in seconds.  
`RG2` is the duration of the postdelay, in seconds.  
`offset` is the frequency offset relative to the synthesizer frequency, in Hz.

<b>Related:</b>	<code>getarray</code>	Obtain all values from arrayed parameter
	<code>offsetglist</code>	Create offset array from position and gradient-amplitude array
	<code>offsetlist</code>	Create offset array from position and gradient amplitude
	<code>poffset</code>	Return frequency offset based on position
	<code>poffset_list</code>	Create offset array from position array
	<code>position_offset</code>	Return frequency offset based on position
	<code>position_offset_list</code>	Create offset array from position array
	<code>shapelist</code>	Create pulse-shape list from base pattern and offset array
	<code>shapedpulselist</code>	Perform shaped pulses from shape list on observe channel
	<code>shapedpulseoffset</code>	Perform shaped pulse on observe channel with offset

### **shapelistpw** Return exact pulse width of shaped-pulse pattern

**Syntax:**

```
value=shapelistpw(pattern,width)
char *shapename;
double width;      /* nominal duration of the
pulse, seconds */
return value:      /* exact duration of pulse,
seconds */
```

**Description:** Return the exact pulse width of a shaped pulse with the name `pattern` and a desired width.

**Arguments:** `pattern` is the root name of a `.RF` file in `shapelib` containing the shape name.

`width` is the requested duration of the shaped pulse, in seconds.

The return value is the exact duration of the pulse, in seconds.

Related:	<code>init_rfpattern</code>	Obtain all values from arrayed parameter
	<code>Shaped_pulse</code>	Create offset array from position and gradient-amplitude array
	<code>offsetlist</code>	Create offset array from position and gradient amplitude
	<code>poffset</code>	Return frequency offset based on position
	<code>poffset_list</code>	Create offset array from position array
	<code>position_offset</code>	Return frequency offset based on position
	<code>position_offset_list</code>	Create offset array from position array
	<code>shapelist</code>	Create pulse-shape list from base pattern and offset array
	<code>shapedpulselist</code>	Perform shaped pulses from shape list on observe channel
	<code>shapedpulseoffset</code>	Perform shaped pulse on observe channel with offset

### `shaped_pulse`

#### Perform shaped pulse on observe channel

Syntax: `shaped_pulse(pattern,width,phase,RG1,RG2)`  
`char *pattern; /* name of .RF text file */`  
`double width; /* duration of pulse, seconds */`  
`codeint phase; /* real-time variable for phase */`  
`double RG1; /* duration of predelay, seconds */`  
`double RG2; /* duration of postdelay, seconds */`

Description: Set the quadrature phase, gate the observe channel on and off at the current power level with amplifier blanking and unblinking, and apply a shaped-pulse pattern. `shaped_pulse` is preceded by a predelay and followed by a postdelay. The associated amplifier is unblanked if it is in pulse mode and the phase is set, both at the beginning of the predelay. The associated amplifier is blanked at the end of the postdelay if it is in pulsed mode.

**Arguments:** `pattern` is the root name of a text file with a `.RF` extension in `shapelib`, containing the shape name.  
`width` is the duration of the pulse, in seconds.  
`phase` is a  $90^\circ$  phase multiplier of the observe channel. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*) or a real-time table (`t1` to `t60`).  
`RG1` is the duration of the predelay, in seconds.  
`RG2` is the duration of the postdelay, in seconds.

**Examples:** `shaped_pulse("gauss",pw,v1,rof1,rof2);`

<b>Related:</b>	<code>decshaped_pulse</code>	Perform shaped pulse on first decoupler
	<code>dec2shaped_pulse</code>	Perform shaped pulse on second decoupler
	<code>dec3shaped_pulse</code>	Perform shaped pulse on third decoupler
	<code>dec4shaped_pulse</code>	Perform shaped pulse on fourth decoupler
	<code>rgpulse</code>	Perform pulse on observe channel

**shapedgradient****Perform shaped gradient pulse on any one axis**

**Syntax:** `shapedgradient(pattern,width,daclvl,axis,loops,wait)`

```
char *pattern;          /* name of .GRD file */
double width;          /* duration of the gradient
shape, seconds */
double daclvl;         /* gradient amplitude of
pulse, in DAC units */
char axis;              /* gradient channel 'x',
'y', or 'z' */
int loops;             /* number of loops */
int wait;              /* WAIT or NOWAIT */
```

**Description:** Apply a shaped gradient pulse of duration `width` and gradient amplitude `daclvl` to the selected gradient channel `axis`, using a shape, determined by the argument `pattern` in `shapelib`. The shape is repeated `loops` times. If the value of `wait` is `NOWAIT`, the next pulse sequence statement is executed immediately after the beginning of the shape to allow for execution of shaped pulses while the gradient is on. If the value of `wait` is `WAIT`, the next statement executes at the conclusion of the gradient pulse.

**Arguments:** `pattern` is the root name of a `.GRD` file in `shapelib` containing the gradient shape name.  
`width` is the requested duration of the gradient shape, in seconds. The minimum duration is the

number of elements times 10  $\mu$ s. The duration of every element must be a multiple of 50 ns. The shaped gradient software rounds each element to a multiple of 50 ns. If the requested width is disallowed or differs from the actual width by more than 2%, a warning message is displayed.

`daclvl` sets amplitude of the gradient pulse in whole-number DAC units from -32767 to 32767. A value of 0 produces no gradient.

`axis` selects the desired gradient channel. It can be 'x', 'X', 'y', 'Y', or 'z', 'Z'.

`loops` is a value from 0 to 255 that repeats the gradient shape.

`wait` is a keyword, either WAIT or NOWAIT, that determines the time to the next statement. For a value of WAIT the time is `width*loops`. For a value of NOWAIT the time is 0.

Examples: `shapedgradient("hsine",0.02,32767,'y',1,NOWAIT);`

Related:	<code>rgradient</code>	Set DAC level of any one gradient axis
	<code>shapedgradient</code>	Perform shaped gradient pulse on any one axis
	<code>zgradpulse</code>	Perform gradient pulse on z axis

## **simpulse**

### **Perform simultaneous pulses, observe and first decoupler**

Syntax: `simpulse(obswidth,decwidth,obsphase,decphase,RG1,RG2)`  
`)`  
`double obswidth,decwidth; /* pulse durations, seconds */`  
`int obsphase,decphase; /* real-time quadrature-phase multipliers */`  
`double RG1; /* duration of predelay, seconds */`  
`double RG2; /* duration of postdelay, seconds */`

Description: Set the quadrature phase, the gates and blanking of the observe channel and first decoupler to produce simultaneous pulses on these two channels with a predelay and postdelay. The shorter of the two pulses is centered on the longer pulse. The amplifiers of both channels are unblanked and the quadrature phase is set at the beginning of the predelay. The amplifiers of both channels are blanked at the end of the postdelay if they are in pulsed mode.

If they are not identical, the absolute difference in the two pulse widths must be greater than or equal to 0.1  $\mu$ s to avoid delays less than the minimum of

0.05  $\mu$ s. The longer pulse width will be truncated to that of the shorter pulse width. If one of the pulse widths is 0.0, `simpulse` will execute no events and it will not interrupt waveform decoupling that may be in progress on that channel. The `simpulse` statement will not be executed if both pulse widths are zero.

**Arguments:** `obswidth` and `decwidth` are the duration, in seconds, of the pulses on the observe channel and first decoupler, respectively.

`obsphase` and `decphase` are 90° phase multipliers of the observe channel and the first decoupler, respectively. The value must be a real-time variable (`v1` to `v42`, `oph` *etc*), a real-time constant (`zero`, `one`, *etc*), or a real-time table (`t1` to `t60`).

`RG1` is the duration of the predelay in seconds.

`RG2` is the duration of the postdelay in seconds.

**Examples:** `simpulse(pw,pp,v1,v2,0.0,rof2);`

**Related:**

<code>decrgpulse</code>	Perform pulse on first decoupler
<code>rgpulse</code>	Perform pulse on observe channel
<code>sim3pulse</code>	Perform simultaneous pulses, observe, first, and second decouplers
<code>sim4pulse</code>	Perform simultaneous pulses, observe, first, second, and third decouplers

### **sim3pulse**

#### **Perform simultaneous pulses, observe, first and second decouplers**

**Syntax:**

```
sim3pulse(obswidth,decwidth,dec2width,obsphase,decphase,dec2phase,
RG1,RG2)
double obswidth,decwidth,dec2width;
/* pulse durations, seconds */
int obsphase,decphase,dec2phase;
/* real-time quadrature-phase multipliers */
double RG1; /* duration of predelay, seconds */
double RG2; /* duration of postdelay, seconds */
```

**Description:** Set the quadrature phase, the gates and blanking of the observe channel, the first and the second decouplers to produce simultaneous pulses on these three channels with a predelay and postdelay. The pulses are all centered on the same time-point. The amplifiers of all channels are unblanked and the quadrature phase is set at the beginning of the predelay. The amplifiers of all channels are blanked at the end of the postdelay if they are in pulsed mode.

If they are not identical, the absolute difference in any two pulse widths must be greater than or equal to 0.1  $\mu\text{s}$  to avoid delays less than the minimum of 0.05  $\mu\text{s}$ . The longer pulse width will be truncated to that of the shorter pulse width. If any of the pulse widths is 0.0, `sim3pulse` will execute no events and it will not interrupt waveform decoupling that may be in progress on that channel. The `sim3pulse` statement will not be executed if all pulse widths are zero.

**Arguments:** `obswidth`, `decwidth`, and `dec2width` are the durations, in seconds, of the pulses on the observe channel, first and second decouplers, respectively.

`obsphase`, `decphase`, and `dec2phase` are 90° phase multipliers of the observe channel, the first and second decouplers, respectively. The values must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*) or a real-time table (`t1` to `t60`).

`RG1` is the duration of the predelay in seconds.

**Examples:** `sim3pulse(pw,p1,p2,oph,v10,v1,rof1,rof2);`  
`sim3pulse(pw,0.0,p2,oph,zero,v1,rof1,rof2);`

**Related:**

<code>decrgpulse</code>	Perform pulse on first decoupler
<code>rgpulse</code>	Perform pulse on observe channel
<code>simpulse</code>	Perform simultaneous pulses, observe and first decoupler
<code>sim4pulse</code>	Perform simultaneous pulses, observe, first, second, and third decouplers

### `sim4pulse`

#### Perform simultaneous pulses, observe, first, second and third decouplers

**Syntax:**

```
sim4pulse(obswidth,decwidth,dec2width,dec3width,obsphase,decphase,dec2phase,dec3phase,RG1,RG2)
double obswidth,decwidth,dec2width,dec3width;
/* pulse durations, seconds */
int obsphase,decphase,dec2phase,dec3phase;
/* real-time quadrature-phase multipliers */
double RG1; /* duration of predelay, seconds */
double RG2; /* duration of postdelay, seconds */
```

**Description:** Set the quadrature phase, the gates and blanking of the observe channel, the first, second and third decouplers to produce simultaneous pulses on these four channels, with a predelay and postdelay. The pulses are all centered on the same time-point. The amplifiers of all channels are unblanked and the quadrature phase is set at the beginning of the predelay. The amplifiers of all channels are blanked

at the end of the postdelay if they are in pulsed mode.

If they are not identical, the absolute difference in any two pulse widths must be greater than or equal to 0.1  $\mu$ s to avoid delays less than the minimum of 0.05  $\mu$ s. The longer pulse width will be truncated to that of the shorter pulse width. If any of the pulse widths is 0.0, `sim4pulse` will execute no events and it will not interrupt waveform decoupling that may be in progress on that channel. The `sim4pulse` statement will not be executed if all pulse widths are zero.

**Arguments:** `obswidth`, `decwidth`, `dec2width`, and `dec3width` are the durations, in seconds, of the pulses on the observe channel, first, second, and third decouplers, respectively.

`obsphase`, `decphase`, `dec2phase`, and `dec3phase` are 90° phase multipliers of the observe channel, the first, second, and third decouplers, respectively. The values must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*) or a real-time table (`t1` to `t60`).

`RG1` is the duration of the predelay in seconds.

`RG2` is the duration of the postdelay in seconds.

**Examples:** `sim4pulse(pw,p1,p2,p3,oph,v10,v1,v2,rof1,rof2);`  
`sim4pulse(pw,0.0,p2,p3,oph,zero,v1,v2,rof1,rof2);`

**Related:**

<code>decrgpulse</code>	Perform pulse on first decoupler
<code>rgpulse</code>	Perform pulse on observe channel
<code>simpulse</code>	Perform simultaneous pulses, observe and first decoupler
<code>sim3pulse</code>	Perform simultaneous pulses, observe, first, and second decouplers

### **simshaped\_pulse**

### **Perform simultaneous shaped pulses, observe and first decoupler**

**Syntax:** `simshaped_pulse(obspattern,decpattern,obswidth,decwidth,obsphase,decphase,RG1,RG2)`  
`char *obspattern,*decpattern;`  
`/* names of .RF text files */`  
`double obswidth, decwidth;`  
`/* pulse durations, seconds */`  
`codeint phase; /* real-time`  
`quadrature phase multiplier */`  
`double RG1; /* duration of`  
`predelay, seconds */`  
`double RG2; /* duration of`  
`postdelay, seconds */`

**Description:** Set the quadrature phase, gate the observe channel and first decoupler on and off at the current power level with amplifier blanking and unblanking ,and apply shaped-pulse patterns to both channels. `shaped_pulse` is preceded by a predelay and followed by a postdelay. The associated amplifier is unblanked if it is in pulse mode and the phase is set, both at the beginning of the predelay. The associated amplifier is blanked at the end of the postdelay if it is in pulsed mode.

If they are not identical, the absolute difference in the two pulse widths must be greater than or equal to 0.1  $\mu$ s to avoid delays less than the minimum of 0.05  $\mu$ s. The longer pulse width will be truncated to that of the shorter pulse width. If any of the pulse widths are 0.0, `simshaped_pulse` will execute no events and it will not interrupt waveform decoupling that may be in progress on that channel. The `simshaped_pulse` statement will not be executed if all pulse widths are zero.

If a pattern is set to '', `simshaped_pulse` will execute a square pulse on that channel.

**Arguments:** `obspattern` and `decpattern` are the root names of text files with a `.RF` extension in `shapelib`, containing the shape names for the observe channel and first decoupler, respectively.

**Arguments:** `obswidth` and `decwidth` are the durations of the pulses, in seconds, for the observe channel and first decoupler, respectively.

`obsphase` and `decphase` are a 90° phase multipliers of the observe and first decoupler channels, respectively. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*) or a real-time table (`t1` to `t60`).

`RG1` is the duration of the predelay, in seconds.

`RG2` is the duration of the postdelay, in seconds.

**Examples:** `simshaped_pulse("gauss", "hrm180", pw, p1, v2, v5, rof1, rof2);`

<b>Related:</b>	<code>decrgpulse</code>	Perform pulse on first decoupler
	<code>decshaped_pulse</code>	Perform shaped pulse on first decoupler
	<code>rgpulse</code>	Perform pulse on observe channel
	<code>shaped_pulse</code>	Perform shaped pulse on observe channel
	<code>sim3shaped_pulse</code>	Perform simultaneous shaped pulses, observe, first, and second decouplers



sim4shaped_pulse	Perform simultaneous shaped pulses, observe, first, second, and third decouplers
simpulse	Perform simultaneous pulses, observe and first decoupler
sim3pulse	Perform simultaneous pulses, observe, first, and second decouplers
sim4pulse	Perform simultaneous pulses, observe, first, second, and third decouplers

**sim3shaped\_pulse****Perform simultaneous shaped pulses observe, first and second decouplers**

**Syntax:** `sim4shaped_pulse(obspattern,decpattern,dec2pattern,obswidth,decwidth,dec2width,obsphase,decphase,dec2phase,RG1,RG2)`  
`char *obspattern,*decpattern,*dec2pattern;`  
`/* names of .RF text files */`  
`double obswidth,decwidth,dec2width;`  
`/* pulse durations, seconds */`  
`int obsphase,decphase,dec2phase;`  
`/* real-time quadrature-phase multipliers */`  
`double RG1; /* duration of`  
`predelay, seconds */`  
`double RG2; /* duration of`  
`postdealy, seconds */`

**Description:** Set the quadrature phase, gate the observe channel first and second decouplers on and off at the current power level with amplifier blanking and unblinking, and apply shaped-pulse patterns to all channels. `sim3shaped_pulse` is preceded by a predelay and followed by a postdelay. The associated amplifiers are unblanked if they are in pulse mode and the phase is set, at the beginning of the predelay. The associated amplifiers are blanked at the end of the postdelay if they are in pulsed mode.

If they are not identical, the absolute difference in any two pulse widths must be greater than or equal to 0.1  $\mu$ s to avoid delays less than the minimum of 0.05  $\mu$ s. The longer pulse width will be truncated to that of the shorter pulse width. If any of the pulse widths is 0.0, `sim3shaped_pulse` will execute no events and it will not interrupt waveform decoupling that may be in progress on that channel. The `sim3shaped_pulse` statement will not be executed if all pulse widths are zero.

If a pattern is set to '', `sim3shaped_pulse` will execute a square pulse on that channel.

**Arguments:** `obspattern`, `decpattern`, and `dec2pattern` are the root names of text files with a .RF extension in

shapelib, containing the shape names for the observe channel first and second decouplers, respectively.

obswidth, decwidth, and dec2width are the durations of the pulses, in seconds, for the observe channel first and second decouplers, respectively.

obsphase, decphase, and dec2phase are a 90° phase multipliers of the observe and first decoupler channels, respectively. The value must be a real-time variable (v1 to v42, oph, etc), a real-time constant (zero, one, etc) or a real-time table (t1 to t60).

RG1 is the duration of the predelay, in seconds.

RG2 is the duration of the postdelay, in seconds.

**Examples:**

```
sim3shaped_pulse("gauss", "hrm180", "sinc", pw, p1, p2,
v2, v5, v6, rof1, rof2);
sim3shaped_pulse("dummy", "hrm180", "sinc", 0.0,
p1, p2, v2, v5, v6, rof1, rof2);
```

<b>Related:</b>	decrpulse	Perform pulse on first decoupler
	decshaped_pulse	Perform shaped pulse on first decoupler
	rgpulse	Perform pulse on observe channel
	shaped_pulse	Perform shaped pulse on observe channel
	simshaped_pulse	Perform simultaneous shaped pulses, observe and first decoupler
	sim4shaped_pulse	Perform simultaneous shaped pulses, observe, first, second, and third decouplers
	simpulse	Perform simultaneous pulses, observe and first decoupler
	sim3pulse	Perform simultaneous pulses, observe, first, and second decouplers
	sim4pulse	Perform simultaneous pulses, observe, first, second, and third decouplers

### sim4shaped\_pulse

### Perform simultaneous shaped pulses, observe, first, second and third decouplers

**Syntax:**

```
sim4shaped_pulse(obspattern, decpattern, dec2pattern, d
ec3pattern, obswidth, decwidth, dec2width, dec3w
idth, obsphase, decphase, dec2phase, dec3phase, R
G1, RG2)
char
*obspattern, *decpattern, *dec2pattern, *dec3pa
ttern;
/* names of .RF text files */
```

```
double obswidth, decwidth, dec2width, dec3width;
/* pulse durations, seconds */
int obsphase, decphase, dec2phase, dec3phase;
/* real-time quadrature-phase multipliers */
double RG1;          /* duration of
predelay, seconds */
double RG2;          /* duration of
postdelay, seconds */
```

**Description:** Set the quadrature phase, gate the observe channel first, second and third decouplers on and off at the current power level with amplifier blanking and unblinking, and apply shaped-pulse patterns to all channels. `sim4shaped_pulse` is preceded by a predelay and followed by a postdelay. The associated amplifiers are unblanked if they are in pulse mode and the phase is set, at the beginning of the predelay. The associated amplifiers are blanked at the end of the postdelay if they are in pulsed mode.

If they are not identical, the absolute difference in any two pulse widths must be greater than or equal to 0.1  $\mu$ s to avoid delays less than the minimum of 0.05  $\mu$ s. The longer pulse width will be truncated to that of the shorter pulse width. If any of the pulse widths is 0.0, `sim4shaped_pulse` will execute no events and it will not interrupt waveform decoupling that may be in progress on that channel. The `sim3shaped_pulse` statement will not be executed if all pulse widths are zero.

If a pattern is set to '', `sim4shaped_pulse` will execute a square pulse on that channel.

**Arguments:** `obspattern`, `decpattern`, `dec2pattern`, and `dec3pattern` are the root names of text files with a `.RF` extension in `shapelib`, containing the shape names for the observe channel, first, second and third decouplers, respectively.

`obswidth`, `decwidth`, `dec2width`, and `dec3width` are the durations of the pulses, in seconds, for the observe channel, first, second and third decouplers, respectively.

`obsphase`, `decphase`, `dec2phase`, and `dec3phase` are a 90° phase multipliers of the observe channel, first, second, and third decoupler channels, respectively. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*) or a real-time table (`t1` to `t60`).

`RG1` is the duration of the predelay in seconds.

`RG2` is the duration of the postdelay in seconds.

**Examples:** `sim4shaped_pulse("gauss", "hrm180", "sinc", "sinc", pw, p1, p2, p3, v2, v5, v6, v7, rof1, rof2);`  
`sim4shaped_pulse("dummy", "hrm180", "sinc", "sinc", 0.0, p1, p2, p3, v2, v5, v6, v7, rof1, rof2);`

<b>Related:</b>	<code>decrgpulse</code>	Perform pulse on first decoupler
	<code>decshaped_pulse</code>	Perform shaped pulse on first decoupler
	<code>rgpulse</code>	Perform pulse on observe channel
	<code>shaped_pulse</code>	Perform shaped pulse on observe channel
	<code>simshaped_pulse</code>	Perform simultaneous shaped pulses, observe and first decoupler
	<code>sim3shaped_pulse</code>	Perform simultaneous shaped pulses, observe, first, and second decouplers
	<code>simpulse</code>	Perform simultaneous pulses, observe and first decoupler
	<code>sim3pulse</code>	Perform simultaneous pulses, observe, first, and second decouplers
	<code>sim4pulse</code>	Perform simultaneous pulses, observe, first, second, and third decouplers

#### **sp#off**

#### **Turn off spare line (# = 1, 2, or 3)**

**Syntax:** `sp1off()`, `sp2off()`, `sp3off()`

**Description:** Turn off the specified user spare line connector for high-speed device control.

**Examples:** `sp1off();`  
`sp3off();`

**Related:** `sp#off` Turn off spare line (# = 1,2 or 3)  
`sp#on` Turn on spare line (# = 1,2 or 3)

#### **sp#on**

#### **Turn on spare line (# = 1, 2, or 3)**

**Syntax:** `sp1on()`, `sp2on()`, `sp3on()`

**Description:** Turn on the specified user-dedicated spare line connector for high-speed device control.

**Examples:** `sp1on();`  
`sp3on();`

**Related:** `sp#off` Turn off spare line (# = 1,2 or 3)  
`sp#on` Turn on spare line (# = 1,2 or 3)

**spinlock****Perform waveform spinlock on observe channel**

Syntax: `spinlock(pattern,90_pulselength,tipangle_resoln,phase,ncycles)`  
`char *pattern; /* name of .DEC text file */`  
`double 90_pulselength; /* 90-degree pulse length in seconds */`  
`double tipangle_resoln; /* tip-angle resolution */`  
`int phase; /* real-time quadrature-phase multiplier */`  
`int ncycles; /* number of cycles */`

Description: Execute a spinlock with an integer number of cycles of programmable decoupling on the observe channel under waveform control. `spinlock` unblanks and blanks the associated amplifier and gates the observe channel on and off for patterns without an explicit gate column. It is good practice to unblank the associated amplifier with `obsunblank`, at least 2.0  $\mu$ s before `spinlock`.

Arguments: `pattern` is the root name of a .DEC file in `shapelib` containing the waveform shape name.

`90_pulselength` is the pulse time-duration, in seconds, for an element, a block of elements, or an integer divisor of all elements in the pattern, which is labeled with a tip-angle duration equal to 90°. Often, `90_pulselength` is set equal  $1/\text{dmf}$ , where `dmf` is a step rate.

`tipangle_resoln` is the smallest common divisor of the tip-angle durations of all the elements of the pattern.

For many patterns, `90_pulselength` is the actual 90° pulse length and the elements are labeled with tip-angle durations that are multiples of 90. In this case, `tipangle_resoln` is 90.0.

For many other patterns, `90_pulselength` is set to be the duration of the minimum stepsize of which all elements are multiples. In these cases, `tipangle_resoln` is nominally 90 and elements in the pattern have tip-angle durations that are multiples of 90.

For some patterns `90_pulselength` is the actual 90° pulse length, but elements of the pattern have arbitrary flip angles that are multiples of a `tipangle_resoln` that is less than 90° (*c.f.* 1.0°). In this case, `tipangle_resoln` is the divisor (*c.f.* 1.0) and for good practice it is also a divisor of 90.0.

`phase` is a 90° multiplier for the phase of the observe channel. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*) or a real-time table (`t1` to `t60`).

`ncycles` is the number of times that the spin-lock pattern is to be executed.

Examples: `spinlock("mlev16",pw90,90.0,v1,50);`  
`spinlock(locktype,pw,resol,v1,cycles);`

Related: `decspinlock` Perform waveform spinlock on second decoupler  
`dec2spinlock` Perform waveform spinlock on second decoupler  
`dec3spinlock` Perform waveform spinlock on third decoupler  
`dec4spinlock` Perform waveform spinlock on fourth decoupler

**startacq****Initialize explicit acquisition**

Syntax: `startacq(delay)`  
`double delay; /* time duration before the next event*/`

Description: Initialize the VNMRS digital receiver for acquisition, when explicit `sample` statements are used for acquisition. The statement `startacq` first executes a `rcvron` statement with the delay `rof3`. If `rof3` is not defined, a delay of 2.0  $\mu$ s is used.

The value of `delay` is usually set with the parameter `alfa` and typically `alfa = 4.0–6.0  $\mu$ s`. Sequences for solid-state NMR often use the parameter `ad` with the same value. `startacq` finishes with a duration of `delay - rof3`. If `delay < rof3` or 2.0  $\mu$ s, `startacq` has a minimum duration of `rof3` or 2.0  $\mu$ s.

The `startacq` statement holds the acquisition of the first point until the first `sample` statement is executed. For a usual, continuous acquisition, `startacq` is immediately followed by `sample(np/(2.0*sw); endacq();`.

For *implicit acquisition*, `startacq` is executed automatically with the `alfa` delay and it should not be included.

For a windowed, *explicit acquisition*, `startacq` is followed by a loop containing `sample` statements and then `endacq()`. If `startacq` is used explicitly before the loop that contains `sample` statements, *continuous sampling*, then zeros will be inserted between `sample` statements before downsampling. If `startacq` is not included before a windowed explicit acquisition loop, the `startacq` statement will be inserted automatically with a delay of `alfa`. In this case, *explicit sampling*, data will be obtained only during the `sample` statements and the blocks of data will be concatenated.

Examples: `startacq(alfa);`  
`startacq(ad);`

Related:	acquire	Acquire data explicitly
	rcvroff	Turn on receiver and blank observe amplifier
	rcvron	Turn off receiver and unblank observe amplifier
	sample	Acquire data explicitly during time delay
	startacq	Initialize explicit acquisition

**startacq\_obs****Initialize explicit acquisition in parallel section**

Syntax: `startacq_obs(delay)`

```
double delay; /* time duration before
the next event */
```

Description: Analogous to the `startacq` statement but to be used in "obs" parallel sections of a pulse sequence.

Related:	acquire_obs	Acquire data explicitly in parallel section
	acquire_rcvr	Acquire data explicitly in parallel section
	startacq_rcvr	Initialize explicit acquisition in parallel section
	endacq_obs	End explicit acquisition in parallel section
	endacq_rcvr	End explicit acquisition in parallel section
	parallelacquire_obs	Acquire data explicitly in parallel section
	parallelacquire_rcvr	Acquire data explicitly in parallel section

**startacq\_rcvr****Initialize explicit acquisition in parallel section**

Syntax: `startacq_rcvr(delay)`

```
double delay; /* time duration before
the next event */
```

Description: Analogous to the `startacq` statement but to be used in "rcvr" parallel sections of a pulse sequence.

Related:	acquire_obs	Acquire data explicitly in parallel section
	acquire_rcvr	Acquire data explicitly in parallel section
	startacq_obs	Initialize explicit acquisition in parallel section

<code>endacq_obs</code>	End explicit acquisition in parallel section
<code>endacq_rcvr</code>	End explicit acquisition in parallel section
<code>parallelacquire_obs</code>	Acquire data explicitly in parallel section
<code>parallelacquire_rcvr</code>	Acquire data explicitly in parallel section

**starthardloop**

**Start hardware loop (obsolete)**

Syntax: `status(count)`  
`int count; /* real-time variable with loop count */`

Description: Start a real-time loop with `count` repetitions. This statement is identical to the `loop` statement, except that the loop index is not accessible. The `starthardloop` statement is obsolete. Use the `loop` - `endloop` statements instead.

Arguments: `count` is the number of loop repetitions. It must be a real-time variable (`v1` to `v42`) or an index (`id2`, `id3`, *etc*).

Related: `endhardloop` End hardware loop  
`endloop` End real-time loop  
`loop` Start real-time loop

**status**

**Set status of decoupler and homospoil**

Syntax: `status(state)`  
`int state; /* index: A, B, C, ..., Z */`

Description: Set decoupler and homospoil gating based on parameters. The *global* variables and parameters that control status are strings: `dm` (first decoupler mode), `dmm` (first decoupler modulation mode), `dm2` (second decoupler mode), `dmm2` (second decoupler modulation mode), `dm3` (third decoupler mode), `dmm3` (third decoupler modulation mode), `dm4` (fourth decoupler mode), `dmm4` (fourth decoupler modulation mode), `xm` (observe-channel mode), and `xmm` (observe-channel modulation mode).

The character values of the parameters for decoupler mode are 's', 'y', and 'n' and they determine whether the transmitter is on-synchronously, on-asynchronously, or off. The characters of the decoupler modulation-mode parameters control waveform decoupling. See the *Command and Parameter* reference for the list of choices.



The functions executed by the status statement are determined by the individual character values of the status parameters, where character 0 controls when `state = A`, character 1 controls when `state = B`, and so forth. If a pulse sequence has more status statements than there are characters in the value of the particular parameter, control reverts to the last character. Thus if `dm = 'ny'`, `status(C)` will look for the third letter, find none, use the second letter 'y', and leave the decoupler on.

The individual status characters can be accessed in a sequence by designating an array index starting at 0. For example, `dm[0]` is the value that controls the decoupler mode during `status(A)`.

The states do not have to be arranged monotonically in the pulse sequence. You can write a pulse sequence that starts with `status(A)`, goes later to `status(B)`, then back to `status(A)`, then to `status(C)`, *etc.* It is a usual convention that `status(A)` controls the `d1` delay and the last character controls acquisition. Often, this character corresponds to `status(C)`, (character 2), whether or not `status(B)` is present in the sequence.

The characters of the homospoil parameter `hs` control the statement `hsdelay`. An `hsdelay` statement will execute with a homospoil pulse if the corresponding status character in `hs` is 'y'. If the character is 'n' `hsdelay` will execute without a homospoil pulse. Thus if `hs = 'ny'`, all `hsdelay` statements that occur during `status(B)` will begin with a homospoil pulse. The `hs` parameter will often have one less character than the other status parameters if the last status period is used only for acquisition. An `hsdelay` statement will never be found during an acquisition and thus a character is not needed.

Arguments: `state` sets the status mode to A, B, C, ...Z. The status modes are controlled by the respective characters of the *global* status variables and their respective parameters, for example, `dm` and `dmm`.

Examples: `status(A) ;`

Related:	<code>hsdelay</code>	Execute time delay with optional homospoil pulse
	<code>setstatus</code>	Set decoupler status of any channel
	<code>statusdelay</code>	Execute status statement within time delay

## **statusdelay**

### **Execute status statement within time delay**

Syntax: `statusdelay(state,time)`  
`int state; /* A, B, C, ... Z */`

```
double time; /* duration of the statement,
seconds */
```

Description: Executes the status statement within a delay specified by the argument *time*.

Arguments: *state* specifies the status mode with the values A,B,C,...Z. The status modes are controlled by the respective characters of the *global* status variables and their respective parameters, for example *dm* and *dmm*.

Examples: *time* is the duration of the *statusdelay* statement.  
`statusdelay(A,d1);`  
`statusdelay(B,0.000010);`

Related:	<code>hsdelay</code>	Execute time delay with optional homospoil pulse
	<code>setstatus</code>	Set decoupler status of any channel
	<code>status</code>	Set status of decoupler and homospoil

**stepsize**

**Set small-angle phase stepsize**

Syntax: `stepsize(step_size,channel)`  

```
double step_size; /* small-angle phase
stepsize */
int channel; /* OBSch, DECch,
DEC2ch, DEC3ch, DEC4ch*/
```

Description: Set the small-angle phase stepsize of any channel.

Arguments: *step\_size* is a value in degrees that sets the phase increment that corresponds to one unit of a real-time phase multiplier.

*channel* is a *global* integer constant that designates the observe channel *OBSch*, first decoupler *DECch*, second decoupler *DEC2ch*, third decoupler *DEC3ch*, and fourth decoupler *DEC4ch*.

Examples: `stepsize(30.0,OBSch);`

Related:	<code>decstepsize</code>	Set small-angle phase stepsize of first decoupler
	<code>dec2stepsize</code>	Set small-angle phase stepsize of second decoupler
	<code>dec3stepsize</code>	Set small-angle phase stepsize of third decoupler
	<code>dec4stepsize</code>	Set small-angle phase stepsize of fourth decoupler
	<code>obsstepsize</code>	Set small-angle phase stepsize of observe channel

stepsize	Set small-angle phase stepsize of any channel
xmtrphase	Set small-angle phase of observe channel

**sub****Subtract real-time integer values**

Syntax: `sub(vi,vj,vk)`  
`codeint vi; /* real-time variable`  
`for minuend */`  
`codeint vj; /* real-time variable`  
`for subtrahend */`  
`codeint vk; /* real-time variable`  
`for difference */`

Description: Set the value of `vk` equal to the difference of integer values of `vi` and `vj`.

Arguments: `vi` contains the value of the minuend, `vj` contains the value of the subtrahend, and `vk` contains the resulting difference. Each argument must be a real-time variable (`v1` to `v42`, `oph`, *etc*).

Examples: `sub(v2,v5,v6);`

Related:	<code>add</code>	Add real-time integer values
	<code>assign</code>	Assign real-time integer value using real-time integer
	<code>dbl</code>	Double real-time integer value
	<code>decr</code>	Decrement real-time integer value
	<code>divn</code>	Divide real-time integer values
	<code>hlv</code>	Assign half the value of real-time integer
	<code>incr</code>	Increment real-time integer value
	<code>initval</code>	Assign real-time integer value using numeric value
	<code>mod2</code>	Assign real-time integer value modulo 2
	<code>mod4</code>	Assign real-time integer value modulo 4
	<code>modn</code>	Assign real-time integer value modulo n
	<code>mult</code>	Multiply real-time integer values

**swift\_acquire****Execute SWIFT rf pulses and gated acquire pulse train**

Syntax: `Syntax:swift_acquire(swiftshape, pw, predelay)`  
`char swiftshape[]; /* SWIFT shape`  
`filename */`  
`double pw; /* rf pulse width*/`  
`double predelay; /* predelay */`

Description: The `swift_acquire` statement executes the SWIFT rf pulses and gated acquire pulse train [JMR vol 181 (2006) pp342-349]. The standard or implicit acquire

is replaced by the `swift_acquire`. The SWIFT rf pulses and gated acquire train executes for the duration of acquisition time, `at`. The attenuator power level for the rf pulse train should be set before the `swift_acquire` statement using an `obspower` statement. The `swift_acquire` statement can be executed in parallel with an oblique shaped gradient in a NOWAIT mode. It can also be included inside a real-time loop statement, such as `loop(v1,v2)`. This is necessary if the SWIFT pulse train is to be executed while changing frequency of the rf and carrier in real-time. It can be accomplished using a `create_offset_list` followed by `voffsetch` statements inside the real-time loop. As SWIFT experiment tends to acquire large amounts of data in very short periods of time, appropriate data transfer and buffering have to be setup on the DDR controller using the `nfmod` parameter. A value of 256 for `nfmod` may be appropriate under many conditions of `nf`, `loop` counts, `at`, `TR`, `np`, and so on. The parameter `dp` can be set to 'n' to minimize the amount of data to be transferred to the host. The factor `pw*swiftmodfreq` gives the duty cycle of rf pulse train. In the case, where `swiftmodfreq` is not defined, it defaults to `pw*sw`.

**Arguments:** `swiftshape` is the name of the rf shape file to be executed.

`pw` is the length of the rf pulse between the acquire events. `pw` is essentially "tau-p" in the above mentioned reference.

`predelay` is the delay after the rf pulse and before the acquire, similar to `alfa` parameter (not currently implemented).

Other parameters that are used by `swift_acquire`, though not presented as arguments, are:

- `swiftrof1`, `swiftrof2`, `swiftrof2`- optional pre- and post-delays around individual `pw` long rf events, with corresponding functionality as `rof1`, `rof2`, and `rof3` respectively.

- `swiftmodfreq` – the modulation rate for the rf gating scheme. It can be independent of `sw` parameter. However, if `swiftmodfreq` is not defined, the modulation rate for rf gating defaults to `sw`.

**Examples:** A simple SWIFT test sequence is included below:

```
/* swift_test */

#include <standard.h>
```

```

#define LISTSIZE 524288

void pulsesequence()
{
    char swiftshape[MAXSTR];
    double numloops = getval("numloops");
    double offsetlist[LISTSIZE];
    double deltaf = -1000.0;

    int i;
    if (numloops > LISTSIZE)
        abort_message("numloops is bigger
than LISTSIZE! abort!\n");

    for(i=0; i<numloops; i++)
        offsetlist[i] = -50000.0 + (del-
taf*i);

    getstr("swiftshape",swiftshape);

    create_offset_list( &offsetlist[0],
numloops, OBSch, 1);

    /* equilibrium period */
    status(A);
    obspower(tpwr);
    delay(d1);

    /* --- tau delay --- */
    status(B);
    delay(d2);
    rgpulse(pw, oph, rof1, rof2);
    initval(numloops, v10);
    loop(v10, v11);
        voffsetch(1, v11, OBSch);
        swift_acquire(swiftshape, pw, 0.0);
        delay(d2);
    endloop(v11);
}

```

### 3 Pulse Sequence Statement Reference

Related: `create_offset_` Create table of frequency offsets  
`list`  
`voffsetch`  
`loop` Start real-time loop

## T

text_error	Send error message to VnmrJ
text_message	Send message to VnmrJ
triggerSelect	Select trigger input on MRI User Panel
tsadd	Add integer to table elements
tsdiv	Divide integer into table elements
tsmult	Multiply integer with table elements
tssub	Subtract integer from table elements
ttadd	Add table to second table
ttdiv	Divide table into second table
ttmult	Multiply table by second table
ttsub	Subtract table from second table
txphase	Set quadrature phase of observe channel

**text\_error****Send error message to VnmrJ**

**Syntax:** text\_error(message, varnames)  
char \*message /\* formatted string  
containing the message \*/  
varnames /\* char, int or double,  
used in message \*/

**Description:** Send a formatted error message to VnmrJ and write the message into the file userdir+'/psg.error'. The formatting is similar to the C printf statement.

