VnmrJ User Programming

Varian, Inc. Inova and MercuryPlus NMR Systems
With VnmrJ 2.2D Software
Pub. No. 01-999379-00, Rev. A 0808
VnmrJ User Programming
Varian, Inc. Inova and MercuryPlus NMR Systems
With VnmrJ 2.2D Software
Pub. No. 01-999379-00, Rev. A 0808

Revision history
A Draft A 051908 Initial release for VnmrJ 2.2D

Applicability of manual:
Varian, Inc. Inova and Mercury-vx and Mercury-plus NMR spectrometer systems with
VnmrJ 2.2D software installed.

Technical contributors: Dan Iverson, Boban John, Frits Vosman, Hung Lin, Debbie
Mattiello, George Gray.

Technical writer and editor: Everett Schreiber

Copyright ©2008 by Varian, Inc.
3120 Hansen Way, Palo Alto, California 94304
www.varianinc.com
1-800-356-4437
All rights reserved. Printed in the United States.

The information in this document has been carefully checked and is believed to be
entirely reliable. However, no responsibility is assumed for inaccuracies. Statements in
this document are not intended to create any warranty, expressed or implied.
Specifications and performance characteristics of the software described in this manual
may be changed at any time without notice. Varian reserves the right to make changes in
any products herein to improve reliability, function, or design. Varian does not assume
any liability arising out of the application or use of any product or circuit described
herein; neither does it convey any license under its patent rights nor the rights of others.
Inclusion in this document does not imply that any particular feature is standard on the
instrument.

UNITY, INOVA, MERCURY-VX, MERCURY-PLUS, MERCURY, Varian, Inc. NMR
Spectrometer Systems, VnmrJ, VNMR, MAGICAL II, Magnex, AutoLock, AutoShim,
AutoPhase, limNET, ASM, and SMS are registered trademarks or trademarks of Varian,
Inc.

Dell, the Dell logo, OptiPlex, Precision, Dimension, Inspiron and Axim are registered
trademarks or trademarks of Dell Computer Corporation. Red Hat is a registered
trademark of Red Hat, Inc. Linux is a registered trademark of Linus Torvalds in the
United States and in other countries. Ethernet is a registered trademark of Xerox
Corporation. VxWORKS and VxWORKS POWERED are registered trademarks of
WindRiver Inc. Other product names in this document are registered trademarks or
trademarks of their respective holders.
Overview of Contents

Introduction .......................................................................................................................... 21
Chapter 1. MAGICAL II Programming.................................................................................. 23
Chapter 2. Pulse Sequence Programming........................................................................... 51
Chapter 3. Pulse Sequence Statement Reference............................................................ 133
Chapter 4. Linux Level Programming............................................................................... 267
Chapter 5. Parameters and Data ....................................................................................... 273
Chapter 6. Panels, Toolbars, and Menus ......................................................................... 301
Index .................................................................................................................................. 339
Table of Contents

Introduction 21

Chapter 1. MAGICAL II Programming................................................................. 23

1.1 Working with Macros ...................................................................................... 23
Writing a Macro ................................................................................................... 24
Executing a Macro .............................................................................................. 24
Transferring Macro Output ................................................................................ 26
Loading Macros into Memory ........................................................................... 26

1.2 Programming with MAGICAL ........................................................................... 27
Tokens ................................................................................................................ 28
Variable Types ................................................................................................... 31
Arrays ................................................................................................................ 32
Expressions ........................................................................................................ 34
Input Arguments ................................................................................................ 35
Name Replacement ............................................................................................ 36
Conditional Statements ...................................................................................... 36
Loops ................................................................................................................. 37
Macro Length and Termination ......................................................................... 38
Command and Macro Tracing ........................................................................... 38

1.3 Relevant VnmrJ Commands .............................................................................. 38
Spectral Analysis Tools ..................................................................................... 39
dres Measure linewidth and digital resolution .............................................. 39
dsn Measure signal-to-noise ............................................................................ 39
dsmmax Calculate maximum signal-to-noise ................................................. 39
gtll Get line frequency and intensity from line list ....................................... 39
gtll Get frequency limits of a specified region .............................................. 39
integ Find largest integral in specified region ................................................. 39
mark Determine intensity of the spectrum at a point ...................................... 40
nll Find line frequencies and intensities ......................................................... 40
numreg Return the number of regions in a spectrum ...................................... 40
peak Find tallest peak in specified region ...................................................... 40
select Select spectrum or 2D plane without displaying it .............................. 40

Input/Output Tools ............................................................................................. 40
apa Plot parameters automatically .............................................................. 40
banner Display message with large characters ............................................ 40
clear Clear a window ....................................................................................... 41
echo Display strings and parameter values in text window ......................... 41
format Format a real number or convert a string for output ......................... 41
input Receive input from keyboard .............................................................. 41
lookup Look up and return words and lines from text file ............................ 41
nrecords Determine number of lines in a file ................................................ 41
psgset Set up parameters for various pulse sequences .................................. 41
write Write output to various devices ........................................................... 42

Regression and Curve Fitting ............................................................................ 42
analyze  Generalized curve fitting ........................................  42
autoscale  Resume autoscaling after limits set by scalelimits ....  42
expfit  Least-squares fit to exponential or polynomial curve  42
expl  Display exponential or polynomial curves .................  42
pexpl  Plot exponential or polynomial curves .....................  42
poly0  Display mean of the data in the file regression.inp ....  42
rinput  Input data for a regression analysis .......................  42
scalelimits  Set limits for scales in regression ....................  43

Mathematical Functions ....................................................................................  43
abs  Find absolute value of a number ..............................................  43
acos  Find arc cosine of a number ....................................................  43
asin  Find arc sine of a number ..........................................................  43
atan  Find arc tangent of a number .....................................................  43
atan2  Find arc tangent of two numbers ..............................................  43
averag  Calculate average and standard deviation of input ....  43
cos  Find cosine value of an angle ....................................................  43
exp  Find exponential value of a number ............................  43
ln  Find natural logarithm of a number ..........................  44
sin  Find sine value of an angle .........................................................  44	
tan  Find tangent value of an angle ....................................................  44

Creating, Modifying, and Displaying Macros ...................................................  44
crcm  Create a user macro without using a text editor ...........  44
delcom  Delete a user macro ..........................................................  44
hidecommand  Execute macro instead of command with same name  44
macrocat  Display a user macro on the text window ..............  44
macrocp  Copy a user macro file ........................................................  44
macrodir  List user macros ..............................................................  45
macroedit  Edit a user macro with user-selectable editor .......  45
macrolld  Load a macro into memory ..............................................  45
macrom  Remove a user macro ..........................................................  45
macrosyscat  Display a system macro on the text window .......  45
macrosyscp  Copy a system macro to become a user macro ......  45
macrosysdir  List system macros .....................................................  45
macrosysrm  Remove a system macro ..................................................  45
macrovi  Edit a user macro with vi text editor ..................  45
mstat  Display memory usage statistics ..............................  45
purge  Remove a macro from memory ..............................................  46
record  Record keyboard entries as a macro .........................  46

Miscellaneous Tools ..........................................................................................  46
axis  Provide axis labels and scaling factors .........................  46
beepoff  Turn beeper off .................................................................  46
beepon  Turn beeper on .................................................................  46
bootup  Macro executed automatically when VnmrJ is started  46
exec  Execute a VnmrJ command ......................................................  46
exists  Determine if a parameter, file, or macro exists..........  46
focus  Send keyboard focus to VNMR input window ................  46
gap  Find gap in the current spectrum .........................................  47
getfile  Get information about directories and files .............  47
graphis  Return the current graphics display status .............  47
length  Determine length of a string .............................................  47
listenoff  Disable receipt of messages from send2Vnmr .........  47
listenon  Enable receipt of messages from send2Vnmr ..........  47
login  User macro executed when VnmrJ activated .................  48
Chapter 2. Pulse Sequence Programming ..................................................... 51

2.1 Application Type and Execpars Programming ........................................ 51
    apptypes ............................................................................................................. 52
    execpar Parameters ........................................................................................... 52

2.2 Overview of Pulse Sequence Programming ............................................ 55
    Spectrometer Differences .................................................................................. 55
    Pulse Sequence Generation Directory .............................................................. 55
    Compiling the New Pulse Sequence ................................................................. 56
    Troubleshooting the New Pulse Sequence ......................................................... 57
    Creating a Parameter Table for Pulse Sequence Object Code ......................... 58
    C Framework for Pulse Sequences .................................................................... 58
    Implicit Acquisition .......................................................................................... 60
    Acquisition Status Codes .................................................................................. 60

2.3 Spectrometer Control .................................................................................. 60
    Creating a Time Delay ....................................................................................... 61
    Pulsing the Observe Transmitter ....................................................................... 62
    Pulsing a Non-Observe Transmitter ................................................................ 64
    Pulsing Channels Simultaneously .................................................................... 65
    Setting Transmitter Quadrature Phase Shifts .................................................... 67
    Setting Small-Angle Phase Shifts ...................................................................... 67
    Controlling the Offset Frequency ..................................................................... 69
    Controlling Observe and Decoupler Transmitter Power ................................... 70
    Status and Gating .............................................................................................. 73
    Interfacing to External User Devices ................................................................ 75

2.4 Pulse Sequence Statements: Phase and Sequence Control ....................... 76
    Real-Time Variables and Constants .................................................................. 77
    Calculating in Real-Time Using Integer Mathematics ...................................... 78
    Controlling a Sequence Using Real-Time Variables ......................................... 79
    Real-Time vs. Run-Time—When Do Things Happen? ....................................... 80
    Manipulating Acquisition Variables ................................................................ 80
    Intertransient and Interincrement Delays ......................................................... 81
    Controlling Pulse Sequence Graphical Display ............................................... 82

2.5 Real-Time AP Tables .................................................................................. 83
    Loading AP Table Statements from Linux Text Files ....................................... 83
    Table Names and Statements .......................................................................... 84
    AP Table Notation ............................................................................................ 84
    Handling AP Tables ......................................................................................... 85
2.16 Programming the Performa XYZ PFG Module .......................................................... 127
    Creating Gradient Tables ...................................................................................... 128
    Pulse Sequence Programming ............................................................................... 128

2.17 Imaging-Related Statements .................................................................................. 129
    Real-time Gradient Statements .............................................................................. 131
    Oblique Gradient Statements ............................................................................... 131
    Global List and Position Statements ..................................................................... 131
    Loops Statements .................................................................................................. 131
    Waveform Initialization Statements ..................................................................... 132

2.18 User-Customized Pulse Sequence Generation ...................................................... 132

Chapter 3. Pulse Sequence Statement Reference .................................................... 133

 abort_message          Send an error to VnmrJ and abort the PSG process ............... 140
 acquire                Explicitly acquire data .......................................................... 140
 add                     Add integer values ................................................................. 141
 apovrride               Override internal software AP bus delay ......................... 141
 apshaped_decpulse       First decoupler pulse shaping via AP bus ....................... 142
 apshaped_dec2pulse      Second decoupler pulse shaping via AP bus ..................... 142
 apshaped_pulse          Observe transmitter pulse shaping via AP bus ................. 143
 assign                  Assign integer values ............................................................. 144
 blankingoff             Unblank amplifier channels and turn amplifiers on ............. 145
 blankingon              Blank amplifier channels and turn amplifiers off .............. 145
 blankoff                Stop blanking observe or decoupler amplifier (obsolete) .... 145
 blankon                 Start blanking observe or decoupler amplifier (obsolete) .... 146
 clearap datatable       Zero all data in acquisition processor memory .................. 146
 create_delay_list      Create table of delays ......................................................... 146
 create_freq_list       Create table of frequencies ................................................ 148
 create_offset_list     Create table of frequency offsets ........................................ 148
 dbl                     Double an integer value ......................................................... 151
 dcplrphase              Set small-angle phase of 1st decoupler ............................ 151
 dcplr2phase             Set small-angle phase of 2nd decoupler ............................ 152
 dcplr3phase             Set small-angle phase of 3rd decoupler ............................. 153
 decblank                Blank amplifier associated with first decoupler ................. 153
 dec2blank               Blank amplifier associated with second decoupler .............. 153
 dec3blank               Blank amplifier associated with third decoupler ................. 154
 dec1vloff               Return first decoupler back to “normal” power .................. 154
 dec1vlon                Turn on first decoupler to full power ............................... 154
 decoff                  Turn off first decoupler ...................................................... 154
 dec2off                 Turn off second decoupler .................................................... 155
 dec3off                 Turn off third decoupler ...................................................... 155
 dec0ffset               Change offset frequency of first decoupler ....................... 155
 dec2offset              Change offset frequency of second decoupler ..................... 155
 dec3offset              Change offset frequency of third decoupler ....................... 155
 dec4offset              Change offset frequency of fourth decoupler ...................... 156
 decon                   Turn on first decoupler ....................................................... 156
 dec2on                  Turn on second decoupler .................................................... 156
 dec3on                  Turn on third decoupler ....................................................... 157
 decphase                Set quadrature phase of first decoupler ............................. 157
 dec2phase               Set quadrature phase of second decoupler ......................... 157
 dec3phase               Set quadrature phase of third decoupler ............................ 157
 dec4phase               Set quadrature phase of fourth decoupler ......................... 158
 decpower                Change first decoupler power level .................................... 158
dec2power 
Change second decoupler power level .......................... 158

dec3power 
Change third decoupler power level ............................ 159

dec4power 
Change fourth decoupler power level ........................... 159

decprgoff 
End programmable decoupling on first decoupler .......... 159

dec2prgoff 
End programmable decoupling on second decoupler ...... 160

dec3prgoff 
End programmable decoupling on third decoupler .. 160

decprgon 
Start programmable decoupling on first decoupler ........ 160

dec2prgon 
Start programmable decoupling on second decoupler ....... 161

decpulse 
Pulse first decoupler transmitter with amplifier gating ... 162

decpwr 
Set first decoupler high-power level, class C amplifier ... 162

decpwrf 
Set first decoupler fine power .................................. 163

dec2pwrf 
Set second decoupler fine power ................................ 163

dec3pwrf 
Set third decoupler fine power ................................... 163

decr 
Decrement an integer value ....................................... 164

decrgpulse 
Pulse first decoupler with amplifier gating .................. 164

dec2rgpulse 
Pulse second decoupler with amplifier gating ............... 165

dec3rgpulse 
Pulse third decoupler with amplifier gating ................. 166

dec4rgpulse 
Pulse fourth decoupler with amplifier gating ................ 166

decshaped_pulse 
Perform shaped pulse on first decoupler ...................... 167

dec2shaped_pulse 
Perform shaped pulse on second decoupler .................... 168

dec3shaped_pulse 
Perform shaped pulse on third decoupler ...................... 169

decspinlock 
Set spin lock waveform control on first decoupler ......... 170

dec2spinlock 
Set spin lock waveform control on second decoupler ...... 170

dec3spinlock 
Set spin lock waveform control on third decoupler ....... 171

decstepsize 
Set step size for first decoupler ............................... 171

dec2stepsize 
Set step size for second decoupler ............................. 172

dec3stepsize 
Set step size for third decoupler .............................. 172

decunblank 
Unblank amplifier associated with first decoupler ... 172

dec2unblank 
Unblank amplifier associated with second decoupler .. 173

dec3unblank 
Unblank amplifier associated with third decoupler ....... 173

delay 
Delay for a specified time ........................................ 173

dhpflag 
Switch decoupling from low-power to high-power .......... 173

divn 
Divide integer values .............................................. 174

dps_off 
Turn off graphical display of statements ..................... 174

dps_on 
Turn on graphical display of statements ....................... 174

dps_show 
Draw delay or pulses in a sequence for graphical display . 174

dps_skip 
Skip graphical display of next statement ................... 177

delsenz 
Execute succeeding statements if argument is nonzero .... 177

dehardloop 
End hardware loop .................................................. 178

dendif 
End execution started by ifzero or elsenz ................. 178

dendloop 
End loop .................................................................. 178

dendmsloop 
End multislice loop ................................................ 178

dendpeoloop 
End phase-encode loop ............................................. 179

dgate 
Device gating (obsolete) .......................................... 180

dgetarray 
Get arrayed parameter values ..................................... 180

dgetelem 
Retrieve an element from a table ............................. 180

dgetorientation 
Read image plane orientation ................................. 181

dgetstr 
Look up value of string parameter ......................... 182

dgetval 
Look up value of numeric parameter ......................... 182

dG_Delay 
Generic delay routine ............................................... 183

dG_Offset 
Frequency offset routine ......................................... 183

dG_Power 
Fine power routine .................................................. 183

dG_Pulse 
Generic pulse routine ............................................... 183
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>hwdshiminit</td>
<td>Initialize next delay for hardware shimming</td>
<td>184</td>
</tr>
<tr>
<td>hlv</td>
<td>Find half the value of an integer</td>
<td>184</td>
</tr>
<tr>
<td>hsdelay</td>
<td>Delay specified time with possible homospoil pulse</td>
<td>185</td>
</tr>
<tr>
<td>idecpulse</td>
<td>Pulse first decoupler transmitter with IPA</td>
<td>186</td>
</tr>
<tr>
<td>idecrgpulse</td>
<td>Pulse first decoupler with amplifier gating and IPA</td>
<td>186</td>
</tr>
<tr>
<td>idelay</td>
<td>Delay for a specified time with IPA</td>
<td>187</td>
</tr>
<tr>
<td>ifzero</td>
<td>Execute succeeding statements if argument is zero</td>
<td>187</td>
</tr>
<tr>
<td>incdelay</td>
<td>Set real-time incremental delay</td>
<td>188</td>
</tr>
<tr>
<td>incgradient</td>
<td>Generate dynamic variable gradient pulse</td>
<td>188</td>
</tr>
<tr>
<td>incr</td>
<td>Increment an integer value</td>
<td>189</td>
</tr>
<tr>
<td>indirect</td>
<td>Set indirect detection</td>
<td>189</td>
</tr>
<tr>
<td>init_rfpattern</td>
<td>Create rf pattern file</td>
<td>189</td>
</tr>
<tr>
<td>init_gradpattern</td>
<td>Create gradient pattern file</td>
<td>191</td>
</tr>
<tr>
<td>initdelay</td>
<td>Initialize incremental delay</td>
<td>191</td>
</tr>
<tr>
<td>initparms_sis</td>
<td>Initialize parameters for spectroscopy imaging sequences</td>
<td>192</td>
</tr>
<tr>
<td>initval</td>
<td>Initialize a real-time variable to specified value</td>
<td>192</td>
</tr>
<tr>
<td>iobspulse</td>
<td>Pulse observe transmitter with IPA</td>
<td>192</td>
</tr>
<tr>
<td>ioffset</td>
<td>Change offset frequency with IPA</td>
<td>193</td>
</tr>
<tr>
<td>ipulse</td>
<td>Pulse observe transmitter with IPA</td>
<td>193</td>
</tr>
<tr>
<td>ipwrf</td>
<td>Change transmitter or decoupler fine power with IPA</td>
<td>194</td>
</tr>
<tr>
<td>ipwm</td>
<td>Change transmitter or decoupler lin. mod. power with IPA</td>
<td>194</td>
</tr>
<tr>
<td>irgpulse</td>
<td>Pulse observe transmitter with IPA</td>
<td>194</td>
</tr>
<tr>
<td>lk_hold</td>
<td>Set lock correction circuitry to hold correction</td>
<td>195</td>
</tr>
<tr>
<td>lk_sample</td>
<td>Set lock correction circuitry to sample lock signal</td>
<td>195</td>
</tr>
<tr>
<td>loadtable</td>
<td>Load table elements from table text file</td>
<td>196</td>
</tr>
<tr>
<td>loop</td>
<td>Start loop</td>
<td>196</td>
</tr>
<tr>
<td>loop_check</td>
<td>Check that number of FIDs is consistent with number of slices, etc.</td>
<td>197</td>
</tr>
<tr>
<td>maggradient</td>
<td>Simultaneous gradient at the magic angle</td>
<td>197</td>
</tr>
<tr>
<td>magradpulse</td>
<td>Gradient pulse at the magic angle</td>
<td>198</td>
</tr>
<tr>
<td>mashapedgradient</td>
<td>Simultaneous shaped gradient at the magic angle</td>
<td>198</td>
</tr>
<tr>
<td>mashapedgradpulse</td>
<td>Simultaneous shaped gradient pulse at the magic angle</td>
<td>199</td>
</tr>
<tr>
<td>mod2</td>
<td>Find integer value modulo 2</td>
<td>200</td>
</tr>
<tr>
<td>mod4</td>
<td>Find integer value modulo 4</td>
<td>200</td>
</tr>
<tr>
<td>modn</td>
<td>Find integer value modulo n</td>
<td>200</td>
</tr>
<tr>
<td>msloop</td>
<td>Multislice loop</td>
<td>200</td>
</tr>
<tr>
<td>mult</td>
<td>Multiply integer values</td>
<td>201</td>
</tr>
<tr>
<td>obl_gradient</td>
<td>Execute an oblique gradient</td>
<td>202</td>
</tr>
<tr>
<td>oblique_gradient</td>
<td>Execute an oblique gradient</td>
<td>202</td>
</tr>
<tr>
<td>obl_shapedgradient</td>
<td>Execute a shaped oblique gradient</td>
<td>203</td>
</tr>
<tr>
<td>obls_shapedgradient</td>
<td>Execute a shaped oblique gradient</td>
<td>203</td>
</tr>
<tr>
<td>obsblank</td>
<td>Blank amplifier associated with observe transmitter</td>
<td>205</td>
</tr>
<tr>
<td>obsoffset</td>
<td>Change offset frequency of observe transmitter</td>
<td>205</td>
</tr>
<tr>
<td>obspower</td>
<td>Change observe transmitter power level</td>
<td>205</td>
</tr>
<tr>
<td>obsprgoff</td>
<td>End programmable control of observe transmitter</td>
<td>206</td>
</tr>
<tr>
<td>obsprgon</td>
<td>Start programmable control of observe transmitter</td>
<td>206</td>
</tr>
<tr>
<td>obspulse</td>
<td>Pulse observe transmitter with amplifier gating</td>
<td>206</td>
</tr>
<tr>
<td>obswrf</td>
<td>Set observe transmitter fine power</td>
<td>207</td>
</tr>
<tr>
<td>obsstepsize</td>
<td>Set step size for observe transmitter</td>
<td>207</td>
</tr>
<tr>
<td>obsunblank</td>
<td>Unblank amplifier associated with observe transmitter</td>
<td>207</td>
</tr>
<tr>
<td>offset</td>
<td>Change offset frequency of transmitter or decoupler</td>
<td>208</td>
</tr>
<tr>
<td>pbox_ad180</td>
<td>Generate adiabatic 180 deg. shapes using Pbox</td>
<td>209</td>
</tr>
<tr>
<td>pbox_mix</td>
<td>Generate mixing shapes using Pbox</td>
<td>209</td>
</tr>
</tbody>
</table>
pboxHT_F1  Generate arbitrary Hadamard encoded shapes in F1 using Pbox 210
pboxHT_F1e Generate Hadamard encoded excitation shapes in F1 using Pbox 210
pboxHT_F1i Generate Hadamard encoded inversion shapes in F1 using Pbox 210
pboxHT_F1s Generate Hadamard encoded sequential inversion shapes . 211
pboxHT_F1r Generate Hadamard encoded refocusing shapes in F1 using Pbox 211