**Related:**

abort_message	Abort PSG at run-time and send message to VnmrJ
psg_abort	Abort PSG Process
text_message	Send message to VnmrJ
warn_message	Send warning message to VnmrJ

**text\_message****Send message to VnmrJ**

**Syntax:** text\_message(message, varnames)  
char \*message /\* formatted string  
containing the message \*/  
varnames /\* char, int or double, used  
in message \*/

**Description:** Send a formatted warning message to VnmrJ. The formatting is similar to the C printf statement.

**Related:**

abort_message	Abort PSG at run-time and send message to VnmrJ
psg_abort	Abort PSG Process
text_error	Send error message to VnmrJ
warn_message	Send warning message to VnmrJ

#### **triggerSelect** Select trigger input on MRI User Panel

**Syntax:** `triggerSelect(a)`  
`int a /* A, B or C*/`

**Description:** Select one of the three input BNCs (INPUT1, INPUT2, or INPUT3) on the MRI GUser panel on the back of the console and connects it to the OUTPUT BNC. The internal input is a 25 kHz square wave.

**Arguments:** a is an integer argument that determines which input is selected:  
A value A Selects INPUT1  
A value B Selects INPUT2  
A value C Selects INPUT3

**Examples:** `triggerSelect(B);`

#### **tsadd**

#### **Add integer to table elements**

**Syntax:** `tsadd(table, scalarval, moduloval)`  
`codeint table; /* real-time table to be modified */`  
`int scalarval; /* integer to be added */`  
`int moduloval; /* modulo value of result */`

**Description:** A real-time operation that adds an integer to all elements of a table.

**Arguments:** table specifies the name of the table (t1 to t60).  
scalarval is an integer to be added to each element of the table.  
moduloval is the modulo value taken on the result of the operation, if moduloval is greater than 0.

**Examples:** `tsadd(t31, 4, 4);`

**Related:**

<code>loadtable</code>	Assign table elements from table text file
<code>settable</code>	Assign integer array to table
<code>tsdiv</code>	Divide an integer into table elements
<code>tsmult</code>	Multiply and integer with table elements
<code>tssub</code>	Subtract an integer from table elements
<code>ttadd</code>	Add table to second table
<code>ttdiv</code>	Divide table into second table
<code>ttmult</code>	Multiply table by second table
<code>ttsub</code>	Subtract table from second table

#### **tsdiv**

#### **Divide integer into table elements**

**Syntax:** `tsdiv(table, scalarval, moduloval)`  
`codeint table; /* real-time table to be`



```

modified */
int scalarval;      /* integer divisor */
int moduloval;     /* modulo value of result
*/

```

**Description:** A real-time operation that divides an integer into all elements of a table.

**Arguments:** `table` specifies the name of the table (t1 to t60). `scalarval` is an integer to be divided into each element of the table. `scalarval` must not equal 0; otherwise, an error is displayed and PSG aborts. `moduloval` is the modulo value taken on the result of the operation, if `moduloval` is greater than 0.

**Examples:** `tsdiv(t31,4,4);`

<b>Related:</b>	<code>loadtable</code>	Assign table elements from table text file
	<code>settable</code>	Assign integer array to table
	<code>tsadd</code>	Add integer to table elements
	<code>tsmult</code>	Multiply integer with table elements
	<code>tssub</code>	Subtract integer from table elements
	<code>ttadd</code>	Add table to second table
	<code>ttdiv</code>	Divide table into second table
	<code>ttmult</code>	Multiply table by second table
	<code>ttsub</code>	Subtract table from second table

## **tsmult**

### **Multiply integer with table elements**

**Syntax:** `tsmult(table,scalarval,moduloval)`

```

codeint table;      /* real-time table to
be modified */
int scalarval;     /* integer multiplier */
int moduloval;     /* modulo value of result
*/

```

**Description:** A real-time operation that multiplies all elements of a table by an integer.

**Arguments:** `table` is the name of the table (t1 to t60). `scalarval` is an integer to be multiplied by each element of the table. `moduloval` is the modulo value taken on the result of the operation, if `moduloval` is greater than 0.

**Examples:** `tsmult(t31,4,4);`

<b>Related:</b>	<code>loadtable</code>	Assign table elements from table text file
	<code>settable</code>	Assign integer array to table
	<code>tsadd</code>	Add integer to table elements
	<code>tsdiv</code>	Divide integer into table elements

tssub	Subtract integer from table elements
ttadd	Add table to second table
ttdiv	Divide table into second table
ttmult	Multiply table by second table
ttsub	Subtract table from second table

**tssub**

**Subtract integer from table elements**

**Syntax:** `tssub(table,scalarval,moduloval)`  
`codeint table; /* real-time table to be modified */`  
`int scalarval; /* integer to be subtracted */`  
`int moduloval; /* modulo value of result */`

**Description:** A real-time operation that subtracts an integer from all elements of a table.

**Arguments:** `table` is the name of the table (t1 to t60).  
`scalarval` is an integer to be subtracted from each element of the table.  
`moduloval` is the modulo value taken on the result of the operation, if `moduloval` is greater than 0.

**Examples:** `tssub(t31,4,4);`

**Related:**

<code>loadtable</code>	Assign table elements from table text file
<code>settable</code>	Assign integer array to table
<code>tsadd</code>	Add integer to table elements
<code>tsdiv</code>	Divide integer into table elements
<code>tsmult</code>	Multiply integer with table elements
<code>ttadd</code>	Add table to second table
<code>ttdiv</code>	Divide table into second table
<code>ttmult</code>	Multiply table by second table
<code>ttsub</code>	Subtract table from second table

**ttadd**

**Add table to second table**

**Syntax:** `ttadd(table_dest,table_mod,moduloval)`  
`codeint table_dest; /* real-time table to be modified */`  
`codeint table_mod; /* real-time table to be added */`  
`int moduloval; /* modulo value of result */`

**Description:** A real-time operation that adds `table_mod` to `table_dest` and returns a new `table_dest`.

**Arguments:** `table_dest` is the name of the destination table (t1 to t60).

`table_mod` is the name of the table (t1 to t60) that modifies `table_dest`. Each element in `table_dest` is modified by the corresponding element in `table_mod` and the result is stored in `table_dest`. The number of elements in `table_dest` must be greater than or equal to the number of elements in `table_mod`.

`moduloval` is the modulo value taken on the result of the operation, if `moduloval` is greater than 0.

Examples: `ttadd(t28,t42,6);`

Related:	<code>loadtable</code>	Assign table elements from table text file
	<code>settable</code>	Assign integer array to table
	<code>tsadd</code>	Add integer to table elements
	<code>tsdiv</code>	Divide integer into table elements
	<code>tsmult</code>	Multiply integer with table elements
	<code>tssub</code>	Subtract integer from table elements
	<code>ttdiv</code>	Divide table into second table
	<code>ttmult</code>	Multiply table by second table
	<code>ttsub</code>	Subtract table from second table

## **ttdiv**

### **Divide table into second table**

Syntax: `ttdiv(table_dest,table_mod,moduloval)`  
`codeint table_dest; /* real-time table to be modified */`  
`codeint table_mod; /* real-time table containing integer divisors */`  
`int moduloval; /* modulo value of result */`

Description: A real-time operation that provides an integer division of `table_dest` by `table_mod` and returns a new `table_dest`.

Arguments: `table_dest` is the name of the destination table (t1 to t60).

`table_mod` is the name of the table (t1 to t60) that modifies `table_dest`. Each element in `table_dest` is modified by the corresponding element in `table_mod`, and the result is stored in `table_dest`. The number of elements in `table_dest` must be greater than or equal to the number of elements in `table_mod`. No element in `table_mod` can be equal to 0.

`moduloval` is the modulo value taken on the result of the operation, if `moduloval` is greater than 0.

Examples: `ttdiv(t28,t42,6);`

Related: `loadtable` Assign table elements from table text file  
`settable` Assign integer array to table  
`tsadd` Add integer to table elements  
`tsdiv` Divide integer into table elements  
`tsmult` Mutiply integer with table elements  
`tssub` Subtract integer from table elements  
`ttadd` Add table to second table  
`ttdiv` Divide table into second table  
`ttsub` Subtract table from second table

**ttmult**

**Multiply table by second table**

Syntax: `ttmult(table_dest,table_mod,moduloval)`  

```
codeint table_dest; /* real-time table to
be modified */
codeint table_mod; /* real-time table
containing multipliers */
int moduloval; /* modulo value of result
*/
```

Description: A real-time operation that multiplies `table_dest` by `table_mod` and returns a new `table_dest`.

Arguments: `table_dest` is the name of the destination table (t1 to t60).

`table_mod` is the name of the table (t1 to t60) that modifies `table_dest`. Each element in `table_dest` is modified by the corresponding element in `table_mod`, and the result is stored in `table_dest`. The number of elements in `table_dest` must be greater than or equal to the number of elements in `table_mod`.

`moduloval` is the modulo value taken on the result of the operation, if `moduloval` is greater than 0.

Examples: `ttmult(t28,t42,6);`

Related: `loadtable` Assign table elements from table text file  
`settable` Assign integer array to table  
`tsadd` Add integer to table elements  
`tsdiv` Divide integer into table elements  
`tsmult` Mutiply integer with table elements  
`tssub` Subtract integer from table elements  
`ttadd` Add table to second table  
`ttdiv` Divide table into second table  
`ttsub` Subtract table from second table

**ttsub****Subtract table from second table**

**Syntax:** `ttsub(table_dest,table_mod,moduloval)`  
`codeint table_dest; /* real-time table to`  
`be modified */`  
`codeint table_mod; /* real-time table to`  
`be subtracted */`  
`int moduloval; /* modulo value of`  
`result */`

**Description:** A real-time operation that subtracts `table_mod` from `table_dest` and returns a new `table_dest`.

**Arguments:** `table_dest` is the name of the destination table (t1 to t60).

`table_mod` is the name of the table (t1 to t60) that modifies `table_dest`. Each element in `table_dest` is modified by the corresponding element in `table_mod` and the result is stored in `table_dest`. The number of elements in `table_dest` must be greater than or equal to the number of elements in `table_mod`.

`moduloval` is the modulo value taken on the result of the operation, if `moduloval` is greater than 0.

**Examples:** `ttsub(t28,t42,6);`

**Related:**

<code>loadtable</code>	Assign table elements from table text file
<code>settable</code>	Assign integer array to table
<code>tsadd</code>	Add integer to table elements
<code>tsdiv</code>	Divide integer into table elements
<code>tsmult</code>	Multiply integer with table elements
<code>tssub</code>	Subtract integer from table elements
<code>ttadd</code>	Add table to second table
<code>ttdiv</code>	Divide table into second table
<code>ttmult</code>	Multiply table by second table

**txphase****Set quadrature phase of observe channel**

**Syntax:** `txphase(multiplier)`  
`codeint multiplier; /* real-time`  
`quadrature-phase multiplier */`

**Description:** Explicitly set the quadrature phase (multiple of 90°) for the observe channel outside of a pulse. If a small-angle phase has been set previously with `xmtrphase`, `txphase` adjusts only the portion of the phase that is a multiple of 90°. Use `xmtrphase(zero)` to clear any small-angle phase if desired.

**Arguments:** `multiplier` is a 90° multiplier for the observe-channel phase. The value must be a real-time variable (v1 to v42, oph, etc), a real-time

### 3 Pulse Sequence Statement Reference

constant (zero, one, *etc*) or a real-time table ( $t_1$  to  $t_{60}$ ).

Examples: `txphase(v3);`

Related:	<code>decphase</code>	Set quadrature phase of first decoupler
	<code>dec2phase</code>	Set quadrature phase of second decoupler
	<code>dec3phase</code>	Set quadrature phase of third decoupler
	<code>dec4phase</code>	Set quadrature phase of fourth decoupler
	<code>xmtrphase</code>	Set small-angle phase of observe channel



If the fields `phase_inc` and `amp_inc` are non zero the waveform pattern contains ramped elements. For each element, the first step has a phase of `phase` and an amplitude `amp`. Each succeeding step in the element is incremented by `phase_inc` and `amp_inc`. Phase is expressed in degrees with 16-bit (0.0055°) resolution and amplitude is expressed in amplitude units 0.0 to 1023.0 units with 16-bit (0.0625 unit) resolution.

The number of steps in a ramped, interpolated element has only a slight effect upon the efficiency of execution. The break point for improved efficiency occurs with pattern elements of about 3 to 5 steps. For long elements interpolation improves efficiency 50 to 100 fold compared to execution of elements with 1 step.

Ramped elements with multiple steps can have variable durations with a flip-angle or time resolution equal to the step size. When using elements with multiple steps, pulses in the waveform can be set with resolution of the step size rather than the element size. With interpolation, the RF controllers can sustain waveform output with a 25 ns or 50 ns step size, though it is necessary to keep the average duration of an elements at about 100 to 300 ns.

One should note that ramped, interpolated elements can have only that slope generated by a constant increment. The interpolator cannot dither (vary) the phase and amplitude increment within an element to produce an arbitrary slope. However it is possible to dither adjacent elements to achieve an average slope over a longer time.

**Arguments:** `pattern` is the root name of the pattern and is the same as argument 1 of `obsprgon`, `decprgon` etc. The `userDECshape` statement does not write a `.DEC` file.

`decpat_struct` is an array of structures of type `DECpattern` with values to create the pattern.

`tipangle_reson` is the smallest common divisor of the tip-angle durations of all the elements of the pattern and is the same as argument 3 of `obsprgon`, `decprgon`, etc.

`mode` is an unknown parameter that should be set to 1

`nsteps` is the number of steps in the pattern.

`channel` is an integer constant to identify the channel on which the waveform will be run and can be `OBSch`, `DECch`, `DEC2ch`, `DEC3ch`, and `DEC4ch`.

**Examples:** The array of structures is best declared with a `malloc` statement as shown below where `pstub` is the array variable name.

```
DECpattern *pstub;
```



```
pstub (DECpattern *)
malloc(NSTEPS*sizeof(struct
_DECpattern);
```

where NSTEPS is an *int* number of waveform elements greater than any expected value of nsteps. This code allocates memory for an array of DECpattern structures, one for each element of the waveform.

Populate the fields of each DECpattern structure for index = 0 to nsteps - 1:

```
pstub[index].tip = tip-angle duration
pstub[index].phase = starting phase
pstub[index].amp = starting amplitude
pstub[index].gate = gate value
pstub[index].phase_inc = phase increment
perstep
pstub[index].amp_inc = amplitude increment
per step
```

Create the pattern for obs with the syntax:

```
userDECshape("mypattern", pstub, 90.0, 1, nsteps, OBSch)
```

where "mypattern" is the root string that has been chosen to name the pattern, 90.0 assumes that the tip-angle durations are multiples of 90.0. Argument 4 must be 1.

This pattern might be run with `obsprgon("mypattern", 50.0e-9, 90.0);` where the step size (90\_pulselength) is 50.0 ns. Here the convention is used that 90.0 degrees of tip-angle resolution represent the step size, independent of the actual tip-angle.

If a pattern is created it must be used at least once. If obsprgon is executed with a conditional so must userDECshape.

Related:	decprgon	Start waveform decoupling on first decoupler
	dec2prgon	Start waveform decoupling on second decoupler
	dec3prgon	Start waveform decoupling on third decoupler
	dec4prgon	Start waveform decoupling on fourth decoupler
	init_decpattern	Create pattern file for waveform decoupling
	obsprgon	Start waveform decoupling on observe channel
	userRFshape	Create shaped-pulse pattern directly

### 3 Pulse Sequence Statement Reference

## V

<code>var_active</code>	Check parameter is used
<code>vdecprwf</code>	Set fine power level of first decoupler in real-time
<code>vdecprwfstepsize</code>	Set step size for real-time fine power of first decoupler
<code>vdec2prwf</code>	Set fine power level of second decoupler in real-time
<code>vdec2prwfstepsize</code>	Set step size for real-time fine power of second decoupler
<code>vdec3prwf</code>	Set fine power level of third decoupler in real-time
<code>vdec3prwfstepsize</code>	Set step size for real-time fine power of third decoupler
<code>vdec4prwf</code>	Set fine power level of fourth decoupler in real-time
<code>vdec4prwfstepsize</code>	Set step size for real-time fine power of fourth decoupler
<code>vdelay</code>	Execute time delay with fixed timebase and real-time count
<code>vdelay_list</code>	Get delay value from delay list with real-time index
<code>vobspwrf</code>	Set fine power level of observe channel in real-time
<code>vobspwrfstepsize</code>	Set step size for real-time fine power of observe channel

**var\_active****Check if parameter is used**

Syntax: `int value = var_active(parname)`

Description: Determine if the numeric parameter `parname` is active (`value = 1`) or inactive (`value = 0`). "Inactive" means that the parameter does not exist in the current parameter table or that it is set to 'n'.

**vdecprwf****Set fine power level of first decoupler in real-time**

Syntax: `vdecprwf(vamplitude)`

`codeint vamplitude; /*real-time variable or table to set fine power */`

Description: Set the fine power in real-time for the first decoupler using `vamplitude`, a real-time variable or a table. The default range of values of `vamplitude` is 0 to 4095, similar to that of the `decprwf` statement. The default amplitude resolution for this statement is 12-bits, because real-time integer and tables may not have fractional values.

To use `vdecprwf` with 16-bit amplitude resolution, include the statement `vdecprwfstepsize(16);` before the use of `vdecprwf` to reset the range as 0 to 65536.

Arguments: `vamplitude` must be a real-time variable (`v1` to `v42`, etc), or a real-time table (`t1` to `t60`) with values of 0 to 4095. The statement `vdecprwfstepsize(N)` (`N = 1, 2, 4, 8` and `16`) resets the maximum value of `vamplitude` to `N*4096 - 1` and sets the amplitude resolution as 12, 13, 14, 15 or 16 bits respectively, if the hardware allows it.

Related:	<code>decprwf</code>	Set fine power level of first decoupler
	<code>vdecprwfstepsize</code>	Set step size for real-time fine power of first decoupler
	<code>vdec2prwf</code>	Set fine power level of second decoupler in real-time
	<code>vdec2prwfstepsize</code>	Set step size for real-time fine power of second decoupler
	<code>vdec3prwf</code>	Set fine power level of third decoupler in real-time
	<code>vdec3prwfstepsize</code>	Set step size for real-time fine power of third decoupler
	<code>vdec4prwf</code>	Set fine power level of fourth decoupler in real-time
	<code>vdec4prwfstepsize</code>	Set step size for real-time fine power of fourth decoupler
	<code>vobspprwf</code>	Set fine power level of observe channel in real-time
	<code>vobspprwfstepsize</code>	Set step size for real-time fine power of observe channel

**vdecprwfstepsize**

**Set step size for real-time fine power of first decoupler**

Syntax: `vdecprwfstepsize(ampstepsize)`  
`double ampstepsize /*multiplier of the fine-power maximum */`

Description: Set the step size for the real-time fine power statement `vdecprwf`. The argument `ampstepsize` can have whole-number values of 1, 2, 4, 8, and 16 and it increases the default maximum of `vamplitude` for the real-time fine power statement `vdec4prwf(vamplitude)`. This `ampstepsize` has no effect upon the maximum of the run-time fine power statement `decprwf`, which remains at 4095.0. The new maximum is `ampstepsize*4096 - 1`. The new maximum allows increased amplitude resolution for a `vamplitude` step of 1. Values of `ampstepsize` of 1, 2, 4, 8 and 16 provide amplitude resolution of 12, 13, 14, 15 and 16 bits respectively, when the transmitter hardware allows it.

The statement `vdecprwfstepsize` is executed at run time and it must precede the use of `vdecprwf` in the sequence. It can be used multiple times in the sequence to reset the maximum.

**Arguments:** `ampstepsize` is a double with recommended whole number values of 1,2,4,8 and 16 to obtain real-time amplitude resolution of 12,13,14,15 and 16 bits, when the transmitter hardware allows it.

<b>Related:</b>	<code>decprwf</code>	Set fine power level of observe channel
	<code>vdecprwf</code>	Set fine power level of first decoupler in real-time
	<code>vdec2prwf</code>	Set fine power level of second decoupler in real-time
	<code>vdec2prwfstepsize</code>	Set step size for real-time fine power of second decoupler
	<code>vdec3prwf</code>	Set fine power level of third decoupler in real-time
	<code>vdec3prwfstepsize</code>	Set step size for real-time fine power of third decoupler
	<code>vdec4prwf</code>	Set fine power level of fourth decoupler in real-time
	<code>vdec4prwfstepsize</code>	Set step size for real-time fine power of fourth decoupler
	<code>vobspwrf</code>	Set fine power level of observe channel in real-time
	<code>vobspwrfstepsize</code>	Set step size for real-time fine power of observe channel

### **vdec2prwf**

#### **Set fine power level of second decoupler in real-time**

**Syntax:** `vdec2prwf(vamplitude)`  
`codeint vamplitude; /* real-time variable or table to set fine power */`

**Description:** Set the fine power in real-time for the second decoupler using `vamplitude`, a real-time variable or a table. The default range of values of `vamplitude` is 0 to 4095, similar to that of the `dec2prwf` statement. The default amplitude resolution for this statement is 12-bits, because real-time integer and tables may not have fractional values.

To use `vdec2prwf` with 16-bit amplitude resolution, include the statement `vdec2prwfstepsize(16);` before the use of `vdec2prwf` to reset the range as 0 to 65536.

**Arguments:** `vamplitude` must be a real-time variable (`v1` to `v42`, etc), or a real-time table (`t1` to `t60`) with values of 0 to 4095. The statement `vdec2prwfstepsize(N)` ( $N = 1, 2, 4, 8$  and 16) resets the maximum value of

vamplitude to  $N \cdot 4096 - 1$  and sets the amplitude resolution as 12, 13, 14, 15 or 16 bits respectively, if the hardware allows it.

Related:	dec2pwr <sub>f</sub>	Set fine power level of observe channel
	vdecpwr <sub>f</sub>	Set fine power level of first decoupler in real-time
	vdecpwr <sub>f</sub> stepsize	Set step size for real-time fine power of first decoupler
	vdec2pwr <sub>f</sub> stepsize	Set step size for real-time fine power of second decoupler
	vdec3pwr <sub>f</sub>	Set fine power level of third decoupler in real-time
	vdec3pwr <sub>f</sub> stepsize	Set step size for real-time fine power of third decoupler
	vdec4pwr <sub>f</sub>	Set fine power level of fourth decoupler in real-time
	vdec4pwr <sub>f</sub> stepsize	Set step size for real-time fine power of fourth decoupler
	vobspwr <sub>f</sub>	Set fine power level of observe channel in real-time
	vobspwr <sub>f</sub> stepsize	Set step size for real-time fine power of observe channel

**vdec2pwr<sub>f</sub>stepsize**

**Set step size for real-time fine power of second decoupler**

Syntax: `vdec2pwrfstepsize(ampstepsize)`  
`double ampstepsize /*multiplier of the fine-power maximum */`

Description: Set the step size for the real-time fine power statement `vdec2pwrf`. The argument `ampstepsize` can have whole-number values of 1, 2, 4, 8, and 16 and it increases the default maximum of `vamplitude` for the real-time fine power statement `vdec4pwrf(vamplitude)`. This stepsize has no effect upon the maximum of the run-time fine power statement `dec4pwrf`, which remains at 4095.0.

The new maximum is  $stepsize \cdot 4096 - 1$ . The new maximum allows increased amplitude resolution for a `vamplitude` step of 1. Values of `ampstepsize` of 1, 2, 4, 8 and 16 provide amplitude resolution of 12, 13, 14, 15 and 16 bits respectively, when the transmitter hardware allows it.

The statement `vdec2pwrfstepsizesize` is executed at run time and it must precede the use of `vdec2pwr` in the sequence. It can be used multiple times in the sequence to reset the maximum.

**Arguments:** `ampstepsizesize` is a double with recommended whole number values of 1,2,4,8 and 16 to obtain real-time amplitude resolution of 12,13,14,15 and 16 bits, when the transmitter hardware allows it.

<b>Related:</b>	<code>dec2pwr</code>	Set fine power level of observe channel
	<code>vdecpwr</code>	Set fine power level of first decoupler in real-time
	<code>vdecpwrfstepsizesize</code>	Set step size for real-time fine power of first decoupler
	<code>vdec2pwr</code>	Set fine power level of second decoupler in real-time
	<code>vdec3pwr</code>	Set fine power level of third decoupler in real-time
	<code>vdec3pwrfstepsizesize</code>	Set step size for real-time fine power of third decoupler
	<code>vdec4pwr</code>	Set fine power level of fourth decoupler in real-time
	<code>vdec4pwrfstepsizesize</code>	Set step size for real-time fine power of fourth decoupler
	<code>vobspwr</code>	Set fine power level of observe channel in real-time
	<code>vobspwrfstepsizesize</code>	Set step size for real-time fine power of observe channel

### **vdec3pwr**

#### **Set fine power level of third decoupler in real-time**

**Syntax:** `vdec3pwr(vamplitude)`  
`codeint vamplitude; /* real-time variable or table to set fine power */`

**Description:** Set the fine power in real-time for the third decoupler using `vamplitude`, a real-time variable or a table. The default range of values of `vamplitude` is 0 to 4095, similar to that of the `dec3pwr` statement. The default amplitude resolution for this statement is 12-bits, because real-time integer and tables may not have fractional values.

To use `vdec3pwr` with 16-bit amplitude resolution, include the statement `vdec3pwrfstepsizesize(16);`

before the use of `vdec3pwr` to reset the range as 0 to 65536.

**Arguments:** `vamplitude` must be a real-time variable (`v1` to `v42`, etc), or a real-time table (`t1` to `t60`) with values of 0 to 4095. The statement `vdec3pwrstepsize(N)` (`N = 1, 2, 4, 8` and `16`) resets the maximum value of `vamplitude` to `N*4096 - 1` and sets the amplitude resolution as 12, 13, 14, 15 or 16 bits respectively, if the hardware allows it.

<b>Related:</b>	<code>dec3pwr</code>	Set fine power level of observe channel
	<code>vdecpwr</code>	Set fine power level of first decoupler in real-time
	<code>vdecpwrstepsize</code>	Set step size for real-time fine power of first decoupler
	<code>vdec2pwr</code>	Set fine power level of second decoupler in real-time
	<code>vdec2pwrstepsize</code>	Set step size for real-time fine power of second decoupler
	<code>vdec3pwrstepsize</code>	Set step size for real-time fine power of third decoupler
	<code>vdec4pwr</code>	Set fine power level of fourth decoupler in real-time
	<code>vdec4pwrstepsize</code>	Set step size for real-time fine power of fourth decoupler
	<code>vobspwr</code>	Set fine power level of observe channel in real-time
	<code>vobspwrstepsize</code>	Set step size for real-time fine power of observe channel

**vdec3pwrstepsize**

**Set step size for real-time fine power of fourth decoupler**

**Syntax:** `Syntax: vdec3pwrstepsize(ampstepsize)`  
`double ampstepsize /*multiplier of the fine-power maximum */`

**Description:** Set the step size for the real-time fine power statement `vdec3pwr`. The argument `ampstepsize` can have whole-number values of 1, 2, 4, 8, and 16 and it increases the default maximum of `vamplitude` for the real-time fine power statement `vdec3pwr(vamplitude)`. This `ampstepsize` has no effect upon the maximum of the run-time fine



power statement `dec3pwr`, which remains at 4095.0.

The new maximum is  $\text{ampstepsize} * 4096 - 1$ . The new maximum allows increased amplitude resolution for a `vamplitude` step of 1. Values of `ampstepsize` of 1,2,4,8 and 16 provide amplitude resolution of 12, 13, 14 15 and 16 bits respectively, when the transmitter hardware allows it.

The statement `vdec3pwrstepsize` is executed at run time and it must precede the use of `vdec3pwr` in the sequence. It can be used multiple times in the sequence to reset the maximum.

**Arguments:** `ampstepsize` is a double with recommended whole number values of 1,2,4,8 and 16 to obtain real-time amplitude resolution of 12,13,14,15 and 16 bits, when the transmitter hardware allows it.

<b>Related:</b>	<code>dec3pwr</code>	Set fine power level of observe channel
	<code>vdecpwr</code>	Set fine power level of first decoupler in real-time
	<code>vdecpwrstepsize</code>	Set step size for real-time fine power of first decoupler
	<code>vdec2pwr</code>	Set fine power level of second decoupler in real-time
	<code>vdec2pwrstepsize</code>	Set step size for real-time fine power of second decoupler
	<code>vdec3pwr</code>	Set fine power level of third decoupler in real-time
	<code>vdec4pwr</code>	Set fine power level of fourth decoupler in real-time
	<code>vdec4pwrstepsize</code>	Set step size for real-time fine power of fourth decoupler
	<code>vobspwr</code>	Set fine power level of observe channel in real-time
	<code>vobspwrstepsize</code>	Set step size for real-time fine power of observe channel

### **vdec4pwr**

#### **Set fine power level of fourth decoupler in real-time**

**Syntax:** `vdec4pwr(vamplitude)`  
`codeint vamplitude /*real-time variable or table to set fine power */`

**Description:** Set the fine power in real-time for the fourth decoupler using `vamplitude`, a real-time variable or a table. The default range of values of `vamplitude` is 0 to 4095, similar to that of the `dec4pwr` statement. The default amplitude resolution for this statement is 12-bits, because real-time integer and tables may not have fractional values.

To use `vdec4pwr` with 16-bit amplitude resolution, include the statement `vdec4pwrstepsize(16);` before the use of `vdec4pwr` to reset the range as 0 to 65536.

**Arguments:** `vamplitude` must be a real-time variable (`v1` to `v42`, *etc*), or a real-time table (`t1` to `t60`) with values of 0 to 4095. The statement `vdec4pwrstepsize(N)` (`N = 1, 2, 4, 8` and `16`) resets the maximum value of `vamplitude` to  $N * 4096 - 1$  and sets the amplitude resolution as 12, 13, 14, 15 or 16 bits respectively, if the hardware allows it.

<b>Related:</b>	<code>dec4pwr</code>	Set fine power level of observe channel
	<code>vdecpwr</code>	Set fine power level of first decoupler in real-time
	<code>vdecpwrstepsize</code>	Set step size for real-time fine power of first decoupler
	<code>vdec2pwr</code>	Set fine power level of second decoupler in real-time
	<code>vdec2pwrstepsize</code>	Set step size for real-time fine power of second decoupler
	<code>vdec3pwr</code>	Set fine power level of third decoupler in real-time
	<code>vdec3pwrstepsize</code>	Set step size for real-time fine power of third decoupler
	<code>vdec4pwrstepsize</code>	Set step size for real-time fine power of fourth decoupler
	<code>vobspwr</code>	Set fine power level of observe channel in real-time
	<code>vobspwrstepsize</code>	Set step size for real-time fine power of observe channel

**vdec4pwrfstpsize****Set step size for real-time fine power of fourth decoupler**

**Syntax:** `vdec4pwrfstpsize(ampstepsize)`  
`double ampstepsize /*multiplier of the`  
`fine-power maximum */`

**Description:** Set the step size for the real-time fine power statement `vdec4pwr`. The argument `ampstepsize` can have whole-number values of 1, 2, 4, 8, and 16 and it increases the default maximum of `vamplitude` for the real-time fine power statement `vdec4pwr(vamplitude)`. This `ampstepsize` has no effect upon the maximum of the run-time fine power statement `dec4pwr`, which remains at 4095.0.

The new maximum is  $\text{ampstepsize} * 4096 - 1$ . The new maximum allows increased amplitude resolution for a `vamplitude` step of 1. Values of `ampstepsize` of 1, 2, 4, 8 and 16 provide amplitude resolution of 12, 13, 14, 15 and 16 bits respectively, when the transmitter hardware allows it.

The statement `vdec4pwrfstpsize` is executed at run time and it must precede the use of `vdec4pwr` in the sequence. It can be used multiple times in the sequence to reset the maximum.

**Arguments:** `ampstepsize` is a double with recommended whole number values of 1, 2, 4, 8 and 16 to obtain real-time amplitude resolution of 12, 13, 14, 15 and 16 bits, when the transmitter hardware allows it.

<b>Related:</b>	<code>dec4pwr</code>	Set fine power level of observe channel
	<code>vdecpwr</code>	Set fine power level of first decoupler in real-time
	<code>vdecpwrfstpsize</code>	Set step size for real-time fine power of first decoupler
	<code>vdec2pwr</code>	Set fine power level of second decoupler in real-time
	<code>vdec2pwrfstpsize</code>	Set step size for real-time fine power of second decoupler
	<code>vdec3pwr</code>	Set fine power level of third decoupler in real-time
	<code>vdec3pwrfstpsize</code>	Set step size for real-time fine power of third decoupler
	<code>vdec4pwr</code>	Set fine power level of fourth decoupler in real-time

vobspwrf	Set fine power level of observe channel in real-time
vobspwrfstepsize	Set step size for real-time fine power of observe channel

**vdelay**

**Set delay with fixed timebase and real-time count**

Syntax: `vdelay(timebase,count)`  
`int timebase; /* NSEC, USEC, MSEC, or SEC */`  
`codeint count; /* real-time variable for count */`

Description: Set a delay for a time period equal to the product of the specified timebase and count. If timebase is NSEC, the time base is a step of 12.5 ns and if count has a real-time value less than 4 (< 50.0 ns), vdelay is not executed.

Arguments: timebase is an integer constant set to one of four defined time bases: NSEC (12.5 ns), USEC (microseconds), MSEC (milliseconds), or SEC (seconds).

count is a real-time variable (v1 to v42) or a real-time table (t1 to t60).

Examples: `vdelay(NSEC,v3);`

Related:	delay	Execute time delay
	hsdelay	Execute time delay with optional homospoil pulse

**vdelay\_list**

**Get delay value from delay list with real-time index.**

Syntax: `vdelay_list(list_number,vindex)`  
`int list_number; /* same index as create_delay_list */`  
`codeint vindex; /* real time variable */`

Description: Provides a means of indexing into previously created delay lists using a real-time variable or a table. The indexing into the list is from 0 to N-1, where N is the number of items in the list. The delay table has to have been created with the create\_delay\_list statement.

Arguments: tlist\_number is the number between 0 and 255 for each list. This number must match the list\_number used when creating the table.

vindex is a real-time variable (v1 to v14) or a table (t1 to t60).

Examples: 

```
pulsesequence()
{
  int ndelay, listnum;
  double delay[256];
  ...
  /* compute values in delay array */
  ...
  /* initialize delay list */
  listnum =
  create_delay_list(delay, ndelay, 1);
  ...
  vdelay_list(listnum, v5); /* get
  element specified by v5 from delay list
  "listnum" */
}
```

Related:	<code>create_delay_list</code>	Create table of delays
	<code>delay</code>	Delay for a specified time
	<code>hsdelay</code>	Delay specified time with possible homospoil pulse
	<code>incdelay</code>	Real time incremental delay
	<code>initdelay</code>	Initialize incremental delay
	<code>vfreq</code>	Select frequency from table
	<code>voffset</code>	Select frequency offset from table
	<code>vdelay</code>	Set delay with fixed timebase and real-time count

**vobspwrf****Set fine power level of observe channel in real-time**

Syntax: 

```
vobspwrf(vamplitude)
codeint vamplitude /* a real-time variable
to set fine power */
```

Description: Set the fine power in real-time for the observe channel using `vamplitude`, a real-time variable or a table. The default range of values of `vamplitude` is 0 to 4095, similar to that of the `obspwrf` statement. The default amplitude resolution for this statement is 12-bits, because real-time integer and tables may not have fractional values.

To use `vobspwrf` with 16-bit amplitude resolution, include the statement `vobspwrfstepsize(16)`;

before the use of `vobspwrf` to reset the range as 0 to 65536.

Arguments: `vamplitude` must be a real-time variable (`v1` to `v42`, etc), or a real-time table (`t1` to `t60`) with values of 0 to 4095. The statement `vobspwrfstepsize(N)` (`N = 1,2,4,8` and `16`) resets the maximum value of `vamplitude` to  $N*4096 - 1$  and sets the amplitude resolution as 12, 13, 14, 15 or 16 bits respectively, if the hardware allows it.

Related:	<code>obspwrf</code>	Set fine power level of observe channel
	<code>vdecpwrf</code>	Set fine power level of first decoupler in real-time
	<code>vdecpwrfstepsize</code>	Set step size for real-time fine power of first decoupler
	<code>vdec2pwrf</code>	Set fine power level of second decoupler in real-time
	<code>vdec2pwrfstepsize</code>	Set step size for real-time fine power of second decoupler
	<code>vdec3pwrf</code>	Set fine power level of third decoupler in real-time
	<code>vdec3pwrfstepsize</code>	Set step size for real-time fine power of third decoupler
	<code>vdec4pwrf</code>	Set fine power level of fourth decoupler in real-time
	<code>vdec4pwrfstepsize</code>	Set step size for real-time fine power of fourth decoupler
	<code>vobspwrfstepsize</code>	Set step size for real-time fine power of observe channel

**vobspwrfstepsize**

**Set step size for real-time fine power of fourth decoupler**

Syntax: `vobspwrfstepsize(ampstepsize)`  
`double ampstepsize /*multiplier of the fine-power maximum */`

Description: Set the step size for the real-time fine power statement `vobspwrf`. The argument `ampstepsize` can have whole-number values of 1,2,4,8,and 16 and it increases the default maximum of

vamplitude for the real-time fine power statement vdec4pwrff(vamplitude). This ampstepsize has no effect upon the maximum of the run-time fine power statement dec4pwrff, which remains at 4095.0.

The new maximum is ampstepsize\*4096 - 1. The new maximum allows increased amplitude resolution for a vamplitude step of 1. Values of ampstepsize of 1,2,4,8 and 16 provide amplitude resolution of 12, 13, 14 15 and 16 bits respectively, when the transmitter hardware allows it.

The statement vobspwrffstepsize is executed at run time and it must precede the use of vobspwrff in the sequence. It can be used multiple times in the sequence to reset the maximum.

**Arguments:** ampstepsize is a double with recommended whole number values of 1,2,4,8 and 16 to obtain real-time amplitude resolution of 12,13,14,15 and 16 bits, when the transmitter hardware allows it.

<b>Related:</b>	obspwrff	Set fine power level of observe channel
	vdecpwrff	Set fine power level of first decoupler in real-time
	vdecpwrffstepsize	Set step size for real-time fine power of first decoupler
	vdec2pwrff	Set fine power level of second decoupler in real-time
	vdec2pwrffstepsize	Set step size for real-time fine power of second decoupler
	vdec3pwrff	Set fine power level of third decoupler in real-time
	vdec3pwrffstepsize	Set step size for real-time fine power of third decoupler
	vdec4pwrff	Set fine power level of fourth decoupler in real-time
	vdec4pwrffstepsize	Set step size for real-time fine power of fourth decoupler
	vobspwrff	Set fine power level of observe channel in real-time

## W

<code>warn_message</code>	Send warning message to VnmrJ
<code>writeMRIUserByte</code>	Set user byte on the MRI User Panel

### `warn_message`

#### Send warning message to VnmrJ

Syntax: `warn_message(message, varnames)`

```
char *message      /* a formatted string containing the
message */
varnames           /* char, int or double, used in message */
```

Description: Send a formatted warning message to VnmrJ and cause a beep. The formatting is similar to the C `printf` statement.

Related:	<code>abort_message</code>	Abort PSG at run-time and send message to VnmrJ
	<code>psg_abort</code>	Abort PSG Process
	<code>text_error</code>	Send error message to VnmrJ
	<code>text_message</code>	Send message to VnmrJ

### `writeMRIUserByte`

#### Set user byte on MRI User panel

Syntax: `writeMRIUserByte(a)`  
`codeint a; /* real-time variable for byte */`

Description: Write the eight output lines from the 15-pin D-connector User Out on the MRI GUser panel on the back of the console from the real-time variable `a`. The write is performed synchronously with the experiment.