pe_gradient Oblique gradient with phase encode in one axis .................. 212
pe2_gradient Oblique gradient with phase encode in two axes .................. 212
pe3_gradient Oblique gradient with phase encode in three axes ................ 213
pe_shapedgradient Oblique shaped gradient with phase encode in one axis ... .. 213
pe2_shapedgradient Oblique shaped gradient with phase encode in two axes ... .. 214
pe3_shapedgradient Oblique shaped gradient with phase encode in three axes .. .. 215
peloop Phase-encode loop ....................................................................... 215
phase_encode_gradient Oblique gradient with phase encode in one axis ........ 216
phase_encode3_gradient Oblique gradient with phase encode in three axes ........ 217
phase_encode_shapedgradient Oblique shaped gradient with PE in one axis .......... 217
phase_encode3_shapedgradient Oblique shaped gradient with PE in three axes ....... 218
phaseshift Set phase-pulse technique, rf type A or B ............................. 219
poffset Set frequency based on position ................................................. 220
poffset_list Set frequency from position list ........................................ 220
position_offset Set frequency based on position ...................................... 220
position_offset_list Set frequency from position list ................................ 221
power Change power level ................................................................. 221
psg_abort Abort the PSG process ............................................................. 222
pulse Pulse observe transmitter with amplifier gating ............................... 222
putCmd Send a command to VnmrJ from a pulse sequence ......................... 223
pwrf Change transmitter or decoupler fine power .................................... 224
pwr  Change transmitter or decoupler linear modulator power .................... 224
rcurovision  Turn on receiver gate and amplifier blanking gate .................... 225
rcvoff  Turn off receiver gate and amplifier blanking gate .......................... 225
readuserap Read input from user AP register ........................................ 226
recoff Turn off receiver gate only ......................................................... 227
recon Turn on receiver gate only ............................................................ 227
rgpulse Pulse observe transmitter with amplifier gating ................................ 227
rgradient  Set gradient to specified level ................................................ 228
r1power Change power level ................................................................. 228
rlpwrf Set transmitter or decoupler fine power (obsolete) .......................... 229
rlpwr Set transmitter or decoupler linear modulator power .......................... 229
rotate Sets the standard oblique rotation angles ...................................... 230
rot_angle Sets user defined oblique rotation angles .................................. 230
rotoperiod Obtain rotor period of MAS rotor .......................................... 230
rotorsync Gated pulse sequence delay from MAS rotor position .................... 230
setautoincrement Set autoincrement attribute for a table .......................... 232
setdivnfactor Set divn-return attribute and divn-factor for table ................... 232
setreceiver Associate the receiver phase cycle with a table .......................... 232
setstatus Set status of observe transmitter or decoupler transmitter ............. 233
settable Store an array of integers in a real-time table ............................. 233
setuserap Set user AP register ............................................................... 234
shapelpulse Perform shaped pulse on observe transmitter ............................ 234
shaped_pulse Perform shaped pulse on observe transmitter ............................ 234
shapedgradient Generate shaped gradient pulse ........................................ 235
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shaped2Dgradient</td>
<td>Generate arrayed shaped gradient pulse</td>
</tr>
<tr>
<td>shapedincgradient</td>
<td>Generate dynamic variable gradient pulse</td>
</tr>
<tr>
<td>shapedvgradient</td>
<td>Generate dynamic variable shaped gradient pulse</td>
</tr>
<tr>
<td>simpulse</td>
<td>Pulse observe and decouple channels simultaneously</td>
</tr>
<tr>
<td>sim3pulse</td>
<td>Pulse simultaneously on 2 or 3 rf channels</td>
</tr>
<tr>
<td>sim4pulse</td>
<td>Simultaneous pulse on four channels</td>
</tr>
<tr>
<td>simshaped_pulse</td>
<td>Perform simultaneous two-pulse shaped pulse</td>
</tr>
<tr>
<td>sim3shaped_pulse</td>
<td>Perform a simultaneous three-pulse shaped pulse</td>
</tr>
<tr>
<td>sli</td>
<td>Set SLI lines</td>
</tr>
<tr>
<td>sp#off</td>
<td>Turn off specified spare line (Inova #=1 to 5)</td>
</tr>
<tr>
<td>sp#on</td>
<td>Turn on specified spare line (Inova #=1 to 5)</td>
</tr>
<tr>
<td>spinlock</td>
<td>Control spin lock on observe transmitter</td>
</tr>
<tr>
<td>starthardloop</td>
<td>Start hardware loop</td>
</tr>
<tr>
<td>status</td>
<td>Change status of decoupler and homospoil (z-shim coil)</td>
</tr>
<tr>
<td>statusdelay</td>
<td>Execute the status statement with a given delay time</td>
</tr>
<tr>
<td>stepsize</td>
<td>Set small-angle phase step size</td>
</tr>
<tr>
<td>sub</td>
<td>Subtract integer values</td>
</tr>
<tr>
<td>text_error</td>
<td>Send a text error message to VnmrJ</td>
</tr>
<tr>
<td>text_message</td>
<td>Send a message to VnmrJ</td>
</tr>
<tr>
<td>tsadd</td>
<td>Add an integer to table elements</td>
</tr>
<tr>
<td>tsdiv</td>
<td>Divide an integer into table elements</td>
</tr>
<tr>
<td>tsmult</td>
<td>Multiply an integer with table elements</td>
</tr>
<tr>
<td>tssub</td>
<td>Subtract an integer from table elements</td>
</tr>
<tr>
<td>ttadd</td>
<td>Add a table to a second table</td>
</tr>
<tr>
<td>ttdiv</td>
<td>Divide a table into a second table</td>
</tr>
<tr>
<td>ttmult</td>
<td>Multiply a table by a second table</td>
</tr>
<tr>
<td>tssub</td>
<td>Subtract a table from a second table</td>
</tr>
<tr>
<td>txphase</td>
<td>Set quadrature phase of observe transmitter</td>
</tr>
<tr>
<td>vagradient</td>
<td>Variable angle gradient</td>
</tr>
<tr>
<td>vgradpulse</td>
<td>Variable angle gradient pulse</td>
</tr>
<tr>
<td>var_active</td>
<td>Checks if the parameter is being used</td>
</tr>
<tr>
<td>vashapedgradient</td>
<td>Variable angle shaped gradient</td>
</tr>
<tr>
<td>vashapedgradpulse</td>
<td>Variable angle shaped gradient pulse</td>
</tr>
<tr>
<td>vdelay</td>
<td>Set delay with fixed timebase and real-time count</td>
</tr>
<tr>
<td>vdelay_list</td>
<td>Get delay value from delay list with real-time count</td>
</tr>
<tr>
<td>vfreq</td>
<td>Select frequency from table</td>
</tr>
<tr>
<td>vgradient</td>
<td>Set gradient to a level determined by real-time index</td>
</tr>
<tr>
<td>voffset</td>
<td>Select frequency offset from table</td>
</tr>
<tr>
<td>vsetuserap</td>
<td>Set user AP register using real-time variable</td>
</tr>
<tr>
<td>warn_message</td>
<td>Send a warning message to VnmrJ</td>
</tr>
<tr>
<td>xgate</td>
<td>Gate pulse sequence from an external event</td>
</tr>
<tr>
<td>xmtroff</td>
<td>Turn off observe transmitter</td>
</tr>
<tr>
<td>xmtron</td>
<td>Turn on observe transmitter</td>
</tr>
<tr>
<td>xmtrphase</td>
<td>Set transmitter small-angle phase</td>
</tr>
<tr>
<td>zero_all_gradients</td>
<td>Zero all gradients</td>
</tr>
<tr>
<td>zgradpulse</td>
<td>Create a gradient pulse on the z channel</td>
</tr>
</tbody>
</table>

**Chapter 4. Linux Level Programming**

4.1 Linux and VnmrJ

4.2 Linux Reference Guide

Command Entry

File Names
File Handling Commands ................................................................. 268
Directory Names .............................................................................. 268
Directory Handling Commands ......................................................... 268
Text Commands ................................................................................ 269
Other Commands .............................................................................. 269
Special Characters ............................................................................ 269

4.3 Linux Commands Accessible from VnmrJ ........................................ 270
Opening a Text Editor from VnmrJ ...................................................... 270
Opening a Shell from VnmrJ ............................................................... 270

4.4 Background VNMR ........................................................................ 270
Running VNMR Command as a Linux Background Task ..................... 270
Running VNMR Processing in the Background .................................... 271

4.5 Shell Programming ........................................................................ 272
Shell Variables and Control Formats .................................................. 272
Shell Scripts ....................................................................................... 272

Chapter 5. Parameters and Data ......................................................... 273

5.1 VnmrJ Data Files ........................................................................... 273
Binary Data Files .............................................................................. 273
Data File Structures .......................................................................... 275
VnmrJ Use of Binary Data Files ............................................................ 278
Storing Multiple Traces ...................................................................... 279
Header and Data Display .................................................................. 280

5.2 FDF (Flexible Data Format) Files .................................................. 280
File Structures and Naming Conventions ............................................. 280
File Format ....................................................................................... 281
Header Parameters ........................................................................... 282
Transformations ............................................................................... 284
Creating FDF Files .......................................................................... 284
Splitting FDF Files ........................................................................... 285

5.3 Reformatting Data for Processing .................................................. 285
Standard and Compressed Formats ..................................................... 286
Compress or Decompress Data ............................................................ 287
Move and Reverse Data ..................................................................... 287
Table Convert Data ........................................................................... 287
Reformatting Spectra ........................................................................ 287

5.4 Creating and Modifying Parameters .............................................. 288
Parameter Types and Trees ................................................................. 289
Tools for Working with Parameter Trees ............................................. 289
Format of a Stored Parameter ............................................................. 292

5.5 Modifying Parameter Displays in VNMR ....................................... 294
Display Template ............................................................................... 294
Conditional and Arrayed Displays ...................................................... 296
Output Format ................................................................................... 297

5.6 User-Written Weighting Functions ................................................. 297
Writing a Weighting Function ............................................................. 298
Chapter 6. Panels, Toolbars, and Menus ......................................................... 301

6.1 Parameter Panel Features ............................................................................. 301

6.2 Using the Panel Editor .................................................................................. 301
   Starting the Panel Editor ............................................................................. 301
   Editing Existing Panel Elements .................................................................. 303
   Adding and Removing Panel Elements ....................................................... 304
   Saving Panel Changes ............................................................................... 305
   Exiting the Panel Editor ............................................................................ 307

6.3 Panel Elements ............................................................................................. 307
   Element Style ............................................................................................. 308
   Panel Element Attributes ......................................................................... 308
   Panel Elements ......................................................................................... 309

6.4 Creating a New Panel ................................................................................... 323
   Writing Commands .................................................................................... 323
   Creating a New Panel Layout ................................................................. 324
   Creating a New Page ............................................................................... 324
   Defining and Populating a Page ............................................................... 325
   Saving and Retrieving a Panel Element ................................................... 325
   Files Associated with Panels .................................................................... 326
   Sizing Panels ......................................................................................... 327

6.5 Graphical Toolbar Menus ........................................................................... 327
   Editing the Toolbar Menu ......................................................................... 327
   Graphics Toolbar Parameters .................................................................... 328
   Icons ....................................................................................................... 328
   Menu File Description Example, dconi ..................................................... 328

Index 339
List of Figures

Figure 1. Amplifier Gating ............................................................................................................ 62
Figure 2. Pulse Observe and Decoupler Channels Simultaneously ........................................ 66
Figure 3. Waveform Generator Offset Delay ............................................................................ 122
Figure 4. Magnet Coordinates as Related to User Coordinates. ............................................. 283
Figure 5. Single-String Display Template with Output .......................................................... 295
Figure 6. Multiple-String Display Template ........................................................................... 296
Figure 7. Panel Editor Window .............................................................................................. 302
Figure 8. Panel and Locator when Panel Editor is Open ....................................................... 302
Figure 9. Dir Menu .................................................................................................................. 306
Figure 10. Type Menu ............................................................................................................ 306
## List of Tables

Table 1. Reserved Words in MAGICAL ................................................................. 28
Table 2. Order of Operator Precedence (Highest First) in MAGICAL .................. 29
Table 3. Variable Types in Pulse Sequences ...................................................... 59
Table 4. Delay-Related Statements ................................................................. 61
Table 5. Observe Transmitter Pulse-Related Statements .................................... 62
Table 6. Decoupler Transmitter Pulse-Related Statements .................................. 64
Table 7. Simultaneous Pulses Statements ......................................................... 65
Table 8. Transmitter Quadrature Phase Control Statements ............................... 67
Table 9. Phase Shift Statements ..................................................................... 68
Table 10. Frequency Control Statements ....................................................... 70
Table 11. Power Control Statements .............................................................. 71
Table 12. Gating Control Statements .............................................................. 73
Table 13. Mapping of User AP Lines .............................................................. 76
Table 14. Integer Mathematics Statements .................................................... 78
Table 15. Pulse Sequence Control Statements .............................................. 79
Table 16. Statements for Controlling Graphical Display of a Sequence .............. 82
Table 17. Statements for Handling AP Tables ................................................ 85
Table 18. Parameter Value Lookup Statements .............................................. 89
Table 19. Global PSG Parameters .................................................................... 89
Table 20. Imaging and Other Variables .......................................................... 92
Table 21. Hardware Looping Related Statements .......................................... 104
Table 22. Number of Events for Statements in a Hardware Loop .................... 106
Table 23. Rotor Synchronization Control Statements ....................................... 108
Table 24. Shapelib File Suffix List .................................................................... 109
Table 25. RF Patterns ..................................................................................... 110
Table 26. Decoupler Patterns ......................................................................... 111
Table 27. Gradient Patterns ........................................................................... 111
Table 28. Shaped Pulse Statements ............................................................... 112
Table 29. Programmable Control Statements .............................................. 114
Table 30. Spin Lock Control Statements ........................................................ 115
Table 31. AP Bus Delay Constants ............................................................... 117
Table 32. Statements for Pulse Shaping Through the AP Bus .......................... 118
Table 33. AP Bus Overhead Delays ................................................................ 119
Table 34. Example of AP Bus Overhead Delays for status Statement ............... 121
Table 35. Multidimensional PSG Variables .................................................. 124
Table 36. Gradient Control Statements .......................................................... 125
Table 37. Delays for Obliquing and Shaped Gradient Statements .................... 126
Table 38. Performa XYZ PFG Module Statements ......................................... 129
Table 39. Imaging-Related Statements ......................................................... 130
Table 40. Commands for Reformatting Data ................................................ 286
Table 41. Commands for Working with Parameter Trees ................................. 290
Table 42. Common Attributes of Panel Elements ........................................ 308
Table 43. Panels and Locations .................................................................... 326
Table 44. Acquisition Status Codes ........................................................... 333
Introduction

VnmrJ software provides NMR users with a programming environment for customizing the system software and the operator interface. This manual covers MAGICAL programming, pulse sequence programming, and manipulating parameters and data.

Overview of this Manual

This manual explains how to use these capabilities:

- Chapter 1, “MAGICAL II Programming,” describes MAGICAL II (MAGnetics Instrument Control and Analysis Language), a powerful software application that enables full automation of spectrometer operation and data analysis using macros.
- Chapter 2, “Pulse Sequence Programming,” covers pulse sequence programming using Varian’s powerful and extensive set of pulse sequence statements.
- Chapter 3, “Pulse Sequence Statement Reference,” is an alphabetical reference to each pulse sequence statement in VnmrJ.
- Chapter 4, “Linux Level Programming,” is an overview of the operating system.
- Chapter 5, “Parameters and Data,” covers manipulating parameters, using data files, modifying parameter displays, and writing user-defined weighting functions.

Notational Conventions

The following notational conventions are used throughout all VnmrJ manuals:

- **Typewriter-like characters** identify VnmrJ and UNIX commands, parameters, directories, and file names in the text of the manual. For example:
  The `shutdown` command is in the `/etc` directory.
- **Typewriter-like characters** also show text displayed on the screen, including the text echoed on the screen as commands are entered. For example:
  `Self test completed successfully.`
- Text shown between angled brackets `<...>` in a syntax entry is optional. For example, if the syntax is `seqgen s2pul<.c>`, entering the “`.c`” suffix is optional, and typing `seqgen s2pul.c` or `seqgen s2pul` is functionally the same.
- Lines of text containing command syntax, examples of statements, source code, and similar material are often too long to fit the width of the page. To show that a line of text had to be broken to fit into the manual, the line is cut at a convenient point (such as at a comma near the right edge of the column), a backslash (`\`) is inserted at the cut, and the line is continued as the next line of text. This notation will be familiar to C programmers. Note that the backslash is not part of the line and, except for C source code, should not be typed when entering the line.
- Because pressing the Return key is required at the end of almost every command or line of text typed on the keyboard, use of the Return key is mentioned only in cases where it is *not* used. This convention avoids repeating the instruction “press the Return key” throughout most of this manual.
- Text with a change bar (like this paragraph) identifies new VnmrJ material.
Chapter 1. MAGICAL II Programming

Sections in this chapter:

- 1.1 “Working with Macros,” this page
- 1.2 “Programming with MAGICAL,” page 27
- 1.3 “Relevant VnmrJ Commands,” page 38

Many of the actions performed on an NMR spectrometer are performed many times a day. VnmrJ software incorporates a high-level macro programming language designed for NMR called “NMR” language, MAGICAL II™ (MAGnetics Instrument Control and Analysis Language, version II—usually just called MAGICAL in this chapter) to make these actions easier. Many commands used in VnmrJ are macros (see /vnmr/maclib).

1.1 Working with Macros

- “Writing a Macro,” page 24
- “Executing a Macro,” page 24
- “Transferring Macro Output,” page 26
- “Loading Macros into Memory,” page 26

A *macro* is a user-defined command containing a long series of commands and parameter changes that are enter one by one. A spectrum is plotted with a scale under the spectrum, and parameters on the page using the following sequence of commands:

```
pl
pscale
hpa
page
```

A macro, called `plot`, containing these commands can be written. A macro called `plot_2d` using certain parameters to routinely plot 2D spectra can be written:

```
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 give access to NMR information like 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 problem: find 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) that 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 $height=-$height endif "If negative, make positive"
vs=180*vs/$height "Adjust the vertical scale"

As written, the macro vsadj has four lines:

- The material in double-quotation marks (the first line and parts of other lines) are comments. MAGICAL permits comments, and as is good programming practice, this example is filled with comments to explain what is happening.
- 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 (e.g., 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 (like vs, which 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 out of existing commands, second-level macros out of first-level macros and commands, and so on. Suppose we created a macro plot, for example, we might also create a macro setuph, another macro acquireh, and yet another macro processh. Now we might create a “higher-level” macro, H1, which is equivalent to setuph acquireh processh plot. Perhaps we have created two more similar macros, C13 and APT. Now we might create yet another higher-level macro HCAPT, equivalent to H1 C13 APT. At every step of the way, the power of the macro increases, but 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 along with acquisition setup commands, etc. 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 Varian software.

Executing a Macro

When any program is executed, the command interpreter first checks to see if it is a standard VnmrJ command. If the program is not a command, the command interpreter then
1.1 Working with Macros

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. For example, there is a built-in VnmrJ command named wft. If someone happens to write a macro also named wft, the macro wft will never get executed because the VnmrJ command wft takes precedence. To get around this restriction, the hidecommand command can 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 is now executable directly. 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').

Macro files can reside in four separate locations:

1. In the user’s maclib directory.
2. In the directory pointed to by the maclibpath parameter (if maclibpath is defined in the user's global parameter file).
3. In the directory pointed to by the sysmaclibpath parameter (if defined).
4. In the system maclib directory.

When macros are executed, the four locations are searched in this order. The first location found is the one that is used. 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 it finds about which location contains 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 changes to it are available to all users. Each user also has their 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 is 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 (examples are 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 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,” page 292), a macro by the name _x (that is, the same name as 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.

If the macro needs to know what macro invoked it, that information is stored by the string parameter `macro` available in each parameter set.

### 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 into 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:

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

For example, the command `peak` (described on page 40) finds the height and frequency of the tallest peak. Entering the command:

```plaintext
peak:r1,r2
```

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

```plaintext
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 thus would be the equivalent of a “tallest line” command (similar to but different than the command `nl` to position the cursor at the nearest line).

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

```plaintext
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, `dll`, also produces one output—the number of lines. Entering

```plaintext
dll:$n
```

reads the number of lines into variable `$n`. `dll` 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 faster execution speed is desirable, the parsed form of the macro, user or system, can be loaded into 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 systems (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 in 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 a 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 is *unchanged*. Executing the 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 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 the personal version in *maclib* directory is subsequently be executed.

Performance is improved if a macro called 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.

### 1.2 Programming with MAGICAL

- “Tokens,” page 28
- “Variable Types,” page 31
- “Arrays,” page 32
- “Expressions,” page 34
- “Input Arguments,” page 35
- “Name Replacement,” page 36
- “Conditional Statements,” page 36
- “Loops,” page 37
- “Macro Length and Termination,” page 38
- “Command and Macro Tracing,” page 38

MAGICAL has many features, including tokens, variables, expressions, conditional statements, and loops. To program in MAGICAL, be aware of the main features described in this section.
Tokens

A token is a character or characters that is taken by the language as a single “thing” 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 characters _ $ #. The underline _ counts as a letter. Upper and lower case letters are different. The first letter of identifiers, except temporary variable identifiers, must be a letter. Temporary variable identifiers start with the dollar-sign ($) character. Identifiers can be any length (but be reasonable). Examples of identifiers are pcon, _pw, or $height.

Reserved Words

The identifiers listed in Table 1 are reserved words and may not be used otherwise. Reserved words are recognized in both upper and lower case formats (e.g., do not use either \textit{and} or \textit{OR} except as a reserved word).

\begin{table}[h]
\centering
\begin{tabular}{lllll}
\hline
abort & else & not & trunc \\
abortoff & elseif & or & typeof \\
aborton & endif & repeat & then \\
and & endwhile & return & until \\
break & if & size & while \\
do & mod & sqrt \\
\hline
\end{tabular}
\caption{Reserved Words in MAGICAL.}
\end{table}

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. The integer and fraction parts both 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, such as

‘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"
1.2 Programming with MAGICAL

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 first. Within each group, operator precedence in expressions is from left to right, except for the logical group, where the respective members are listed in order of precedence.

**Table 2. Order of Operator Precedence (Highest First) in MAGICAL**

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

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 results.

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, e.g.,
  
  '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 when the end of a line occurs.

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

  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, e.g.,

  /* 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 it takes 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 `real(variable)` command creates a real variable without a value, where `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)`, where `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:

<table>
<thead>
<tr>
<th>name</th>
<th>group</th>
<th>array size</th>
</tr>
</thead>
<tbody>
<tr>
<td>basictype</td>
<td>display</td>
<td>group</td>
</tr>
<tr>
<td>subtype</td>
<td>max./min. values</td>
<td>protection status</td>
</tr>
<tr>
<td>active</td>
<td>step size</td>
<td></td>
</tr>
</tbody>
</table>

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 \texttt{rt} and \texttt{rtp} commands are not used, these parameters do act like global parameters in 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—be sure that some other macro does not change the value of this variable in the meantime.

Variables stored in \texttt{/vmnrsys/global} are not experiment-based and retain their values even when \texttt{jexp(experiment\_number)}, \texttt{rt<('file'<,'nolog'>)}, or \texttt{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.

A special initialization is required in one situation. When the first use of a string variable is as the return argument from a procedure, it must be initialized first by setting it to a null string. For example, a line such as

\begin{verbatim}
input('Input Your Name: '):$name
\end{verbatim}

produces an error. Use instead

\begin{verbatim}
$name=' ' input('Input Your Name: '):$name.
\end{verbatim}

By definition, local variables are lost upon 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 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 error message:

\begin{verbatim}
Variable "variable\_name" doesn't exist.
\end{verbatim}

**Arrays**

Both global and local variables, whether real or string, can be arrayed. Array elements are referred to by square brackets ([...]), such as \texttt{pw[1]}. Indices for the array can be fixed numbers (\texttt{pw[3]}), global variables (\texttt{pw[r1]}), or local variables (\texttt{pw[$i]}). Of course, the index must not exceed the size of the array. Use the \texttt{size} operator to determine the array size. For example, the statement \texttt{r1=size('d2')} sets \texttt{r1} to number of elements in variable \texttt{d2}. If the variable has only a single value, \texttt{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 \texttt{llfrq} and \texttt{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, \texttt{pw[4]} or \texttt{llfrq[5]}, for example.
1.2 Programming with MAGICAL

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 or 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/swl,2/swl,3/swl
```