Arguments: The argument is a real-time variable from where the byte is written.

Examples: `writeMRIUserByte(v2);`



## X

xgate	Gate pulse sequence from external tachometer signal
xmtroff	Turn off observe channel
xmtron	Turn on observe channel
xmtrphase	Set small-angle phase of observe channel

**xgate****Gate pulse sequence from external tachometer signal**

Syntax: `xgate(events)`  
`double events; /* number of external events */`

Description: Halt all channels of a pulse sequence. The pulse sequence continues after `events` external events have occurred.

Arguments: `events` is the number of external events.

Examples: `xgate(2.0);`

Related:	<code>rotorperiod</code>	Obtain period of external tachometer signal
	<code>rotorsync</code>	Execute time delay based on external tachometer signal

**xmtroff****Turn off observe channel**

Syntax: `xmtroff()`

Description: Explicitly gate off the observe transmitter outside of a pulse. Amplifier blanking state is unchanged.

Related:	<code>decoff</code>	Turn off first decoupler
	<code>dec2off</code>	Turn off second decoupler
	<code>dec3off</code>	Turn off third decoupler
	<code>dec4off</code>	Turn off fourth decoupler
	<code>xmtron</code>	Turn on observe channel

**xmtron****Turn on observe channel**

Syntax: `xmtron()`

Description: Explicitly gate on the observe channel in the pulse sequence outside of a pulse. Amplifier blanking state is unchanged. The associated amplifier must

be previously unblanked with `obsunblank` at least  $2.0\ \mu\text{s}$  before `xmtron`. Follow `xmtron` with a delay, `xmtroff`, and an optional `obsblank`.

Related:	<code>decon</code>	Turn on first decoupler
	<code>dec2on</code>	Turn on second decoupler
	<code>dec3on</code>	Turn on third decoupler
	<code>dec4on</code>	Turn on fourth decoupler
	<code>xmtroff</code>	Turn off observe channel

**`xmtrphase`****Set small-angle phase of observe channel**

Syntax: `xmtrphase(multiplier)`  
`codeint multiplier; /* real-time`  
`phase-step multiplier */`

Description: Set the phase as a product of `multiplier` and a phase `stepsize`, which is set in degrees by the `obsstepsize` statement. If `obsstepsize` has not been used, the default `stepsize` is  $90^\circ$ .

The `xmtrphase` statement sets the total phase as a sum of a quadrature part, a multiple of  $90^\circ$ , and a small-angle part,  $0^\circ$  to  $90^\circ$ . The small-angle phase of VNMR5 has a phase resolution of  $360.0/8192$  or about  $0.044^\circ$  and the small-angle phase is set to the nearest step.

The `xmtrphase` statement overrides the quadrature part of the phase, set by any previous `txphase` statement. The `txphase` statement overrides only the quadrature part of the phase. Use `xmtrphase(zero)` to remove small-angle phase and return to only quadrature phases.

You should be aware that `rgpulse` can be set only in the quadrature phase. Set a small-angle phase cycle with `xmtrphase` before the pulse and use `zero` as the quadrature-phase multiplier for the pulse.

Arguments: `multiplier` is a small-angle phaseshift multiplier for the first decoupler. The value must be a real-time variable (`v1` to `v42`, `oph`, *etc*), a real-time constant (`zero`, `one`, *etc*) or a real-time table (`t1` to `t60`).

Examples: `xmtrphase(v1);`

Related:	<code>dcplrphase</code>	Set small-angle phase of first decoupler
----------	-------------------------	--

dcplr2phase	Set small-angle phase of second decoupler
dcplr3phase	Set small-angle phase of third decoupler
dcplr4phase	Set small-angle phase of fourth decoupler
obsstepsize	Set small-angle phase stepsize of observe channel
txphase	Set quadrature phase of observe channel

Z

zero_all_gradients	Zero gradient DAC level for all axes
zgradpulse	Perform gradient pulse on z axis

**zero\_all\_gradients**

**Zero gradient DAC level for all axes**

Syntax: `zero_all_gradients()`  
 Description: Set the gradients of the X, Y, and Z axes to zero.  
 Examples: `magradient(3.0);`  
`delay(0.001);`  
`zero_all_gradients();`

Related:	<code>mashapedgradient</code>	Perform three-axis shaped gradient at magic angle
	<code>obl_shaped3gradient</code>	Perform three-axis oblique shaped gradient, three patterns
	<code>pe3_shaped3gradient</code>	Perform oblique gradient, one pattern, phase encode three axes
	<code>phase_encode3oblshapedgradient</code>	Perform general oblique shaped gradient, phase encode three axes

**zgradpulse**

**Create a gradient pulse on the z channel**

Syntax: `zgradpulse(daclvl,width)`  
`double daclvl; /* gradient amplitude, DAC units */`  
`double width; /* duration of gradient pulse, seconds */`

Description: Create a gradient pulse on the Z channel with an amplitude of `daclvl`, in DAC units and a duration `width`, in seconds. The gradient amplitude is reset to 0 at the end of the pulse. The `gradalt` parameter can be used to multiply the amplitude on alternative scans.

Arguments: `daclvl` is the gradient amplitude in DAC units, -32767 to 32767.  
`width` is the duration of the gradient pulse, in seconds.

Examples: `zgradpulse(1234.0,d2);`

Related:	<code>rgradient</code>	Set DAC level of any one gradient axis
	<code>shapedgradient</code>	Perform shaped gradient pulse on any one axis



## 4 Linux Level Programming

Linux and VnmrJ 406

Linux Reference Guide 407

Linux Commands Accessible from VnmrJ 411

Background VNMR 412

Shell Programming 414

Several books have been written on every aspect and level of UNIX and much of it also applies to Linux, the open-source version of UNIX. This manual does not replace that material.



## Linux and VnmrJ

The VnmrJ software is a complete NMR work environment and VnmrJ users do not need to work directly with the operating system aside from login, logout, and starting VnmrJ. The operating system runs the workstation at all times. The user starts VnmrJ by clicking on the VnmrJ icon after completing the login procedure. Operators assigned to a Walkup account remain within the VnmrJ environment and use the VnmrJ switch operator function and login screen.

Linux provides "tools" to perform almost anything short of complex mathematical manipulations, search through your files, sort line lists, report who is on the system, run a program unattended, and more. Use the online help provided with Linux and other published third-party references to learn about these tools.

## Linux Reference Guide

This section is a brief overview of the operating system and its associated commands.

Command Entry

File Names

File Handling Commands

Directory Names

Directory Handling Commands

Text Commands

Other Commands

Special Characters

This is a brief overview of the operating system and its associated commands.

### Command entry

Single command entry	<code>commandname</code>
Command names	Generally lowercase, case-sensitive
Multiple command separator	<code>;</code> (semicolon) or new line
Arguments	<code>commandname arg1 arg2</code>

## File names

Typical (shorthand names usually used)	<code>/vnmr/fidlib/fid1d</code>
Level separator	<code>/</code> (forward slash)
Individual filenames	Any number of characters (256 unique)
Characters in filenames	Underline, period often used
First character in filename	First character unrestricted

## File handling commands

Delete (unlink) a file(s)	<code>rm filenames</code>
Copy a file	<code>cp filename newfilename</code>
Rename a file	<code>mv filename newfilename</code>
Make an alias (link)	<code>ln target linkname</code>
Sort files	<code>sort filenames</code>
Tape backup	<code>tar</code>
Package files	<code>zip</code>

## Directory names

Home directory for each user	Directory assigned by the administrator
Working directory	Current directory the user is in
Shorthand for current directory	<code>.</code> (single period)
Shorthand for parent directory	<code>..</code> (two periods)
Shorthand for home directory	<code>~</code> (tilde character)
Root directory	<code>/</code> (forward slash)

## Directory handling commands

Create (or make) a directory	<code>mkdir directoryname</code>
Rename a directory	<code>mv dirname newdirname</code>
Remove an empty directory	<code>rmdir directoryname</code>
Delete directory and all files in it	<code>rm -r directoryname</code>



List files in a directory, short list	<code>ls directoryname</code>
List files in a directory, long list	<code>ls -l directoryname</code>
Copy file(s) into a directory	<code>cp filenames directoryname</code>
Move file(s) into a directory	<code>mv filenames directoryname</code>
Show current directory	<code>pwd</code>
Change current directory	<code>cd newdirectoryname</code>

## Text commands

Edit a text file using <code>vi</code> editor	<code>vi filename</code>
Edit a text file using <code>ed</code> editor	<code>ed filename</code>
Edit a text file using <code>textedit</code> editor	<code>textedit filename</code>
Display first part of a file	<code>head filename</code>
Display last part of a file	<code>tail filename</code>
Concatenate and display files	<code>cat filenames</code>
Compare two files	<code>cmp filename1 filename2</code>
Compare two files deferentially	<code>diff filename1 filename2</code>
Print file(s) on line printer	<code>lp filenames</code>
Search file(s) for a pattern	<code>grep expression filenames</code>
Find spelling errors	<code>spell filename</code>

## Other commands

Pattern scanning and processing	<code>awk pattern filename</code>
Change file protection mode	<code>chmod newmode filename</code>
Display current date and time	<code>date</code>
Summarize disk usage	<code>du -k</code>
Report free disk space	<code>df -k filesystem</code>
Kill a background process	<code>kill process-id</code>
Sign on to system	<code>login username</code>
Send mail to other users	<code>mail</code>
Print out UNIX manual entry	<code>man commandname</code>
Process status	<code>ps</code>

Convert quantities to another scale	<code>units</code>
Who is on the system	<code>w</code>
System identification	<code>uname -a</code>

## Special characters

Send output into named file	<code>&gt; filename</code>
Append output into named file	<code>&gt;&gt; filename</code>
Take input from named file	<code>&lt; filename</code>
Send output from first command to input of second command (pipe)	<code> </code> (vertical bar)
Wildcard character for a single character in filename operations	<code>?</code>
Wildcard character for multiple characters in filename operations	<code>*</code>
Run program in background	<code>&amp;</code>
Abort the current process	Control-C
Logout or end of file	Control-D

## Linux Commands Accessible from VnmrJ

Several commands are accessible directly from VnmrJ, including the `vi`, `edit`, `shell`, `shelli`, and `w` commands.

### Opening a text editor from VnmrJ

Entering `vi(file)` or `edit(file)` from VnmrJ opens a text editor screen for editing the name of the file given in the argument (e.g., `vi('myfile')`). Exiting from the editor closes the editing window.

A useful Linux and UNIX editing program is `vi`. The UNIX text editors, `ed` and `textedit`, and the Linux `gedit` that are easier to learn than `vi`, but `vi` is the a widely used Linux and UNIX text editor because of its features. A text editor is necessary to prepare or edit text files, such as macros, menus, and pulse sequences (short text files such as those used to annotate spectra are usually edited in simpler ways).

### Opening a shell from VnmrJ

Entering the `shell` command from VnmrJ without any argument opens a normal Linux or UNIX shell. Entering `shell` with the syntax:

```
shell(command) <:$var1,$var2,...>
```

executes the operating system `command` given, displays any text lines generated, and returns control to VnmrJ when finished. The results of the command line are returned to the variables `$var1`, `$var2...` if return arguments are present. Each variable receives a single display line.

`shell` calls involving pipes (`|`) or input redirection (`<`) require either an extra pair of parentheses or the addition of `;` `cat` to the shell command string, for example:

```
shell('(ls -t|grep May)'):$list
shell('ls -t|grep May; cat'):$list
```

To display information about who is on to the operating system, enter the `w` command from VnmrJ.

## Background VNMR

This section describes running VNMR commands and processing in the background.

### Running VNMR Command as a Linux background task

VNMR commands can be executed as a Linux background task by using the command

```
Vnmr ?mback <?n#> command _string <&>
```

where `-mback` is a keyword (entered exactly as shown), `-n#` defines that processing will occur in experiment # (e.g., `-n2` sets experiment 2), and `command _string` is a VNMR command or macro. If `-n#` is omitted, processing occurs in experiment 1. If more than one command is to be executed, place double quote marks around the command string, e.g.

```
"printon dg printoff"
```

Linux background operation (`&`) is possible, as in `Vnmr -mback wft2da &`. Use of redirection (`>` or `>>`) with background processing is recommended:

```
Vnmr -mback ?n3 wft2da > vnmroutput &
```

The `vbg` shell script is also available to run VNMR processing in the background.

All text output, both normal text window output and the typical two-letter prompts that appear in the upper right ("FT", "PH", *etc.*), are directed to the UNIX output window.

Note the following characteristics of the `Vnmr` command:

- Full multi-user protection is implemented. If user `vnmr1` is logged in and using experiment 1, and another person logs in as `vnmr1` from another terminal and tries to use the background `Vnmr`, the second `vnmr1` receives the message "experiment 1 locked" if that person tries to use experiment 1. The second user can, however, use other experiments.
- Pressing Control-C does *not* work. Typing the command shown cannot be aborted with Control-C.
- Operation within VNMR is possible using shell.

```
shell ('Vnmr -mback -n2 wftda')
```

- Plotting is possible.

```
Vnmr -mback -n3 "pl pscale pap page"
```

- Printing is possible.

```
Vnmr -mback "printon dg printoff"
```

## Running VNMR processing in the background

The `vbg` shell script runs VNMR processing in the background. The main requirements are that `vbg` must be run from within a shell and that no foreground or other background processes can be active in the designated experiment. Open a terminal window and start `vbg` in the following form:

```
vbg # command_string <prefix>
```

where `#` is the number of an experiment (from 1 to 9) in the user's directory in which the background processing is to take place, `command_string` is one or more VNMR commands and macros to be executed in the background (double quotes surrounding the string are mandatory), and `prefix` is the name of the log file, making the full log file name `prefix_bgf.log` (e.g., to perform background plotting from experiment 3, enter `vbg 3 "vsadj pl pscale pap page" plotlog`).

The default log file name is `#_bgf.log`, where `#` is the experiment number. The log file is placed in the experiment in which the background processing takes place. Refer to the *Command and Parameter Reference* for more information on `vbg`.

## Shell Programming

The shell variables execute commands given either from a terminal or those contained in a file. Files containing commands and control flow notation, called *shell scripts*, can be created, allowing users to build their own commands. This section provides a short overview of such programming; refer to the Linux and UNIX literature for more information.

### Shell variables and control formats

As a programming language, the shell provides string-valued variables: \$1, \$2,... The number of variables is available as \$# and the file being executed is available as \$0. Control flow is provided by special notation, including *if*, *case*, *while*, and *for*. The following format is used:

<pre>if command-list (not Boolean) then command-list else command-list fi</pre>	<pre>while command-list do command-list done</pre>
<pre>case word in pattern) command-list;; ... esac</pre>	<pre>for name (in w1 w2) do command-list done</pre>

### Shell scripts

The following shows two ways to write a shell script for the same command. In both scripts, the command name *lower* is selected by the user and the intent of the command is to convert a file to lower case, but the scripts differ in features.

The first script:

```
: lower --- command to convert a file to lower case
: usage   lower filename
: output  filename.lower
tr '[A-Z]' '[a-z]' < $1 > $1.lower
```

The second script:

```
: lower --- a command to convert a file to lower case
: usage   lower filename or lower inputfile outputfile
: output  filename.lower or output file
case $# in
  1) tr '[A-Z]' '[a-z]' <$1 > $1.lower;;
  2) tr '[A-Z]' '[a-z]' <$1 > $2;;
  *) echo "Usage: lower filename or lower \
        inputfile outputfile";;
esac
```

In the first script, only one form of input is allowed. In the

second script, not only is a second form of input allowed, but a prompt explaining how to use `lower` appears if the user enters `lower` without any arguments. Notice that in both scripts, a colon is used to identify lines containing comments (and that each script is carefully commented).

## Data backup under Linux

This section describes backup of data using Linux.

### General considerations

Data backup is an important task that should not be underestimated. Although modern computer hard- and software should be, in principle, quite robust against “environmental” risks, the possibility remains that the spectrometer host (and/or data station computers) can still be affected by power failures, unauthorized intruders (“hackers”) etc.

Some of these risks can and should be minimized wherever possible: If the NMR lab is situated in an environment where power outages occur rather frequently (more than once per year), the purchase of an uninterruptible power supply (UPS) should be considered, at least for the spectrometer host computer. If the network the spectrometer host is connected to is inherently not very safe (no firewall towards the external internet, for example), setting up the personal Linux firewall may be advisable.

The presence of a data backup can tremendously help in such cases. It should therefore be considered to back up the following data structures, with descending order of importance:

- 1 NMR data directories
- 2 Application directories (these often contain personalized settings and customized macros, sequences etc.) should be backed up at least once using CD/DVD, for example
- 3 Personal home directories of VnmrJ users (“/home/vnmr1”, for example) can be re-created quite easily after re-installing VnmrJ, but any personal settings or customizations would be lost. Therefore, home directories should be backed up at least once, after personalization and customization, using CD/DVD for example. Running instead a continuous, automatic backup ensures being able to get back to the point of work where it was stopped.
- 4 The VnmrJ home directory (/vnmr) should not hold, in principle, any personalized data apart from system packages like Biopack, Solidspack or VnmrJ patches that cannot be re-installed rather easily.



- 5 The Red Hat operating system (/usr, /dev etc.) itself is considerably more difficult to back up into a state which enables the user to functionally re-create the operating system. Although such (typically stand-alone, bootable CDs/DVDs) exist to do this, it should not be covered here. A recovery Red Hat DVD is supplied together with the Dell Linux computer that enables the re-installation of the operating system. See also the “Linux installation for VnmrJ” manual.

One of the safest ways to back up data – setting up a RAID 1 (disk mirror with two disks, possible as “software RAID” without special controller) or RAID 5 (redundant system with minimum 3 disks, usually requires special controller or BIOS) requires doing so before Linux installation and shall not be covered in this guide.

In the following sections, three typical backup mechanisms shall be covered which are supported by Red Hat Linux and VnmrJ: Backup on CD/DVD media, data mirroring using VnmrJ tools and backup using rsync on either local or remote filesystems.

## Data backup on CDs/DVDs

The probably least demanding and cheapest way to do data backup is to use DVD-R media. CD-R media are, although they could be used as well, in principle, are not very convenient for NMR data due to their rather limited capacity of approx. 700 MB versus approx. 4.5 GB of typical DVDs.

Advantages: Using DVDs for data backup requires no hard- or software setup, all hard- and software required is already installed, Backups can easily be stored in a different place than the computer that is to be backed up (in case of fire, theft concerns etc.).

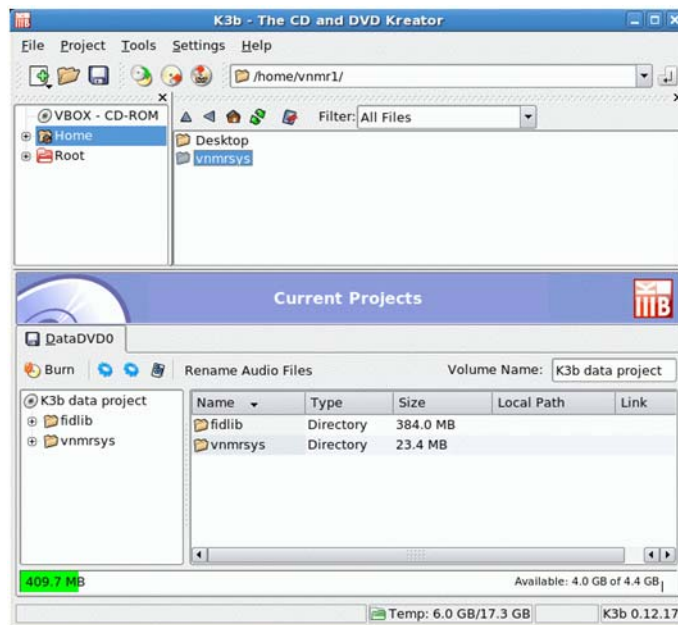
Disadvantages: Backup via DVDs cannot be automated; it always requires human intervention to start it and to insert multiple DVDs if necessary. The process is also rather slow (approx. 10 min to burn a 4.5 GB DVD). Hence, it should preferentially be used to back up smaller amounts of NMR data and/or the user home directories from time to time or as an occasional redundancy backup.

Your Dell spectrometer host is already equipped with a DVD burner and Red Hat Enterprise Linux contains a convenient DVD burning tool “K3b”. To launch the program, select “K3b” from the Linux Applications ? Sound & Video menu.

K3b is quite self-explanatory: After launching it for the first time, it may ask for confirmation of the DVD burner hardware (click “OK” here). After that, select a “New data CD” or “New data DVD” project:



Now drag-n-drop the directories to be backed up into the window and click the “Burn” button on the centre left of the window (alternatively, select “Project Burn” from the menu or press Ctrl-B).



On the following popup window you can choose to either burn a CD/DVD on the fly, keep or remove the ISO image or just create an ISO image without actually burning a CD/DVD (in the latter case, just click “Save” instead of “Burn”).

## Installation of a second hard disk

One of the cheapest and fastest ways to backup data besides using CDs or DVDs is to regularly copy the data on a second hard disk. This disk can be either attached to an external USB port or mounted inside the spectrometer host computer and attached to a free serial ATA (SATA) connector.

In principle, large, current-generation USB sticks could also be used as data backup, but you should take note that the FLASH chip inside USB sticks can only be overwritten approx. 10,000 times. This property makes USB memory sticks not very safe as a backup medium and hence they should only be used for occasional data transfer.

### External USB hard disk:

Attach the external hard disk to one of the external USB connectors, provide power to the harddrive and switch it on. The hard disk should be automatically recognized, just like a USB memory stick.

Typically, USB devices show up in the /media directory. It

may be handy to create a (soft) link to that directory so that the crontab entry/VnmrJ preferences don't need to be changed when changing the backup drive - in this case, don't create a /backup directory but type instead:

```
ln -sf /media/USB_DRIVE_NAME /backup
```

### Internal SATA hard disk:

- 1 Obtain a SATA hard disk and matching SATA data transfer cable (at least for Dell 390, T3400 and T3500 computers, the connector plug on the hard disk side must have a 90 degree angle, otherwise the computer's cover cannot be closed after mounting the hard disk).
- 2 Shut the computer down and power it off.
- 3 Mount the hard disk using the mounting bracket inside the computer and attach power and SATA cables to the drive. Connect the SATA cable to an unused SATA channel.
- 4 Power the computer up and press F2 to enter the BIOS setup.
- 5 Go to the "Drives" menu and switch on the SATA channel to which you connected the SATA cable (for example SATA-1). Save the BIOS settings and reboot the computer.

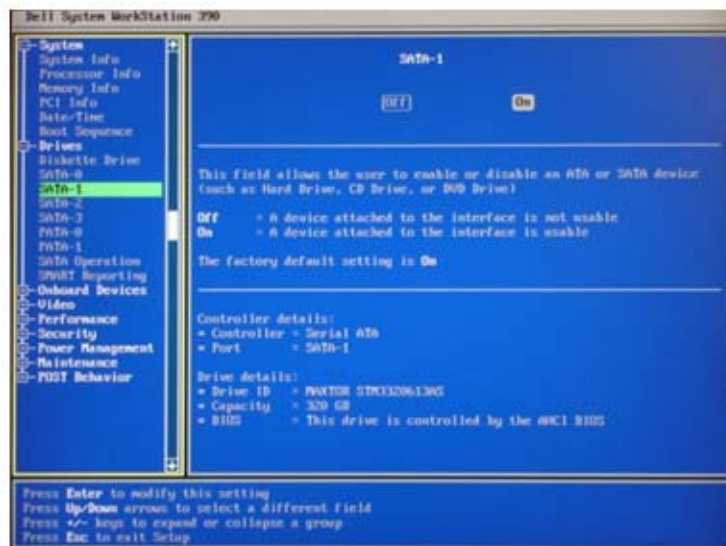
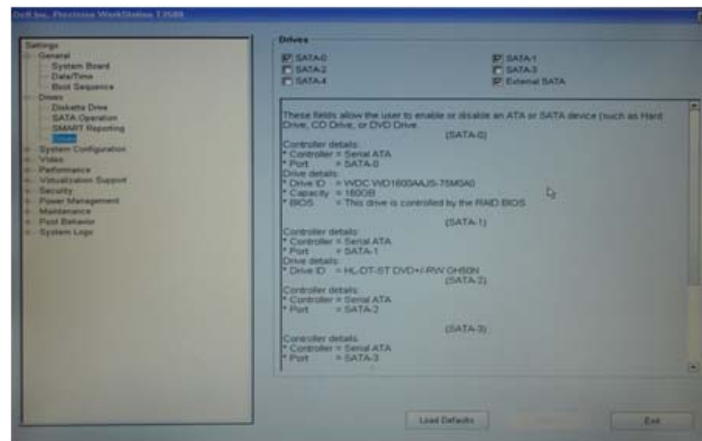
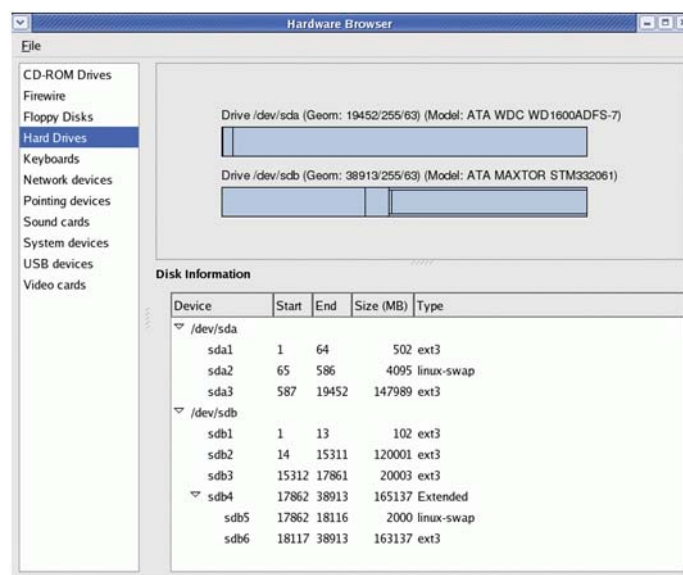


Figure 3 BIOS Drives screen of a Dell 390N



**Figure 4** BIOS Drives screen of a Dell T3500

- 6 Check if the second hard disk is detected by Linux: Open the hardware browser by selecting “System ? Administration ? Hardware” from the Linux menu. Select “Hard drives”. The second hard disk should typically be recognized as device `/dev/sdb` (the first hard disk is `/dev/sda`). The following screenshot shows a second drive `/dev/sdb` that had already been formatted to contain three partitions (if the drive does not contain any partitions yet, it may appear in the “Disk information” list but not graphically on top):



- 7 Format the disk. Enter:

```
fdisk /dev/sdb
```

Note: Make sure you do NOT type here the name of the system hard disk – typically “/dev/sda” as you can erase your main hard disk! If you are unsure about this operation, seek advice from Agilent.

- Type “p” to show the current partition table (type “m” to see all commands)
- If any partitions are present on the second hard disk, delete them with “d”
- Now create a new primary partition (typically, fill the backup hard disk with a single partition). Enter “n” for a new partition, followed by “p” for a primary partition, “1” for the first partition. Choose the size in cylinders (typically, hit “Return” to fill the entire disk).
- Enter “p” again to verify the partition(s) has/have been created as desired. If not, delete the partition(s) and create afresh.

Note: all changes so far are done in memory – only the next step actually writes them to the hard disk header.

- Enter “w” to write this partition table to disk and exit.
- 8** Now create a filesystem on the new partition(s). Type, for example:

```
mkfs.ext3 /dev/sdb1
```

Note: This may take a few minutes.

- 9** The new partition should be mounted on system bootup. To achieve this, enter the partition into the filesystem table /etc/fstab. Open the file with a text editor like “gedit”:

```
gedit /etc/fstab
```

Add a new line that reads like this (adapt for your requirements):

```
/dev/sdb1/backupext3defaults 0 0
```

- 10** If it doesn’t exist yet, create a directory that will be the mount point for the hard disk, like

```
mkdir /backup
```

- 11** You can now mount the new partition without having to reboot the computer (at next reboot, the partition will be mounted automatically). Type:

```
mount /backup
```

The partition should show up correctly now when checking the filesystems. To check, type:

```
df -h
```

to display the filesystems in human-readable format (MB, GB etc).

Your hard disk should now be set up to be used as a backup medium.

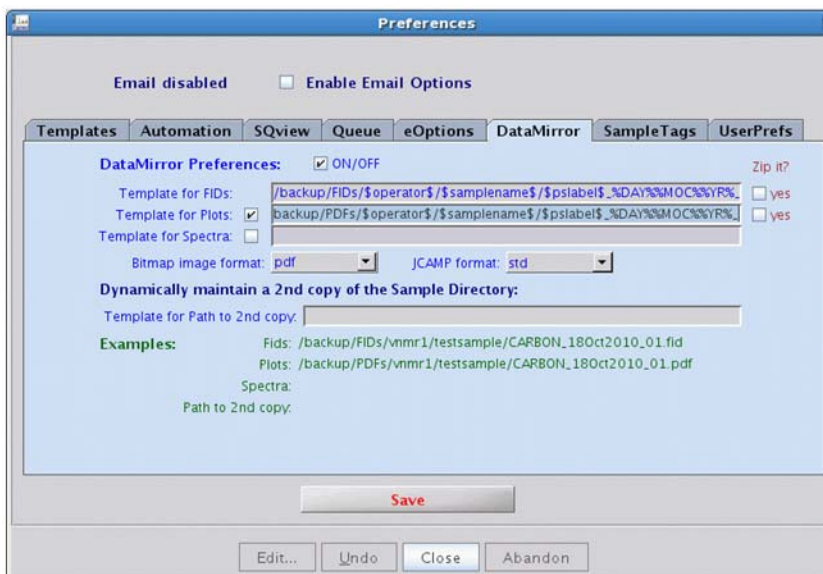
## Data mirroring using VnmrJ tools

VnmrJ 3 contains a functionality to routinely mirror NMR data to a second harddrive or remotely mounted filesystem. It is activated from the VnmrJ Preferences (under the Edit menu), Data Mirror tab. Data backup using mirroring from VnmrJ requires the presence (and setup) of either a second backup hard disk or a mounted remote filesystem on another computer. The data will be backed up in a synchronous way – each time an FID is saved on disk, it will be saved (mirrored) in the second location as well. Please refer also to the “Automation User Guide”, chapter 9.5.

*Advantages:* Easy to set up from VnmrJ itself.

*Disadvantages:* In the case of a network mounted filesystem, if the network connection should be lost before or during data transfer, all data attempted to mirror will not be transferred and will be missing from the backup. In such a case the missing data will have to be copied over by hand as soon as the network is up again. Hence, this type of data mirroring should only be used if the “uptime” of the network can be guaranteed to a very high degree or if the target medium is a second harddrive inside or attached to the spectrometer host.

To use data mirroring from VnmrJ, first install and/or setup your backup medium that you want to mirror the data to. Please refer to section 6.6.2 on how to do this. Then use the VnmrJ Preferences to set up the data mirroring:



- 1 Open the Preferences window from the Edit ? Preferences menu (see also the “Automation” manual).
- 2 Select the “DataMirror” tab.
- 3 If not done already, switch on data mirroring by selecting the “DataMirror Preferences” checkbox on top.
- 4 Tick the checkbox of all types of data that shall be mirrored: FIDs, PDFs etc. File compression via ZIP can also be selected by the “Zip it?” checkboxes on the right.
- 5 For every data type, enter a save template, that includes the path to the backup medium. The mirror template can be similar to the normal data save template (as set up on the “Templates” tab) like, in this example,

```
/backup/FIDs/$operator$/$samplename$/$pslabel$_%D
AY%%MOC%%YR%_
```

(or whatever your local templates look like) but also could be set up completely differently from the data save templates. Refer to the autoname entry in the Command & Parameter Reference manual or type

```
man('autoname')
```

in the VnmrJ command line to see details on template usage.

## Automatic data backup using rsync

This is a description on how to set up automated file saving



between two hard disks or Linux computers using remote synchronization – rsync. One hard disk or computer is the client, the other acts as a (file) server.

*Advantages:* Safe, works even if network connection is intermittent. Fast. Data transfer can be piped through secure shell (SSH) and therefore encrypted. Can be set up to run at specific intervals or dates.

*Disadvantages:* Requires special setup described here.

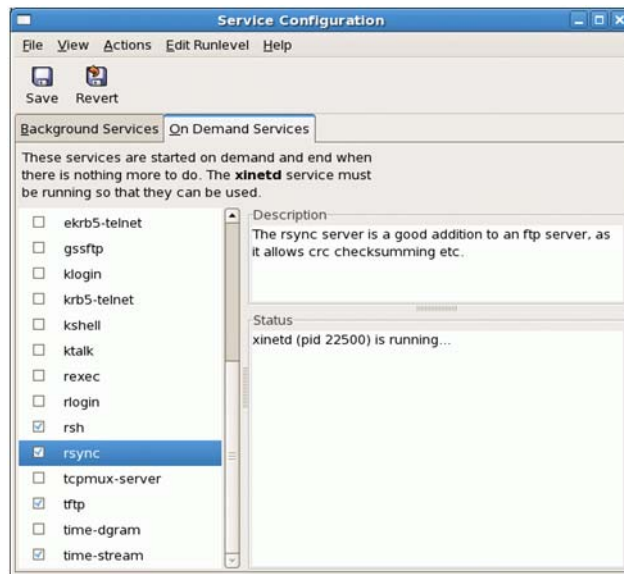
rsync compares the date stamps of all files in two directories – the source and the target directory. If the source directory contains newer files (like recently touched/updated files) than the target directory, only those files will be copied, preserving (in “archive mode”) all file settings like owner, group and date.

For checking the date of the relevant files, rsync taps into the internal filesystem table which already contains this information. Hence, rsync runs very fast and uses less resources than typical FTP scripts that have to find out this information actively at every runtime. Typically, rsync is run via a crontab script once or several times per day (like once per hour or even more often).

Because of its “passive” behavior, rsync will still work if, for example, a mounted drive on another network computer is unavailable for a certain time (network down): If the drive cannot be found, no data are copied. As soon as the network is available again, more “new” data are found and the complete backlog is copied to the backup drive without further intervention. Because of this intrinsically higher data safety rsync is normally preferable over “simple” data mirroring (see section 6.6.3), but it requires more steps to set it up, as described in this chapter.

A prerequisite for rsync to work is that the rsync server daemon is active on the computer that actively sends or fetches the data. This can be checked by going to the System > Administration > Server Settings > Services menu.

With RedHat Linux versions before 5.3, the rsync service is found in the main list whereas with RHEL 5.3, it is under the “On Demand Services” tab:



If the service is not active yet, click the checkbox and press “Save”.

With RedHat versions before 5.3, it is also necessary to restart the xinet daemon if you newly added rsync to the running servers. To restart xinet, select it from the list and click the “Restart” button. With later RHEL versions this is not required as rsync runs on demand.

## Backup on a second hard disk

Please see section 6.6.2 on how to prepare a second hard disk for backup use. After preparation, the data backup can be set up:

Create a cron job doing an rsync call every few minutes (rsync is very fast from hard disk to hard disk and can run e.g. all 5-10 minutes). Add a line to the file /etc/crontab reading like this (in this example, VnmrJ was set up such that the NMR data is written to /home/data):

```
* /5 * * * * user rsync -aqu /home/data/ /backup
a  b c d e f   g   h   i           j
```

The arguments in the crontab file are:

**a** Minute (0-59) – “5” would result in the command being executed at every 5th minute of every hour, “\*/5” would result in command execution every five minutes.

**b** Hour (0-23) – “\*” means execution at every hour of the day, \*/5 would result in command execution every five hours.

**c** Day of the month (1-31) – “\*” means execution at every day of the month, \*/5 would result in command execution every five days.

**d** Month of the year – “\*” means execution at every month of the year.

**e** Day of the week (0-6, Sunday=0) – “\*” means execution at every day of the week

**f** User as who the command should be executed (for example “root” or “vnmr1”)

**g** Command to be executed

**h** Command arguments (here, “a” = archive, “q” = quiet, “u” = ...

**i** Directory to be backed up (note the trailing “/”- this is important if only the content of the /home/data directory shall be copied to /backup – not the directory “data” itself)

**j** Target directory (can have a trailing “/”, but does not have to)

The moment the new entry is saved to /etc/crontab, it will start executing. If “\*/5” is selected for the minutes, for example, data backup should start five minutes later.

Another possibility to create crontab jobs is by typing

```
crontab -e
```

in a terminal shell. This will create a personal crontab (instead of a system wide crontab when editing /etc/crontab). In this case, the “user” entry is not required as the user who typed “crontab -e” is the user will be executing the command contained in the crontab entry.

To check whether a personal crontab is already running, type

```
crontab -l
```

See also `man(crontab)` for more options.

Note: Make sure the last item in the crontab line (here “/backup”) does NOT end with an (accidentally added, for example by copy-n-paste) empty string! The rsync command

will not be executed in this case.

## Backup on a remote mounted filesystem

Data backup from a spectrometer host to a remote mounted filesystem can be done in two ways:

Option A: The spectrometer host copies “pushes” the data to the backup server

Option B: The backup server copies “pulls” the data from the spectrometer host

### Option A

This setup allows backing up data from the spectrometer host to a file server which exports one of its hard disks/partitions to the network via NFS:

- 1 Log in as root user on the spectrometer host and create a new directory that will be used as mount point for the backup harddrive, for example:

```
mkdir /backup
```

- 2 Edit the file `/etc/fstab` to create a new entry for the remote mount point. Add a new line similar to this example:

```
backup_server:/target_directory /backup nfs
defaults 0 0
```

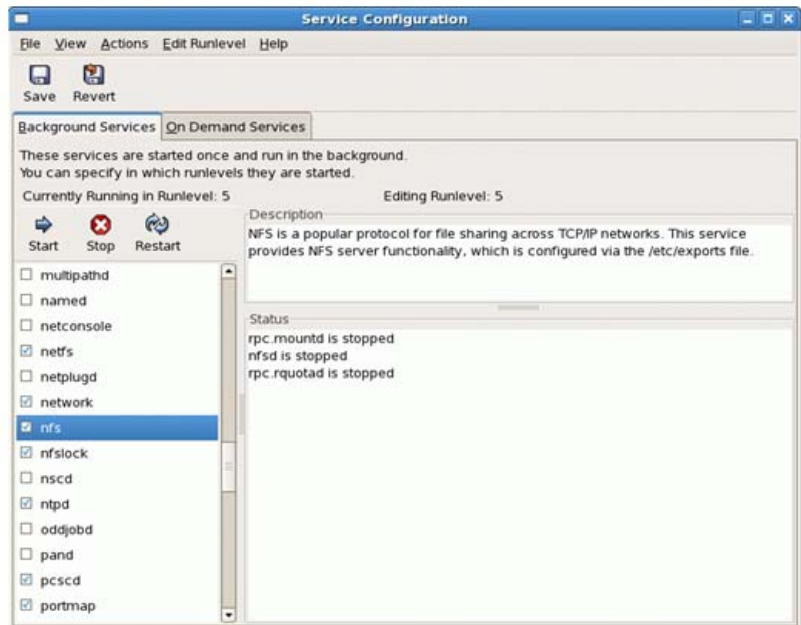
- 3 Try out if you can already mount it (the entry in `/etc/fstab` will make sure it is mounted upon every reboot):

```
mount /backup
```

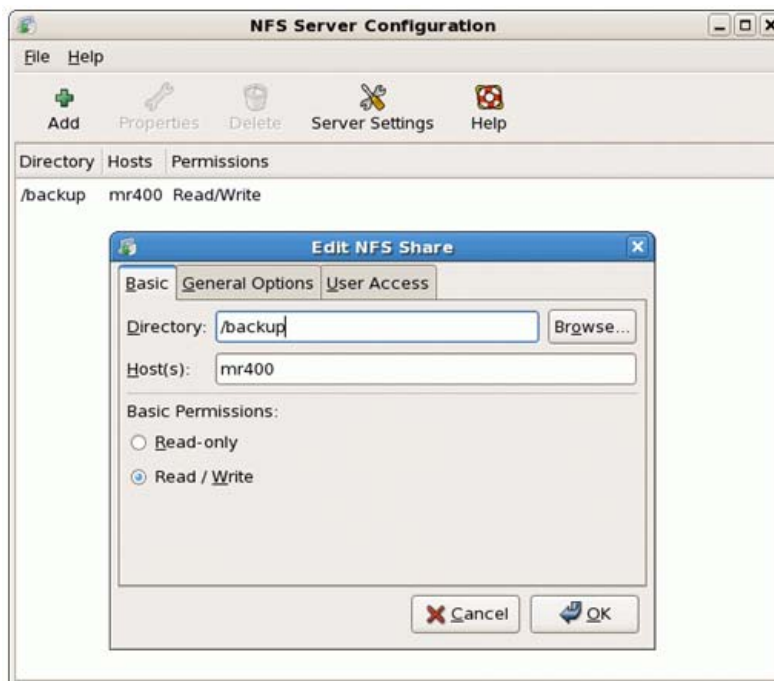
If it succeeds, continue with step 4)

If it fails, maybe the target directory hasn't been exported properly. Here is what needs to be done to export a directory via NFS if the remote computer is a (Red Hat) Linux machine as well:

- a On the (Red Hat) Linux backup server, log in as root and open the System > Administration > Server Settings > Services menu
- b Check if the “nfs” service is running. If not, select the checkbox and click “Start” and “Save”:



- c Now open the System > Administration > Server Settings > NFS:



- d Click “Add” to export a new directory via NFS.
  - e Select the Directory to be exported
  - f f read/write access shall be restricted to a few (spectrometer) hosts, enter their names or IP addresses into the “Host(s)” field.
  - g Select “Read/Write” permissions.
  - h If you want all other network computers to have Read-only access, add a new NFS entry for the same directory but with a “\*” in the “Host(s)” field and selecting “Read-only” permissions.
- 4) Create a cron job doing an rsync call every few minutes. Add a line to the file /etc/crontab reading something like this (see section “Backup to a second hard disk” above for a detailed description of the crontab file, VnmrJ was set up such that the NMR data is written to /home/data):

```
*/5 * * * * user rsync -aqu /home/data/ /backup
```

Note: Make sure the last item in the crontab line (here “/backup”) does NOT end with an (accidentally added, for example by copy-n-paste) empty string! The rsync command will not be executed in this case.

### Option B

The second setup backs up data by “pulling” them from the spectrometer host to the file server. Now the spectrometer host has to export its hard disk/partitions to the network via NFS:

- 1 Log in as root user on the backup server and create a new directory that will be used as mount point for the backup harddrive, for example:

```
mkdir /backup
```

- 2 Create a directory where the backup should be stored, for example “/home/backup/400MR”:

```
mkdir /home/backup/400MR
```

- 3 Edit the file /etc/fstab to create a new entry for the remote mount point. Add a new line similar to this example:

```
Spectrometer_host:/source_directory /backup nfs  
defaults 0 0
```

In the present example, the entry could therefore be:

```
400MR:/home/data /backup nfs defaults 0 0
```

- 4 Try out if you can already mount it (the entry in /etc/fstab will make sure it is mounted upon every reboot):

```
mount /backup
```

If it succeeds, continue with step 5)

If it fails, maybe the target directory hasn't been exported properly. Here is what needs to be done to export a directory via NFS if the remote computer is a (Red Hat) Linux machine as well (for screenshots see part A) above):

- a On the (Red Hat) Linux backup server, open the System > Administration > Server Settings > Services menu
  - b Check if the "nfs" service is running. If not, select the checkbox and hit "Start" and "Save":
  - c Now open the System > Administration > Server Settings > NFS.
  - d Click "Add" to export a new directory via NFS.
  - e Select the Directory to be exported
  - f If read/write access shall be restricted to a few (spectrometer) hosts, enter their names or IP addresses into the "Host(s)" field.
  - g Select "Read/Write" permissions.
  - h If you want all other network computers to have Read-only access, add a new NFS entry for the same directory but with a "\*" in the "Host(s)" field and selecting "Read-only" permissions.
- 5 Create a cron job doing an rsync call every few minutes. Add a line to the file /etc/crontab reading something like this (see section "Backup to a second hard disk" above for a detailed description of the crontab file). Now "/backup" is the source and "/home/backup/400MR" the target directory:

```
*/5 * * * * user rsync -aqu /backup/ /home/backup/400MR
```

The moment the new entry is saved to /etc/crontab, it will start executing. If "\*/5" is selected for the minutes, for example, data backup should start five minutes later.

## Backup on a remote computer via Secure Shell (ssh)

In certain situations (mostly because of safety concerns, firewalls etc.), backup server access cannot be granted directly via NFS but data traffic and user login has to be encrypted. This can be achieved using Secure Shell (SSH).

As SSH requires user login, each operation would require typing the password, making an automated backup essentially impossible. To still allow a safe SSH login without having to enter the password, a highly encrypted



private/public key pair can be generated that replaces the password.

Data backup from a spectrometer host to a remote mounted filesystem can again be done in two ways, only it is done via SSH:

Option A: The spectrometer host copies “pushes” the data to the backup server

Option B: The backup server copies “pulls” the data from the spectrometer host

Which of the two options should be used is usually determined by the question whether the rsync service is available on the backup server or not. Often, SSH is possible even on Microsoft Windows-based machines while both SSH and rsync are typically available on Unix-based servers only.

### Option A

This setup allows backing up data from the spectrometer host to a remote file server by “pushing” the data via rsync and SSH. The following recipe only works if the file server is also a Linux or Unix computer. In this example, the user “vnmr1” shall back up the content of /home/data on the NMR spectrometer host “400MR” to the “/home/backup” directory of the backup server “NMRbackup”.

- 1 Log in as root on the spectrometer host (“400MR”).
- 2 Make sure both rsync and ssh are enabled by checking the "System > Administration > Server Settings > Services" tool: Choose "rsync" and "ssh", save (start the daemons if necessary) and exit.
- 3 Check if SSH and rsync have been enabled on the remote backup server (this typically has to be checked with the local IT department, a user login also must be provided – let’s assume a username “backupuser”).
- 4 Generate a private/public key pair on the NMR host:  
Type:

```
mkdir /home/nmruser/cron
cd /home/nmruser/cron
ssh-keygen -t dsa -b 2048 -f nmrhost_rsync_key
```

(“nmruser” would here be “vnmr1”, while “nmrhost” is the network name of the NMR spectrometer host, here, “400MR”)

The output should be:

**Generating public/private dsa key pair.****Enter passphrase ...:** (press enter here)**Enter same passphrase again...:** (press enter again)

- 5 Now copy the public key to the backup server:

```
scp nmrhost_rsync_key.pub serveruser@serverhost:/home/
serveruser
```

("serveruser" being the backup user name, here  
"backupuser" on the backup server and "serverhost" its  
hostname, here "NMRbackup")

```
ssh serveruser@serverhost (type in password)
```

```
mkdir .ssh (if it doesn't exist)
```

```
mv nmrhost_rsync_key.pub .ssh/
```

```
cd .ssh
```

```
touch authorized_keys
```

```
chmod 600 authorized_keys
```

```
cat nmrhost_rsync_key.pub >> authorized_keys
```

If more than one NMR spectrometer host shall send its backup data to the server, repeat steps 4) and 5) on the other NMR hosts and copy their nmrhost\_rsync\_key.pub public keys to the backup server.

- 6 Now test rsync via ssh (should NOT ask for a password anymore). Log in to the spectrometer host again and type (all one line):

```
rsync -aqu -e "ssh -i /home/nmruser/cron/
nmrhost_rsync_key"/nmrhost_dir
/serveruser@serverhost:/serverhost_dir
```

In the present example, this would be (all one line):

```
rsync -aqu -e "ssh -i
/home/vnmr1/cron/400MR_rsync_key"/home/data/
backupuser@NMRbackup:/home/NMRbackup
```

If this still asks for a password, it is likely that some permissions on the private key (on the NMR host), of the public "authorized\_keys" file, .ssh directory and home directory (on nmrhost) are not set securely enough: Neither "group" nor "other" should have write permissions on any of these files or directories. In this case, ssh deems the procedure not to be secure and asks for the password.

- 7 If data backup has started in the correct way, you can now create the crontab entry. Add a line to the file

/etc/crontab of the spectrometer host reading something like this (see section “Backup to a second hard disk” above for a detailed description of the crontab file):

```
* /5 * * * * nmruser rsync -aqu -e "ssh
-i /home/nmruser/cron/nmrhost_rsync_key"/nmrhost_d
ir /serveruser@serverhost:/serverhost_dir
```

In the present example, this would be (Note: The lines above and below represent one single line.):

```
* /5 * * * * vnmr1 rsync -aqu -e "ssh -i
/home/vnmr1/cron/400MR_rsync_key"/home/data/
backupuser@NMRbackup:/home/NMRbackup
```

If more than one NMR spectrometer host shall send its backup data to the server, add the same line to the crontab file of the other NMR hosts as well. Try to avoid overlapping the backup times of the different computers – use, for example “5 \* \* \* \*” for NMR host 1, “10 \* \* \* \*” for NMR host 2 etc. to start their backups at 5 minutes past the hour for NMR host 1, 10 minutes past the hour for NMR host 2 etc.

Note: Make sure the last item in the crontab line (here “/backup”) does NOT end with an (accidentally added, for example by copy-n-paste) empty string! The rsync command will not be executed in this case.

## Option B

The second setup backs up data by “pulling” them from the spectrometer host to the file server via rsync and SSH. The following recipe only works if the file server is also a Linux or Unix computer. In this example, the user “backupuser” shall back up the content of /home/data on the NMR spectrometer host “400MR” to the “/home/backup” directory of the backup server “NMRbackup” and is hence requires the inverse process of setup A).

- 1 Log in as root on the spectrometer host (“400MR”).
- 2 Make sure both rsync and ssh are enabled by checking the "System > Administration > Server Settings > Services" tool: Choose "rsync" and "ssh", save (start the daemons if necessary) and exit.
- 3 Check if SSH and rsync have been enabled on the remote backup server (this typically has to be checked with the local IT department, a user login also must be provided – let’s assume a username “backupuser”).

- 4 Generate a private/public key pair on the backup server. Log in to the backup server as the provided serveruser login (here “backupuser”). Type:

```
mkdir /home/serveruser/cron
cd /home/serveruser/cron
ssh-keygen -t dsa -b 2048 -f serverhost_rsync_key
```

(“serverhost” is the network name of the backup server, here, “NMRbackup”)

The output should be:

**Generating public/private dsa key pair.**

**Enter passphrase ...:** (press enter here)

**Enter same passphrase again...:** (press enter again)

- 5 Now copy the public key to the NMR host (all one line):

```
scp serverhost_rsync_key.pub nmruser@nmrhost:
/home/nmruser
```

(“nmruser” being the NMR user name, here “backupuser” on the backup server and “serverhost” its hostname, here “NMRbackup”)

```
ssh nmruser@nmrhost (type in password)
mkdir .ssh (if it doesn't exist)
mv serverhost_rsync_key.pub .ssh/
cd .ssh
touch authorized_keys
chmod 600 authorized_keys
cat serverhost_rsync_key.pub >> authorized_keys
```

If the data of more than one NMR spectrometer host shall be “pulled” to the backup server, repeat step 5) for the other NMR hosts and copy the serverhost\_rsync\_key.pub public key to the other NMR spectrometer hosts.

- 6 Now test rsync via ssh (should NOT ask for a password anymore). Log in to the backup server again and type (all one line):

```
rsync -aqu -e "ssh -i
/home/serveruser/cron/serverhost_rsync_key"
/nmruser@nmrhost:/nmrhost_dir /server_dir
```

In the present example, this would be be (all one line):

```
rsync -aqu -e "ssh -i
/home/backupuser/cron/NMRbackup_rsync_key"
vnmr1@400MR:/home/data/ /home/backup
```

If this still asks for a password, it is likely that some permissions on the private key (on the backup server), of the public "authorized\_keys" file, .ssh directory and home directory (on nmrhost) are not set securely enough: Neither "group" nor "other" should have write permissions on any of these files or directories. In this case, ssh deems the procedure not to be secure and asks for the password.

- 7 If data backup has started in the correct way, you can now create the crontab entry. Add a single line to the file /etc/crontab of the backup server reading something like this (see section "Backup to a second hard disk" above for a detailed description of the crontab file):

```
*/5 * * * * serveruser rsync -aqu -e "ssh -i
/home/serveruser/cron/serverhost_rsync_key"
/nmruser@nmrhost:/nmrhost_dir /server_dir
```

In the present example, this would be (Note: The lines above and below represent one single line!):

```
*/5 * * * * backupuser rsync -aqu -e "ssh -i
/home/backupuser/cron/NMRbackup_rsync_key"
vnmr1@400MR:/home/data/ /home/backup
```

If data of more than one NMR spectrometer host shall be "pulled" to the backup server, add more lines to the crontab file of the server, one per spectrometer host. Try to avoid overlapping the backup times of the different computers – use, for example "5 \* \* \* \*" for NMR host 1, "10 \* \* \* \*" for NMR host 2 etc. to start their backups at 5 minutes past the hour for NMR host 1, 10 minutes past the hour for NMR host 2 etc.





## 5 Parameters and Data

VnmrJ Data Files	440
FDF (Flexible Data Format) Files	452
Reformatting Data for Processing	459
Creating and Modifying Parameters	464
Modifying Parameter Displays in VNMR	473
Modules	478
User-Written Weighting Functions	480
User-Written	484



## VnmrJ Data Files

Although a number of different files are used by VnmrJ to process data, VnmrJ data files use only two basic formats:

*Binary format* – Stores

FIDs and transformed spectra. Binary files consist of a file header describing the details of the data stored in the file, followed by the spectral data in integer or floating point format.

*Text*

*format* – Stores all other forms of data, such as line lists, parameters, and all forms of reduced data obtained by analyzing NMR spectra. The advantage of storing data in text format is that it can be easily inspected and modified with a text editor and can be copied from one computer to another with no major problems. The text on Sun systems use the ASCII format in which each letter is stored in one byte.

### Binary data files

Binary data files are used in the VnmrJ file system to store FIDs and the transformed spectra. FIDs and their associated parameters are stored as `filename.fid` files. A `filename.fid` file is always a directory file containing the following individual files:

- `filename.fid/fid` is a binary file containing the FIDs.
- `filename.fid/procpar` is a text file with parameters used to obtain the FIDs.
- `filename.fid/text` is a text file.

In experiments, binary files store FIDs and spectra. In non-automation experiments, the FID is stored within the experiment, regardless of what the parameter `file` is set to. The path `~username/vnmrsys/expn/acqfil/fid` is the full Linux path to that file. FIDs are stored as either 16- or 32-bit integer binary data files, depending on whether the data acquisition was performed with `dp='n'` or `dp='y'`, respectively.

After a Fourier transform, the experiment file `expn/datdir/data` contains the transformed spectra stored in a 32-bit floating point format. This file always contains complex numbers (pairs of floating point numbers), except



if `pmode=''` was selected in processing 2D experiments. To speed up the display, VnmrJ also stores the phased spectral information in `expn/datdir/phasefile`, where it is available only after the first display of the data. In arrayed or 2D experiments, `phasefile` contains only those traces that have been displayed at least once after the last FT or phase change. Therefore, a user program to access that file can only be called after a complete display of the data.

The directory file, `expn` for current experiment *n*, typically contains the following files:

`expn/curpar` is a text file containing the current parameters.

- `expn/procpar` is a text file containing the last used parameters.
- `expn/text` is a text file.
- `expn/acqfil/fid` is a binary file that stores the FIDs.
- `expn/datdir/data` is a binary file with transformed complex spectrum.
- `expn/datdir/phasefile` is a binary file with transformed phased spectrum.
- `expn/sn` is saved display number *n*.

To access information from one of the experiment files of the current experiment, the user must be sure that each of these files has been written to the disk. The problem arises because VnmrJ tries to keep individual blocks of the binary files in the internal buffers as long as possible to minimize disk accesses. This buffering in memory is not the same as the disk cache buffering that the Linux operating system performs. The `flush` command can be used in VnmrJ to write all data buffers into disk files (or at least into the disk cache, where it is also available for other processes). The command `fsave` can be used in VnmrJ to write all parameter buffers into disk files.

The default directory for the 3D spectral data is `curexp/datadir3d`. The output directory for the extracted 2D planes is the same as that for the 3D spectral data, except that 2D uses the `/extr` subdirectory and 3D uses the `/data` subdirectory. Following are the files and further subdirectories within the `/data` 3D data subdirectory:

- `data1` to `data#` are the actual binary 3D spectral data files. If the option `nfiles` is not entered, the number of data files depends upon the size of the largest 2D plane and the value for the Linux environmental parameter `memsize`.

- `info` is a directory that stores the 3D coefficient text file (`coef`), the binary information file (`procdat`), the 3D parameter set (`procpar3d`), and the automation file (`auto`). The first three files are created by the `set3dproc()` command within VnmrJ. The last file is created by the `ft3d` program.
- `log` is a directory that stores the log files produced by the `ft3d` program. The file `f3` contains all the log output for the  $f_3$  transform. For the  $f$  and  $f$  transforms, there are two log files for each data file- one for the  $f_2$  transform (`f2.#`) and one for the  $f_1$  (`f1.#`). The file `master` contains the log output produced by the master `ft3d` program.

## Data file structures

A data file header of 32 bytes is placed at the beginning of a VnmrJ data file. The header contains information about the number of blocks and their size. It is followed by one or more data blocks. At the beginning of each block, a data block header is stored, which contains information about the data within the individual block. A typical 1D data file, therefore, has the following form:

```
data file header
header for block 1
data of block 1
header for block 2
data of block 2
. . .
```

The data headers allow for 2D hypercomplex data that may be phased in both the  $f_1$  and  $f_2$  directions. To accomplish this, the data block header has a second part for the 2D hypercomplex data. Also, the data file header, the data block header, and the data block header used with all data have been slightly revised. The new format allows processing of FIDs obtained with earlier versions of VnmrJ. The 2D hypercomplex data files with `datafilehead.nbheaders=2` have the following structure:

```
data file header
header for block 1
second header for block 1
data of block 1
header for block 2
second header for block 2
```

data of block 2

. . .

All data in this file are contiguous. The byte following the 32nd byte in the file is expected to be the first byte of the first data block header. If more than one block is stored in a file, the first byte following the last byte of data is expected to be the first byte of the second data block header. Note that these data blocks are not disk blocks; rather, they are a complete data group, such as an individual trace in an experiment. For non-arrayed 1D experiments, only one block will be present in the file.

Details of the data structures and constants involved can be found in the file `data.h`, which is provided as part of the VnmrJ source code license. The C specification of the file header is the following:

```
struct datafilehead
/* Used at start of each data file (FIDs, spectra, 2D) */
{
long nblocks;    /* number of blocks in file */
long ntraces;   /* number of traces per block */
long np;        /* number of elements per trace */
long ebytes;    /* number of bytes per element */
long tbytes;    /* number of bytes per trace */
long bbytes;    /* number of bytes per block */
short vers_id;  /* software version, file_id status bits
*/
short status;   /* status of whole file */
long nbheaders; /* number of block headers per block */
};
```

The variables in `datafilehead` structure are set as follows:

- `nblocks` is the number of data blocks present in the file.
- `ntraces` is the number of traces in each block.
- `np` is the number of simple elements (16-bit integers, 32-bit integers, or 32-bit floating point numbers) in one trace. It is equal to twice the number of complex data points.
- `ebytes` is the number of bytes in one element, either 2 (for 16-bit integers in single precision FIDs) or 4 (for all others).
- `tbytes` is set to  $(np * ebytes)$ .
- `bbytes` is set to  $(ntraces * tbytes + nbheaders * sizeof(struct datablockhead))$ . The size of the `datablockhead` structure is 28 bytes.

- `vers_id` is the version identification of present VnmrJ.
- `nbheaders` is the number of block headers per data block.
- `status` is bits as defined below with their hexadecimal values.

All other bits must be zero.

**Table 47** Bits 0–6: file header and block header status bits (bit 6 is unused)

0	S_DATA	0x1	0 = no data, 1 = data
1	S_SPEC	0x2	0 = FID, 1 = spectrum
2	S_32	0x4	*
3	S_FLOAT	0x8	0 = integer, 1 = floating point
4	S_COMPLEX	0x10	0 = real, 1 = complex
5	S_HYPERCOMPLEX	0x20	1 = hypercomplex

\* If S\_FLOAT=0, S\_32=0 for 16-bit integer, or S\_32=1 for 32-bit integer.

If S\_FLOAT=1, S\_32 is ignored.

**Table 48** Bits 7–14: file header status bits (bits 10 and 15 are unused)

7	S_ACQPAR	0x80	0 = not Acqpar, 1 = Acqpar
8	S_SECND	0x100	0 = first FT, 1 = second FT
9	S_TRANSF	0x200	0 = regular, 1 = transposed
11	S_NP	0x800	1 = np dimension is active
12	S_NF	0x1000	1 = nf dimension is active
13	S_NI	0x2000	1 = ni dimension is active
14	S_NI2	0x4000	1 = ni2 dimension is active

Block headers are defined by the following C specifications:

```
struct datablockhead
/* Each file block contains the following header */
{
short scale;      /* scaling factor */
short status;    /* status of data in block */
short index;     /* block index */
short mode;      /* mode of data in block */
long ctcount;   /* ct value for FID */
float lpval;     /* f2 (2D-f1) left phase in phasefile */
float rpval;     /* f2 (2D-f1) right phase in phasefile */
float lvl;       /* level drift correction */
float tlt;       /* tilt drift correction */
};
```

status is bits 0–6 defined the same as for file header status. Bits 7–11 are defined below (all other bits must be zero):

**Table 49** Bits 7–11

7	MORE_BLOCKS	0x80	0 = absent, 1 = present
8	NP_CMPLX	0x100	0 = real, 1 = complex
9	NF_CMPLX	0x200	0 = real, 1 = complex
10	NI_CMPLX	0x400	0 = real, 1 = complex
11	NI2_CMPLX	0x800	0 = real, 1 = complex

Additional data block header for hypercomplex 2D data:

```
struct hypercplxhead
{
short s_spare1;   /* short word: spare */
short status;    /* status word for block header */
short s_spare2;   /* short word: spare */
short s_spare3;   /* short word: spare */
long l_spare1;   /* long word: spare */
float lpval1;    /* 2D-f2 left phase */
float rpval1;    /* 2D-f2 right phase */
float f_spare1;  /* float word: spare */
float f_spare2;  /* float word: spare */
};
```

Main data block header mode bits 0–15:

**Table 50** Bits 0–3: bit 3 is currently unused

0	NP_PHMODE	0x1	1 = ph mode
1	NP_AVMODE	0x2	1 = av mode
2	NP_PWRMODE	0x4	1 = pwr mode

**Table 51** Bits 4–7: bit 7 is currently unused

4	NF_PHMODE	0x10	1 = ph mode
5	NF_AVMODE	0x20	1 = av mode
6	NF_PWRMODE	0x40	1 = pwr mode

**Table 52** Bits 8–11: bit 11 is currently unused

8	NI_PHMODE	0x100	1 = ph mode
9	NI_AVMODE	0x200	1 = av mode
10	NI_PWRMODE	0x400	1 = pwr mode

**Table 53** Bits 12–15: bit 15 is currently unused

12	NI2_PHMODE	0x8	1 = ph mode
13	NI2_AVMODE	0x100	1 = av mode
14	NI2_PWRMODE	0x2000	1 = pwr mode

**Table 54** Usage bits for additional block headers  
(hypercmplxhead.status)

U_HYPERCOMPLEX	0x2	1 = hypercomplex block structure
----------------	-----	----------------------------------

The actual FID data are typically stored as pairs of floating-point numbers. The first represents the real part of a complex pair and the second represents the imaginary component. In phase-sensitive 2D experiments, "X" and "Y" experiments are similarly interleaved. The format of the data points and the organization as complex pairs must be specified in the data file header.

## VnmrJ use of binary data files

To understand how VnmrJ uses individual binary data files, consider the example of a simple Fourier transform followed by the display of the spectrum. The FT is performed with the command `ft`, which does the following:

- Copy processing parameters from `curpar` into `propar`.
- If FID is not in the `fid` file buffer, open the `fid` file (if not already open) and load it into buffer. Initialize the data file with the proper size (using parameter `fn`). Store the FID in the data file buffer.
- Apply dc drift correction and first point correction.
- Apply weighting function, if requested.
- Zero fill data, if required.
- Fourier transform data in data file buffer.

At this point, the data file buffer contains the complex spectrum. Unless other FTs are done, which use up more memory space than assigned to the data file buffer, the data is not automatically written to the file `expn/datdir/data` at this time. Joining a different experiment or the command `flush` performs such a write operation.

The `ds` command takes the following steps in displaying the spectrum:

- 1 If the data is not present in the `phasefile` buffer, or if the phase parameters have changed, `ds` tries to open the phase file (if not already open) and load data into the buffer (if the `phasefile` is present). If `ds` is unsuccessful, the data must be phased:
  - a If the data is not in the data file buffer, `ds` opens the data file (if not already open) and loads it into the buffer.
  - b `ds` initializes the `phasefile` buffer with the proper size (using the same parameter `fn` as used for last FT).
  - c `ds` calculates the phased (or absolute value) spectrum and stores it in the `phasefile` buffer.
- 2 `ds` calculates the display and displays the spectrum.

The `phasefile` buffer now contains the phased spectrum. Unless other displays are done, which use up more memory space than assigned to the `phasefile` buffer, the data is not automatically written to the file `expn/datdir/phasefile` at this time. Joining a different experiment or entering the command `flush` performs such a write operation.

Depending on the nature of the data processing, the two files `data` and `phasefile` will contain different information, as follows:

- *After a 1D FT* - `data` contains a complex spectrum, which can be used for phased or absolute value displays.
- *After a 1D display* - `phasefile` contains either phased or absolute value data, depending on which type of display had been selected.
- *After a 2D FID display* - `data` contains the complex FIDs, floated and normalized for different scaling during the 2D acquisition. `phasefile` contains the absolute value or phased equivalent of this FID.
- *After the first FT in a 2D experiment* - `data` contains the once-transformed spectra. This is equivalent to the interferograms, if the `data` file is properly reorganized (see  $f_1$  and  $f_2$  traces in “[Storing multiple traces](#)”). If a display is done now, `phasefile` contains phased (or absolute value) half-transformed spectra or interferograms.
- *After the second FT in a 2D experiment* - `data` contains the fully transformed spectra, and after a display, `phasefile` contains the equivalent phased or absolute-value spectra.

## Storing multiple traces

Arrayed experiments are handled in VnmrJ by storing the multiple traces of arrayed experiments in one file. To allow this, the file is divided into several blocks, each containing one trace. Therefore, in an arrayed experiment, the files `fid`, `data`, and `phasefile` typically contain the same number of blocks. The number of traces in an arrayed experiment is identical to the parameter `arraydim`. The only complication when working with such data files in arrayed experiments might be that there are "holes" in such files. The holes occur if not all FIDs are transformed or displayed. They do not present a problem as long as a user program uses a "seek" operation just to position the file pointer at the right point in the file and does not try to read traces that have never been calculated.

You can look at 2D experiments as a special case of an arrayed experiment; however, the situation is complicated by the fact that the data often has to be transposed. After the first FT, the resulting spectra are transposed to become the FIDs used for the second FT, and after the second FT, the



user might want to work on traces in either the  $f_1$  or  $f_2$  direction. Furthermore, some types of symmetrization and baseline correction algorithms may have to work on traces in both directions at the same time. The situation is complicated by the fact that the "in place" matrix transposition of large data sets is a very complex operation, requiring many disk accesses. Therefore, this can not be used in a system that has to transform large non-symmetric data sets in a short time.

"Out of place" transpositions are not acceptable for large data sets because they double the disk space requirements of the large 2D experiments. Therefore, VnmrJ software uses a storage format in the 2D data file that allows access to both rows and columns at the same time. Because of the proprietary nature and complexity of the algorithm involved, it is not presented here. The storage format is used only in `datdir/data`.

2D FIDs are stored the same way as 1D FIDs. Transformed 2D data sets are stored in `data` in large blocks of typically 256K bytes. This means that multiple traces are combined to form a block. Within one block, the data is not stored as individual traces, but is scrambled to make access to rows and columns as fast as possible.

Phased 2D data is stored in `phasefile` in the same large blocks as in `data`, but the traces within each block are stored sequentially in their natural order. Both traces along  $f_1$  and  $f_2$  are stored in the same file. The first block(s) contain traces number 1 to  $f_n$  along the  $f_1$  axis; the next block(s) contains traces number 1 to  $f_{n1}$  along the  $f_2$  axis. Note again that `phasefile` will only contain data if the corresponding display operation has been performed. Therefore, in most typical situations, where only a display along one of the two 2D axes is done, `phasefile` will contain only the block(s) for the traces along  $f_1$  or a 'hole' followed by the block(s) for the traces along  $f_2$ . Furthermore, in large experiments, where multiple blocks must be used to store the whole data set, only a 'full' display will ensure that all blocks were actually calculated.

## Header and data display

The VnmrJ commands `ddf`, `ddff`, and `ddfp` display file headers and data. `ddf` displays the data file in the current experiment. Without arguments, only the file header is displayed. Using

```
ddf<(block_number,trace_number,first_number)>, ddf
displays a block header and part of the data set of that
block is displayed. block_number is the block number,
default 1. trace_number is the trace number within the
block, default 1. first is the first data element number
within the trace,
default 1.
```

The `ddff` command displays the FID file in the current experiment and the `ddfp` command displays the phase file in the current experiment. Without any arguments, both display only the file header. Using the same arguments as the `ddf` command, `ddff` and `ddfp` display a block header and part of the data of that block is displayed. The `mstat` command displays statistics of memory usage by VnmrJ commands.

## Binary files in VnmrJ and byte order

There are two competing, incompatible standards for storing binary data in the computing industry:

- In systems with "big-endian" architecture (e.g., Motorola MC680x0, Motorola PowerPC, Sun SPARC), multi-byte data words are stored with the most significant byte (MSB) first;
- In systems with "little-endian" architecture (e.g., Intel x86), multi-byte data words are stored with the least significant byte (LSB) first.

This implies that, in general, when reading binary data that were written on a system with the "other" architecture, software, in general, needs to perform a byte-swapping operation (0,1 -> 1,0; 0,1,2,3 -> 3,2,1,0; 0,1,2,3,4,5,6,7 -> 7,6,5,4,3,2,1,0).

For over two decades, Varian used computing platforms with "big-endian" architecture (first Motorola MC680x0, then Sun SPARC). VnmrJ now runs on Intel x86-based computers. To avoid working with two incompatible binary data formats, VnmrJ has retained the "big-endian" data format, i.e., *all* binary VnmrJ data files are written in "MSB mode"; if running on Intel x86 PC platforms, VnmrJ performs a byte swapping operation when reading binary files, and the bytes

are also swapped prior to writing data to disk. The key is that all binary data files used in connection with VnmrJ are use MSB byte ordering, independent of the architecture on which they were created.

## FDF (Flexible Data Format) Files

The FDF file <XREF>format was developed to support the Image Browser, chemical shift imaging (CSI), and single-voxel spectroscopy (SVS) applications. When these applications were under development, the current VnmrJ file formats for image data were not easily usable for the following reasons:

- The data and parameters describing the data were separated into two files. If the files were ever separated, there would be no way to use or understand the data.
- The data file had embedded headers that were not needed and provided no useful purpose.
- There was no support or structure for saving multislice data sets or a portion of a multislice data set as image files.

FDF was developed to make it similar to VnmrJ formats, with parameters in an easy-to-manipulate ASCII format and a data header that is not fixed so that parameters can be added. This format makes it easy for users and different applications to manipulate the headers and add needed parameters without affecting other applications.

### File structures and naming conventions

Several file structure and naming conventions have been developed for more ease in using and interpreting files. Applications should not assume certain names for certain file; however, specific applications may assume default names when outputting files.

#### Directories

The directory naming convention is <name>.dat. The directory can contain a parameter file and any number of FDF files. The name of the parameter file is `procpar`, a standard VnmrJ name.

#### File names

Each type of file has a different name in order to make the file more recognizable to the user. For image files, the name is `image [nnnn].fdf`, where `nnnn` is a numeric string from 0000 to 9999. For volumes, the name is `volume [nnnn].fdf`, where `nnnn` is also a numeric string from 0000 to 9999. Programs that read FDF files should not depend on these

names because they are conventions and not definitions.

### Compressed files

Although not implemented at this time, compression will be supported for the data portion of the file. The headers will not be compressed. A field will be put in the header to define the compression method or to identify the command to uncompress the data.

## File format

The format of an FDF file consists of a header and data.

Figure 5 is an example of an FDF header. The header is in ASCII text and its fields are defined by a data definition language. Using ASCII text makes it easy to decipher the image content and add new fields, and is compatible with the ASCII format of the `procpa` file. The fields in the data header can be in any order except for the magic number string, which are the first characters in the header, and the end of header character `<null>`, which must immediately precede the data. The fields have a C-style syntax. A correct header can be compiled by the C compiler and should not result in any errors.

The data portion is binary data described by fields in the header. It is separated from the header by a null character.

```
#!/usr/local/fdf/startup
int rank=2;
char *spatial_rank="2dfov";
char *storage="float";
int bits=32;
char *type="absval";
int matrix[]={256,256};
char *abscissa[]{"cm","cm"};
char *ordinate[]{"intensity"};
float span[]={-10.000000,-15.000000};
float origin[]={5.000000,6.911132};
char *nucleus[]{"H1","H1"};
float nucfreq[]={200.067000,200.067000};
float location[]={0.000000,-0.588868,0.000000};
float roi[]={10.000000,15.000000,0.208557};
float orientation[]={0.000000,0.000000,1.000000,-1.000000,
0.000000,0.000000,0.000000,1.000000,0.000000};
checksum=0787271376;

<zero>
```

Figure 5 Example of an FDF Header

## Header parameters

The fields in the data header are defined in this section.

### Magic number

The magic number is an ASCII string that identifies the file as a FDF file. The first two characters in the file must be `#!`, followed by the identification string. Currently, the string is `#!/usr/local/fdf/startup`.

### Data set dimensionality or rank fields

These entries specify the data organization in the binary portion of the file.

- `rank` is a positive integer value (1, 2, 3, 4,...), giving the number of dimensions in the data file (e.g., `int rank=2;`).
- `matrix` is a set of rank integers, giving the number of data points in each dimension (e.g., for `rank=2`, `float matrix[]={256,256};`).
- `spatial_rank` is a string ("none", "voxel", "1dfov", "2dfov", "3dfov") for the type of data (e.g., `char *spatial_rank="2dfov";`).

### Data content fields

The following entries define the data type and size.

- `storage` is a string ("integer", "float") that defines the data type (e.g., `char *storage="float";`).
- `bits` is an integer (8, 16, 32, or 64) that defines the size of the data (e.g., `float bits=32;`).
- `type` is a string ("real", "imag", "absval", "complex") that defines the numerical data type (e.g., `char *type="absval";`).

### Data location and orientation fields

The following entries define the user coordinate system and specify the size and position of the region from which the data was obtained. Magnet Coordinates as Related to User Coordinates illustrates the coordinate system. Vectors that correspond to header parameters are shown in **boldface**.

- orientation specifies the orientation of the user reference frame ( $x, y, z$ ) with respect to the magnet frame ( $X, Y, Z$ ). orientation is given as a set of nine direction cosines, in the order:

$$d_{11}, d_{12}, d_{13}, d_{21}, d_{22}, d_{23}, d_{31}, d_{32}, d_{33}$$

where:

$$x = d_{11}X + d_{12}Y + d_{13}Z$$

$$y = d_{21}X + d_{22}Y + d_{23}Z$$

$$z = d_{31}X + d_{32}Y + d_{33}Z$$

and

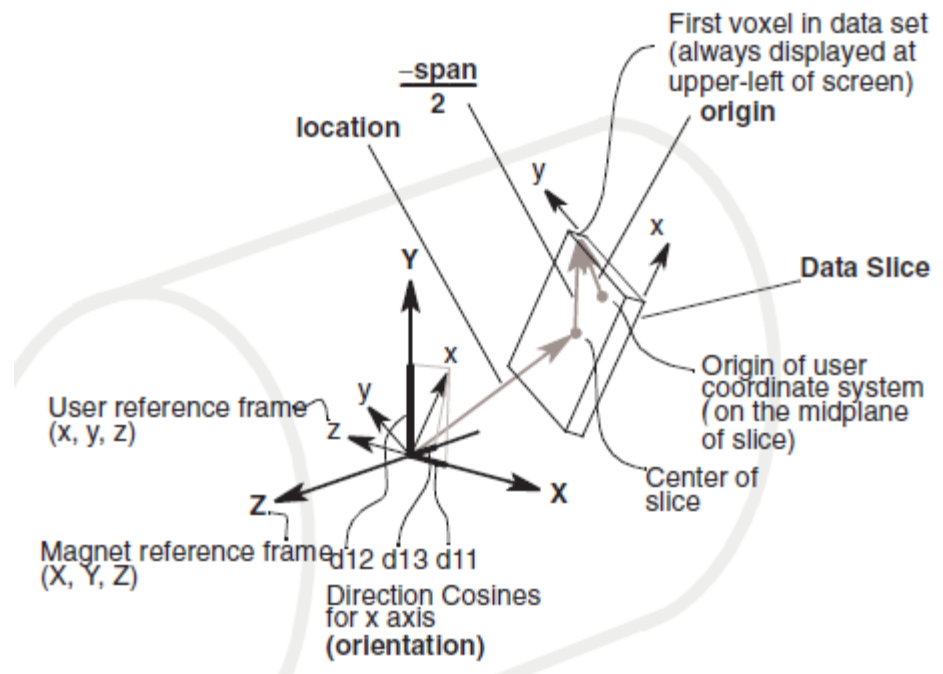
$$X = d_{11}x + d_{21}y + d_{31}z$$

$$Y = d_{12}x + d_{22}y + d_{32}z$$

$$Z = d_{13}x + d_{23}y + d_{33}z$$

The value is written as nine floating point values grouped as three triads (e.g., float

orientation[]={0.0,0.0,1.0,-1.0,0.0,0.0,0.0,1.0,0.0};).



**Figure 6** Magnet Coordinates as Related to User Coordinates

- `location` is the position of the center of the acquired data volume relative to the center of the magnet, in the user's co-ordinate system. The position is given in centimeters as a triple (three floating point values) of x, y, z distances (e.g., `float location[]={10.0,15.0,0.208};`).
- `roi` is the size of the acquired data volume (three floating point values), in centimeters, in the user's coordinate frame, not the magnet frame (e.g., `float roi[]={10.0,15.0,0.208};`). Do not confuse this `roi` with ROIs that might be specified inside the data set.