The square brackets can also be used on the right-hand 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.

<table>
<thead>
<tr>
<th>Enter</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>nt=[1]</td>
<td>nt=1</td>
</tr>
<tr>
<td>nt=[1,2,3]</td>
<td>nt=1,2,3</td>
</tr>
<tr>
<td>nt=[1,2,3]*10</td>
<td>nt=10,20,30</td>
</tr>
</tbody>
</table>
Use [ ] to give precedence to expressions, just like ( ).

<table>
<thead>
<tr>
<th>Enter</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>nt=22*[2*3,r2+6,trunc(r3)]+2</td>
<td>nt=22<em>2</em>3+2,22*(r2+6)+2,22*trunc(r3)+2</td>
</tr>
<tr>
<td>d2=[0,1,2,3]/sw1</td>
<td>d2=0/sw1,1/sw1,2/sw1,3/sw1</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Enter</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>nt=[2*[3+4]]</td>
<td>nt=14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Enter</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>nt=[1,2],3</td>
<td>nt=1,2,3</td>
</tr>
<tr>
<td>nt=[1,2],[3,4]</td>
<td>nt=1,2,3,4</td>
</tr>
</tbody>
</table>

**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 $\text{pw}=0.6*pw90$, or as input arguments to commands or macros, such as in $\text{pa}(-5+sc,50+vp)$.

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 $\text{nt}=7$, the statement $\text{nt}=0.5*\text{nt}$ will end with $\text{nt}=3.5$, just as directly entering $\text{nt}=3.5$ would have resulted in $\text{nt}=3$. Other examples of this include the round-off of $\text{fn}$ 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 $\text{pw}=10$, $\text{sw}>=10000$, $\text{at}/2<0.05$, and ($\text{pw}<5) \text{ or } (\text{pw}>10)$.

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 $\$n$, where $n$ 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.

An example of using an input arguments such as $\$1$:

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

Another example, which 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'
${a} = 10.3
pw = 20.5
$b = ${a}
${a}[2]=5
$b = ${a}[2]
$cmd='wft'
{"cmd}
```

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... }
```

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 be sure 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 (d1>30) or ((tn='H1') and (dhp='y'))
    then write('line3','Problem with acquisition parameters')
endif
```

This 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
dendif

This 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 form:
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 form:
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 like $n=$n+1
inside 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 when the command `return` is encountered. This is usually inserted into the macro after testing some condition, as shown in the example below:

```
"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 as do commands. This information is captured by the calling macro using the format `:argument1,argument2,...` Here is an example of returning 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` still will 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`.

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`, where `xx` is a number. Use this on the VnmrJ command line `jFunc(55,'/dev/pts/xx')`. Replace `xx` with the correct number.

The commands `debug('c')` and `debug('C')` turn on and off, respectively, VnmrJ command and macro tracing. 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 which macro or command failed.

If VnmrJ is started when the user logs in, the output goes to a Console window. If no Console window is present, the output goes into a file in the `/var/tmp` directory. This last option is not recommended.

1.3 Relevant VnmrJ Commands

- “Spectral Analysis Tools,” page 39
- “Input/Output Tools,” page 40
- “Regression and Curve Fitting,” page 42
- “Mathematical Functions,” page 43
- “Creating, Modifying, and Displaying Macros,” page 44
- “Miscellaneous Tools,” page 46
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**

**dres**  
Measure linewidth and digital resolution  
Syntax: 
```
dres<(frequency,fractional_height)> : linewidth, resolution
```
Description: Analyzes line defined by current cursor position (cr) for linewidth and digital resolution. frequency overrides cr as the line frequency. fractional_height specifies the height at which linewidth is measured.

**dsn**  
Measure signal-to-noise  
Syntax: 
```
dsn<(low_field,high_field)> : signal_to_noise, noise
```
Description: Measures signal-to-noise of the tallest peak in the displayed spectrum. Noise region, in Hz, is specified by supplying low_field and high_field frequencies or it is specified by the positions of the left and right cursors.

**dsnmax**  
Calculate maximum signal-to-noise  
Syntax: 
```
dsnmax<(noise_region)>
```
Description: Finds best signal-to-noise in a region. noise_region, in Hz, can be specified, or the cursor difference (delta) can be used by default.

**getll**  
Get line frequency and intensity from line list  
Syntax: 
```
getll(line_number) : height, frequency
```
Description: Returns the height and frequency of the specified line number.

**getreg**  
Get frequency limits of a specified region  
Syntax: 
```
getreg(region_number) : minimum, maximum
```
Description: Returns the minimum and maximum frequencies, in Hz, of the specified region number.

**integ**  
Find largest integral in specified region  
Syntax: 
```
integ<(highfield,lowfield)> : size, value
```
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 highfield and lowfield are parameters sp and sp+wp, respectively.
**mark**

Determine intensity of the spectrum at a point

Syntax:
- `mark<(f1_position)>`
- `mark<(left_edge, region_width)>`
- `mark<(f1_position, f2_position)>`
- `mark<(f1_start, f1_end, f2_start, f2_end)>`
- `mark<('trace', <options>)>`
- `mark('reset')`

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 `mark` command in the box mode finds the maximum intensity and the coordinate(s) of the maximum intensity.

**nll**

Find line frequencies and intensities

Syntax: `nll<('pos',<noise_mult>)><:number_lines>

Description: Returns the number of lines using the current threshold, but does not display or print the line list.

**numreg**

Return the number of regions in a spectrum

Syntax: `numreg:number_regions`

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.
1.3 Relevant VnmrJ Commands

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.

format Format a real number or convert a string for output
Syntax: format(real_number,length,precision):string_var
format(string,'upper'|'lower'|'isreal'):return_var
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).

input Receive input from keyboard
Syntax: input<(<prompt>,<delimiter>):var1,var2,...
Description: Receives characters from the keyboard and stores them into one or more string variables. prompt is a string that is displayed on the command line. The default delimiter is a comma.

lookup Look up and return words and lines from text file
Syntax: lookup(options):return1,return2,...,number_returned
Description: Searches a text file for a word and returns to the user subsequent words or lines. options is one or more keywords (‘file’, ‘seek’, ‘skip’, ‘read’, ‘readline’, ‘count’, and ‘delimiter’) and other arguments.

nrecords Determine number of lines in a file
Syntax: nrecords(file):$number_lines
Description: Returns the number of “records,” or lines, in the given file.

psgset Set up parameters for various pulse sequences
Syntax: psgset(file,param1,param2,...,paramN)
Description: Sets up parameters for various pulse sequences using information in a file from the user or system parlib.
**write**  
Write output to various devices  
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.
1.3 Relevant VnmrJ Commands

**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 absolute value of a number.

**acos**  
**Find arc cosine of a number**  
Syntax: `acos(number)<:value>`  
Description: Finds 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 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 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 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.

**cos**  
**Find cosine value of an angle**  
Syntax: `cos(angle)<:value>`  
Description: Finds cosine of an angle given in radians.

**exp**  
**Find exponential value of a number**  
Syntax: `exp(number)<:value>`  
Description: Finds exponential value (base e) of a number.
Chapter 1. MAGICAL II Programming

### ln

**Find natural logarithm of a number**

Syntax:  \( \text{ln(number)} \langle : \text{value} \rangle \)

Description: Finds natural logarithm of a number. To convert to base 10, use \( \log_{10}x = 0.43429 \times \ln(x) \).

### sin

**Find sine value of an angle**

Syntax:  \( \text{sin(angle)} \langle : \text{value} \rangle \)

Description: Finds sine an angle given in radians.

### tan

**Find tangent value of an angle**

Syntax:  \( \text{tan(angle)} \langle : \text{value} \rangle \)

Description: Finds tangent of an angle given in radians.

---

### Creating, Modifying, and Displaying Macros

#### crcom

**Create a user macro without using a text editor**

Syntax:  \( \text{crcom(file,actions)} \)

Description: Creates a user macro file in the user's macro directory. The \actions\ string is the contents of the new macro.

#### delcom

**Delete a user macro**

Syntax:  \( \text{delcom(file)} \)

Description: Deletes a user macro file in the user's macro directory. The \actions\ string is the contents of the new macro.

#### hidecommand

**Execute macro instead of command with same name**

Syntax:  \( \text{hidecommand(command\_name)} \langle : \$\text{new\_name} \rangle \)

hidecommand('? ')

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:  \( \text{macrocat(file1<,file2><,...>)} \)

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

#### macrocp

**Copy a user macro file**

Syntax:  \( \text{macrocp(from\_file,to\_file)} \)

Description: Makes a copy of an existing user macro.
1.3 Relevant VnmrJ Commands

**macrodir**  
List user macros  
Description: Lists 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.

**macrolld**  
Load a macro into memory  
Syntax: macrolld(file)<:dummy>  
Description: Loads a macro, user or system, into memory. If 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.

**macrosyscat**  
Display a system macro on the text window  
Syntax: macrosyscat(file1<,file2><,...>)  
Description: Displays one or more system macro files, where file1, file2, ... are names of macros in the system macro directory.

**macrosyscp**  
Copy a system macro to become a user macro  
Syntax: macrosyscp(from_file,to_file)  
Description: Makes a copy of an existing system macro.

**macrosysdir**  
List system macros  
Description: Lists names of system macros.

**macrosysrm**  
Remove a system macro  
Syntax: macrosysrm(file)  
Description: Removes a system macro from the macro directory.