### Data axes

The data axes entries specify the user co-ordinates of data points. These axes do not tell how to orient the display of the data, but only define what to call the co-ordinates of a given datum. There are no standard header entries to specify the orientation of the data display. Currently, data is always displayed or plotted in the same order that it is stored. The fastest data dimension is plotted horizontally from left to right; the next dimension is plotted vertically from top to bottom.

- `origin` is a set of rank floating point values, giving the user co-ordinates of the first point in the data set (e.g., `float origin[]={5.0,6.91};`).
- `span` is a set of rank floating point values for the signed length of each axis, in user units. A positive value means the value of the particular co-ordinate increases going away from the first point (e.g., `float span[]={-10.000,-15.000};`).
- `abscissa` is a set of rank strings ("hz", "s", "cm", "cm/s", "cm/s2", "deg", "ppm1", "ppm2", "ppm3") that identifies the units that apply to each dimension (e.g., `char *abscissa[]{"cm","cm"};`).
- `ordinate` is a string ("intensity", "s", "deg") that gives the units that apply to the numbers in the binary part of the file (e.g., `char *ordinate[]{"intensity"};`).

### Nuclear data fields

Data fields may contain data generated by interactions between more than one nucleus (e.g., a 2D chemical shift correlation map between protons and carbon). Such data require interpreting the term "ppm" for the specific nucleus if ppm to frequency conversions and properly labeling axes



arising from different nuclei are necessary. To properly interpret ppm and label axes, the identity of the nucleus in question and the corresponding nuclear resonance frequency are needed. These fields are related to the abscissa values "ppm1", "ppm2", and "ppm3" in that 1, 2, and 3 are indices into the nucleus and nucfreq fields. That is, the nucleus for the axis with abscissa string "ppm1" is the first entry in the nucleus field.

- nucleus is one entry ("H1", "F19", same as VnmrJ tn parameter) for each rf channel (e.g., char \*nucleus[]={ "H1", "H1" };).
- nucfreq is the nuclear frequency (floating point) used for each rf channel (e.g., float nucfreq[]={200.067,200.067};).

### Miscellaneous fields

- checksum is the checksum of the data. Changes to the header do not affect the checksum. The checksum is a 32-bit integer, calculated by the gluer program (e.g., int checksum=0787271376;).
- compression is a string with either the command needed to uncompress the data or a tag giving the compression method. This field is not currently implemented.

### End of header

A character specifies the end of the header. If there is data, it immediately follows this character. The data should be aligned according to their data type. For single precision floating point data, the data is aligned on word boundaries. Currently, the end of the header character is <zero> (an ASCII "NUL").

## Transformations

By editing some of the header values, it is possible to make a program that reads FDF data files to perform simple transformations. For example, to flip data left-to-right, set:

```
span'0=-span0
origin'0=origin0-span'0
```

## Creating FDF files

To generate files in the FDF format, the following macros are available to write out single or multi-slice images:

- For the current imaging software—including sequences `sems`, `mems`, and `flash`—use the macro `svib(directory<,'f'|'m'|'i'|'o'>)`, where `directory` is the directory name desired (`.dat` is appended to the name), `'f'` outputs data in the floating point format (this is the default), `'m'` or `'i'` outputs data as 12-bit integer values in 16-bit words, and `'b'` outputs data in 8-bit integer bytes.
- For older style SIS imaging sequences and microimaging sequences, use the macro `svsis(directory<,'f'|'m'>)`, where `directory`, `'f'`, and `'m'` are defined the same as `svib`.

Raw data from the FID file of the current experiment can be saved as an FDF file with the `svfdf(directory)` macro, where `directory` is the name of the directory in which to store the files (`.dat` is appended to the name). Data is saved in multiple files, with one trace per file. The files are named `fid0001.fdf`, `fid0002.fdf`, *etc.* The `propar` file from the current experiment is also saved in the same directory.

Another way to create the FDF files is to edit or create a header defining a set of data with no headers and attach it to the data file with the `fdfgluer` program. Use the syntax `fdfgluer header_file <data_file <output_file>>` (from Linux only). This program takes a `header_file` and a `data_file` and puts them together to form an FDF file. It also calculates a checksum and inserts it into the header. If the `data_file` argument is not present, `fdfgluer` assumes the data as input from the standard input, and if the `output_file` name is not present, `fdfgluer` writes the FDF file to the standard output.

## Splitting FDF files

The `fdfsplit` command takes an FDF file and splits it into its data and header parts. The syntax is `fdfsplit fdf_file data_file header_file` (from Linux only). If the header still has a checksum value, that value should be removed.

## Reformatting Data for Processing

Sometimes, data Reformatting spectraacquired in an experiment has to be reformatted for processing. This is especially true for in-vivo imaging experiments, where time is critical in getting the data. So, experiments are designed to acquire data quickly, but not necessarily in the most desirable format for processing. Reformatting data can also occur in other applications because of a particular experimental procedure.

The VnmrJ processing applications `ft2d` and `ft3d` can accept data in standard, compressed, or compressed-compressed (3D) data formats. There are a number of routines that allow users to reformat their data into these formats for processing. The reformatting routines allow users to compress or uncompress their data (`flashc`), move data around between experiments and into almost any format (`mf`, `mfblk`, `mfdata`, `mftrace`), reverse data while moving it (`rfblk`, `rfdata`, `rftrace`), or use a table of values, in this case a table stored in `tablib`, to sort and reformat scans of data (`tabc`, `tcapply`).

In this section, standard and compressed data are defined, reformatting options are described, and several examples are presented. [Table 55](#) on page 460 summarizes the reformatting commands described in this section. Note that the commands `rsapply`, `tcapply`, `tcclose`, and `tcopen` are for 2D spectrum data; the remaining commands in the table are for FID data.

**Table 55** Commands for Reformatting Data

Commands	
flashc*	Convert compressed 2D data to standard 2D format
mf(<from_exp,>to_exp)	Move FIDs between experiments
mfbk*	Move FID block
mfclose	Close memory map FID
mfdata*	Move FID data
mfopen(<src_expno,>dest_expno)	Memory map open FID file
mftrace*	Move FID trace
rfbk*	Reverse FID block
rfdata*	Reverse FID data
rftrace*	Reverse FID trace
rsapply	Reverse data in a spectrum
tabc<(dimension)>	Convert data in table order to linear order
tcapply<(file)>	Apply table conversion reformatting to data
tcclose	Close table conversion file
tcopen<(file)>	Open table conversion file
* flashc(<'ms'   'mi'   'rare'<,traces><,echoes>)	
mfbk(<src_expno,>src_blk_no,dest_expno,dest_blk_no)	
mfdata(<src_expno,>src_blk_no,src_start_loc,dest_expno, \	
dest_blk_no,dest_start_loc,num_points)	
mftrace(<src_expno,>src_blk_no,src_trace_no,dest_expno	
dest_blk_no,dest_trace_no)	
rfbk(<src_expno,>src_blk_no,dest_expno,dest_blk_no)	
rfdata(<src_expno,>src_blk_no,src_start_loc,dest_expno, \	
dest_blk_no,dest_start_loc,num_points)	
rftrace(<src_expno,>src_blk_no,src_trace_no,dest_expno, \	
dest_blk_no,dest_trace_no)	

## Standard and compressed formats

Usually, when discussing standard and compressed data formats, *standard* means the data was acquired using the arrayed parameters  $n_i$  and  $n_{i2}$ , which specify the number of increments in the second and third dimensions; and *compressed* means using parameter  $n_f$  to specify the increments in the second dimension.

For multi-slice imaging, *standard* means using  $n_i$  to specify the phase-encode increments and  $n_f$  to specify the number of slices and *compressed* means using  $n_f$  to specify the phase-encode increments while arraying the slices.

*Compressed-compressed* means using  $n_f$  to specify the phase-encode increments and slices for 2D or to specify the phase-encode increments in the second and third dimensions for 3D. In compressed-compressed data sets,  $n_f$  can be set

to  $nv*ns$  or  $nv*nv2$ , where  $nv$  is the number of phase-encode increments in the second dimension,  $nv2$  is the number of phase-encode increments in the third dimension, and  $ns$  is the number of slices.

To give another view of data formats, which will help when using the "move FID" commands, each  $ni$  increment or array element is stored as a data block in an FID file and each  $nf$  FID is stored as a trace within a data block in a FID file.

## Compress or decompress data

The most common form of reformatting for imaging has been to use the `flashc` command to convert compressed data sets to standard data sets in order to run `ft2d` on the data. With the implementation of `ft2d('nf', <index>)`, `flashc` is no longer necessary. However, use of `flashc` is still necessary for converting compressed-compressed data to compressed or standard formats.

## Move and reverse data

The commands `mf`, `mfblk`, `mfdata`, and `mftrace` are available to move data around in a FID file or to move data from one experiment FID file to another experiment FID file. These commands give users more control in reformatting their data by allowing them to move entire FID files, individual blocks within a FID file, individual traces within a block of a FID file, or sections of data within a block of a FID file.

To illustrate the use of the "move FID" commands, [Figure 6](#) on page 455 is an example with code from a macro that moves a 3D dataset from an arrayed 3D dataset to another experiment that runs `ft3d` on the data. The `$index` variable is the array index. It works on both compressed-compressed and compressed 3D data.

The "reverse FID" commands `rfbblk`, `rftrace`, and `rfdata` are similar to their respective `mfblk`, `mftrace`, and `mfdata` commands, except that `rfbblk`, `rftrace`, and `rfdata` also reverse the order of the data. The `rfbblk`, `rftrace`, and `rfdata` commands were implemented to support EPI (Echo Planar Imaging) processing. [Listing 3](#) is an example of using these commands to reverse every other FID echo for EPI data. Note that the `mfopen` and `mfclose` commands can significantly speed up the data reformatting by opening and closing the data files once, instead of every time the data set is moved. The `rfbblk`, `rftrace`, and `rfdata` commands can also be used with the "move FID" commands.

**CAUTION**

For speed reasons, the "move FID" and "reverse FID" commands work directly on the FID and follow data links. These commands can modify data returned to an experiment with the `rt` command. To avoid modification, enter the following sequence of VnmrJ commands before manipulating the FID data:

```
cp(curexp+'/acqfil/fid',curexp+'/acqfil/fidtmp')
rm(curexp+'/acqfil/fid')
mv(curexp+'/acqfil/fidtmp',curexp+'/acqfil/fid')
```

---

**Table convert data**

VnmrJ supports reconstructing a properly ordered raw data set from any arbitrarily ordered data set acquired under control of an external AP table. The data must have been acquired according to a table in the `tablib` directory. The command for table conversion is `tabc`.

**Reformatting spectra**

The commands `rsapply`, to reverse a spectrum, and `tcapply`, to reformat a 2D set of spectra using a table, support reformatting of spectra within a 2D dataset. The types of reformatting are the reversing of data within a spectrum and the reformatting of arbitrarily ordered 2D spectrum by using a table. These commands do not change the original FID data, and they may provide some speed improvement over the similar commands that operate on FID data. For 2D data, an `ft1d` command should be applied to the data, followed by the desired reformatting, and then an `ft2d` command to complete the processing.

**Table 56** Listing 2 Code from a "Move FID" Macro

```

if ($seqcon[3] = 'c') and ($seqcon[4] = 'c') then
  "**** Compressed-compressed 3d ****"
  $arraydim = arraydim
  if ($index > $arraydim) then
    write('error','Index greater than arraydim.')
    abort
  endif
  mfbk($index,$workexp,1)
  jexp($workexp)
  setvalue('arraydim',1,'processed')
  setvalue('arraydim',1,'current')
  setvalue('array','','processed')
  setvalue('array','','current')
  ft3d
  jexp($cexpn)
else if ($seqcon[3] = 'c') and ($seqcon[4] = 's') then
  "**** Compressed 3d ****"
  if (ni < 1.5) then
    write('error','seqcon, ni mismatch check parameters.')
    abort
  endif
  $arraydim = arraydim/ni
  if ($index > $arraydim) then
    write('error','Index greater than arraydim.')
    abort
  endif
  $i = 1
  $k = $index
  while ($i <= ni) do
    mfbk($k,$workexp,$i)
    $k = $k + $arraydim
    $i = $i + 1
  endwhile
  jexp($workexp)
  setvalue('arraydim',ni,'processed')
  setvalue('arraydim',ni,'current')
  setvalue('array','','processed')
  setvalue('array','','current')
  ft3d
  jexp($cexpn)

```

**Table 57** Listing 3 Example of Command Reversing Data Order

```

*****
* epirf(<blkno>) - macro to reverse every other FID
* block & trace indicies start at 1 for rfbk,rftrace,rfdata
*****
mfopen
$i=2
while ($i <= nv) do
  rftrace($i,$i)
  $i = $i + 2
endwhile
mfclose

```

## Creating and Modifying Parameters

VnmrJ parameters and the `<XREF>ir` attributes are created and modified with the commands covered in this section. The parameter trees used by these commands are Linux files containing the attributes of a parameter as formatted text.

### Parameter types and trees

The types of parameters that can be created are 'real', 'string', 'delay', 'frequency', 'flag', 'pulse', and 'integer (default is 'real'). In brief, the meaning of these types are as follows (for more details, refer to the description of the `create` command in the *VnmrJ Command and Parameter Reference*):

- 'real' is any positive or negative value.
- 'string' is composed of characters, and can be limited to selected words by enumerating the possible values with the command `setenumerals`.
- 'delay' is a value between 0 and 8190, in units of seconds.
- 'frequency' is positive real number values.
- 'flag' is composed of characters, similar to the 'string' type, but can be limited to selected characters by enumerating the possible values with the command `setenumerals`. If enumerated values are not set, the 'string' and 'flag' types are identical.
- 'pulse' is a value between 0 and 8190, in units of microseconds.
- 'integer' is composed of integers (0, 1, 2, 3,...),

The four parameter tree types are 'current', 'global', 'processed', and 'systemglobal' (the default is 'current'):

- 'current' contains the parameters that are adjusted to set up an experiment. The parameters are from the file `curpar` in the current experiment.
- 'global' contains user-specific parameters from the file `global` in the `vnmrsys` directory of the present Linux user.
- 'processed' contains the parameters with which the data was obtained. These parameters are from the file `propar` in the current experiment.



- 'systemglobal' contains instrument-specific parameters from the text file /vnmr/conpar. The config program is used to define most of these parameters. All users have the same systemglobal tree.

## Tools for working with parameter trees

Table 58 lists commands for creating, modifying, and deleting parameters.

### To create a new parameter

Use `create(parameter<,type<,tree>>)` to create a new parameter in a parameter tree with the name specified by `parameter`. For example, entering `create('a','real','global')` creates a new real-type parameter *a* in the global tree. `type` can be 'real', 'string', 'delay', 'frequency', 'flag', 'pulse', or 'integer'. If the `type` argument is not entered, the default is 'real'. `tree` can be 'current', 'global', 'processed', or 'systemglobal'. If the `tree` argument is not entered, the default is 'current'. See the section above for a description of parameter types and trees. Note that these same arguments are used with all the commands appearing in this section.

**Table 58** Commands for Working with Parameter Trees

<b>Commands</b>	
<code>create (parameter&lt;, type&lt;, tree&gt;&gt;)</code>	Create a new parameter in parameter tree
<code>destroy (parameter&lt;, tree&gt;)</code>	Destroy a parameter
<code>destroygroup (group&lt;, tree&gt;)</code>	Destroy parameters of a group in a tree
<code>display (parameter   '*'   '**'&lt;, tree&gt;)</code>	Display parameters and their attributes
<code>fread (file&lt;, tree&lt;, 'reset'   'value'&gt;&gt;)</code>	Read in parameters from a file into a tree
<code>fsave (file&lt;, tree&gt;)</code>	Save parameters from a tree to a file
<code>getvalue (parameter&lt;, index&gt;&lt;, tree&gt;)</code>	Get value of parameter in a tree
<code>groupcopy (from_tree, to_tree, group)</code>	Copy group parameters from tree to tree
<code>paramvi (parameter&lt;, tree&gt;)</code>	Edit parameter and its attributes using vi
<code>prune (file)</code>	Prune extra parameters from current tree
<code>setdgroup (parameter, dgroup&lt;, tree&gt;)</code>	Set the Dgroup of a parameter in a tree
<code>setenumerals*</code>	Set values of a string parameter in a tree
<code>setgroup (parameter, group&lt;, tree&gt;)</code>	Set group of a parameter in a tree
<code>setlimit*</code>	Set limits of a parameter in a tree
<code>setprotect*</code>	Set protection mode of a parameter
<code>settype (parameter, type&lt;, tree&gt;)</code>	Change type of a parameter
<code>setvalue*</code>	Set value of any parameter in a tree
* <code>setenumerals (parameter, N, enum1, enum2, ... enumN&lt;, tree&gt;)</code>	
<code>setlimit (parameter, maximum, minimum, step_size&lt;, tree&gt;)</code> or	
<code>setlimit (parameter, index&lt;, tree&gt;)</code>	
<code>setprotect (parameter, 'set'   'on'   'off', value&lt;, tree&gt;)</code>	
<code>setvalue (parameter, value&lt;, index&gt;&lt;, tree&gt;)</code>	

### To get the value of a parameter

The value of most parameters can be accessed simply by using their name in an expression; for example, `sw?` or `r1=np` accesses the value of `sw` and `np`, respectively.

However, parameters in the processed tree cannot be accessed this way. Use

`getvalue (parameter<, index><, tree>)` to get the value of any parameter, including the value of a parameter in a processed tree. To make this easier, the default value of `tree` is 'processed'. The `index` argument is the number of a single element in an arrayed parameter (the default is 1).

### To edit or set parameter attributes

Use `paramvi (parameter<, tree>)` to open the file for a parameter in the `vi` text editor to edit the attributes. To open a parameter file with an editor other than `vi`, use `paramedit (parameter<, tree>)`. Refer to entry for `paramedit` in the *VnmrJ Command and Parameter Reference* for information on how to select a text editor other than `vi`. The format of a stored parameter is described in the next section.

Several parameter attributes can be set by the following

commands:

- `setlimit(parameter, maximum, minimum, step_size<, tree>)` sets the maximum and minimum limits and stepsize of a parameter.
- `setlimit(parameter, index<, tree>)` sets the maximum and minimum limits and the stepsize, but obtains the values from the index-th entry of a table in `conpar`.
- `setprotect(parameter, 'set' | 'on' | 'off', bit_vals<, tree>)` sets the protection bits associated with a parameter. The keyword 'set' causes the current protection bits to be replaced with the set specified by `bit_vals` (listed in *VnmrJ Command and Parameter Reference*). 'on' causes the bits specified in `bit_vals` to be turned on without affecting other protection bits. 'off' causes the bits specified in `bit_vals` to be turned off without affecting other protection bits.
- `settype(parameter, type<, tree>)` changes the type of an existing parameter. A string parameter can be changed into a string or flag type, or a real parameter can be changed into a real, delay, frequency, pulse, or integer type.
- `setvalue(parameter, value<, index><, tree>)` sets the value of any parameter in a tree. `setvalue` bypasses normal range checking for parameter entry. It also bypasses any action that would be invoked by the parameter's protection bits.
- `setenumerals(parameter, N, enum1, enum2, ..., enumN<, tree>)` sets possible values of a string-type or flag-type parameter in a parameter tree.
- `setgroup(parameter, group<, tree>)` sets the group (also called the Ggroup) of a parameter in a tree. The group argument can be 'all', 'sample', 'acquisition', 'processing', 'display', or 'spin'.
- `setdgroup(parameter, dgroup<, tree>)` sets the Dgroup of a parameter in a tree. The `dgroup` argument is an integer. The usage of `setdgroup` is set by the application. Only the experimental user interface currently uses this command.

### To display a parameter

Use `display(parameter | '*' | '**'<, tree>)` to display one or more parameters and their attributes from a parameter tree. The first argument can be one of the following three options:

- a parameter name (to display the attributes of that parameter,
- '\*' (to display the name and value of all parameters in a tree), or
- '\*\*' (to display the attributes of all parameters in a tree.

The results are displayed in the Process tab, Text Output.

### To move parameters

Use `groupcopy(from_tree,to_tree,group)` to copy a set of parameters of a group from one parameter tree to another (it cannot be the same tree). `group` is the same keywords as used with `setgroup`.

The `fread(file<,tree<,'reset'|'value'>>)` command reads the parameters from a file and loads them into a tree. The keyword 'reset' causes the tree to be cleared before the new file is read; 'value' causes only the values of the parameters in the file to be loaded. The `fsave(file<,tree>)` command writes parameters from a parameter tree to a file for which the user has write permission. It overwrites any file that exists.

### To destroy a parameter

The `destroy(parameter<,tree>)` command removes a parameter from a parameter tree, while the `destroygroup(group<,tree>)` command removes parameters of a group from a parameter tree. The `group` argument uses the same keywords as used with the `setgroup` command. If the destroyed parameter was an array, the array parameter is automatically updated.

To remove leftover parameters from previous experimental setups, use `prune`. The `prune(file)` command destroys parameters in the current parameter tree that are not also defined in the parameter file specified.

## Format of a stored parameter

To use the `create` command to create a new parameter, or to use the `paramvi` and `paramedit` commands to edit a parameter and its attributes, knowledge of the format of a stored parameter is required. If an error in the format is made, the parameter may not load. This section describes the format.

The stored parameter format is made up of three or more

lines.

**Line 1** contains parameter attributes and has the following fields (given in the same order as they appear in the file):

- `name` is the parameter name, which can be any valid string.
- `subtype` is an integer value for the parameter type: 0 (undefined), 1 (real), 2 (string), 3 (delay), 4 (flag), 5 (frequency), 6 (pulse), 7 (integer).
- `basictype` is an integer value: 0 (undefined), 1 (real), 2 (string).
- `maxvalue` is a real number for the maximum value that the parameter can contain, or an index to a maximum value in the parameter `parmax` (found in */vnmr/conpar*). Applies to both string and real types of parameters.
- `minvalue` is a real number for the minimum value that the parameter can contain or an index to a minimum value in the parameter `parmin` (found in */vnmr/conpar*). Applies to real types of parameters only.
- `stepsize` is a real number for the step size in which parameters or index to a step size in the parameter `parstep` (found in */vnmr/conpar*) can be entered. If `stepsize` is 0, it is ignored. Applies to real types only.
- `Ggroup` is an integer value: 0 (ALL), 1 (SAMPLE), 2 (ACQUISITION), 3 (PROCESSING), 4 (DISPLAY), 5 (SPIN).
- `Dgroup` is an integer value. The specific application determines the usage of this integer.

- protection is a 32-bit word made up of the following bit masks, which are summed to form the full mask:

Bit	Value	Description
0	1	Cannot array the parameter
1	2	Cannot change active/not active status
2	4	Cannot change the parameter value
3	8	Causes <code>_{parameter}</code> macro to be executed upon parameter entry (e.g., if the parameter is named <code>sw</code> , the macro <code>_sw</code> is executed when <code>sw</code> is changed); an error is issued if such a macro is not found
4	16	Avoids automatic redisplay
5	32	Cannot delete parameter
6	64	System parameter for spectrometer or data station
7	128	Cannot copy parameter from tree to tree
8	256	Will not set array parameter
9	512	Cannot set parameter numeral values
10	1024	Cannot change the parameter's group
11	2048	Cannot change protection bits
12	4096	Cannot change the display group
13	8192	Take <code>max</code> , <code>min</code> , <code>step</code> from <code>/vnmr/conpar</code> (systemglobal) parameters <code>parmax</code> , <code>parmin</code> , <code>parstep</code> .
14	16384	Parameter marked for locking ( <code>P_LOCK</code> , see <code>rtx</code> )
15	32768	Global parameter not shared in multiple VnmrJ viewports
16	65536	Force automatic redisplay in VnmrJ parameter templates

- `active` is an integer value: 0 (not active), 1 (active).
- `intptr` is not used (generally set to 64).

**Line 2** or the group of lines starting with line 2, lists the values of the parameter. The first field on line 2 is the number of values the parameter is set to. The format of the rest of the fields on line 2 and subsequent lines, if any, depends on the value of `basictype` set on line 1 and the value entered in the first field on line 2:

- If `basictype` is 1 (real) and the first value on line 2 is any number, all parameter values are listed on line 2, starting in the second field. Each value is separated by a space.
- If `basictype` is 2 (string) and the first value on line 2 is 1, the single string value of the parameter is listed in the second field of line 2, inside double quotes.
- If `basictype` is 2 (string) and the first value on line 2 is greater than 1, the first array element is listed in the second field on line 2 and each additional element is listed on subsequent lines, one value per line. Strings are surrounded by double quotes.

**The last line** of a parameter file lists the enumerable values of a string or flag parameter. This specifies the possible values the string parameter can be set to. The first field is the number of enumerable values. If this number is greater than 1, all of the values are listed on this line, starting in the second field.

For example, here is how a typical real parameter file, named `a`, is interpreted (the numbers in parentheses are not part of the file but are line references in the interpretation):

```
(1) a 31 1e+30 -1e+30 0 0 1 0 1 64
(2) 24.126400
(3) 0
```

This file is made up of the following lines:

- The parameter has the name `a`, subtype is 3 (delay), `basictype` is 1 (real), maximum size is `1e+30`, minimum size is `-1e+30`, `stepsize` is 0, `Ggroup` is 0 (ALL), `Dgroup` is 1 (ACQUISITION), protection is 0 (cannot array the parameter), active is 1 (ON), and `intptr` is 64 (not used).
- Parameter `a` has 1 value, the real number 24.126400.
- Parameter `a` has 0 enumerable values.

As another example, here are the values in a file for the parameter `tof`:

```
(1) tof 5 1 7 7 7 2 1 8202 1 64
(2) 1 1160
```

```
(3) 0
```

The `tof` file is made up of the following lines:

- The parameter has the name `tof`, subtype is 5 (frequency), and basictype is 1 (real). To read the next 3 values, we must jump to the protection field. Because the protection word value is 8202, which is  $8192 + 8 + 2$ , then bit 13 (8192), bit 3 (8), and bit 1 (2) bitmasks are set. Because bit 13 is set, the maximum size, minimum size, and stepsize values (each is 7) are indices into the 7th array value in the parameters `parmax`, `parmin`, and `parstep`, respectively, in the file `conpar`. Because bit 3 is set, this causes a macro to be executed. The bit 1 bitmask (2) is also set, which means that the active/not active status of the parameter cannot be changed. For the remaining fields, `Ggroup` is 2 (ACQUISITION), `Dgroup` is 1 (ACQUISITION), `active` is 1 (ON), and `intptr` is 64 (not used).
- Parameter
- `tof` has 1 value, the real number 1160.
- Parameter
- `tof` has 0 enumerable values.

The following file is an example of a multi element array character parameter, `beatles`:

```
(1) beatles 2 2 8 0 0 2 1 0 1 64
(2) 4 john
(3) paul
    george
    ringo
(4) 0
```

The `beatles` file is made up of the following lines:

- The parameter has the name of `beatles`, subtype is 2 (string), basictype is 2 (string), `8 0 0` is max min step (not really used for strings), `Ggroup` is 2 (acquisition), `Dgroup` is 1 (ALL), `protection` is 0, `active` is 1 (ON), 64 is a terminating number.
- There are four elements to this variable; therefore, it is arrayed.
- `john` is the first element in the array.
- `paul`, `george`, and `ringo` are the other three elements in the array.
- 0 (zero) is the terminating line.



## Modifying Parameter Displays in VNMR

The VNMR <XREF>plotting commands and macros— `ap`, `pap`—are controlled by the template parameters, specifying the content and form of the information plotted. The template parameters have the same name as the respective command or macro; for example, the plot created by the `ap` command is controlled by the parameter `ap` in the experiment's current parameter set.

To modify an existing template parameter, such as `ap`, enter `paramvi('ap')` to use the `vi` text editor, or enter `paramedit('ap')` to use the text editor set by the environmental variable `vnmreditor`.

### Display template

A plot template can have a single string or multiple strings. The first number on the second line of a stored parameter indicates the number of string templates. If the number is 1, the display template is a single string; otherwise, a value greater than 1 indicates the template is multiple strings. [Figure 7](#) shows an example of a single-string display template (actually the parameter `ap`) and the resulting plot.

In a single-string template, the string always starts with a double quote and then repeats the following information for each column in the plot:

```
ap 2 2 1023 0 0 4 1 6 1 64
1
"1:SAMPLE:date,solvent,file;1:ACQUISITION:sw:1,at:3,np:0,fb:0,bs(bs):0,ss(ss):0,
d1:3,d2(d2):6,nt:0,ct:0;1:TRANSMITTER:tn,sfrq:3,tof:1,tpwr:0,pw:3,p1(p1):3;1:DE
COUPLER:dn,dof:1,dm,dmm,dpwr:0,dmf:0;2:SPECIAL:temp:1,gain:0,spin:0,hst:3,p
w90:3,alfa:3;2:FLAGS:il,in,dp,hs;2:PROCESSING:lb(lb):2,sb(sb):3,sbs(sb):3,gf(gf):
3,gfs(gf):3,awc(awc):3,lsfid(lsfid):0,lsfrq(lsfrq):1,phfid(phfid):1,fn:0;2:DISPLAY:sp:
1,wp:1,rfl:1,rfp:1,rp:1,lp:1;2:PLOT:wc:0,sc:0,vs:0,th:0,aig*,dcg*,dmg*;"
0
```

**Figure 7** Single-String Display Template with Output

- Column number (e.g., 2)
- Condition for plot of column (optional, e.g., "4(ni)", see `Conditional` and `arrayed` displays)
- Colon
- Column title (e.g., 2D ACQUISITION)
- Colon

- Parameters to appear in column, separated by commas (for notation, see
- Semicolon

At the end of the string is another double quote. Spaces *cannot* appear anywhere in the string template, except as part of a column title.

Column titles are often in upper case, but need not be, and are limited to 19 characters. More than one title can appear in the same column (such as shown above, SAMPLE and DECOUPLING are both in column 2).

Parameters listed in "plain" form (e.g., `tn,date,math`) are printed either as strings or in a form in which the number of decimal places plotted varies, depending on the value of the parameter.

To plot a specific number of digits past the decimal place, the desired number is placed following a colon (e.g., `sfrq:3,at:3,sw:0`). Extra commas can be inserted to skip rows within a column (e.g., `math,,werr,wexp,`).

The maximum number of columns is 4; each column can have 17 lines of output. As this includes the title(s), fewer than 17 parameters can be displayed in any one column. The entire template is limited to 1024 characters or less.

As an alternative to a single-string template, which tends to be difficult to read, a template can be written as multiple strings, each enclosed in double quotes. The first number indicates the number of strings that follow. Each string must start with a column number. [Figure 8](#) contains the plot template for the parameter `dg2`, which is a typical example of a multiple-string template.

The conditional statement in this example (e.g., `"(numrfch >2)"`) is covered in [Conditional and Arrayed Displays](#)<XREF>.

The title field can contain a string variable besides a literal. If the variable is a real variable, or not present, or equal to the null string, the variable itself is used as the title (e.g., `mystrvar[1]='Example Col 1'` and `mystrvar[2]='Example Col 2'`).

```

6 "1:1st DECOUPLING:dfrq:3,dn,dpwr:0,dof:1,dm,dmm,dmf:0,dseq,dres:1,homo:"
"2(numrfch>2):2nd DECOUPLING:dfrq2:3,dn2,dpwr2:0,dof2:1,dm2,dmm2,dmf2:0,dseq2,dres2:1,homo2:"
"2(numrfch>3):3rd DECOUPLING:dfrq3:3,dn3,dpwr3:0,dof3:1,dseq3,dres3:1,homo3:"
"3(ni2):3D ACQUISITION:d3:3,sw2:1,ni2:0,phase2:0:"
"3(ni2):3D DISPLAY:rp2:1,lp2:1:"
"4(ni2):3D PROCESSING:lb2:3,sb2:3,sbs2(sb2):3,gf2:3,gfs2(gf2):3,awc2:3,wf2:1,proc2,fn2:0:"

```

**Figure 8** Multiple-String Display Template

## Conditional and arrayed displays

Use of parentheses allows the conditional display of an entire column and/or individual parameters. If the real parameter within parentheses is not present, or is equal to 0 or to 'n', then the associated parameter or section is not displayed. In the case of string parameters, if the real number is not present, or is equal to the NULL string or the character 'n', then the associated parameter or section is not displayed. The following examples from the dg template above demonstrate this format:

- p1 (p1):1 means display parameter p1 only when p1 is non-zero.
- sbs (sb):3 means display sbs only when sb is active (not equal to 'n').
- 4(ni):2D PROCESSING: means display the entire "2D PROCESSING" section only when the ni parameter is active and non-zero.

### NOTE

If a parameter is arrayed, the display status is derived from the first value of the array. Thus, if p1 is arrayed and the first value is 0, p1 will not appear; if the first value is non-zero, p1 will appear, with "arrayed" as its parameter value.

Similarly, a multiple variable expression can also be placed within the parentheses for conditional plot of parameters. Each expression must be a valid MAGICAL II expression (see [Chapter 1](#), "MAGICAL II Programming") and must be written so there is no space between the last character of the expression and the closing parenthesis ")".

In summary, if a single variable expression is placed in the parentheses, it is FALSE under the following conditions:

- Variable does not exist.
- Variable is real and equals 0 or is marked inactive.

- Variable is a string variable equal to the NULL string or equal to the character 'n'.

Multiple variable expressions are evaluated the same as in MAGICAL II. If a variable does not exist, it is considered an error.

Examples of multiple parameter expressions include the following:

- `2(numrfch>2):2nd DECOUPLING:` means display entire "2nd DECOUPLING" section only when `numrfch` (number of rf channels) is greater than 2.
- `3((myflag <> 'n') or ((myni > ni) and (mysw < sw))):My Section:` means display entire "My Section" section only when `myflag` is not equal to 'n' or when `myni` is greater than `ni` and `mysw` is less than `sw`.

The asterisk (...\*) is a "special parameter" designator that allows the value of a series of string parameters to be displayed in a single row without names. This is more commonly used with the parameters `aig`, `dcg`, and `dmg`, for example:

```
aig*,dcg*,dmg*
```

For a tabular output of arrayed parameters, square brackets ([...]) are used. For example:

```
1: Sample Table Output:[pw,p1,d1,d2];
```

Notice that all parameters in the column must be in the brackets; thus, the following is not recognized:

```
1: Sample Table Output:[pw,p1,d1],d2;
```

As arrayed variables are normally displayed with `da`, this format is rarely needed.

The field width and digit field options can be used to clean up the display. The first number after the colon is the field width. The next colon is the digit field. For example:

```
1: Sample Table
Output:[pw:6:2,p1:6:2,d1:10:6,d2:10:6];
```

Here, the parameters `pw` and `p1` are plotted in 6 columns with 2 places after the decimal point, while `d1` and `d2` are displayed in 10 columns with 6 places after the decimal point.

## Output format

For plot, each parameter and value occupies 20 characters of space:

- Characters 1 to 8 are the name of the parameter. Parameters with names longer than 8 characters are permitted within VnmrJ itself but cannot be printed with `pap`.
- Character 9 is always blank.
- Characters 10 to 18 are used for the parameter value. Any parameter value exceeding 9 characters (a file name is a common example) is continued on the next line; in this case, character 19 is a tilde "~", which is used to show continuation.
- Character 20 is always blank.

For printing with the `pap` command, which uses the `ap` parameter template, a "da" listing is printed, starting in column 3, so that the template will typically specify only two columns of output. `ap` can specify more than two columns; but if any parameter is arrayed, the listing of that parameter will overwrite the third column. For printing, the maximum number of lines in each column is 64.

## Modules

A module is a file that contains a small or limited parameter set. The purpose of creating a module is to simplify coding such as pulse sequence programming by grouping the parameters required for a given task or pulse sequence element into a module. That module can then be used to load that group of parameters wherever they are needed.

The modules that are supplied with VJ3.1 are located in the `/vnmr/modules` directory.

bip	globalprefs	localprefs	presat	std1D
ChemPack	gradient	lstd1D	protunelist	stdpar
cpglobals	hadamard	N15calib	protunelistglobal	step
cpProtune	HCcalib	nureference	pureshift	studypar
cpQdefaults	hetero1D	0ch_adiabatic	quant	vchmap
Dch_adiabatic	hetero2D	par2D	sampleglobal	walkupQpars
dpfgse	homo2D	par3D	sel1D	wet
fixpar	homodec	parlp	sel2D	zfilter
fixparglobal	iact1D	parlp1	soggy	zqfilter
flow	impress	parlp2	spinlock	

**Figure 9** Modules in `/vnmr/modules`

### Creating modules

Create modules using the `make` option of the `module` command as follows:

- 1 Use the `module` command, the `make` option, and arguments that include the module name and parameters to add to the module. The following example creates a module called `demo_params` that contains entries for three parameters:

```
param_1, param_2, and param_3:
module('make', 'demo_params', 'param_1 param_2
param_3')
```

- 2 Set parameter values using the `modulename_module` macro. In the above example, the macro to load `param_1`, `param_2`, and `param_3` would be called `demo_params_module` and the `modulename_module` macro will be called when the module is loaded.

#### NOTE

Generic\_module template to build the module macro is available in `/vnmr/maclib`.

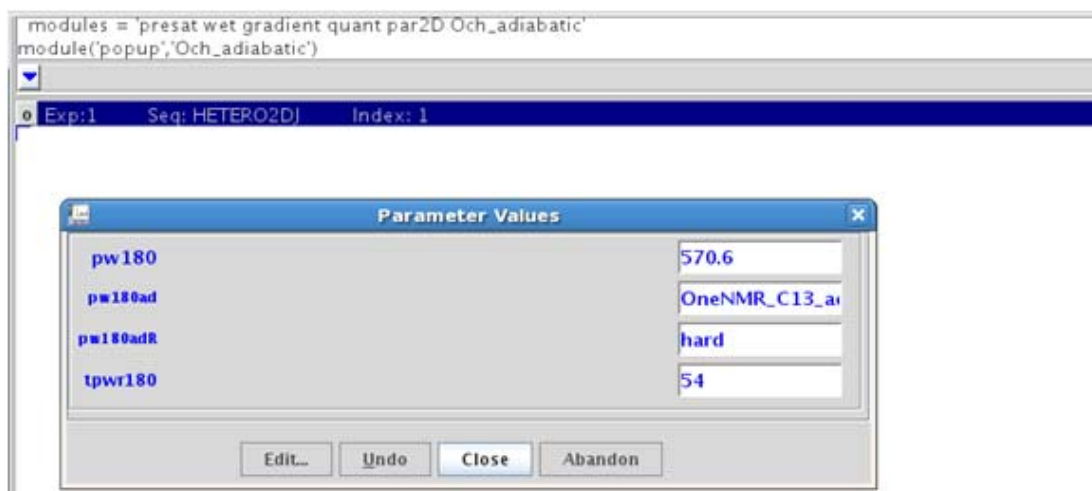
## Viewing modules parameters

The `modules` parameter holds the names of all modules loaded in the current workspace. Enter `modules?` to display the list of current modules.

To see which module parameters are used in the current workspace use the `module` command, `popup` option, and the module name as an argument.

```
module('popup', 'modulename')
```

## Module example—Och\_adiabatic



Using the `Och_adiabatic` module as an example, the `modules` can be used to add parameters for an adiabatic 180-degree pulse using the observe channel. The associated `Och_adiabatic_module` macro parses the probe file, creates the needed shape, and sets resulting parameter values into the current experiment.