**macrovi**  
Edit a user macro with vi text editor  
Syntax: macrovi(file)  
Description: Modifies an existing user macro or creates a new macro using the vi text editor. To edit a system macro, copy it to a personal macro directory first.

**mstat**  
Display memory usage statistics  
Syntax: mstat<(program_id)>
Chapter 1. MAGICAL II Programming

Description: Displays memory usage statistics on macros loaded into memory.

**purge**  
**Remove a macro from memory**  
Syntax: `purge<(file)>`  
Description: Removes a macro from memory, freeing extra memory space. With no argument, removes all macros loaded into memory by `macrold`.

**record**  
**Record keyboard entries as a macro**  
Syntax: `record<(file|'off')>`  
Description: Records keyboard entries and stores the entries as a macro file in the user’s `maclib` directory.

**Miscellaneous Tools**

**axis**  
**Provide axis labels and scaling factors**  
Syntax: `axis('fn'|'fn1'|'fn2'):<axis_label, \`$frequency_scaling,$factor>  
Description: Returns axis labels, the divisor to convert from Hz to units defined by the `axis` parameter with any scaling, and a second scaling factor determined by any `scalesw` type of parameter. The parameter `'fn'|'fn1'|'fn2' describes the Fourier number for the axis.

**beepoff**  
**Turn beeper off**  
Description: Turns beeper sound off. The default is beeper sound on.

**beepon**  
**Turn beeper on**  
Description: Turns beeper sound on. The default is beeper sound on.

**bootup**  
**Macro executed automatically when VnmrJ is started**  
Syntax: `bootup<(foreground)>`  
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.

**exec**  
**Execute a VnmrJ command**  
Syntax: `exec(command_string)`  
Description: Takes as an argument a character string constructed from a macro and executes the VNMR command given by `command_string`.

**exists**  
**Determine if a parameter, file, or macro exists**  
Syntax: `exists(name,type):$exists`
1.3 Relevant VnmrJ Commands

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 Command and Parameter Reference manual for a detailed description of the use of exists for Applications Directory usage.

**focus**  
Send keyboard focus to VNMR input window  
Description: Sends keyboard focus to the VNMR input window.

**gap**  
Find gap in the current spectrum  
Syntax: gap(gap,height):found,position,width  
Description: Looks for a gap between lines of the currently displayed spectrum, where 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.

**getfile**  
Get information about directories and files  
Syntax: getfile(directory,file_index):$file,$file_extension  
getfile(directory):$number_files  
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 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. The command then to send commands to VNMR is /vnmr/bin/send2Vnmr $vnmruser/.talk command where command is any character string (commands, macros, or if statements) normally typed into the VNMR input window.
**Chapter 1. MAGICAL II Programming**

**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**  
**Syntax:** off(parameter|'n'<,tree>)  
**Description:** Makes a parameter inactive. tree is 'current', 'global', 'processed', or 'systemglobal'.

**on**  
**Make a parameter active or test its state**  
**Syntax:** on(parameter|'y'<,tree>)<:$active>  
**Description:** Makes a parameter active or tests the active flag of a parameter. tree is 'current', 'global', 'processed', or 'systemglobal'.

**readlk**  
**Read current lock level**  
**Syntax:** readlk<:lock_level>  
**Description:** 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.

**rtv**  
**Retrieve individual parameters**  
**Syntax:** rtv<(file,par1<,index1<,par2,index2...>>)><:val>  
**Description:** Retrieves one or more parameters from a parameter file to the experiment's current tree. If a return argument is added, rtv instead returns values to macro variables, which avoids creating additional parameters in the current tree. For arrayed parameters, array index arguments can specify which elements to return to the macro. The default is the first element.

**shell**  
**Start a UNIX shell**  
**Syntax:** shell<(command)>:$var1,$var2,...  
If no argument is given, opens a normal UNIX shell. If a UNIX command is entered as an argument, shell executes the command. Text lines usually displayed as a result of the UNIX command given in the argument can be returned to $var1, $var2, etc. shell 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:  
shell('ls -t|grep May; cat')  
or  
shell('(ls -t|grep May)')

**solppm**  
**Return ppm and peak width of solvent resonances**  
**Syntax:** solppm:chemical_shift,peak_width
1.3 Relevant VnmrJ Commands

Description: Returns information about the chemical shift in ppm and peak spread of solvent resonances in various solvents for either $^1$H or $^{13}$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: \texttt{substr(string, word\_number):substring} \texttt{substr(string, index, length):substring}

Description: Picks a substring out of a string. If two arguments are given, substring returns the word\_number word in string. If three arguments, it returns a substring from string where 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: \texttt{textis(command):$yes\_no} \texttt{textis:$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: \texttt{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.
Chapter 1. MAGICAL II Programming
Chapter 2. Pulse Sequence Programming

Sections in this chapter:

- 2.1 “Application Type and Execpars Programming,” page 51
- 2.2 “Overview of Pulse Sequence Programming,” page 55
- 2.3 “Spectrometer Control,” page 60
- 2.4 “Pulse Sequence Statements: Phase and Sequence Control,” page 76
- 2.5 “Real-Time AP Tables,” page 83
- 2.6 “Accessing Parameters,” page 89
- 2.7 “Using Interactive Parameter Adjustment,” page 98
- 2.8 “Hardware Looping and Explicit Acquisition,” page 103
- 2.9 “Pulse Sequence Synchronization,” page 108
- 2.10 “Pulse Shaping,” page 108
- 2.11 “Shaped Pulses Using Attenuators,” page 116
- 2.12 “Internal Hardware Delays,” page 119
- 2.13 “Indirect Detection on Fixed-Frequency Channel,” page 122
- 2.14 “Multidimensional NMR,” page 122
- 2.15 “Gradient Control for PFG and Imaging,” page 125
- 2.16 “Programming the Performa XYZ PFG Module,” page 127
- 2.17 “Imaging-Related Statements,” page 129
- 2.18 “User-Customized Pulse Sequence Generation,” page 132

2.1 Application Type and Execpars Programming

An NMR protocol is a specific set of parameters and methods used to acquire, process, plot, and store NMR data. The parameters also specify the pulse sequence used to acquire the data. NMR protocols can be grouped into classes or types of applications, which often share many of the parameters and methods needed by individual protocols.

VnmrJ uses protocols and application types (apptype) to systematize the development of new NMR protocols. The next section describes how protocols and application types are programmed. The remainder of this chapter describes how to program pulse sequences using the traditional C language. To use the SpinCAD interface for creating pulse sequences, refer to the SpinCAD manual.

The application type concept provides preparation, prescan, processing, and plotting customization based on the type of NMR data.
apptypes

Each apptype has a corresponding macro, which has the same name as the apptype. These macros handle the customization required for that apptype.

**Liquids apptypes**

<table>
<thead>
<tr>
<th>apptype</th>
<th>representative protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td>std1d</td>
<td>Proton, Carbon, Phosphorus, Presat, Apt, Dept</td>
</tr>
<tr>
<td>homo2d</td>
<td>Cosy, Dqcosy, Gcosy, Gdqcosy, Noesy</td>
</tr>
<tr>
<td>hetero2d</td>
<td>Cigar, Cigar2j3j, Ghmhc, Ghmqc, Ghmactoxy, Ghsqc, Ghsqctoxy, Hmbc, Hmqc, Hmqctoxy, Hsqc, Hsqctoxy</td>
</tr>
</tbody>
</table>

**Imaging apptypes**

<table>
<thead>
<tr>
<th>apptype</th>
<th>representative protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td>im1D</td>
<td>press isis steam</td>
</tr>
<tr>
<td>im1Dcsi</td>
<td>presscsi steamcsi</td>
</tr>
<tr>
<td>im1Dglobal</td>
<td>spuls</td>
</tr>
<tr>
<td>im2D</td>
<td>angio gems mens mens sems sems dw</td>
</tr>
<tr>
<td>im2Dcsi</td>
<td>csi2d</td>
</tr>
<tr>
<td>im2Dfse</td>
<td>fsems</td>
</tr>
<tr>
<td>im3D</td>
<td>ct3d, ge3d, ge3dangio, se3d</td>
</tr>
<tr>
<td>im3Dfse</td>
<td>fse3d</td>
</tr>
<tr>
<td>imEPI</td>
<td>epidw epimss epimssn</td>
</tr>
<tr>
<td>imFM</td>
<td>fastestmap</td>
</tr>
</tbody>
</table>

**execpar Parameters**

Five execpar parameters control the execution of the apptype macros: execsetup, execprep, execprescan, execprocess, and execplot. The following two examples show how the execpar parameters are set for std1d and im2D apptypes.

<table>
<thead>
<tr>
<th>std1d apptype</th>
<th>im2D apptype</th>
</tr>
</thead>
<tbody>
<tr>
<td>execsetup = 'std1d(setup)'</td>
<td>execsetup = 'im2D(setup)'</td>
</tr>
<tr>
<td>execprep = ''</td>
<td>execprep = 'im2D(prep)'</td>
</tr>
<tr>
<td>execprescan = ''</td>
<td>execprescan = 'im2D(prescan)'</td>
</tr>
<tr>
<td>execprocess = 'std1d(process)'</td>
<td>execprocess = 'im2D(proc)'</td>
</tr>
<tr>
<td>execplot = 'std1d(plot)'</td>
<td>execplot = ''</td>
</tr>
</tbody>
</table>

These parameters should not be set to specific actions, such as 'ni=256' or 'pcon page'. They should only call the apptype macro with appropriate arguments, which avoids problems if someone wants to change the behavior. Instead of fixing all the old parameter sets, you only need to update one macro.

Files containing these execpar parameters are saved in the /vnmr/execpars directory. You can have private execpar parameters in a /userdir/execpars directory. The Configure EXEC parameters window (under the Utilities menu) allows you to create and update these parameters. Behind the scenes, the execpars macro handles these parameter files. It can read the execpars into the current parameter set, save execpars, create default execpars, or delete execpars.
Standard macros execute the execpar strings. The rules for executing these strings, based on the execpar parameters, are as follows. If the parameter does not exist, or is set to inactive, the execpar string is not executed. Otherwise, the execpar string is executed. Some macros include default behavior. In these cases, if the execpar is set to inactive, the default behavior will occur. If the execpar is set to active and the value is '', no action, including no default action will occur. An example might clarify this. The process macro provides default NMR processing tools. At the beginning of this macro is the execpars handling:

```plaintext
on('execprocess'): $e
if ($e > 0.5) then
  exec(execprocess)
  return
endif
```

The `on` command tests whether the execprocess exists and is active. If it does not exist or is inactive, the $e will be less than 0.5 and the `exec` command and `return` command will not be executed. The rest of the process macro will be executed, giving default behavior. If the parameter is active, the `exec` command will be executed. Now, if `execprocess='`, the `exec` command will return without executing anything. This is followed by `return`, which exits the process macro, avoiding any default processing.

When a protocol is brought into a work space or study queue, the `cqexp` (for liquids) or `sqexp` (for imaging) macro is called. These check if the `execsetup` parameter exists. If it does not, it runs `execpars` to read the execpars for that apptype. Using the rules above, it might execute the `execsetup` string.

The execpars parameters are executed by several other standard macros:

<table>
<thead>
<tr>
<th>Macro</th>
<th>execpar string executed, using above rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire</td>
<td>execprep</td>
</tr>
<tr>
<td>prep</td>
<td>execprep</td>
</tr>
<tr>
<td>settime</td>
<td>execprep</td>
</tr>
<tr>
<td>prescan_gain</td>
<td>execprescan</td>
</tr>
<tr>
<td>process</td>
<td>execprocess</td>
</tr>
<tr>
<td>plot</td>
<td>execplot</td>
</tr>
</tbody>
</table>

As a consequence of the execpars scheme, the `usergo` and `go_seqfil` macros are no longer used. This customization should be handled in the 'setup' or 'prep' section of the apptype macros.

The apptype macros should use the template shown in Listing 1. If there is a first argument, it should be `prep, proc, prescan, or plot`. Additional arguments can be used (setup, process, plot).
Chapter 2. Pulse Sequence Programming

Listing 1. apptype Macro Template

```plaintext
// ********* Parse input *********
$action = 'prep'
$do = ''
if ($# > 0) then
  $action = $1
  if ($# > 1) then
    $do = $2
  endif
endif

// ********* Setup *********
if ($action = 'prep') then
  // apptype preparatory customization
  execseq('prep') // Execute any sequence specific preparation
  // additional apptype preparatory customization

elseif ($action = 'proc') then
  // apptype processing customization
  execseq('proc') // Execute any sequence specific processing
  // additional apptype processing customization

elseif ($action = 'prescan') then
  // apptype prescan customization
  execseq('prescan') // Execute any sequence specific prescan
  // additional apptype prescan customization

elseif ($action = 'plot') then
  // apptype plot customization
  execseq('plot') // Execute any sequence specific plot
  // additional plot prescan customization
endif
```

The `execseq` macro constructs a macro name as

```
$macro = seqfil + '_' + $1
```

and will execute it if it exists. If no argument is given, it defaults to 'prep'. This allows for sequence specific behavior.

Protocol Programming

A protocol is made by defining its parameters and specifying its apptype. The New Protocol window (Utilities->Make a New Protocol) will save the current parameters for that protocol, construct the necessary file so that the protocol is available from the Locator and the Experiment selector, and create a macro which can be used to setup that protocol. For liquids, the macro calls the `cqexp` macro with the protocol name and apptype as the two arguments. For example, the macro for the Proton protocol is

```
cqexp('Proton','std1d')
```
With this information, the `cqexp` macro reads in the execpars for the std1d apptype. It then executes macro defined by the execsetup parameter. In this case, 

```c
execsetup="std1d('setup')".
```

The `std1d` macro gets called with the 'setup' argument. Before calling the command specified by the execsetup parameter, the `cqexp` macro set the parameter macro to its first argument.

The first argument is the name of the specific protocol, so that, in this case, `macro='Proton'`. The apptype macros, (e.g., `std1d`) typically use the macro parameter in order to decide which parameter set should be used.

### 2.2 Overview of Pulse Sequence Programming

- “Spectrometer Differences,” page 55
- “Pulse Sequence Generation Directory,” page 55
- “Compiling the New Pulse Sequence,” page 56
- “Troubleshooting the New Pulse Sequence,” page 57
- “Creating a Parameter Table for Pulse Sequence Object Code,” page 58
- “C Framework for Pulse Sequences,” page 58
- “Implicit Acquisition,” page 60
- “Acquisition Status Codes,” page 60

Pulse sequences can be written in C, a high-level programming language that allows considerable sophistication in the way pulse sequences are created and executed. New pulse sequences can be added to the software by writing and compiling a short C procedure. This process is simplified using the tools provided with VnmrJ.

#### Spectrometer Differences

This manual contains information on how to write pulse sequences for $\text{UNITY} \text{INOVA}$ and $\text{MERCURYplus/-Vx}$ spectrometers. Each spectrometer has different capabilities, so not all statements may be executed on all platforms.

For example, because $\text{MERCURYplus/-Vx}$ hardware differs significantly from $\text{UNITY} \text{INOVA}$ hardware, sections in this manual covering waveform generators and imaging are not applicable to the $\text{MERCURYplus/-Vx}$ even though the pulse sequence programming language is the same. Pay careful attention to comments in the text regarding the system applicability of the pulse sequence statement or technique.

#### Pulse Sequence Generation Directory

Pulse sequence generation (PSG) text files (like `hom2dj.c` in Listing 2) are stored in a directory named `psglib`. The system (`/vnmr/psglib`) and each user have a `psglib` directory.

The user `psglib` is stored in the user’s private directory system (e.g., for user `vnmr1`, in `vnmrsys/psglib`). Some systems use `/space` and Linux uses `/home`. All pulse sequence files stored in these directories are given the extension `.c` to indicate that the file contains C language source code. For instance, a homonuclear-2D-J sequence that may have been written by a user (other than the system administrator) is automatically stored in the
user’s private pulse sequence directory and has a name like home/user/vnmrsys/psglib/hom2dj.c

Numerous sequences are in the standard Varian-supplied directory /vnmr/psglib and in the user library directory /vnmr/userlib/psglib, or a sequence can be written using any of the standard text editors such as vi or textedit. Once a pulse sequence exists, it can subsequently be modified as desired, again using one of a number of text editors.

Compiling the New Pulse Sequence

After a pulse sequence is written, the source code is compiled by one of these methods:

• By entering seqgen(file<.c>) on the VnmrJ command line.
• By entering seqgen file<.c> from a UNIX shell.

For example, entering seqgen('hom2dj') compiles the hom2dj.c sequence in VnmrJ and entering seqgen hom2dj does the same in Linux. Note that a full path, such as ('/home/vnmr1/vnmrsys/psglib/hom2dj.c') or even seqgen('hom2dj.c') is not necessary or possible—the seqgen command knows where to look to find the source code file and knows that it will have a .c extension.

During compilation, the system performs the following steps:

1. If the program dps_ps_gen is present in /vnmr/bin, extensions are added to the pulse sequence to allow a graphical display of the sequence by entering the dps command. Statements dps_off, dps_on, dps_skip, and dps_show can be inserted in the pulse sequence to control the dps display.
2. The source code is passed through the Linux program lint to check for variable consistency, correct usage of functions, and other program details.
3. The source code is converted into 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), in a procedure called link loading, to produce the executable pulse sequence code. This is actually done at run-time. If compilation of the pulse sequence with the dps extensions fails, the pulse sequence is recompiled without the dps extensions.

#include <standard.h>
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);
}
If the executable pulse sequence code is successfully produced, it is stored in the user seqlib directory (e.g., /home/vnmr1/vnmrsys/seqlib). If the user does not have a seqlib directory, it is automatically created.

Like psglib, different seqlib directories exist, including the system directory and each user’s directory. The user’s vnmrsys directory should have directories psglib and seqlib. Whenever a user attempts to run a pulse sequence, the software looks first in the user’s personal directory for a pulse sequence by that name, then in the system directory.

A number of sequences are supplied in /vnmr/seqlib, compiled and ready to use. The source code for each of these sequences is found in /vnmr/psglib. To compile one of these sequences, or to modify a sequence in /vnmr/psglib, copy the sequence into the user’s psglib, make any desired modifications, then compile the sequence using seqgen. (seqgen will not compile sequences directly in /vnmr/psglib). All sequences in /vnmr/psglib have an appropriate macro using them.

**Troubleshooting the New Pulse Sequence**

During the process of pulse sequence generation (PSG) with the seqgen command, the user-written C procedure is passed through a utility to identify incorrect C syntax or to hint at potential coding problems. If an error occurs, a number of messages usually are displayed. Somewhere among them are these statements:

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

As a rule of thumb, focus on the lines in the name.errors text file that begin with the name of the pulse sequence enclosed in double quotes followed by the line number and those that begin with a line number in parentheses. In both cases, a brief description of the problem is also displayed. If the line of code looks correct, often the preceding line of code is the culprit. Note that a large number of error messages can be generated from the same coding error.

If a warning occurs, the following message appears:

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:

This message means that although the pulse sequence has some inconsistent C code that may produce run-time errors, the pulse sequence did compile. Three warnings to watch for are the following:

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 may be generated by less than optimum usage of the ix variable:
conversion from long may lose accuracy

An example can be found in a few of the earlier pulse sequences implementing TPPI. The following construct, which was taken from an older version of hmqc.c, generates the warning:

```c
if (iphsae == 3)
{
    t1_counter = ((int) (ix - 1)) / (arraydim / ni);
    initval((double) (t1_counter), v14);
}
```

Changing these lines to
if (iphase == 3)
  initval((double) ((int)((ix - 1) / (arraydim / ni) \\
  + 1e-6)), v14);

avoids the warning and also provides for roundoff of the floating point expression to give proper TPPI phase increments.

Even the above expression can fail under some circumstances. That construction will not work for 3D and 4D experiments. With the availability of increment counters such as id2, id3, and id4, and the predefined phase1 variable, this example can be rewritten as

if (phase1 == 3)
  assign(id2, v14);

The second warning generally suggests an uninitialized variable:

parameter_name may be used before set

This should be corrected; otherwise, unpredictable execution of the pulse sequence is likely. A common cause is the use of a user variable without first using a getval or getstr statement on the variable.

The third warning generally suggests that a variable is defined within the pulse sequence that has the same name as one of the standard PSG variables.

parameter_name redefinition hides earlier one

This warning is normally avoided by renaming the variable in the pulse sequence or, if the variable corresponds to a standard PSG variable, by removing the variable definition and initialization from the pulse sequence and just using the standard PSG variable. A list of the standard PSG variable names is given in “Accessing Parameters,” page 89.

Finally, if the pulse sequence program is syntactically correct, the following message is displayed:

Done! Pulse sequence now ready to use.

Creating a Parameter Table for Pulse Sequence Object Code

The ability to modify or customize acquisition parameters to fit a given user-created pulse sequence is provided by a small number of commands. These commands make it possible to perform the following operations on an existing parameter table:

• Create new parameters
• Control the display and enterability of parameters
• Control the limits of the parameter
• Create a parameter table for \( n \)-dimensional experiments

The commands that enable the creation and modification of parameters are discussed in section 5.4 “Creating and Modifying Parameters,” page 288.

C Framework for Pulse Sequences

Each pulse sequence is built onto a framework written in the C programming language. Look again at the hom2dj sequence in Listing 2. The absolutely essential elements of this framework are these:

```c
#include <standard.h>
pulsesequence()
{
}
```
This framework must be included exactly as shown. Between the two curly braces ({}), pulse sequence statements are placed, each ending with a semicolon.

The majority of pulse sequence statements allow the user to control pulses, delays, frequencies, and all functions necessary to generate pulse sequences. Most are in the general form statement(argument1,argument2,...), where statement is the name of the particular pulse sequence statement, and argument1,argument2,... is the information needed by that statement in order to function.

Many of these arguments are listed as real number. Because of the flexibility of C, a real-number argument can take three different forms: variable (e.g., d1), constant (e.g., 3.4, 20.0e–6), or expression (e.g., 2.0*pw, 1.0–d2).

Times, whether delays or pulses, are determined by the type of acquisition controller board used on the system:

- Data Acquisition Controller boards: times can be specified in increments as small as 12.5 ns with a minimum of 100 ns.
- Output boards and the MERCURYplus/-Vx: times can be specified in increments as small as 0.1 μs. The smallest possible time interval in all other cases is 0.2 μs, or 0.

Any pulse widths or delays less than the minimum generate a warning message and are then eliminated internally from the sequence. (Note that time constants within a pulse sequence are always expressed in seconds.)

A series of internal, real-time variables named v1, v2, ..., v14 are provided to perform calculations in real-time (by the acquisition computer) while the pulse sequence is executing. Real-time variables are discussed in detail later in this chapter. For now, note that all of the phases, and a small number of the other arguments to the pulse sequence statements discussed here, must be real-time variables. A real-time variable must appear as a simple argument (e.g., v1), and cannot be replaced by anything else, including an integer, a real number, a “regular” variable such as d1, or an expression such as v1+v2.