To implement a pulse sequence module:

- 1 Create the module:  

```
module('make', 'modulename', 'parametrlist')
```
- 2 Create the corresponding macro:  

```
modulename_module macro
```
- 3 Add the new module name to the current list in the `modules` parameter of the current workspace.
- 4 Use the Edit Parlib utility (**Tools/Study Clones/Edit Parlib**) to create the protocol for the new pulse sequence. The list of active modules will be shown in the Edit Parlib popup and the active modules will be loaded when that protocol is executed.

## User-Written Weighting Functions

The parameter `wfile` can be<XREF> set to the name of the file containing a user-written weighting function. If the parameter `wfile` (or `wfile1` or `wfile2`) does not exist, it can be created with the following command:

```
create('wfile','flag')
setgroup('wfile','processing')
setlimit('wfile',15,0,0).
```

If `wfile` exists, but `wfile=""` (two single quotes), VnmrJ does not look for the file: `wfile` is inactive. To enable user-written weighting functions, set `wfile=filename`, where `filename` is the name of the executable weighting function (enclosed in single quotes) that was created by compiling the weighting function source code with the shell script `wtgen` (a process described in the next section).

VnmrJ first checks if `filename` exists in `wtlib` subdirectory of the user's private directory. If the file exists, VnmrJ then checks if the file `filename.wtp`, which may contain the values for up to ten internal weighting parameters, exists in the current experiment directory. If `filename.wtp` does not exist in the current experiment directory, the ten internal weighting parameters are set to 1.

VnmrJ executes the `filename` program, using the optional file `filename.wtp` as the source for parameter input. The output of the program is the binary file `filename.wtf` in the current experiment directory. This binary file contains the weighting vector that will be read by VnmrJ. The total weighting vector used by VnmrJ is a vector-vector product of this external, weighting vector and the internal VnmrJ weighting vector, the latter being calculated from the parameters `lb`, `gf`, `gfs`, `sb`, `sbs`, and `awc`. The `awc` parameter still provides an overall additive contribution to the total weighting vector. Although the external weighting vector cannot be modified with `wti`, the total weighting vector can be modified with `wti` by modifying the internal VnmrJ weighting vector.

### NOTE

Only a single weighting vector is provided for both halves of the complex data set—real and imaginary data points of the complex pair are always weighted by the same factor.

---

If the `filename` program does not exist in a user's `wtlib`



subdirectory, VnmrJ looks for a text file in the current experiment directory with the name `filename`. This file contains the values for the external weighting function in floating point format (for example, 0.025, but not 2.5e-2) with one value per line. If the number of weighting function values in this file is less than the number of complex FID data points (that is,  $np/2$ ), the user-weighting function is padded out to  $np/2$  points, using the last value in the `filename` text file.

## Writing a weighting function

Weighting functions must follow this format, similar to pulse-sequence programs:

```
#include "weight.h"
wtcalc(wtpntr, npoints, delta_t)
int npoints;          /* number of complex data points */
float *wtpntr,        /* pointer to weighting vector */
delta_t;              /* dwell time */

{
...                  /* user-written part */
}
```

The variable `wtpntr` is a pointer and must be dealt with differently than an ordinary variable such as `delta_t`. `wtpntr` contains the address in memory of the first element of the user-calculated weighting vector; `*wtpntr` is the value of that first element. The `*wtpntr++=x` statement implies that `*wtpntr` is set equal to  $x$  and the pointer `wtpntr` is subsequently incremented to the address of the next element in the weighting vector.

The following examples show the filename program set by `wtfilename=filename`.

**Source file** `filename.c` in a user's `vnmrSYS/wtlib` directory:

```
#include "weight.h"
wtcalc(wtpontr, npoints, delta_t)
int npoints;          /* number of complex data points */
float *wtpontr,      /* pointer to weighting vector */
delta_t;             /* dwell time */

{
int i;
for (i = 0; i < npoints; i++)
    *wtpontr++ = (float) (exp(-(delta_t*i*wtconst[0])));
/* wtconst[0] to wtconst[9] are 10 internal weighting */
/* parameters with default values of 1 and type float. */
}
```

**Optional parameter file** `filename.wtp` in the current experiment directory:

```
0.35    /* value placed in wtconst[0] */
-2.4    /* value placed in wtconst[1] */
...     /* etc. */
```

**Text file** `filename` in the current experiment directory:

```
0.9879  /* value of first weighting vector element */
0.8876  /* value of second weighting vector element */
-0.2109 /* value of third weighting vector element */
0.4567  /* value of fourth weighting vector element */
...     /* etc. */
0.1234  /* value of last weighting vector element */
```

## Compiling the weighting function

The macro/shellsript `wtgen` is used to compile `filename` as set by parameter `wtfilename` into an executable program. The source file is `filename.c` stored in a user's `vnmrSYS/wtlib` directory. The executable file is in the same directory and has the same name as the source file, but with no file extension. The syntax for `wtgen` is `wtgen(file<.c>)` from `VnmrJ` or `wtgen file<.c>` from a shell.

The `wtgen` macro allows the compilation of a user-written weighting function that can subsequently be executed from within `VnmrJ`. The shellsript `wtgen` can be run from within

a shell by typing the name of the shellsript file name, where the `.c` file extension is optional. `wtgen` can also be run from within VnmrJ by executing the macro `wtgen` with the file name in single quotes.

The following functions are performed by `wtgen`:

- Checks for the existence of the `bin` subdirectory in the VnmrJ system directory and aborts if the directory is not found.
- Checks for files `usrwt.o` and `weight.h` in the `bin` subdirectory and aborts if either of these two files cannot be found there.
- Checks for the existence of the user's directory and creates this directory if it does not already exist.
- Establishes, in the `wtlib` directory, soft links to `usrwt.o` and `weight.h` in the directory `/vnmr/bin`.
- Compiles the user-written weighting function, which is stored in the `wtlib` directory, link loads it with `usrwt.o`, and places the executable program in the same directory. Any compilation and/or link loading errors are placed in the file `name.errors` in `wtlib`.
- Removes the soft links to `usrwt.o` and `weight.h` in the `bin` subdirectory of the VnmrJ system directory.

The name of the executable program is the same as that for the source file without a file extension. For example, `testwt.c` is the source file for the executable file `testwt`.

## User-Written

### FID files

Introduce computed data into your experiment by using the command `makefid(input_file <,element_number,format>)`. The `input_file` argument, which is required, is the name of a file containing numeric values, two per line. The first value is assigned to the X (or real) channel; the second value on the line is assigned to the Y (or imaginary) channel. Arguments specifying the element number and the format are optional and may be entered in either order.

The argument `element_number` is any integer larger than 0. If this element already exists in your FID file, the program will overwrite the old data. If not entered, the default is the first element or FID. `format` is a character string with the precision of the resulting FID file and can be specified by one of the following:

'dp=n'	single precision (16-bit) data
'dp=y'	double precision (32-bit) data
'16-bit'	single precision (16-bit) data
'32-bit'	double precision (32-bit) data

If an FID file already exists, `format` is the precision of data in that file. Otherwise, the default for `format` is 32 bits.

The number of points comes from the number of numeric values read from the file. Remember that it reads only two values per line.

If the current experiment already contains a FID, the `format` and the number of points from that present in the FID file cannot be changed. Use the command `rm(curexp+'/acqfil/fid')` to remove the FID.

The `makefid` command does not look at parameter values when establishing the format of the data or the number of points in an element. Thus, if the FID file is not present, it is possible for `makefid` to write a FID file with a header that does not match the value of `dp` or `np`. Use the `setvalue` command if any changes are needed as the active value is in the `processed` tree.

The `makefid` command can modify data returned to an experiment by the `rt` command, because after an `rt`, the FID file in the current experiment, `curexp+'/acqfil/fid'`, may be a symbolic link only, pointing to the saved FID. So, any change to the FID in the experiment will effectively occur on the original, saved data. To avoid alteration of the original data, enter the following sequence of VnmrJ commands on the saved data before running `makefid`:

```
cp(curexp+'/acqfil/fid',curexp+'/acqfil/fidtmp')
rm(curexp+'/acqfil/fid')
mv(curexp+'/acqfil/fidtmp',curexp+'/acqfil/fid')
```

These commands ensure that the FID file in the current experiment is not a symbolic link. The command `writefid(textfile<,element_number>)` writes a text file using data from the selected FID element. The default element number is 1. The program writes two values per line—the first is the value from the X (or real) channel, and the second is the value from the Y (or imaginary) channel.





## 6 User Space Customization

Customize User Space with dousermacro [488](#)

Customizing Default Experiment Parameters using user<pslabel> [489](#)

Customizing Output - Plotting [490](#)



## Customize User Space with `dousermacro`

A new command has been added to VnmrJ 3 which increases the flexibility of the software on a per-user basis by incorporating the concept of a “usermacro” feature into many standard system macros.

When the `dousermacro` command is invoked with the `($0)` argument in any system macro, the user’s local macro library is searched for a macro of the form `user<macroname>`. If such a macro exists, it is executed at that location in the parent macro.

For example:

- The `useroperatorlogin` and `useroperatorlogout` macros are executed in the `operatorlogin` and `operatorlogout` macros.
- The `userbootup` macro is executed in the system `bootup` macro.
- The `userprocess` macro is executed in the system `process` macro.
- The `userplot` macro is executed in the system `plot` macro.
- The `usergo` macro is executed when an acquisition is started.

For a complete list of `dousermacro` calls, in a terminal window type:

```
grep dousermacro /vnmr/maclib/*
```

The `dousermacro` enhances the ability of users and operators to individually change the behavior of their local account without impacting other users while preserving the system macros from modification.



## Customizing Default Experiment Parameters using `user<pslabel>`

Another place that the `dousermacro` philosophy is invoked to allow easy, user specific customizations is the `dousermacro($seqfil)` call that is made by the standard experiment setup macro, `cpsetup`. By invoking the `$seqfil` argument to the `dousermacro` command, the user's local macro library is searched for a `user<pslabel>` macro and, if that macro exists, it is executed as part of the protocol setup macro. This allows each user to customize the behavior of each protocol in their user space without affecting any other users or changing any systemfiles. For example:

- Addition of a macro named `userPROTON` to the local macro library allows the user to change the default set up for PROTON in that user account. If the `userPROTON` macro was simply set to `nt=16`, then every time PROTON was added to a study or invoked in the current workspace in that user's account, the number of transients would be set to 16.
- Creation of a `usergHSQCAD` macro in the local macro library that read:

```
trace='f1' fn=4096 fn1=fn gaussian nt=4 ni=96
nullflg='n' mult=0
```

would set the display to show the heteronuclear axis horizontally, set the Fourier number for `f2` to 4096 points, set the Fourier number for `f1` equal to that of `f2`, calculate the apodization function in both dimensions as a Gaussian decay, set the number of transients to 4, set the number of `t1` increments to 96, turn off the TANGO gradient pulse sequence element for suppression of 1-bond responses, and turn off multiplicity editing for the `gHSQCAD` protocol in this user account.

It is important to note that this `user<pslabel>` feature is invoked in the individual protocol set-up macro. There is no equivalent function for a cloned user study. Rather, each individual protocol in such a study would be parameterized as defined by the `user<pslabel>` macros.

## Customizing Output - Plotting

Individual plotting parameters for each protocol can be adjusted at set-up by the creation of a `<pslabel>_plot` macro in the user's local macro library. For example:

- Creating a macro named `CARBON_plot` that contains `wc=wcmax-20 sc=10` would set the default width of all CARBON plots to 20 mm less than the maximum width available for the current plotter, and then set the starting point of the plot to 10 mm from the right-hand edge of the page.

User level, protocol-specific plotting can be automatically invoked when a plot is generated by the use of the `p1_<pslabel>` macro feature. Such a macro supersedes the system plotting facility for that protocol. For example:

- Creation of a `p1_DQCOSY` macro in the local macro library that read:

```
pcon('pos',32) pcon('neg',4) pap page
```

would produce a default plot of a DQCOSY spectrum that consisted of 32 positive contours, 4 negative contours, and a list of all parameters.

There is also a mechanism that allows individualized plotting for a specific type of sample. For example, if a user wanted to collect data on a number of samples containing a certain molecule and then plot those results in a format specific to those samples. For example, let's assume that a user wanted to create a specific experiment to investigate ibuprofen. The procedure to do so would be:

- Set up all acquisition parameters as desired for the analysis, collect a spectrum on a representative sample of ibuprofen using those parameters, and load that data set into the current workspace.
- Next, create a macro that when executed on the command line generates a plot as desired, including text, integrations, inset regions, etc. For this example, let's assume that this plotting macro is called `ibuprofen_1`.
- Create the `execplot` parameter by typing `create('execplot','string')` on the command line.
- Set the value of the new `execplot` parameter to `ibuprofen_1` by typing `execplot=ibuprofen_1` on the command line.

- Use the `Edit Parlib` tool to create a new protocol called `Ibuprofen` and include the parameter `execplot` in the `lock parameters` field.

A new button named “Ibuprofen” will appear in the experiment selector corresponding to the current experiment and, when executed, this protocol will set up an experiment using the current parameters and use the macro `ibuprofen_1` to control plotting output.





## 7 Panels, Toolbars, and Menus

Parameter Panel, Toolbar, and Menu Properties [494](#)

Using the Panel Editor [495](#)

Panel Elements [508](#)

Creating a New Panel [534](#)

Panel Style Guidelines [540](#)

Graphical Toolbar Menus [545](#)



## Parameter Panel, Toolbar, and Menu Properties

The parameter panels as well as vertical panels, menus, and most toolbars in VnmrJ are built using xml files that are stored in the `/vnmr/templates` directory. While parameter panels are saved under `templates/layout/NAME_OF_PULSESEQUENCE`, vertical panels are saved under `templates/layout/toolPanel`. The System, User, Hardware Toolbar and all Menu files are stored in `templates/VnmrJ/interface`.

An exception is the Graphical Toolbar: being a descendant of the Vnmr tool menu (which used to reside in `/vnmr/menulib`), its files are of similar MAGICAL build and are stored in `/vnmr/menujlib`.

Parameter panel items may display strings, expressions, and parameter values. Some parameter panels are general and are shared with all pulse sequences and some are customized to meet the requirements of individual pulse sequences.

The liquids and solids interfaces use panels in the Start, Acquire, and Process folders. The imaging interface has an additional folder labeled Image. The LC-NMR interface has an additional folder labeled LC/MS. Panels are selected by clicking on the tab at the top of the window. Each panel contains a number of pages, and the pages are selected by clicking on the page tab at the left.

Panels in the Spectroscopy and LC-NMR interfaces use the name of the pulse sequence, `seqfil`. Imaging interface panels utilize the parameter `layout` to select the panels that need to be displayed. The imaging gems protocol sets `layout = 'gems'`. The parameter can be set to `layout = seqfil`. Using the `layout` parameter facilitates sequence development; a panel does not have to be created because a sequence is recompiled under a different name.

To determine whether 2D-type rather than the default 1D-type panels should be displayed, a file "DEFAULT" is copied to the panel directory. See [“Files associated with panels”](#) on page 537 for more information.

## Using the Panel Editor

The Panel Editor is used to customize parameter panels in the VnmrJ interface, as described in the sections:

- “Starting the panel editor” on page 495
- “Editing existing panel elements” on page 497
- “Adding and removing panel elements” on page 499
- “Adding user-defined sampletags” on page 502
- “Saving panel changes” on page 504

### Starting the panel editor

- 1 Click **Edit** on the main menu.
- 2 Select **Parameter Pages...** to display the Panel Editor window. See [Figure 10](#).

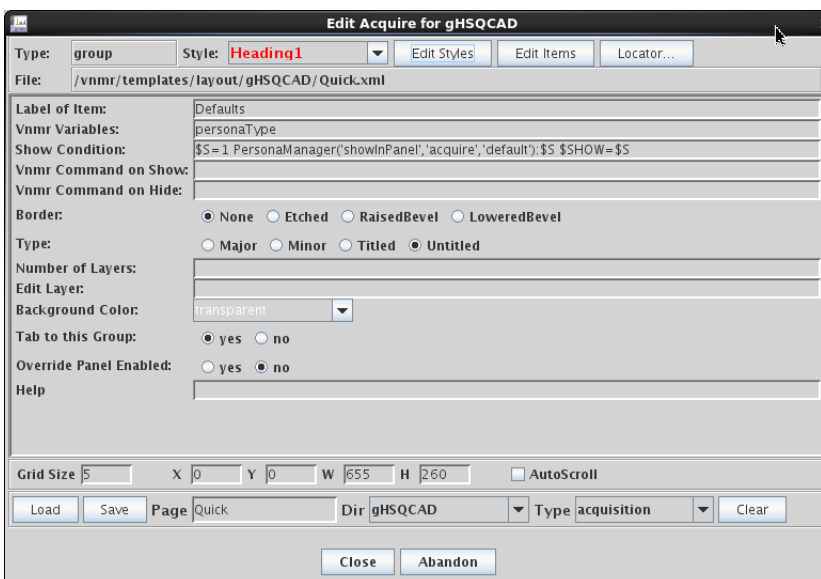


Figure 10 Panel Editor window

3 The current page is displayed in the edit mode with a grid. See Figure 11.

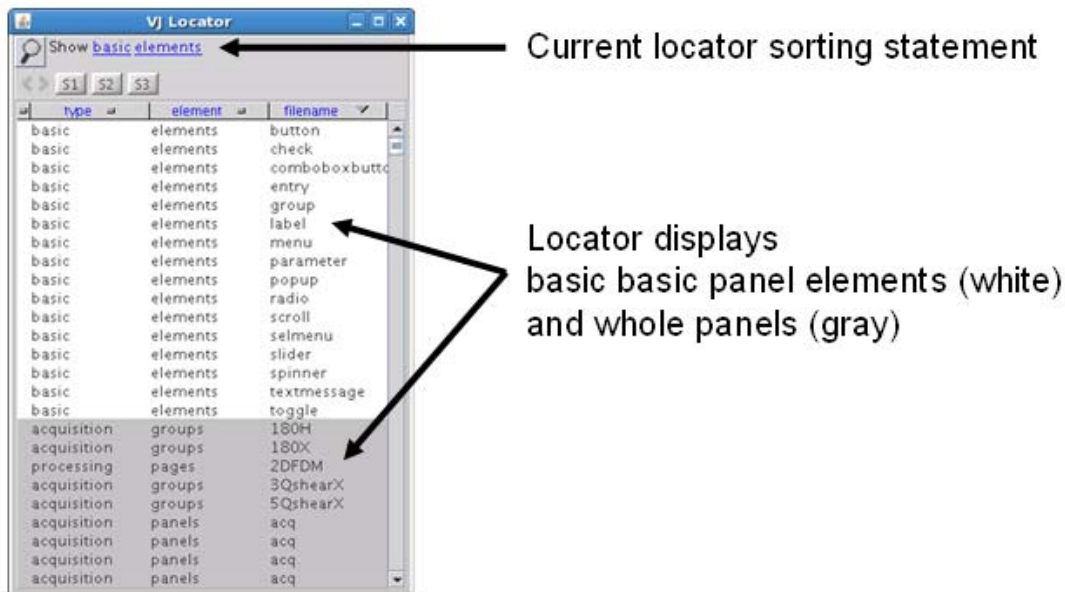
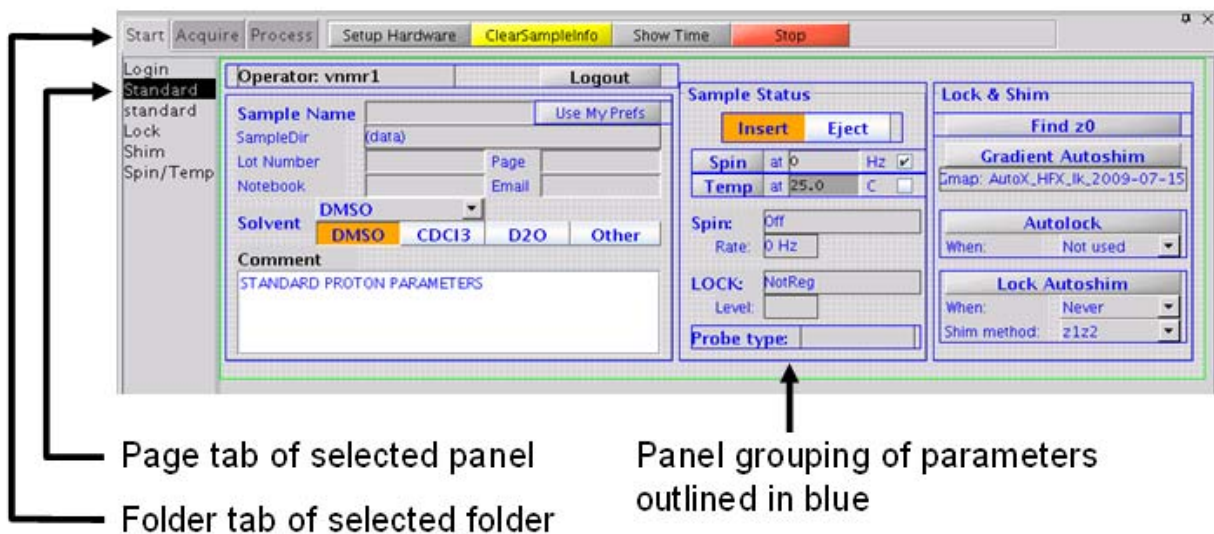


Figure 11 Panel and Locator when Panel Editor is Open. The default grid size is 5.

4 Click **Tools**.

5 Select **Locator** to display the locator panel (Figure 11), and the basic elements used to build a panel.



## Editing existing panel elements

### Selecting a panel element

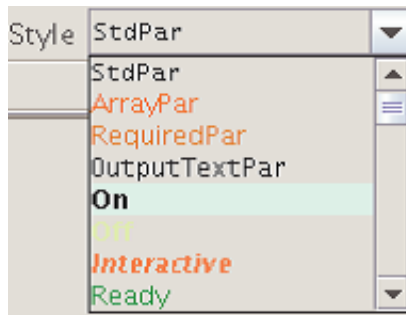
To select a panel element:

- 1 Double-click an element (button, toggle, group, etc.) to select it. The selected element is highlighted in yellow and is ready for editing.
- 2 Double-click an empty area within the group to select a group.
- 3 Double-click an empty area within the page to select a page.
- 4 Double-click an empty area outside a page to select a folder.
- 5 Expand the panel display area until it is larger than the page if there is no area outside the page.

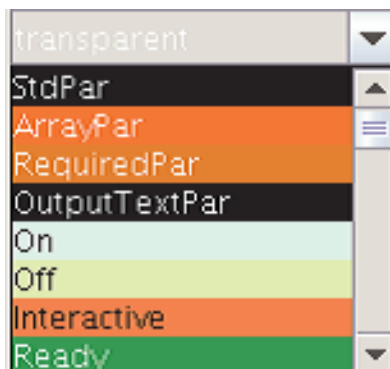
### Viewing or changing a panel element attribute

The panel editor displays the attributes of a selected element. A list of elements and their attributes is given in Panel Elements. The panel editor can also set the following attributes:

**Style** The Style drop-down menu sets the font, style, size, and color of the element.



**Background color** The Background Color drop-down menu sets the background color of the element.



**Location** Move an element to a new location using one of the following methods:

- Drag the element to the desired location with the mouse.
- Use the arrow keys to move the element to the new location.
- Enter the position in the **entry boxes X** (horizontal) and **Y** (vertical) in pixels. The top left corner is X=0, Y=0.

**Size** Resize an element using one of the following methods:

- Drag the edges of the element with the mouse.
- Hold the control key down and resize the element using the arrow keys.
- Enter the size in the **entry boxes W** (Width) and **H** (Height) in pixels.
- Copy an element to a location on the page and press **Enter**.

### Changing the grid size

The default grid size is 5. The grid size can be changed as follows:

- 1 Enter a **new value** in the field next to Grid Size.
- 2 Press **Enter**.

### Editing styles

The **Edit Styles** button opens the **Display Options** editor.



The editor is used for setting the styles of panel elements.

Changing the font, style, size, or color in Display Options changes all elements in the interface of that style.

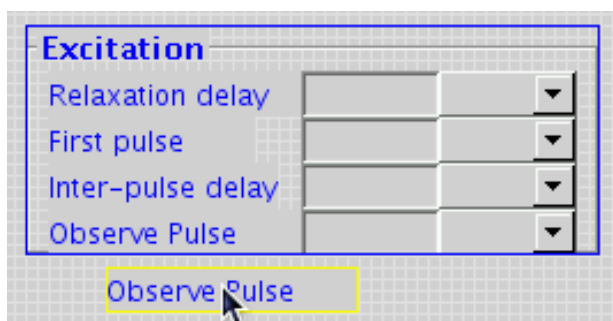
## Adding and removing panel elements

### Selecting an element from the locator

- 1 Select an element in the Locator.
- 2 Drag the element from the locator to the desired position on the page.

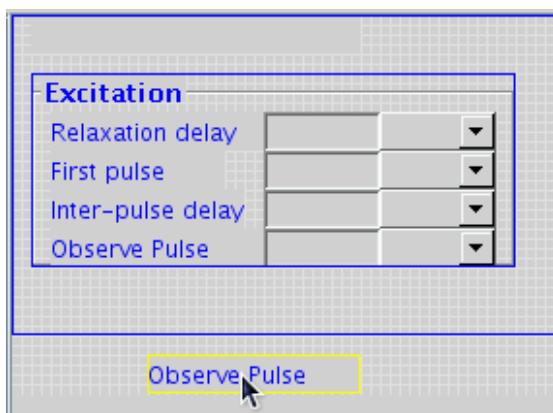
### Copying an element to another location on the same page

- 1 Select an existing element or group of elements by double-clicking it (make sure the borders are highlighted).
- 2 Hold the control key down and drag it to the new location—a new element is automatically created.



### Copying an element between pages within a folder

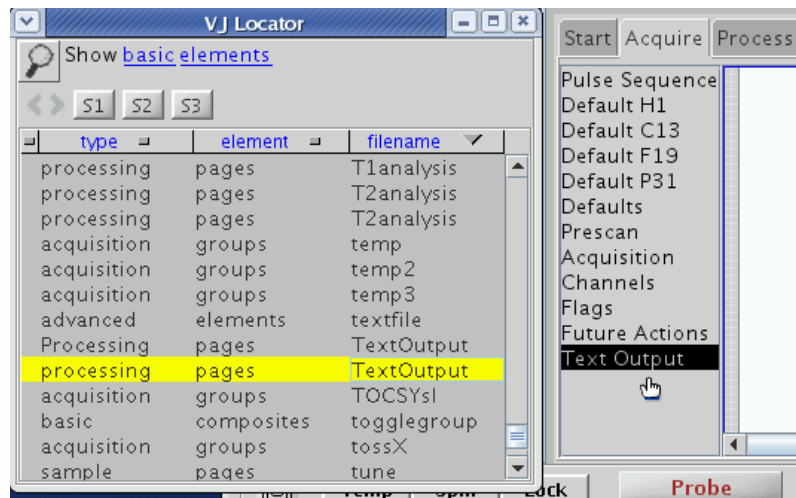
- 1 Select an existing element or group of elements on a page by double-clicking it (make sure that the borders are highlighted).
- 2 Hold the control key down and drag it to a location outside the page. Use the arrow keys to move the copy outside the page if the area outside the page cannot be viewed.
- 3 Select a new page to the left in the page tab list.
- 4 Move the copied element within the new page.



### Creating a new page from the Locator

- 1 Select **Show all elements** in the Locator.
- 2 Find the **page** element in the Locator.
- 3 Drag the **page** element into the parameter panel or into the tab list to the left of the **parameter** panel in the appropriate folder.

**New Page** appears as the tab on the left.



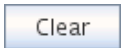
- 4 Change the position and size of the page using one of the following methods:
  - a Use the mouse buttons to click on an edge or corner and drag the page to a new size.
  - b Use the ctrl-arrow keys to resize the page.
  - c Type in values for width (**W**) and height (**H**) in the template editor.

### Copying an existing page from the Locator

- 1 Set the columns of the locator to show **type**, **directory**, and **filename**.
- 2 Find the required page in the Locator.
- 3 Drag the page in to the tab list to the left of the panels in the appropriate folder.

The page will appear as a new tab in the list.

### Removing panel items

- 1 Select the panel element, group, page, or folder to remove by double-clicking it.
- 2 Click the **Clear** button at the lower right corner of the template editor.  This removes all items from the page or folder, or deletes the selected item or group (highlighted with yellow border).

The item can also be dragged to the trash.



### Using the Item Editor

The Item Editor can be used to edit individual visual elements, composite elements or complex groups and pages.

To open the Item Editor,

- 1 Click **Edit** on the main menu.
- 2 Select **Parameter Pages...** to display the Panel Editor window. See [Figure 10](#).
- 3 In the Panel Editor window, click **Edit Items**. The Item Editor window appears. See [Figure 12](#) on page 502.

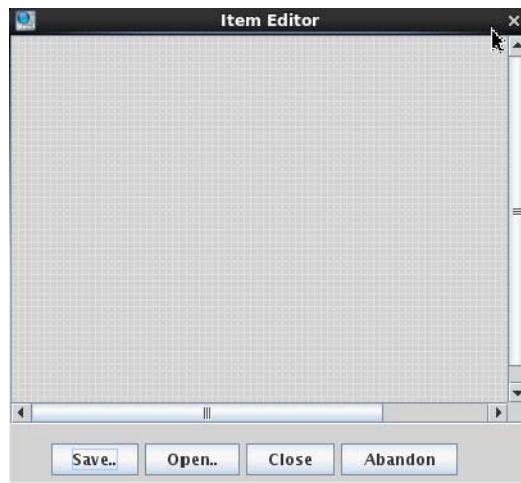


Figure 12 Item Editor window

#### 4 The Item Editor supports the following

- Drag items to the work area from the Locator or Browser
- Move items to and from the work area from the standard panels area using drag and drop, copy-drag and drop
- Right-click a selected item, and use the pop-up menu to cut, copy, or paste. Accelerator key shortcuts for these menu items are available by pressing ctrl-x, ctrl-c, and ctrl-p on your keyboard.
- The properties and attributes of items selected in the Item Editor can be modified and saved as “panelitems” in the same way that these operations are supported for objects selected in the previous editable areas.

5 Click **Save** to save the sets of items in the Item Editor work area.

6 Click **Open** to open a previously-saved set of items.

### Adding user-defined sampletags

A `sampletag` is a parameter that is tightly associated with a given Study. While many common parameters are already included in the list (e.g., `samplename`, `sampleowner`, `studyowner`, `emailaddr`, `researcher`, etc.), it is expected that users may want to add local, site-specific sampletags for their data. Global parameters should not be used as

sampletags.

### How to add a sampletag

As an example for how a new sampletag would be created and used, assume that you want to create a new sampletag called `faculty_name`. The procedure for this is as follows:

- 1 On the command line, type:
 

```
create('faculty_name', 'string')
faculty_name='Prof1' to create the faculty_name
variable, then fill that variable with the value 'Prof1'.
```
- 2 Next, open the **Preferences** panel (**Edit > Preferences**) and click the **SampleTags** tab.
- 3 Under **UserSamp Tags**, add the parameter `faculty_name` to the list and click **Save sample tags**. This adds the new parameter to the `sampletags` list.

The value of the new sampletag will now be stored with every Study, even if that value is an empty string.

### How to add a sampletag to the template

A common reason for creating a new sampletag is the desire to customize the data-save templates. Once the new sampletag is available, the process to add it to the template is straightforward. Continuing with the `faculty_name` example (“[How to add a sampletag](#)” on page 503):

- 1 Open the **Preferences** panel (**Edit > Preferences**) and click the **SampleTags** tab. Add `/${faculty_name}` to the study directory template. Often, this might be just before a `/${studyowner}` variable. This will create a new directory, as needed, based on the values of `faculty_name` and `studyowner`, for each study that is collected in automation (e.g., a folder for each faculty member, with subfolders for each student inside the faculty folders).

Each time someone attempts to run an experiment, the system will now pop up a window and make them fill in a value for `faculty_name` (assuming that they did not populate this variable before submitting their experiment).

- 2 An alternative approach would be to select **Edit > Preferences** and add `faculty_name` to the **UserPrefsRememberance** list under the **UserPref** tab. Once each operator uses the **Edit > Operator Preferences** pop-up to fill in a value for `faculty_name`, the system will store that value for each operator and automatically populate it.

- 3 An additional solution would be to add a widget to the interface that displays the new variable and allows users to change that variable on a per-sample basis. There are two ways to do this:
  - a A Simple Entry Field Swap: In the Study Queue, click **New Study** and then select **Edit > Parameter Pages**. This opens the panel editor tool (Equation 10 on page 495). For more information, see “Using the Panel Editor” on page 495. For this example, the simplest option is to replace an unused field in the current panel with the new parameter name. For example, double-click on the label for “Notebook:” and change the value of “Label of item” in the panel editor to “Faculty:”. Next, double-click on the entry field next to the label in the panel and change the variable `notebook` to `faculty_name` in all three places it appears on the panel editor. Finally, double-click on the outermost blue box in the panel to select the entire group, select “save” from the lower left-hand side of the panel editor, and close the panel editor. The panel will now show a field containing the value of `faculty_name` and allow users to enter this parameter as needed.
  - b Create a Drop-down Menu Item: This requires a more in-depth understanding of the parameter panel editing tools. In short, a menu item can be added to the standard panel as described in part a., but rather than just recycling a fixed entry field, a new “filemenu” field can be created. This would allow users to select the value for `faculty_name` from a drop-down list that is maintained by the administrator. The “solvents” menu item can be used as an example.

## Saving panel changes

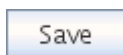
### Saving element or group changes

- 1 Double-click on an element or group.  
The **Save** button is followed by the element type, an entry field for specifying the name of the saved element, and is grayed out until a name is specified.
- 2 Enter a name and press **Enter**.  
The group, including all elements within the group, is saved when saving a group. See Table 2 for saving locations.
- 3 Select a choice from the **Type** menu to set the element type.



The element type may be used for searching for all elements of this type in the Locator. It does not impose any restrictions on the use of the element.

- 4 Press the **Save** button to save the element or group.



Saves the element or group under the name in the panelitems directory. The page then has a reference to the named item within it.

- 5 To reload an element or group from disk, press the **Load** button.



Loads the element or group using the file name in the element or group entry field.

A panel item may be saved in one folder using this method and copied into another folder by dragging it from the Locator.

### Saving page changes

Double-click on an empty space within a page, or click on a tab on the left to select a page.

- 1 The **Save** button is followed by **Page**, an entry field for specifying the page name, and is grayed out if no name is specified.
- 2 Enter a name and press **Enter**.
- 3 Select a choice from the **Dir** menu.

Selecting a pulse-sequence name or layout saves the page in a directory for the pulse sequence or layout. Saving to a default directory makes it available to all sequences.

The default directory is `default_name` if the file `DEFAULT` exists in the directory and contains `set default default_name`. Otherwise, the default directory will be `default`. The directory for many 2D liquids sequences is `default2d`.

- 4 Select a choice from the **Type** menu to set the page type.

The page type is used for searching all pages of this type in the Locator. It does not impose any restrictions on the use of the page.

- 5 Press the **Save** button to save the page in the layout directory.
- 6 Press the **Load** button to reload a page from disk.

### Saving folder changes

Double-click the area outside a page. Expand the panel display area until larger than the page if no area is available.

- 1 The **Save** button is followed by **Folder** and an entry field for specifying the folder name. The folder name must be one of the system types: `sample`, `acq`, `proc`, or `aip` if Imaging.
- 2 Select a directory to save the folder in the **Dir** menu.

The folder is saved in a directory for the pulse sequence or layout if a pulse-sequence name or layout is selected. Select **default** to save it in the default directory available to all sequences.

The default directory is `default_name` if the file `DEFAULT` exists in the directory and has contents set `default default_name`. Otherwise, the default directory will be `default`.

- 3 Select a choice from the **Type** menu to set the folder type.
 

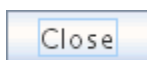
The folder type may be used for searching for all folders of this type in the Locator. It does not impose any restrictions on the use of the folder.
- 4 Press the **Save** button to save the folder specifying the order of pages in it.
- 5 Press the **Load** button to reload a folder from disk.

### Exiting the Panel Editor

Use one of the following options to exit the panel editor:

#### Exit and temporarily save changes

- 1 Click the **Close** button.



Closes the panel editor; unsaved changes are retained only for the current VnmrJ session. Changes are not saved when VnmrJ is exited.

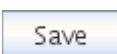
- 2 Changes are saved and retained only for the current VnmrJ session. Changes are lost when VnmrJ is exited.

#### Exit and apply or abandon changes

Apply the changes to the current VnmrJ session, save the

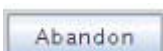
changes for the next VnmrJ session, or abandon the changes as follows:

- 1 Double-click an element or group. The **Save** button is followed by the element type and an entry field for specifying the name of the saved element. **Save** is grayed out until a name is specified.
- 2 Enter a name and press **Enter**. The group, including all elements within the group, is saved when saving a group.
- 3 Select a choice from the **Type** menu to set the element type. The element type may be used for searching for all elements of this type in the Locator. It does not impose any restrictions on the use of the element.
- 4 Do one of the following:
  - Press the **Save** button to save the element or group.



Saves the element or group under the name in the panelitems directory. The page then has a reference to the named item within it.

- Exit and make no changes. Click the **Abandon** button to exit and make no changes.



Exits the panel editor, discards unsaved changes, and reloads previously saved pages.

- 5 Press the **Load** button to reload an element or group from disk.



Loads the element or group using the file name in the element or group entry field.

- 6 Click the **Close** button to exit the panel editor.

## Panel Elements

This section describes how to add and modify panel elements.

### Element style

The font style (plain, bold, italic, bold-italic), size, font, and color that are selected in the **Style** section at the top of the panel editor window determine the appearance of the text associated with the element. The specifics of an element style can be modified by clicking **Edit Styles** in the panel editor window or by selecting **Display Options...** from the top menu **Edit**.

Changing the appearance of a given style will immediately affect any existing elements that use that style.

### Panel element attributes

Commonly used panel element attributes are listed in [Table 59](#).

Table 59 Common Attributes of Panel Elements

Attribute	Description
Label of item	Text label of item.
Icon of item	Icon of item. This is used only for some elements (button, label).
Label justification	Justification of label of item. Choices are Left, Right, and Center.
Vnmr variables	VNMR parameters that can change the Value of item, Enable condition, or Show condition of the item.
Value of item	The value of the item. This string is a MAGICAL expression that sets the value of \$VALUE. The value of some items (check box, radio, toggle) can be either true (1) or false (0). Other items (comboboxbutton, menu, selmenu) match a value from the Value of choices. For other items (entry, textmessage), it is a number or string to display.
Decimal Places	The number of decimal places to truncate to in a real expression in Value of item.
Vnmr command	The command sent when the item is executed or selected. This string is a MAGICAL expression that can use \$VALUE, which is read from the value entered in or set by the item.

Table 59 Common Attributes of Panel Elements

Vnmr command2	The command sent when the item is deselected. This is used only by some items (check box, radio, toggle).
Enable condition	The expression that determines whether an item is active or not. This string is a MAGICAL expression that sets \$ENABLE or \$VALUE, which can evaluate to either active (1), inactive (0), or disabled (-1). A disabled item does not allow the item to change the parameter value, while an inactive item simply changes the background color but still allows parameter entry.
Label of choices	Text labels used in a menu or comboboxbutton.
Value of choices	Values in a menu or comboboxbutton used to set the Vnmr command.
Status parameter	Parameter from the acquisition or hardware status. A status parameter can change the item or value of the item to display. Status parameters cannot be used in combination with MAGICAL expressions. They are mutually exclusive from Vnmr variables. The status parameter is any of the names listed by the command <code>Infostat</code> .
Show condition	The expression that determines whether a group is shown or not. This string is a MAGICAL expression that sets the value of \$SHOW or \$VALUE, which evaluates to show (1) or hide (0).
Vnmr command on show	In a group, the command sent when the group is shown. This string is a MAGICAL expression.
Vnmr command on hide	The command sent when the group is hidden. This string is a MAGICAL expression.
Editable	Sets whether or not text may be entered in the item (yes or no).

## Panel elements

### Button



A button causes an action to occur in VnmrJ. The command behind a button is anything that can be written in a macro or entered on the command line.

The button attributes are:

- Label of item
- Icon of item
- Enable condition
- Vnmr command
- Background color
- Vnmr variables— enable/disable a button based on the parameter value.
- Status parameter—enable/disable a button based on the status parameter value.
- Enable status values— list of status parameter values that enable the button.

Example: The **Acquire Profile** button in the sems layout is a button.

Attribute	Value
Label of item	Acquire Profile
Icon of item	
Vnmr variables	
Enable condition	
Vnmr command	au
Status variables	
Enable status values	
Background color	transparent

**Check**  **Z-filter**

The check box element selects and deselects some mode or state, often as a yes or no selection. It is presented as a small square box to the left of a label.

The attributes of a check box are:

- Value of element – the check box is checked if \$VALUE evaluates to a positive integer.
- Enable condition
- Vnmr command
- Vnmr command2

Example: Inversion Recovery is a check box .

Label of Item	Inversion Recovery
Vnmr variables:	ir
Value of element:	\$VALUE = (ir='y')
Vnmr command:	ir='y'
Vnmr command2:	ir='n'

The commands in the **Vnmr command** and **Vnmr command2** fields are executed when the check box is selected or deselected. The parameter `ir` is set to `y` when the box is selected, and when the box is deselected, `ir` is set to `n`.

The **Value of element** field determines, based on the current value of `ir`, whether the check box is shown as selected or deselected. Thus, this element needs to "listen to" the parameter `ir`, which requires `ir` to be in the **Vnmr variables** list.

**Vnmr command** and the **Value of element** must be consistent.

### Comboboxbutton

The **comboboxbutton** button provides a number of choices using a drop-down menu. Selecting an option from the menu sets the menu item. The Vnmr command is executed by clicking the button, which is specified in the menu, while a **menu** type button executes the command already after the selection has been made.

The attributes of a comboboxbutton are:

- Label of item
- Vnmr variables
- Value of item
- Enable condition
- Vnmr command
- Label of Choices
- Value of Choices
- Editable

An example is a **comboboxbutton** that displays the number of complex transform points  $fn/2$ :

Attribute	Value
Vnmr variables	<code>fn</code>
Value of item	<code>\$VALUE = fn/2</code>
Enable condition	<code>on('fn'):\$ENABLE</code>
Vnmr command	<code>fn = \$VALUE * 2</code>
Label of Choices	<code>"64" "128" "256" "512" "1024" "2048"</code>
Value of Choices	<code>"64" "128" "256" "512" "1024" "2048"</code>
Editable	<code>Yes</code>



**Entry field** 

Use the entry element to directly enter values for VnmrJ parameters.

The entry field attributes are:

- Vnmr Variables
- Value of item
- Enable condition
- Vnmr Command
- Decimal places
- Disable style
- Status parameter

The number of transients or averages, `nt`, is an example:

Attribute	Value
Vnmr Variables:	<code>nt</code>
Value of item:	<code>\$VALUE = nt</code>
Enable condition	
Vnmr Command:	<code>nt = \$VALUE '</code>
Decimal places:	
Disable style:	
Status parameter:	

The entry field created in the above example functions as follows after exiting the editor:

Enter a value into the entry field, for example, 4.

Enter a list of values into an entry field a parameter that can be arrayed (`nt=1,1,1,1,1`). The value is displayed as the string array.

String parameters require enclosing the `$VALUE` with quotes: `n1=' $VALUE`. All math functions must be done to a value prior to assigning it to a VnmrJ parameter, for example, `te` in the imaging interface:

Vnmr Command: `te = $VALUE/1000`

Entering a list of values for `te` in this case, for example, 10, 20, 30, 40, results in dividing only the last value by 1000 and the array ends up with the values 10, 20, 30, 0.04.

Enclose \$VALUE in square brackets, [] to force the math to be applied to all entered values.

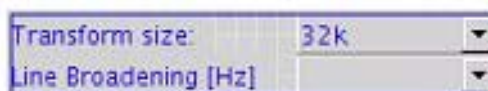
Vnmr Command:  $t_e = [\$VALUE]/1000$

The value is correctly divided by 1000 for all entered values.

This is only an issue for entry fields which allow arbitrary values. The options for the entered value are predefined for menus, check boxes, etc.

Depending on the Enable condition, entries have a different appearance: they are either active (\$ENABLE=1, left in the picture), inactive (0, middle), or disabled (-1, right). A disabled entry does not allow the item to change the parameter value, while an inactive item simply changes the background color but still allows parameter entry.

**Group**



Groups are used to delineate a collection of basic elements that are connected. There are four types of groups: Major, Minor, Titled, and Untitled.

Group attributes are:

- Label of item
- Vnmr variables
- Show condition
- Vnmr Command on Show
- Vnmr Command on Hide
- Type
- Number of Layers
- Edit Layer
- Background Color
- Tab to this Group Disabled
- Override Panel Enabled

While Major and Minor groups exist mainly because of legacy and compatibility reasons, the main difference is whether a group shows a title (if a title was entered) or not.

Major groups are outlined with a visible border and can have a label associated at the top of the group.

Minor groups can have a label but it is not displayed..

Groups can be surrounded by a border that can be either Etched, RaisedBevel, or LoweredBevel, see below for examples. The default for groups other than Major is no border.

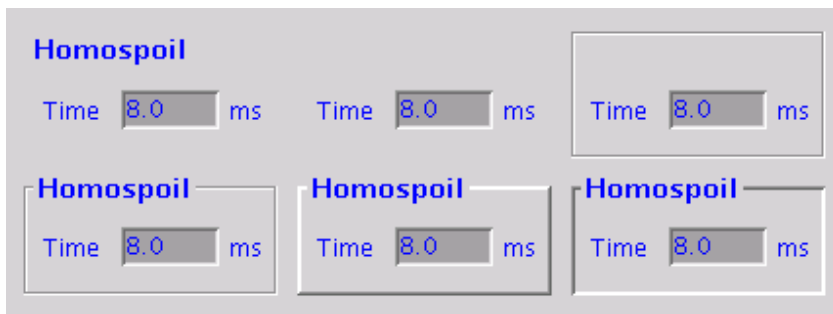


Figure 13 Group types shown for six copies of the same group. First row, left to right: No border/Titled, No border/Untitled, Etched border/Untitled. Second row: Etched/Titled, RaisedBevel/Titled, LoweredBevel/Titled.

Panel groups may also be mutable. Their contents can change depending on other parameters. Multiple layers are available. Set the number of layers > 1 to enable this property. Use the editor to select the current active for editing. Mutable.

Populate a group by first placing the group on the page, sizing it to hold all the elements to be added to it, and placing the individual elements inside the group.

A group cannot be created around existing elements. Placing a new (empty) group on top of existing elements gives the appearance of placing those elements in the group but none of the elements can be selected because they are behind the group. A group cannot be resized to encompass neighboring elements. Place elements inside a group by moving the elements into the group one by one.

An element within a group cannot be resized so that the element extends beyond the group. An element that extends beyond the group is no longer considered part of the group and it cannot be selected from within the group. The element must reside inside the group. The group cannot be moved or resized to cover the element.

Groups can be hidden using Show Condition as `false`. `false` is a negative integer or 0; `true` is a positive integer. For example, a group might only be suitable for display if the

parameter relax is set to 'y'. In this case, the value of the Show Condition can be calculated by a MAGICAL expression, for example:

```
"if relax='y' then $SHOW=1 else $SHOW=0 endif"
alternatively,
'$SHOW=(relax='y')'
```

For this attribute, \$SHOW is equivalent to \$VALUE, and either may be used.

The same space on the page can be used for different groups having a functionality determined by the value of a parameter. Editing this type of group is best done by separating the groups on the page and at the end of the editing process repositioning the groups on top of one another. It is important, in this case, that the groups not fit within each other.

Groups can have multiple layers, hidden (muted) or shown, depending on the show condition for a particular layer. **Number of Layers** sets the number of layers in a group. **Edit Layer** is the number of the layer being edited between 1 and the number of layers. To jump to and edit another layer, first select/ double-click the main panel frame surrounding all, then enter a different layer number to be edited next.

Number of Layers:	3
Edit Layer:	1

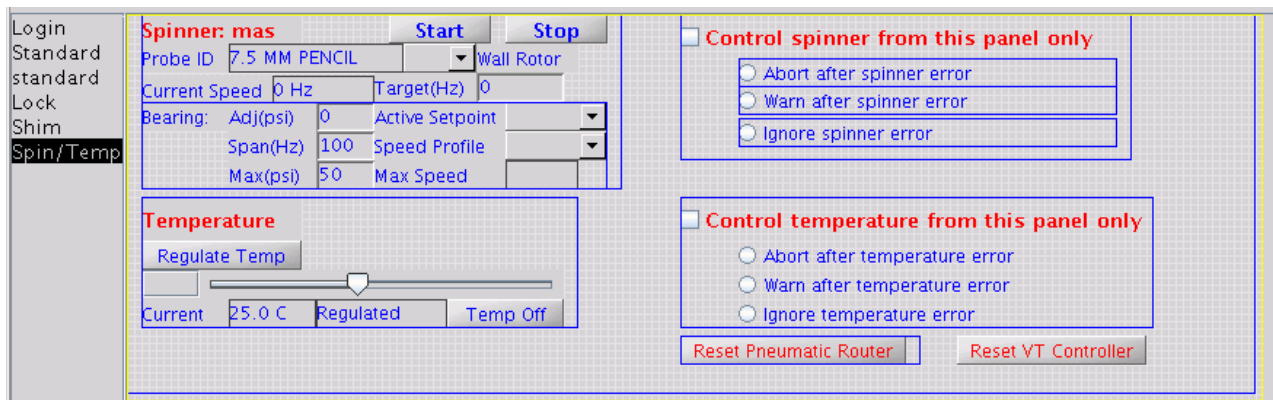


Figure 14 "Spin/Temp" Panel with three layers, layer 1 is currently open in the panel editor. Note the additional panel group (currently yellow/selected)

**Label** 

The label element is a non-interactive label with a pre-defined text. Labels are typically used to give a title, or a description of some other field.

The attributes of a label are:

- Label of item
- Icon of item
- Vnmr variables
- Used for setting the Enable condition
- Enable condition
- Changes label's appearance, but cannot make it invisible. Put the label in a group and set the show condition on the group to make the label invisible.
- Label justification

Attribute	Value
Label of item	Transform size
Icon of item	
Vnmr variables	
Enable condition	
Label justification	Left

Example: **Transform size** in front of an entry field for entering the number of transformed points.

**Menu** 

The menu element gives a number of choices in a drop-down menu. Selecting an option from the menu executes the specified Vnmr command, and displays the last selected option in the menu.

The attributes of a menu are:

- Value of element
- Enable condition
- Vnmr command
- Label of choices
- Value of choices
- Editable

The menu for  $np$  in which the value displayed is the number of complex pairs, that is,  $np/2$  is an example:

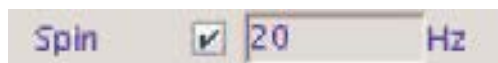
Attribute	Value
Vnmr variables	$np$
Value of element	$\$VALUE = np/2$
Vnmr command	$np = \$VALUE*2$
Label of choices	"32" "64" "128" "256" "512" "1024"
Value of choices	"32" "64" "128" "256" "512" "1024"

The menu displays and returns the number of complex pairs and the value of  $np$  is adjusted through multiplying and dividing by 2. To illustrate that the "Label of choices" and "Value of choices" do not need to be identical, an alternative implementation would be to have the menu return the number of data points but display the number of complex pairs:

Attribute	Value
Value of element	$\$VALUE = np$
Vnmr command	$np = \$VALUE$
Label of choices	"32" "64" "128" "256" "512" "1024"
Value of choices	"64" "128" "256" "512" "1024" "2048"

A value not included in the list of choices can be typed in if the menu is editable. The typed-in value is added to the list of label choices and to the list of value choices.

#### Parameter



The parameter element offers a combination of a label, a check box, an entry field, and a menu (typically used for selecting the units of the parameter in question). Each of these sub-elements is optional. The elements within a parameter are:

Parameter element function	Description
Label	Style permits changing font and units.
Check box	Enables or disables selected conditions.
Entry field	Enter a value with optional decimal places

Units	Label for parameter units. Selected from a menu is optional.
-------	--

**Entry Size** and **Unit Size** establish the size of the label box, and **Units Label** adds the required unit's description at the end of the box.

The label and menu have the same font.

The following example uses fixed units. Type **Label**.

Attribute	Value
Parameter Name:	temp
Label of item:	Temperature
Enable Condition:	<code>vnmrinfo('get','tempExpControl'):\$tc=0 then \$ENABLE=-1 else on('temp'):\$ENABLE endif</code>
Vnmr variables:	temp tin
Checkbox Enable Condition:	<code>vnmrinfo('get','tempExpControl'):\$tc \$ENABLE=\$tc*2-1</code>
Checkbox Value:	<code>on('temp'):\$VALUE</code>
Checkbox Vnmr Command:	<code>on('temp') tin=Y</code>
Checkbox Vnmr Command2:	<code>off('temp') tin='n'</code>
Entry value:	<code>\$VALUE=temp</code>
Entry size:	80
Entry Vnmr Command:	<code>temp=\$VALUE tin=Y</code>
Entry Decimal Places:	1
Entry Disable Styles:	Grayed out
Units Enable:	Label
Units size:	30
Units Label:	C
Units value:	
Units Vnmr Command:	
Menu Choice Label:	
Menu Choice Value:	

<b>Attribute</b>	<b>Value</b>
Parameter Name:	d1
Label of item:	Relaxation Delay
Enable Condition:	\$SHOW=1
Vnmr variables:	d1
Checkbox Enable Condition	:
Checkbox Value:	
Checkbox Vnmr Command:	
Checkbox Vnmr Command2:	
Entry value:	vnmrunits('get','d1'):\$VALUE
Entry size:	50
Entry Vnmr Command:	vnmrunits('set','d1',\$VALUE)
Entry Decimal Places:	2
Entry Disable Styles:	Grayed out
Units Enable:	Menu
Units size:	10
Units Label:	
Units value:	parunits('get','d1'):\$VALUE
Units Vnmr Command:	parunits('set','d1','\$VALUE')
Menu Choice Label:	's' 'ms' 'us'
Menu Choice Value:	'sec' 'ms' 'us'
Radio button	



### Radio button Full Partial Off

Radio buttons are used when a few mutually exclusive choices are available for a particular state. Whenever one option is selected, the others are deselected. If there are more than 3-4 choices, a menu is a better element to use.

A collection of radio buttons related to a particular parameter must be within a single group to separate them from other sets of radio buttons, even if the groups of radio buttons use different parameters. The radio buttons must be explicitly programmed to be mutually exclusive.

The attributes of a radio button are:


- Label of item
- Vnmr variables
- Value of element
- Enable condition
- Vnmr command
- Vnmr command2

An example of two radio buttons is the selection of either **chess** or **wet water suppression** method in the steam protocol. The chess and wet buttons have the following attributes:

Attribute	Chess	WET
Vnmr variables	wss	wss
Value of element	\$VALUE= (wss= 'chess' )	\$VALUE= (wss= 'wet' )
Enable condition	\$VALUE= (ws= 'y' )	\$VALUE= (ws= 'y' )
Vnmr command	wss= 'chess'	wss= 'wet'

Vnmr command2 is not used.

This example also shows an example of using the Enable condition to gray out the radio button if water suppression (ws) is turned off altogether (ws = 'n').

**Scroll** 

The scroll element adjusts a parameter with increment and decrement buttons (typically up and down arrow scroll buttons respectively). The parameter's current value is displayed to the left of the scroll buttons. Each click of the left mouse button on an arrow selects a new value for the parameter in the direction implied by the button to the current parameter. Changes in the parameter, from value to value, do not have to be equally spaced. The parameter value can be a number or a string. “Spinner” on page 524 is similar and does require a defined step size.

The attributes of a scroll are:

- Vnmr variables
- Value of element
- Enable condition
- Vnmr command
- Value of choices

An example is the selection of a decoupling modulation mode from a defined list:

Attribute	Value
Vnmr variables	dmm
Value of element	\$VALUE = dmm
Enable condition	
Vnmr command	dmm = '\$VALUE'
Value of choices	"ccc" "ccw" "ccg"

**Selmenu** 

The selmenu (select menu) element is similar to a menu and gives a number of choices in a drop-down menu. The difference between a menu and a selmenu is that the selmenu always displays the same text (the "Label"), regardless of what was last selected. The exception to this is if the selmenu is "Editable", in which case it displays the last selected option.

The attributes of a selmenu are the same as for a menu.



The slider element adjusts a parameter with a slider. The current value of the parameter is displayed to the left of the slider. The value is incremented by clicking the left mouse button or decremented by clicking the right mouse button in the scale or dragging the slider to the right (increase) or to the left (decrease). The value can also be set by entering it in the entry box to the left of the slider.

The attributes of a slider are:

- Vnmr variables
- Value of element
- Enable condition
- Vnmr command
- Status parameter
- Limits parameter
- Min displayed value
- Max displayed value
- Coarse adjustment value
- Fine adjustment value
- Number of digits to display
- Ms between updates while dragging

The limits parameter is a Vnmr parameter name used to control the range of the slider.

The Min/Max displayed value entries control the range of the slider. These entries are inactive when there is an entry in the status parameter box and the limits box.

The Coarse/Fine adjustment values establish how much the value changes when the slider is moved by clicking the right or left mouse button in the scale.

Ms between updates while dragging establishes the delay in reacting to the slider.

Example:

Attribute	Value
Vnmr variables	
Value of element	
Enable condition	
Vnmr command	setshim ('Z0', \$VALUE)
Status parameter	Z0
Limits parameter	Z0
Min displayed value	
Max displayed value	
Coarse adjustment value	10
Fine adjustment value	1
Number of digits to display	6
Ms between updates while dragging	0

### Spinner

The spinner element applies a defined stepsize change to the value of a parameter using increment and decrement buttons (typically up and down arrow buttons). The range of values is set by the minimum and maximum displayed value attributes. “[Scroll](#)” on page 522 is similar but does not require a defined stepsize.

The attributes of a spinner are:

- Vnmr variables
- Value of item
- Enable condition
- Vnmr command
- Status parameter
- Limits parameter
- Min displayed value
- Max displayed value
- Mouse click adjustment value

Example:

Attribute	Value
Vnmr variables	vtairflow
Value of item	\$VALUE = vtairflow
Enable condition	\$SHOW = (vtairflow>6)
Vnmr command	
Status parameter	
Limits parameter	
Min displayed value	7.0
Max displayed value	25.0
Mouse click adjustment value	1.0

### Textmessage

Acquired Points 1024

The textmessage element displays a non-interactive label that displays an expression and the current value of the expression. The display is updated if the expression's value changes. The expression can not be changed using this element.

The attributes of a textmessage are:

- Status parameter to display
- Vnmr Variables
- Enable condition
- Vnmr expression to display
- Number of digits

Example:

Attribute	Value
Status parameter to display	
Vnmr Variables	np
Enable condition	
Vnmr expression to display	\$VALUE = np/2
Number of digits	0
Toggle button	

**Toggle button**

A toggle button is used to execute one action when the button is selected and another action when the button is deselected. Clicking the toggle button runs an action, and the button changes to appear pressed in. Clicking the button again runs the other action, and the button is released. An example of such a toggle button is FID shimming, which starts FID shim acquisition until the button is clicked a second time to abort acquisition.

A different use for a toggle button is in switching between two mutually exclusive states, such as inserting or ejecting a sample. For this usage, two or more toggle buttons are placed within a group.

The attributes of a toggle button are:

- Label of item
- Vnmr variables
- Value of item
- Enable condition
- Vnmr command  
(executed when the button is selected)
- Vnmr command2  
(executed when the button is deselected)
- Status variables
- Selecting status values
- Enabling status values

Example:

	Button 1	Button 2
Label of item	Insert	Eject
Vnmr variables		
Value of item		
Enable condition		
Vnmr command		
Vnmr command2		
Status variables	air	air
Selecting status values	insert	eject
Enabling status values	eject	insert

## Advanced panel elements

The advanced panel elements are described here and are accessed using the Locator.

### Dial



A dial is a non-interactive display of the value of any parameter. It is typically used to display the FID area while shimming or setting the lock. A parameter cannot be set using the dial.

The attributes of a dial are:

- Vnmr variables
- Value of item
- Enable condition
- Status variable
- Min value
- Max value
- Max value elastic
- Number of hands
- Digital readout
- Show max value
- Max marker color
- Show pic slice
- Show color bars

Example:

Attribute	Value
Vnmr variables	fidarea
Value of item	\$VALUE = fidarea
Enable condition	
Status variable	
Min value	0.0

Max value	1000.0
Max value elastic	no
Number of hands	2
Digital readout	yes
Show maximum value	yes
Max marker color	GraphForeground
Show pie slice	yes
Show color bars	yes

### Filemenu

A filemenu is used when the choices and values associated with a menu are given in a file. This is useful for having a dynamic menu, in which entries may be added or removed (typically by macros) during a session. The filemenu contains pairs of strings on multiple lines. Spaces in strings are contained within double quotes.

The attributes of a filemenu are:

- Label of item
- Selection variables
- Content variables
- Value of item
- Enable condition
- Vnmr command
- Menu source
- Menu type
- Show dot files

An example of a filemenu is the orientation menu in Plan page in the Imaging interface, in which the orientation of previously acquired data is dynamically added to the orientation menu during a study:

Attribute	Value
Label of item	
Selection variables	orient planValue
Content variables	sqdir studyid
Value of item	\$VALUE = planValue



<b>Enable condition</b>	
Vnmr command	<code>iplanType = 0 planValue='\$VALUE' setgplan('\$VALUE')</code>
Menu source	<code>\$VALUE=sqdir+'/plans'</code>
Menu type	<code>file</code>
Show dot files	<code>yes</code>

## Page

The page element has the same attributes as a group element, with a size of a whole page and "Tab to this Group" enabled. The page size may be changed for particular use. The position of the page should always be X=0, Y=0. See the description of the group element for further details.

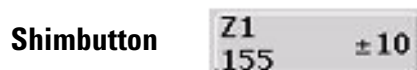
Page attributes are:

- Label of item
- Vnmr variables
- Show condition
- Vnmr Command on Show
- Type
- Vnmr Command on Hide
- Number of Layers
- Edit Layer
- Background Color
- Tab to this Group Enabled
- Override Panel Enabled

## Selfitemenu

The selfitemenu element is similar to a filemenu and gives a number of choices in a drop-down menu. However, the difference between a filemenu and a selfitemenu is that the selfitemenu always displays the same text (the Label), regardless of what was last selected. The exception to this is if the selfitemenu is "Editable", in which case it displays the last selected option.

The attributes of a selfitemenu are the same as for a filemenu.



This button is typically used to adjust the shims. It can be used for any numerical Vnmr or status parameter.

A shimbutton displays a text (the "Label"), the current value of the parameter, and a stepsize. The parameter value is adjusted in steps by clicking the mouse buttons: left and right mouse button to increase and decrease the parameter value, respectively. The value can be entered directly by holding the shift key while clicking the value with the left mouse button. The stepsize can be changed by clicking the middle mouse button (goes through 3 values), or a new stepsize can be entered by holding the shift key while clicking the middle mouse button.

The attributes of a shimbutton are:

- Vnmr variables
- Value of item
- Label
- Vnmr command
- Status variable
- Limits parameter
- Min allowed value
- Max allowed value
- Pointy style
- Rocker style
- Arrow feedback
- Arrow color
- Values wrap around

Example:

Attribute	Value
Vnmr variables	
Value of item	
Label	Z0
Vnmr command	setshim ('Z0', \$VALUE)
Status parameter	Z0
Limits parameter	Z0
Min allowed value	

Max allowed value	
Pointy style	False
Rocker style	True
Arrow feedback	True
Arrow color	GraphForeground
Values wrap around	False

### Shimset

The shimset element displays an entire set of shim buttons, corresponding to the shim hardware.

Z1 0 ±10	X1 0 ±1	X3 0 ±1	Z3X 0 ±1	X4 0 ±1	Z4X 0 ±1	Z3X3 0 ±1
Z2 0 ±1	Y1 0 ±1	Y3 0 ±1	Z3Y 0 ±1	Y4 0 ±1	Z4Y 0 ±1	Z3Y3 0 ±1
Z3 0 ±1	X2 0 ±1	X22 0 ±1	Z2X2Y2 0 ±1		Z3X2Y2 0 ±1	Z4X2Y2 0 ±1
Z4 0 ±1	Y2 0 ±1	Y22 0 ±1	Z2XY 0 ±1		Z3XY 0 ±1	Z4XY 0 ±1
Z5 0 ±1	XY 0 ±1	ZXY 0 ±1	ZX3 0 ±1		Z2X3 0 ±1	Z5X 0 ±1
Z6 0 ±1	Z7 0 ±1	X2Y2 0 ±1	ZX2Y2 0 ±1	ZY3 0 ±1	Z2Y3 0 ±1	Z5Y 0 ±1

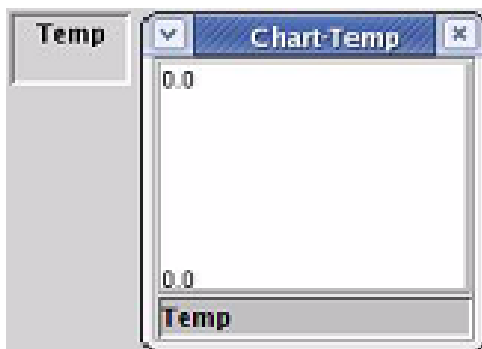
Figure 15 Shimset

The attributes of a shimset are:

- Border type
- Freeze layout
- Vnmr shim set parameter
- Vnmr shim set value
- Shim setting command
- Status parameter for shim
- Vnmr variables for shim
- Vnmr expression for shim

### Status button

A status button displays a popup window that shows the temporal change in a given parameter.



The attributes of a statusbutton are:

- Status Title
- Chart Window Title
- Status Color
- Chart Max Points
- Status Variable
- Display Value
- Vnmr variables
- Min value
- Value of item
- Max value
- Vnmr command
- Chart Show Range
- Vnmr command2
- Chart Background Color
- Chart Foreground Color

Example of the FID shim button.

Attribute	Value
Status Title	FID Shim
Status Color	fg
Status Variable	
Vnmr variables	fidarea
Value of item	\$VALUE=fidarea

Vnmr command	fid_scan
Vnmr command2	aa('exit FID shim')
Chart Window Title	FID Shim area
Chart Max Points	200
Display Value	no
Min value	0
Max value	1000
Chart Show Range	True
Chart Background Color	StdPar
Chart Foreground Color	StdPar

### Textfile

The textfile window displays the text file corresponding to the file path value. The file contents can change and the display updates whenever a file name variable updates. For example, if `n1` is listed as a *file name variable*, setting `n1 = n1` will update the display.

The attributes of a textfile are:

- File name variables
- Value of file path
- Vnmr command
- Enable condition
- Editable
- Wrap lines

## Creating a New Panel

This section describes how to create a new VnmrJ panel that will appear in the Parameter Panel area.

### Writing commands

The panel editor uses the MAGICAL command syntax and a special variable, \$VALUE, which is a local variable for each attribute associated with a panel element.

The variable, \$VALUE, holds the value of the entry in an entry field. The value in the entry field may be a real or string value, the output evaluation of a boolean expression (1, or 0), or the result from the evaluation of an expression. A string variable, in an expression, is set in single quotes, for example: `p1pat = '$VALUE'`. Other local panel variables are \$SHOW and \$ENABLE. See [Table 59](#) on page 508.

### Creating a new panel layout

A new layout is created using either a blank panel or an existing layout that is similar to the required new panel. Use an existing layout (existing user layouts are located in `~/vnmrsys/templates/layout`) by copying an existing user layout and giving it a new name or by copying an existing system layout to the user directory and renaming the copied layout. An example:

```
cp -r /vnmr/imaging/templates/layout/gems
~/vnmrsys/templates/layout/mygems
```

Load the **protocol**, pulse sequence, and/or parameter set that will use the new layout.

- 1 Set `seqfil` or `layout='mygems'`.

The current panels are edited if `seqfil` or `layout` is not set to the new name.

- 2 Open the **panel editor**:
  - a Click **Edit** on the main menu.
  - b Select **Parameter Pages**.
- 3 Modify any page.
- 4 Double click within a page or select the tab to the left. The selected page border is highlighted in yellow.
- 5 Click the **Save** button and save the page.

Save the entire folder if new pages were created:

- 1 Select the folder by double-clicking in the area outside the page grid.

Nothing is highlighted in yellow and at the bottom of the panel editor window, the Save button is followed by the word Folder and an entry field.

- 2 Click the **Save** button.

The folders (three for liquids and four for imaging) must be named: Start, Acquire, Process, Image (imaging only), and LC/MS (LC-NMR only and uses the file LcMs.xml). Arbitrary names cannot be used. The name of the file that governs the Start folder is always `sample.xml`, the Acquire folder `acq.xml`, the Process folder `proc.xml`, and for the fourth folder in the imaging interface `aip.xml` (advanced imaging processing).

Varian standard imaging pages use the following convention:

- Pages in the Start folder start with **samp**
- Pages in the Acquire folder with **acq**
- Pages in the Process folder with **proc**
- Pages in the Image folder with **aip**

Press the **Clear** button outside the current page to **delete all pages** in the current folder. Click the **Abandon** button to reload the folder and pages from disk before clicking the **Save** button if the **Clear** button is clicked by error. See “[VnmrJ Data Files](#)” on page 440.

## Creating a new page

- 1 Select **Show all elements** in the Locator.
- 2 Find the **page** element.
- 3 Drag the **page** element into the tab list to the left of the panels in the appropriate folder.

The New Page appears as the tab on the left.

- 4 Change the size of the page by using one of the following:
  - The mouse buttons and clicking on a corner and dragging the page to a new size.
  - The ctrl-arrow keys to resize the page.
  - Type in values for size W(width) and H(height).

## Defining and populating a page

- 1 Save the page.

Select the entire page by double-clicking somewhere within the page frame, but not on any of the elements within the page. The entire page frame is highlighted in yellow. At the bottom of the panel editor window, the name of the page is shown in an entry field to the right of the Save button.

- 2 Click the **Save** button and save the page.

The keyword **Page** appears between the Save button and the entry field. Use either the original name (already shown) or enter a new name. The page **Type** is provided for refining a search of pages for future use.

Undo any changes made since the most recent save by clicking the **Load** button to reload the file that is saved on disk.

The **Clear** button deletes the current page in the folder. Click the **Abandon** button to reload the page from the disk before clicking the **Save** button or closing the editor if the **Clear** button is clicked by mistake.

## Saving and retrieving a panel element

Save and retrieve a panel element for use in a different panel as follows:

- 1 Double-click an element.

The **Save** button is followed by the element type and an entry field for specifying the name of the saved element.

- 2 Enter a name and press **Enter**.

**Saving a group** saves the group and all elements within the group.

- 3 Set the element type from the menu (acquisition, advanced, basic, display, imaging, plotting, processing, and sample) for easy Locator search.
- 4 To retrieve a saved element, use the Locator to find the element (try sorting alphabetically by name or by type) and drag it on to the required page.
- 5 Saving an individual element is merely a tool to save and retrieve an element for use in a different panel.



## Files associated with panels

See [Table 60](#) for panels and locations.

Table 60 Panels and Locations

Panel owner	Panel	Location	Comment
VnmrJ	Spectroscopy	/vnmr/templates/layout	Named according to the pulse sequence name <code>seqfil</code> and used by all interfaces. Vertical panels are in the "toolPanels" subdirectory.
	Imaging	/vnmr/imaging/templates/layout	Imaging panels determined by the VnmrJ parameter layout. Vertical panels are in the "toolPanels" subdirectory.
	LC-NMR	/vnmr/lc/templates/layout	LC-NMR panels not shared with the experimental interface. Vertical panels are in the "toolPanels" subdirectory.
User	user defined	~user/vnmrsys/templates/layout	User panels named according to the pulse-sequence name or layout.
User	user defined groups	~user/vnmrsys/VnmrJ/panelitems	User groups of choosable name to be used on any user panel.

The panel search path is defined in **Applications...** dialog box in the **Edit** menu, in directories allowed by the VnmrJ administrator under `appdir`. The default is in the user's home directory in `vnmrsys/templates/layout`, then optionally an application-dependent directory (e.g. `/vnmr/imaging/templates/layout`), and finally `/vnmr/templates/layout`.

Panels are first searched for in the pulse-sequence directory, then in the default directory. If the file `DEFAULT` exists in the pulse sequence or layout directory and has contents `set default default_name`, an additional default directory of `default_name` will be searched.

Search path examples:

**Gems panels in the imaging interface:**

- `~/vnmrsys/templates/layout/gems`
- `/vnmr/imaging/templates/layout/gems`
- `/vnmr/templates/layout/gems`
- `~/vnmrsys/templates/layout/default`
- `/vnmr/imaging/templates/layout/default`
- `/vnmr/templates/layout/default`

**COSY panels in the walkup interface, with a DEFAULT file of set default default2d.**

- `~/vnmrsys/templates/layout/COSY`
- `/vnmr/walkup/templates/layout/COSY`
- `/vnmr/templates/layout/COSY`
- `~/vnmrsys/templates/layout/default2d`
- `/vnmr/walkup/templates/layout/default2d`
- `/vnmr/templates/layout/default2d`
- `~/vnmrsys/templates/layout/default`
- `/vnmr/walkup/templates/layout/default`
- `/vnmr/templates/layout/default`

**Panel elements and groups are saved in**

`templates/VnmrJ/panelitems` in either `vnmrsys` or `/vnmr`.

## Sizing panels

The panel size is determined by the number of pixels on the page when the page is created. Scroll bars appear automatically if the panel size is reduced and all the elements in the panel cannot be displayed. Scroll bars also appear automatically when the text is too long to be displayed in elements that support scroll bars. The `Textfile` element is an example. Some elements that contain text do not support scroll bars and display a portion of the text with an "..." to indicate that not all the text is displayed.

The default environment variable setting for VnmrJ is `squish=1.0` to maintain the size of the font when VnmrJ is resized. Set VnmrJ to automatically resize the fonts as follows:

- 1 Log in as the system administrator, typically `vnmr1`.
- 2 Open a terminal window.
- 3 Enter `cd /vnmr/bin`
- 4 Enter `cp VnmrJ VnmrJ.backup`

Edit the `/vnmr/bin/VnmrJ` script and set the parameter `squish=0.5` to automatically resize the fonts. Use `vi` or any ASCII text editor provided with the operating system.

- 5 Save the change and exit the editor.
- 6 Restart VnmrJ to make VnmrJ read the new value of the parameter.

## Panel Style Guidelines

Below are some hints and guidelines that will help you to create panels that are clear, readable and are consistent with the existing set of parameter panels. Examples are shown to illustrate common mistakes and possible improvements.

### Structuring

You may want to give your panel a bit of (sub-)structure by using, for example, etched group borders and group titles:

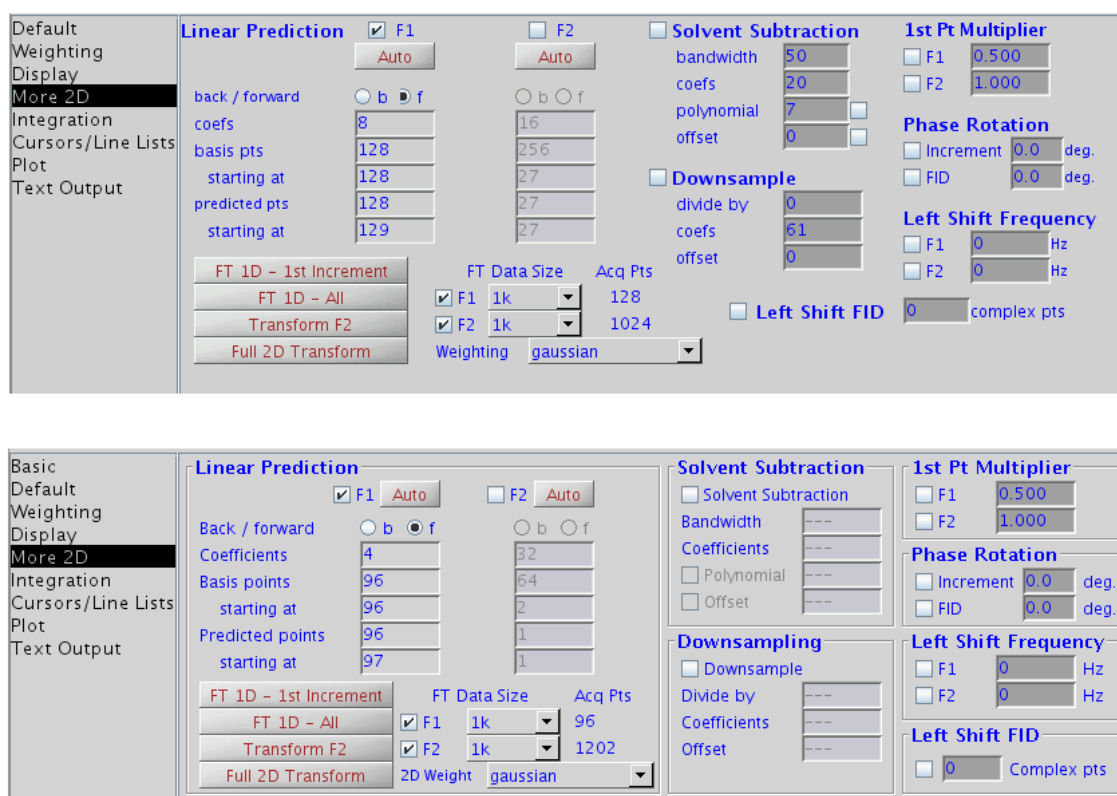


Figure 16 Less structured panel (top) and a panel with "etched" group borders and titles (bottom)

### Alignment

Align items on a panel wherever possible to help maintain clarity and make the panel easy to read. Alignment should be maintained on both left- and right-hand edges of panel labels and widgets. Try to align all units horizontally as well. Vertical alignment (of groups, titles etc.) is often more difficult to achieve other than at the very top of a panel. See Figure 16.

Commonly, a minimum distance of 10 units between items on a panel and (etched, raised, or lowered) group borders should be kept.

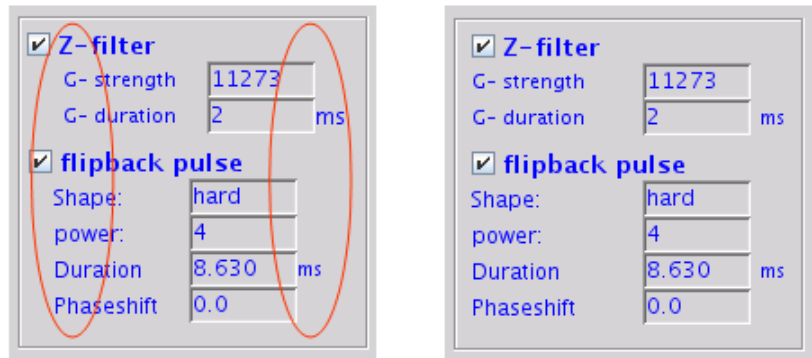


Figure 17 Sloppy alignment and widgets too near to the left group border (left) and improved alignment and distances (right)

### Sizes

Keep the number of different vertical sizes of labels, buttons, or menus, etc. to a minimum – preferably, use only one size. Having too many sizes will make proper alignment difficult. The typical size used for most entries, labels, and buttons is 20.

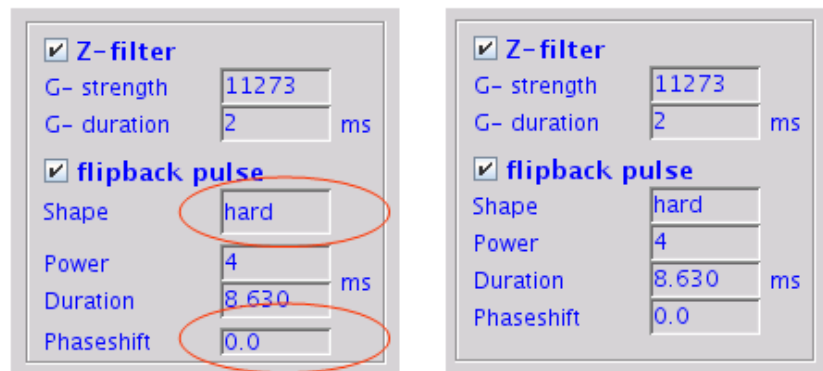


Figure 18 Too many label and entry sizes (left) and only one size (right)

### Font size

Avoid choosing the size of text labels (and textmessages, dropdown menus, etc.) such that it just matches (or is smaller than) the size of the text. This will lead to an undesirable (and frequently unnecessary) automatic scaling down of the text font size, making the text less readable.

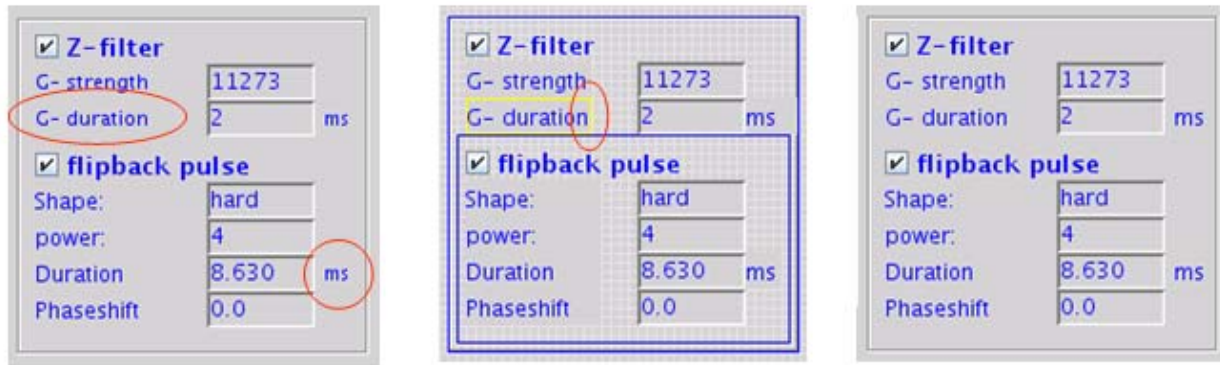


Figure 19 Too small label size with respect to its text length (left,center), improvement by resizing the indicated labels by 5-10 units each (right)

**Text, labels**

Avoid adding colons at the end of a parameter description – they are not necessary. Use a capital letter at the beginning of each entry. At least try to be consistent throughout the panel.



Figure 20 Differences in upper/lower case labels, some labels but not all using colons (left) and cleaned-up panel (right)

**Colors**

Do not to use too many colors on a panel. Use colors only sparingly to highlight items – red, in general, should be avoided unless used only for warnings and/or "dangerous" buttons such as "Delete".

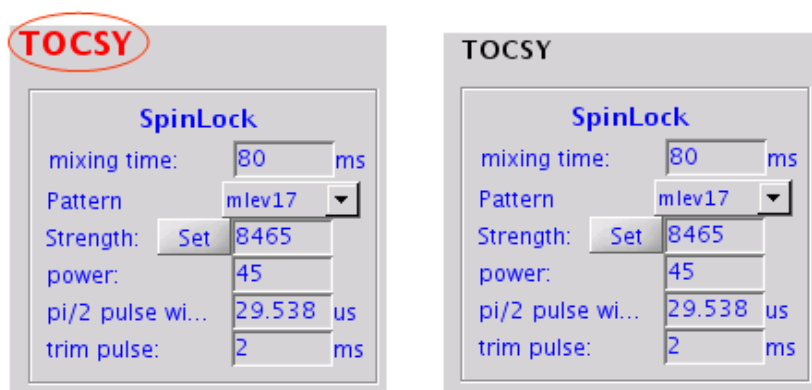


Figure 21 Use of large red font for a normal title (left) and "milder" version (right).

### IUPAC units

Use correct scientific/SI/IUPAC units and abbreviations wherever possible. Correct units: "s", "ms", "Hz", "kHz". Incorrect: "sec", "msec", "hz", "KHz".

### Decimal places

Watch out how many decimal places make sense for a particular entry. For example, in Figure 18, an 'entry' item is used to display (and set) the value of a real (floating-point) parameter `gtz`. The actual value of `gtz` is 1.6 ms but with "Decimal Places" set to "0", the displayed value gets rounded to 2 ms. Here, setting "Decimal Places" to an empty string would be more appropriate – this allows any necessary decimals to be displayed. Similarly, allowing too many decimal places (for example, 4 and more) is often unnecessary as these may be invisible anyway because of the size of the entry.

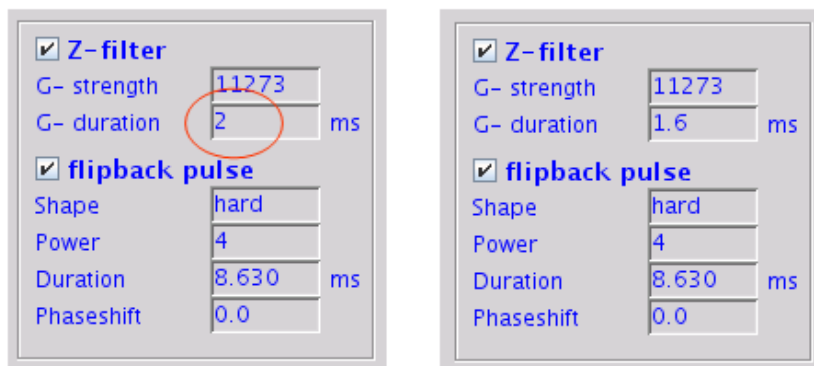


Figure 22 Entry with too few (0) decimal places (left) and correct (1 or leaving decimals undefined) decimal places (right)

Table 61 Common measures and properties on default VnmrJ 3 panels

Attribute	Size in units/pixels
Grid size	5
Horizontal distance of panel items to the left and right group borders	10
Size of Label, entry, and buttons	20
Horizontal distance between two adjacent groups, distance to left panel border	5
Text	General text is usually in font style "Label1" (default blue). Highlighted text may be formatted with "Label3" (default brown) or, when really important, in "Label2" (default red).
Groups	Most large groups on parameter panels are of the type "Titled" and have an Etched border. Title font style: "Heading2" (blue)
Buttons	Buttons are generally labelled with font style "Label3" (default brown) or, for large, very often used convenience buttons, "Heading4" (default brown).



## Graphical Toolbar Menus

This section describes how to edit the graphical toolbar menus.

### Editing the Toolbar menu

The graphics toolbar menu is invoked with the command `menu(filename)` and `filename` is the name of a file in the directory `menujlib` that exists in any of the following locations:

- `/vnmr`
- `$HOME/vnmrsys`
- any `appdir` accessible path

Menus and toolbars are a macro containing other macros. The definitions are plain text files and can be edited using `vi` or any ASCII text editor supplied with the operating system.

### Graphics toolbar parameters

Each button displayed in the graphics toolbar menu is specified by three attributes that are set by the index of three arrayed global parameters: `micon`, `mlabel`, and `mstring`. The following global parameters are associated with the **Graphic Toolbar** menus:

Parameter	Description
<code>micon</code>	Saves the name of the GIF file associated with the button in <code>micon[i]</code> . This parameter is typically arrayed with one icon for each button and is set when a menu is called. A <code>noicon.gif</code> is used if an icon does not exist.
<code>mlabel</code>	Stores the tooltip for a menu button. This parameter is typically arrayed with one tooltip for each button in the menu. This parameter is set whenever a menu is called.  <code>mlabel[i]</code> contains the tooltip for the <code>i</code> th button.
<code>mstring</code>	Stores command or macro strings to be executed when a VnmrJ menu button is clicked. Usually the <code>mstring</code> parameter is arrayed with a string for each button in the menu. The string can be any string of commands that can appear in a macro or on the command line. This parameter is set whenever a menu is called.

---

The following rules apply:

— No new lines (that is, carriage returns) in the text string.

— Single quotes in the text string must be replaced by reverse single quotes (`'...'`) or by the escape sequence back slash with single quote (`\'...'`).

— The length for the text string is subject to a maximum. A menu string can simply contain the name of a macro.

---

## Icons

VnmrJ icons available to all users are `.gif` files located in:  
`/vnmr/iconlib`

Size all button icons to  $24 \times 24$  pixels. Use any graphics editor that can create a `.gif` file.

## Menu file description example, `dconi`

The following is a line by line description of the `dconi` menu file.

The line numbers in the listing for the `dconi` menu file are for reference only and are not part of the file.

The header contains comments and file history. It is not required but it is a good practice to provide this or similar information when creating new or editing existing menu files.

Line 1 is the first line of the menu and checks the graphics mode display.

Lines 2 through 9 establish the conditions for displaying the `dconi` menu.

Lines 10 through 12 initialize the `mlabel`, `micon`, and `mstring` to null strings to clear any traces of a previous menu.

### Button 1

Lines 14 through 22 establish the first button (`$vjm=1`) as a toggle between the cursor and box modes (`crmode='b'`). The temporary parameter `$vjm` is used as button index.

Line 16 sets the label for the button to `Cursor` (`mlabel[$vjm]='Cursor'`) and the icon to `2D1cur.gif` (`micon[$vjm]='2D1cur.gif'`) when the cursor operation is in the box mode (`crmode='b'`).

Clicking on the button changes the button to `Box`  
`(mlabel[$vjm]='Box')` and the icon to `2D2cur.gif`  
`(micon[$vjm]='2D2cur.gif')` when it is in the cursor mode

Line 22 specifies the command to toggle `dconi` between modes:

```
mstring[$vjm]='dconi('toggle')
```

### Button 2

Line 24 through 28 establish the next button (`$vjm=$vjm+1`) and the `mlabel`, `micon`, and `mstring` strings are set to define the name, icon, and `VnmrJ` command string.

### Buttons 3 through 7

Line 29 through 53 increment the index to the next buttons (`$vjm=$vjm+1`) and the `mlabel`, `micon`, and `mstring` strings are set to define the name, icon, and command string.

### Button 8

The button (lines 54 through 57) is similar to buttons 2 through 7 with the inclusion of conditional statement in the parameter `mstring` on line 56.

### Buttons 9 through 11

The buttons (lines 59 through 72) are similar to buttons 2 through 8.

### Buttons 12 and 13

Line 74 is the `if` part of an `if then else endif` condition that reads the value of the parameter `appmode`.

Lines 75 through 84 are the `then` part of the `if then else endif` statement.

Lines 76 through 83 specify button attributes for display scaling if the statement in line 74 is true.

Line 85 is the `else` part of an `if then else endif` statement.

Lines 86 through 95 specify button attributes for display scaling if the statement in line 74 is false.

Line 96 is `endif` part of an `if then else endif` condition.

**Button 14**

This button (lines 98 through 101) is similar to buttons 2 through 7.

**Button 15**

This button (lines 103 through 108) is optionally displayed depending on the value of the parameter `appmode`. The construct is similar to **Button 12** without the `else` statement.

**Button 16**

The return button action (lines 103 through 108) is determined by the conditions set in lines 113, 116, and 119 as part of a nested set of `if then else endif` statements.

Line 122 is the `endif` statement associated with the initial `if then else` on lines 2 through 9.

```
"@(#)dconi 5.9 03/08/07 Copyright (c) 1991-2007 Varian,
Inc. All Rights Reserved."
```

```
" ***** "
" ****  M E N U :   D C O N I   ****  "
" ***** "
```

Line  
number

```

1 graphis:$vjmgd
2 if (($vjmgd <> 'dconi') and ($vjmgd <> 'dpcon')
3 and ($vjmgd <> 'dcon') and ($vjmgd <> 'ds2d')) then
4   if (lastmenu<>'') then
5     menu(lastmenu) lastmenu=''
6   else
7     menu('display_2D')
8   endif
9 else
10 mlabel=''
11 mstring=''
12 micon=''
13
14 $vjm=1
```

```

15     if (crmode = 'b') then
16         mlabel[$vjm]='Cursor'
17         micon[$vjm]='2D1cur.gif'
18     else
19         mlabel[$vjm]='Box'
20         micon[$vjm]='2D2cur.gif'
21     endif
22     mstring[$vjm]='dconi('toggle')'
23
24     $vjm=$vjm+1
25     mlabel[$vjm]='Show Full Spectrum'
26     micon[$vjm]='2Dfull.gif'
27     mstring[$vjm]='mfaction(\mfzoom\',0)'
28
29     $vjm=$vjm+1
30     mlabel[$vjm]='Zoom in'
31     micon[$vjm]='1Dexpand.gif'
32     mstring[$vjm]='mfaction(\mfzoom\',1)'
33
34     $vjm=$vjm+1
35     mlabel[$vjm]='Zoom out'
36     micon[$vjm]='1Dzoomout.gif'
37     mstring[$vjm]='mfaction(\mfzoom\',-1)'
38
39     $vjm=$vjm+1
40     mlabel[$vjm]='Zoom mode'
41     mstring[$vjm]='setButtonMode(2)'
42     micon[$vjm]='ZoomMode.gif'
43
44     $vjm=$vjm+1
45     mlabel[$vjm]='Pan & Stretch Mode'
46     mstring[$vjm]='setButtonMode(3)'
47     micon[$vjm]='1Dspwp.gif'
48
49     $vjm=$vjm+1
50     mlabel[$vjm]='Trace'

```

```

51     mstring[$vjm]='dconi('trace')'
52     micon[$vjm]='2Dtrace.gif'
53
54     $vjm=$vjm+1
55     mlabel[$vjm]='Show/Hide Axis'
56     mstring[$vjm]='if(mfShowAxis=1) then mfShowAxis=0 else
57     mfShowAxis=1 endif repaint'
58     micon[$vjm]='1Dscale.gif'
59
60     $vjm=$vjm+1
61     mlabel[$vjm]='Projections'
62     mstring[$vjm]='newmenu('dconi_proj') dconi('restart')'
63     micon[$vjm]='2Dvhproj.gif'
64
65     $vjm=$vjm+1
66     mlabel[$vjm]='Redraw'
67     mstring[$vjm]='dconi('again')'
68     micon[$vjm]='recycle.gif'
69
70     $vjm=$vjm+1
71     mlabel[$vjm]='Rotate'
72     mstring[$vjm]='if trace='f2' then trace='f1' else
73     trace='f2' endif dconi('again')'
74     micon[$vjm]='2Drotate.gif'
75
76     if appmode='imaging' then
77
78         $vjm=$vjm+1
79         mlabel[$vjm] = 'Scale +7%'
80         mstring[$vjm] = 'vs2d=vs2d*1.07 dconi('redisplay')'
81         micon[$vjm]='2Dvs+20.gif'
82         $vjm=$vjm+1
83         mlabel[$vjm] = 'Scale -7%'
84         mstring[$vjm] = 'vs2d=vs2d/1.07 dconi('redisplay')'
85         micon[$vjm]='2Dvs-20.gif'
86     else

```

```

86     $vjm=$vjm+1
87     mlabel[$vjm] = 'Scale +20%'
88     mstring[$vjm] = 'vs2d=vs2d*1.2 dconi('again')'
89     micon[$vjm]='2Dvs+20.gif'
90     $vjm=$vjm+1
91     mlabel[$vjm] = 'Scale -20%'
92     mstring[$vjm] = 'vs2d=vs2d/1.2 dconi('again')'
93     micon[$vjm]='2Dvs-20.gif'
94
95     endif
96
97     $vjm=$vjm+1
98     mlabel[$vjm]='Phase2D'
99     mstring[$vjm]='newmenu('dconi_phase') dconi('trace')'
100    micon[$vjm]='1Dphase.gif'
101
102    if appmode<>'imaging' then
103        $vjm=$vjm+1
104        mlabel[$vjm]='Peak Picking'
105        mstring[$vjm]='newmenu('112d') dconi('restart')'
106        micon[$vjm]='2Dpeakmainmenu.gif'
107    endif
108
109    $vjm=$vjm+1
110    mlabel[$vjm]='Return'
111    micon[$vjm]='return.gif'
112    if (lastmenu<>'') then
113        mstring[$vjm]='menu(lastmenu) lastmenu='''
114    else
115        if appmode='imaging' then
116            mstring[$vjm]='menu('main')'
117        else
118            mstring[$vjm]='menu('display_2D')'
119        endif
120    endif
121

```

**122** endif





## Appendix A Status Codes

Status Codes 554



## Status Codes

Codes marked with a double asterisk (\*\*) apply only to *Whole Body Imaging* systems.

**Table 62** Acquisition Status Codes

<b>Done codes:</b>	11. FID complete
	12. Block size complete (error code indicates bs number completed)
	13. Soft error
	14. Warning
	15. Hard error
	16. Experiment aborted
	17. Setup completed (error code indicates type of setup completed)
	101. Experiment complete
	102. Experiment started
<b>Error codes:</b>	<b>Warnings</b>
	101. Low-noise signal
	102. High-noise signal
	103. ADC overflow occurred
	104. Receiver overflow occurred*
	<b>Soft errors</b>
	200. Maximum transient completed for single precision data
	201. Lost lock during experiment (LOCKLOST)
	<b>Spinner errors:</b>
	301. Sample fails to spin after 3 attempts to reposition (BUMPFALL)
	302. Spinner did not regulate in the allowed time period (RSPINFAIL)
	303. Spinner went out of regulation during experiment (SPINOUT)
	395. Unknown spinner device specified (SPINUNKNOWN)
	396. Spinner device is not powered up (SPINNOPOWER)

**Table 62** Acquisition Status Codes

397. RS-232 cable not connected from console to spinner (SPINRS232)
398. Spinner does not acknowledge commands (SPINTIMEOUT)
<b>VT (variable temperature) errors:</b>
400. VT did not regulate in the given time vtime after being set
401. VT went out of regulation during the experiment (VTOUT)
402. VT in manual mode after auto command (see Oxford manual)
403. VT safety sensor has reached limit (see Oxford manual)
404. VT cannot turn on cooling gas (see Oxford manual)
405. VT main sensor on bottom limit (see Oxford manual)
406. VT main sensor on top limit (see Oxford manual)
407. VT sc/ss error (see Oxford manual)
408. VT oc/ss error (see Oxford manual)
495. Unknown VT device specified (VTUNKNOWN)
496. VT device not powered up (VTNOPOWER)
497. RS-232 cable not connected between console and VT (VTRS232)
498. VT does not acknowledge commands (VTTIMEOUT)
<b>Sample changer errors:</b>
501. Sample changer has no sample to retrieve
502. Sample changer arm unable to move up during retrieve
503. Sample changer arm unable to move down during retrieve
504. Sample changer arm unable to move sideways during retrieve
505. Invalid sample number during retrieve
506. Invalid temperature during retrieve
507. Gripper abort during retrieve
508. Sample out of range during automatic retrieve
509. Illegal command character during retrieve*
510. Robot arm failed to find home position during retrieve*
511. Sample tray size is not consistent*

**Table 62** Acquisition Status Codes

512. Sample changer power failure during retrieve*
513. Illegal sample changer command during retrieve*
514. Gripper failed to open during retrieve*
515. Air supply to sample changer failed during retrieve*
525. Tried to insert invalid sample number*
526. Invalid temperature during sample changer insert*
527. Gripper abort during insert*
528. Sample out of range during automatic insert
529. Illegal command character during insert*
530. Robot arm failed to find home position during insert*
531. Sample tray size is not consistent*
532. Sample changer power failure during insert*
533. Illegal sample changer command during insert*
534. Gripper failed to open during insert*
535. Air supply to sample changer failed during insert*
593. Failed to remove sample from magnet*
594. Sample failed to spin after automatic insert
595. Sample failed to insert properly
596. Sample changer not turned on
597. Sample changer not connected to RS-232 interface
598. Sample changer not responding*
<b>Shimming errors:</b>
601. Shimming user aborted*
602. Lost lock while shimming*
604. Lock saturation while shimming*
608. A shim coil DAC limit hit while shimming*
<b>Autolock errors:</b>
701. User aborted (ALKABORT)*
702. Autolock failure in finding resonance of sample (ALKRESFAIL)
703. Autolock failure in lock power adjustment (ALKPOWERFAIL)*

**Table 62** Acquisition Status Codes

704. Autolock failure in lock phase adjustment (ALKPHASFAIL)*
705. Autolock failure, lost in final gain adjustment (ALKGAINFAIL)*
<b>Autogain errors</b>
801. Autogain failure, gain driven to 0, reduce $p_w$ (AGAINFAIL)
<b>Hard errors</b>
901. Incorrect PSG version for acquisition
902. Sum-to-memory error, number of points acquired not equal to np
903. FIFO underflow error (a delay too small?)*
904. Requested number of data points (np) too large for acquisition*
905. Acquisition bus trap (experiment may be lost)*
<b>SCSI errors:</b>
1001. Recoverable SCSI read transfer from console*
1002. Recoverable SCSI write transfer from console**
1003. Unrecoverable SCSI read transfer error*
1004. Unrecoverable SCSI write transfer error*
<b>Host disk errors:</b>
1101. Error opening disk file (probably a UNIX permission problem)*
1102. Error on closing disk file*
1103. Error on reading from disk file*
1104. Error on writing to disk file*
<b>RF Monitor errors:</b>
1400. An RF monitor trip occurred but the error status is OK **
1401. Reserved RF monitor trip A occurred **
1402. Reserved RF monitor trip B occurred **
1404. Excessive reflected power at quad hybrid **
1405. STOP button pressed at operator station **
1406. Power for RF Monitor board (RFM) failed **
1407. Attenuator control or read back failed **

**Table 62** Acquisition Status Codes

1408. Quad reflected power monitor bypassed **
1409. Power supply monitor for RF Monitor board (RFM) bypassed **
1410. Ran out of memory to report RF monitor errors **
1411. No communication with RF monitor system **
1431. Reserved RF monitor trip A1 occurred on observe channel **
1432. Reserved RF monitor trip B1 occurred on observe channel **
1433. Reserved RF monitor trip C1 occurred on observe channel **
1434. RF Monitor board (PALI/TUSUPI) missing on observe channel **
1435. Excessive reflected power on observe channel **
1436. RF amplifier gating disconnected on observe channel **
1437. Excessive power detected by PALI on observe channel **
1438. RF Monitor system (TUSUPI) heartbeat stopped on observe channel **
1439. Power supply for PALI/TUSUPI failed on observe channel **
1440. PALI asserted REQ_ERROR on observe channel (should never occur) **
1441. Excessive power detected by TUSUPI on observe channel **
1442. RF power amp: overdrive on observe channel **
1443. RF power amp: excessive pulse width on observe channel **
1444. RF power amp: maximum duty cycle exceeded on observe channel **
1445. RF power amp: overheated on observe channel **
1446. RF power amp: power supply failed on observe channel **
1447. RF power monitoring disabled on observe channel **
1448. Reflected power monitoring disabled on observe channel **
1449. RF power amp monitoring disabled on observe channel **

**Table 62** Acquisition Status Codes

1451. Reserved RF monitor trip A2 occurred on decouple channel **
1452. Reserved RF monitor trip B2 occurred on decouple channel **
1453. Reserved RF monitor trip C2 occurred on decouple channel **
1454. RF Monitor board (PALI/TUSUPI) missing on decouple channel **
1455. Excessive reflected power on decouple channel **
1456. RF amplifier gating disconnected on decouple channel **
1457. Excessive power detected by PALI on decouple channel **
1458. RF Monitor system (TUSUPI) heartbeat stopped on decouple channel **
1459. Power supply for PALI/TUSUPI failed on decouple channel **
1460. PALI asserted REQ_ERROR on decouple channel (should never occur) **
1461. Excessive power detected by TUSUPI on decouple channel **
1462. RF power amp: overdrive on decouple channel **
1463. RF power amp: excessive pulse width on decouple channel **
1464. RF power amp: maximum duty cycle exceeded on decouple channel **
1465. RF power amp: overheated on decouple channel **
1466. RF power amp: power supply failed on decouple channel **
1467. RF power monitoring disabled on decouple channel **
1468. Reflected power monitoring disabled on decouple channel **
1469. RF power amp monitoring disabled on decouple channel **
1501. Quad reflected power too high **
1502. RF Power Monitor board not responding **
1503. STOP button pressed on operator's station **
1504. Cable to Operator's Station disconnected **
1505. Main gradient coil over temperature limit **
1506. Main gradient coil water is off **

**Table 62** Acquisition Status Codes

1507. Head gradient coil over temperature limit **
1508. RF limit read back error **
1509. RF Power Monitor Board watchdog error **
1510. RF Power Monitor Board self test failed **
1511. RF Power Monitor Board power supply failed **
1512. RF Power Monitor Board CPU failed **
1513. ILI Board power failed **
1514. SDAC duty cycle too high **
1515. ILI Spare #1 trip **
1516. ILI Spare #2 trip **
1517. Quad hybrid reflected power monitor BYPASSED **
1518. SDAC duty cycle limit BYPASSED **
1519. Head Gradient Coil errors BYPASSED **
1520. Main Gradient Coil errors BYPASSED **
1531. Channel 1 RF power exceeds 10s SAR limit **
1532. Channel 1 RF power exceeds 5min SAR limit **
1533. Channel 1 peak RF power exceeds limit **
1534. Channel 1 RF Amp control cable error **
1535. Channel 1 RF Amp reflected power too high **
1536. Channel 1 RF Amp duty cycle limit exceeded **
1537. Channel 1 RF Amp temperature limit exceeded **
1538. Channel 1 RF Amp pulse width limit exceeded **
1539. Channel 1 RF Power Monitoring BYPASSED **
1540. Channel 1 RF Amp errors BYPASSED **
1551. Channel 2 RF power exceeds 10s SAR limit **
1552. Channel 2 RF power exceeds 5 min SAR limit **
1553. Channel 2 peak RF power exceeds limit **
1554. Channel 2 RF Amp control cable error **
1555. Channel 2 RF Amp reflected power too high **
1556. Channel 2 RF Amp duty cycle limit exceeded **
1557. Channel 2 RF Amp temperature limit exceeded **
1558. Channel 2 RF Amp pulse width limit exceeded **



**Table 62** Acquisition Status Codes

1559. Channel 2 RF Power Monitoring BYPASSED **
1560. Channel 2 RF Amp errors BYPASSED **



# Index

## Symbols

? (question mark) notation (UNIX), 410  
.(single period) notation (UNIX), 408  
.. (double period) notation (UNIX), 408  
... notation (display template file), 476  
.fdf file extension, 452  
.fid file extension, 440  
", 410  
(semicolon) notation (UNIX), 407  
\* (asterisk) notation (display template), 476  
/ notation (UNIX), 408  
& (ampersand) notation (UNIX), 410  
| (vertical bar) notation (UNIX), 410  
~ (tilde) notation (UNIX), 408

## Numerics

1D data file, 442  
1D display, 448  
1D Fourier transform, 448  
2D data file, 449  
2D FID display, 448  
2D FID storage, 449  
2D hypercomplex data, 442  
2D phased data storage, 449  
3D coefficient text file, 442  
3D parameter set, 442  
3D spectral data default directory, 441

## A

abort current process (UNIX), 410  
acquisition bus trap, 557  
ADC overflow warning, 554  
alias (UNIX), 408  
ampersand (&) character, 410  
ap command, 473  
ap parameter, 473, 477  
arraydim parameter, 448  
arrayed experiment, 448  
ASCII format, 440  
asterisk (\*) character, 410, 476  
auto file, 442  
Autogain, see automatic gain, 557  
Autolock, see automatic lock, 556  
automatic execution of macros, 470  
automatic gain  
  errors, 557  
automatic lock  
  errors, 556  
automation file, 442  
awc parameter, 480

awk command (UNIX), 409

## B

background process (UNIX), 409  
background processing, 412, 413  
binary files, 440  
binary information file, 442  
block size complete, 554  
bs parameter, 554  
buffering in memory, 441

## C

cat command (UNIX), 409  
cd command (UNIX), 409  
change current directory, 409  
checksum of FDF file data, 457  
chmod command (UNIX), 409  
cmp command (UNIX), 409  
coef file, 442  
comparing two files (UNIX), 409  
complex pair of FID data, 446  
compressed data format, 460  
compressed files, 453  
Compressed-compressed data format, 460  
concatenate and display files (UNIX), 409  
config command, 465  
conpar file, 465, 469  
copying files (UNIX), 408, 409  
cp command (UNIX), 408, 409  
create command, 465, 468  
creating  
  directories (UNIX), 409  
  FDF files, 457  
  new parameter, 465  
curpar file, 441, 464  
current experiment files, 441  
current parameter tree, 464  
current parameters text file, 441  
current-type parameter tree, 464

## D

data block, 442  
data block header, 442  
data buffers, 441  
data directory, 441  
data file, 441, 447, 448, 449  
data file header, 442  
data file in current experiment, 450  
data portion of FDF file, 453

data transposition, 448  
data.h file, 443  
datablockhead structure, 445  
datadir3d directory, 441  
datafilehead structure, 443  
date command (UNIX), 409  
dc drift correction, 447  
ddf command, 450  
ddff command, 450  
ddfp command, 450  
DECch, DEC2ch, DEC3ch devices, 184  
delay  
  parameter type, 464  
deleting files (UNIX), 408  
destroy command, 468  
destroygroup command, 468  
dg2 parameter, 474  
Dgroup field, 469  
Dgroup of a parameter, 467  
diff command (UNIX), 409  
differentially compare files (UNIX), 409  
disk blocks, 443  
disk cache buffering, 441  
disk file errors, 557  
display command, 467  
displaying  
  date and time (UNIX), 409  
  FID file, 450  
  file headers, 450  
  memory usage, 450  
  part of file (UNIX), 409  
done codes, 554  
dp parameter, 440  
ds command, 447  
du command (UNIX), 409

## E

ed command (UNIX), 409, 411  
edit command, 411  
editing  
  parameter attributes, 466  
  text files, 409  
end of file (UNIX), 410  
enumerated values of a parameter, 471  
error codes, 554  
error during acquisition, 554  
expn directory file, 441  
extr directory, 441  
extracted 2D planes, 441

## F

f3 file, 442  
 FDF files  
   attach header to data file, 458  
   creating, 457  
   directory naming convention, 452  
   format, 453  
   header format, 453  
   magic number, 454  
   splitting data and header parts, 458  
   transformations of data, 457  
   why developed, 452  
 fdf files, 452  
 fdfgluer command, 458  
 fdfspl command, 458  
 FID complete, 554  
 FID data, 446  
 fid file, 441, 447  
 fid file extension, 440  
 FID files, 440, 450, 484  
 FIFO underflow error, 557  
 file  
   binary format, 440  
   header of binary file, 440  
   protection mode (UNIX), 409  
   text format, 440  
 first point correction, 447  
 flag-type parameter, 464  
 flashc command, 461  
 flexible data format files. See FDF files, 452  
 flush command, 441, 447  
 format of weighting function, 481  
 forward slash notation (UNIX), 408  
 Fourier transform process, 447  
 fread command, 468  
 frequency-type parameter, 464  
 fsave command, 441, 468  
 ft command, 447  
 ft3d command, 442

## G

gedit, 411  
 getvalue command, 466  
 Ggroup, 467, 469  
 global file, 464  
 global list, 184  
 global-type parameter tree, 464  
 grep command (UNIX), 409  
 gripper abort, 555  
 group of parameters, 467  
 groupcopy command, 468

## H

half-transformed spectra, 448  
 head command (UNIX), 409  
 header of FDF file, 453  
 high-noise signal, 554  
 home directory for user (UNIX), 408

host disk errors, 557  
 hypercmplxhead structure, 445

## I

image file names, 452  
 imaginary component of FID data, 446  
 info directory, 442  
 integer-type parameter, 464  
 interferograms, 448

## K

kill command (UNIX), 409

## L

last used parameters text file, 441  
 Linux  
   shell, 411  
   tools, 406  
 list files in a directory (UNIX), 409  
 lists  
   global, 184  
   offset, 184  
 ln command, 408  
 log directory, 442  
 log files, 413, 442  
 login command (UNIX), 409  
 login procedure, 406  
 logout (UNIX), 410  
 lower shell script, 414  
 low-noise signal, 554  
 lp command (UNIX), 409  
 ls command (UNIX), 409

## M

macro  
   automatic execution, 470  
 magic number, 454  
 mail command (UNIX), 409  
 makefid command, 484  
 man command (UNIX), 409  
 manual entry (UNIX), 409  
 matrix transposition, 449  
 maximum value of parameter, 469  
 memory usage by VNMR commands, 450  
 memsize parameter (UNIX), 441  
 mf command, 461  
 mfbk command, 461  
 mfd data command, 461  
 mftrace command, 461  
 minimum value of parameter, 469  
 mkdir command (UNIX), 408  
 move data in FID file, 461  
 move FID commands, 461  
 moving files into a directory, 409  
 mstat command, 450  
 multiple command separator (UNIX), 407  
 multiple trace or arrayed experiments, 448

multiuser protection, 412  
 mv command (UNIX), 408, 409

## N

notation (UNIX), 410  
 np parameter, 557

## O

OBSch device, 184  
 offset lists, 184  
 operating system, 406

## P

package files, 408  
 pap command, 477  
 paramedit command, 466, 473  
 parameter  
   attributes, 469  
   create new parameter, 465  
   enumerable values, 471  
   maximum value, 469  
   minimum value, 469  
   template, 473  
   trees, 464  
   typical parameter file, 471  
   values, 471  
 parameters  
   accessing the value, 466  
   change type, 467  
   conditional display, 475  
   display field width, 476  
   display formats, 477  
   editing attributes, 466  
   get value, 466  
   protection bits, 467  
   step size, 469  
   types, 464  
 paramvi command, 466, 468, 473  
 parent directory (UNIX), 408  
 parmax parameter, 469  
 parmin parameter, 469  
 parstep parameter, 469  
 pattern scanning and processing (UNIX), 409  
 phase file in the current experiment, 450  
 phased 2D data storage, 449  
 phased spectral information, 441  
 phased spectrum, 447  
 phasefile file, 441, 447, 448, 449  
 phase-sensitive 2D NMR, 446  
 pipe, 410  
 pmode parameter, 441  
 print files (UNIX), 409  
 procdat file, 442  
 process status (UNIX), 409  
 processed-type parameter tree, 464  
 procp ar file, 441, 452, 453, 464  
 procp ar3d file, 442  
 protection bits, 467, 470

prune command, 468  
 ps command (UNIX), 409  
 pulse-type parameter, 464  
 pw parameter, 557  
 pwd command, 409

## Q

quadrature detection, 446  
 question mark (?) character, 410

## R

read parameters from a file, 468  
 real component of FID data, 446  
 real-type parameter, 464, 467  
 receiver overflow warning, 554  
 reformatting data for processing, 459  
 reformatting spectra, 462  
 removing an empty directory (UNIX), 408  
 renaming a directory (UNIX), 408  
 renaming a file (UNIX), 408  
 reverse a spectrum, 462  
 reverse FID commands, 461  
 reverse order of data, 461  
 RF monitor errors, 557  
 rfbk command, 461  
 rfdata command, 461  
 rftace command, 461  
 rm command (UNIX), 408  
 rmdir command (UNIX), 408  
 root directory (UNIX), 408  
 RS-232 cable, 555  
 rsapply command, 462  
 rt command, 485  
 run program in background, 410

## S

sample changer  
   errors, 555  
 saved display file, 441  
 SCSI errors, 557  
 searching files for a pattern (UNIX), 409  
 semicolon (;) notation (UNIX), 407  
 send mail to other users (UNIX), 409  
 set3dproc command, 442  
 setdgroup command, 467  
 setenumeral command, 464, 467  
 setgroup command, 467  
 setlimit command, 467  
 setprotect command, 467  
 settype command, 467  
 setvalue command, 467, 484  
 shell command, 411, 412  
 shell programming, Linux and Unix, 414  
 shell scripts, 414  
 single period notation (UNIX), 408  
 sn file, 441  
 sort command (UNIX), 408  
 sort files (UNIX), 408

spell command (UNIX), 409  
 spelling errors check (UNIX), 409  
 spinner errors, 554  
 square brackets notation, 476  
 standard data format, 460  
 step size  
   parameters, 469  
 stored format of a parameter, 468  
 storing multiple traces, 448  
 string template, 473  
 string-type parameter, 464, 467  
 sum-to-memory error, 557  
 svfdf macro, 458  
 svib macro, 458  
 svsis macro, 458  
 Syntax for Controlling Parallel Channels, 130  
 system identification, 410  
 systemglobal-type parameter tree, 465

## T

tabc command, 462  
 tail command (UNIX), 409  
 tape backup (UNIX), 408  
 tar command (UNIX), 408  
 tcapply command, 462  
 template parameters, 473  
 text file, 441  
 text format files, 440  
 textedit command (UNIX), 409, 411  
 tilde character notation (display templates), 477  
 tilde character notation (UNIX), 408  
 total weighting vector, 480  
 transformations of FDF data files, 457  
 transformed complex spectrum storage file, 441  
 transformed phased spectrum storage file, 441  
 transformed spectra storage files, 440  
 two periods notation (UNIX), 408  
 type of parameter, 467  
 types of parameters, 464, 469

## U

uname command (UNIX), 410  
 units command (UNIX), 410  
 UNIX  
   commands, 407  
   shell, 411  
   text commands, 409  
 user-written weighting function, 480

## V

values of a parameter, 471  
 vbg shell script (UNIX), 413  
 vertical bar notation (UNIX), 410  
 vi command (Linux and UNIX), 411  
 vi command (UNIX), 409  
 vi command (VNMR), 411

vi text editor, 466  
 VNMR  
   source code license, 443  
 Vnmr command (UNIX), 412  
 VnmrJ  
   background processing, 413  
 VT errors, 555  
 vtime parameter, 555

## W

w command, 411  
 w command (UNIX), 410  
 warning error codes, 554  
 weighting function, 447, 480, 481  
 who is on the system (UNIX), 410  
 wildcard character (UNIX), 410  
 working directory (UNIX), 408  
 writing parameter buffers into disk files, 441  
 wtcalf function, 481  
 wtf file extension, 480  
 wtfile parameter, 480, 482  
 wtfile1 parameter, 480  
 wtfile2 parameter, 480  
 wtgen shell script, 480, 482  
 wti command, 480  
 wtlb directory, 480, 482  
 wtp file extension, 480

## X

X channel, 446

## Y

Y channel, 446

## Z

zero fill data, 447  
 zip, 408







© Agilent Technologies, Inc.

Printed in USA, 2013

G7446-90520