Any variables chosen for use in a pulse sequence must be declared. Most variables are of type double, while integers are of type int, and strings, such as dmm, are of type char with dimension MAXSTR. Table 3 lists the length of these basic types on the computer. Many variables that refer to parameters used in an experiment are already declared (see “Accessing Parameters,” page 89).

Table 3. Variable Types in Pulse Sequences

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Length (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>character</td>
<td>8</td>
</tr>
<tr>
<td>short</td>
<td>short integer</td>
<td>16</td>
</tr>
<tr>
<td>int</td>
<td>integer</td>
<td>32</td>
</tr>
<tr>
<td>long</td>
<td>long integer</td>
<td>32</td>
</tr>
<tr>
<td>float</td>
<td>floating point</td>
<td>32</td>
</tr>
<tr>
<td>double</td>
<td>double-precision floating point</td>
<td>64</td>
</tr>
</tbody>
</table>

A codeint is a 16 bit integer (as opposed to a float or char). Real-time variables are of type codeint and are 16 bit integers on UNITY/INOVA, MERCURYplus, and MERCURY-Vx. A framework including variable declarations of the main types might look like this:

```c
#include <standard.h>
pulsesequence()
```
{  
double delta;    /* declare delta as double */  
char xpolar[MAXSTR];    /* declare xpolar as char */  
...  
}

Implicit Acquisition

The hom2dj.c pulse sequence listing in Listing 2 on page 56 has one notable omission—data acquisition. In most pulse sequences, the sequence of events consists of a series of pulses and delays, followed at the very end by the acquisition of an FID; the entire process is then repeated for the desired number of transients, and then again (for arrayed and nD experiments) for subsequent elements of the arrayed or nD experiment.

In all these cases, pulse sequences use implicit acquisition, that is, following the pulse sequence as written by the user, an FID is automatically (implicitly) acquired. This acquisition is preceded by a delay that includes the parameter alfa with a delay based on the type of filter and the filter bandwidth. In addition, the phase of all channels of the spectrometer (except the receiver) is set to zero at this time.

Some pulse sequences are not described by this simple model; many solids NMR sequences are in this category, for example. These sequences use explicit acquisition, in which the preacquisition and acquisition steps must be explicitly programmed by the user. This method is described further in “Hardware Looping and Explicit Acquisition,” page 103.

Acquisition Status Codes

Whenever wbs, wnt, wexp, or werr processing occurs, the acquisition condition that initiated that processing is available from the parameter acqstatus. This acquisition condition is represented by two numbers, a “done” code and an “error” code. The done code is set in acqstatus[1] and the error code is set in acqstatus[2]. Macros can take different actions depending on the acquisition condition.

The done codes and error codes are listed in Table 44 and in the file acq_errors in /vnmr/manual. For example, a werr command could specify special processing if the maximum number of transients is accumulated. The appropriate test would be the following:
if (acqstatus[2] = 200) then
  "do special processing, e.g. dp='y' au"
endif

2.3 Spectrometer Control

- “Creating a Time Delay,” page 61
- “Pulsing the Observe Transmitter,” page 62
- “Pulsing a Non-Observe Transmitter,” page 64
- “Pulsing Channels Simultaneously,” page 65
- “Setting Transmitter Quadrature Phase Shifts,” page 67
- “Setting Small-Angle Phase Shifts,” page 67
- “Controlling the Offset Frequency,” page 69
- “Controlling Observe and Decoupler Transmitter Power,” page 70
2.3 Spectrometer Control

More than 200 pulse sequence statements are available for pulse sequence generation (PSG). This section starts the discussion of each statement by covering statements intended primarily for spectrometer control. For discussion purposes, the statements in this section are divided into categories: delay-related, observe transmitter pulse-related, decoupler transmitter pulse-related, simultaneous pulses, transmitter phase control, small-angle phase shift, frequency control, power control, and gating control.

Creating a Time Delay

The statements related to time delays are delay, hsdelay, idelay, vdelay, initdelay, and incdelay. Table 4 summarizes these statements.

Table 4. Delay-Related Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>delay(time)</td>
<td>Delay specified time</td>
</tr>
<tr>
<td>hsdelay(time)</td>
<td>Delay specified time with possible hs pulse</td>
</tr>
<tr>
<td>idelay(time,string)</td>
<td>Delay specified time with IPA</td>
</tr>
<tr>
<td>incdelay(count,index)</td>
<td>Set real-time incremental delay</td>
</tr>
<tr>
<td>initdelay(time_increment,index)</td>
<td>Initialize incremental delay</td>
</tr>
<tr>
<td>vdelay(timebase,count)</td>
<td>Set delay with fixed timebase and real-time count</td>
</tr>
</tbody>
</table>

The main statement to create a delay in a pulse sequence for a specified time is the statement delay(time), where time is a real number (e.g., delay(d1)). The hsdelay and idelay statements are variations of delay:

- To add a possible homospoil pulse to the delay, use hsdelay(time). If the homospoil parameter hs is set to 'y', then at the beginning of the delay, hsdelay inserts a shim coil '3' homospoil pulse of length hst seconds (refer to the description of status).

- To cause interactive parameter adjustment (IPA) information to be generated when gf or go('acqi') is entered, use idelay(time,string), where string is the label used in acqi. If go is entered, idelay is the same as delay. See “Using Interactive Parameter Adjustment,” page 98, for details on IPA. IPA and idelay are not available on the MERCURYplus/-Vx.

To set a delay to the product of a fixed timebase and a real-time count, use vdelay(timebase,count), where timebase is NSEC (defined below), USEC (microseconds), MSEC (milliseconds), or SEC (seconds) and count is one of the real-time variables (v1 to v14). For predictable acquisition, the real-time variable should have a value of 2 or more. If timebase is set to NSEC, the delay depends on the type of acquisition controller board in the system:

- Systems with a Data Acquisition Controller board:
  - The minimum delay is a count of 0 (50 ns), and a count of n corresponds to a delay of (50 + (12.5*n)) ns.
- The vdelay statement is not available on the MERCURYplus/-Vx.

Use initdelay(time_increment,index) or incdelay(count,index) to enable a real-time incremental delay. A maximum of five incremental delays (set by index) can be defined in one pulse sequence. The following steps are required to set up an incremental delay (initdelay and incdelay are not available on the MERCURYplus/-Vx):
1. Enter `initdelay(time_increment,index)` to initialize the time increment and delay.
   The argument `time_increment` is the time increment that will be multiplied by the `count` (a real-time variable) for the delay time, and `index` is one of the indices `DELAY1, DELAY2, ..., DELAY5` (e.g., `initdelay(1.0/sw, DELAY1)` or `initdelay(1.0/sw1, DELAY2)`).

2. Set the increment delay by specifying its index and the multiplier count using `incdelay(count,index)` (e.g., for `incdelay(v3, DELAY2)`, when `v3=0`, the delay is `0*(1/sw1)`).

### Pulsing the Observe Transmitter

Statements related to pulsing the observe transmitter are `rgpulse`, `irgpulse`, `pulse`, `ipulse`, `obspulse`, and `iobspulse`. Table 5 summarizes these statements.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iobspulse(string)</code></td>
<td>Pulse observe transmitter with IPA</td>
</tr>
<tr>
<td><code>ipulse(width,phase,string)</code></td>
<td>Pulse observe transmitter with IPA</td>
</tr>
<tr>
<td><code>irgpulse(width,phase,RG1,RG2,string)</code></td>
<td>Pulse observe transmitter with IPA</td>
</tr>
<tr>
<td><code>obspulse()</code></td>
<td>Pulse observe transmitter with amp. gating</td>
</tr>
<tr>
<td><code>pulse(width,phase)</code></td>
<td>Pulse observe transmitter with amp. gating</td>
</tr>
<tr>
<td><code>rgpulse(width,phase,RG1,RG2)</code></td>
<td>Pulse observe transmitter with amp. gating</td>
</tr>
</tbody>
</table>

Note that `observe transmitter` does not refer to a specific physical channel, but to that physical channel used for observe.

Use `rgpulse(width, phase, RG1, RG2)` as the main statement to pulse the observe transmitter in a sequence, where `width` is the pulse width, `phase` (a real-time variable) is the pulse phase, and `RG1` and `RG2` are defined as:

- `RG1` is the delay during which any needed phase shift is performed and the linear amplifier is gated on and then allowed to stabilize prior to executing the rf pulse, and `RG2` is the delay after the pulse after gating off the amplifier. Thus, receiver gating is a misnomer: `RG1` and `RG2` set amplifier gating, as shown in Figure 1. The receiver is off during execution of the pulses and is only gated on immediately before acquisition.

- On the `MERCURYplus/-Vx`, the receiver and amplifiers are tied together such that when the amplifier is on, the receiver is automatically turned off and when the receiver is on, the amplifier is off.

![Figure 1. Amplifier Gating](image-url)
Some further information about RG1 and RG2:

- Typically, RG1 is a few microseconds and RG2 is probe-dependant.
- The phase of the pulse is set at the beginning of RG1. The phase settling time is about:
  0.2μsec on Unity/Inova.
  0.2μsec on Mercuryplus/Vx.
- A transmitter gate is also switched during RG1. The switching time for this gate is less than:
  0.1μsec on Unity/Inova systems.

For systems with linear amplifiers, an rf pulse can be unexpectedly curtailed if the amplifier goes into thermal shutdown. Thermal shutdown can be brought about if the amplifier duty cycle becomes too large for the average power output.

The remaining statements for pulsing the observe transmitter are variations of rgpulse:

- To pulse the observe transmitter the same as rgpulse but with RG1 and RG2 set to the parameters rof1 and rof2, respectively, use pulse (width, phase). Thus, pulse (width, phase) and rgpulse (width, phase, rof1, rof2) are exactly equivalent.
- To pulse the observe transmitter the same as pulse but with width preset to pw and phase preset to oph, use obspulse(). Thus, obspulse() is exactly equivalent to rgpulse (pw, oph, rof1, rof2).
- To pulse the observe transmitter with rgpulse, pulse, or obspulse, but generate interactive parameter adjustment (IPA) information when go or go('acqi') is entered, use irgpulse (width, phase, RG1, RG2, string), ipulse (width, phase, string), or iobspulse (string), respectively. The string argument is used as a label in acqi. If go is entered, the IPA information is not generated. For details on IPA, see “Using Interactive Parameter Adjustment,” page 98. IPA is not available on Mercuryplus/Vx systems.

The ampmode parameter gives override capability over the default selection of amplifier modes. Unless overridden, the observe channel is set to the pulse mode, other used channels are set to the CW (continuous wave) mode, and any unused channels are set to the idle mode. By using values of d, p, c, and i for the default, pulse, CW, and idle modes, respectively, ampmode can override the default modes. For example, ampmode='dcp' selects default behavior for the first two amplifiers and forces the third channel amplifier into the pulse mode.

The selection of rf channels can be independently controlled with the rfchannel parameter. Single-channel broadband systems do not need rfchannel to set up a normal HMOC experiment (tn='H1', dn='C13'). The software recognizes that you cannot do this experiment and swaps the two channels automatically to make the experiment possible.

The rfchannel parameter becomes important if, for example, if running an HMOC experiment with the X-nucleus using channel 3 with a three-channel spectrometer is required. Instead of rewriting the pulse sequence, create rfchannel (by entering create('rfchannel', 'string')), and set rfchannel, in this example set, rfchannel='132'. Channels 2 and 3 are effectively swapped without changing the sequence.

Similarly, to observe on channel 2, run S2PUL with rfchannel='21'.

The rfchannel mechanism only works for pulse sequences that eliminate all references to the constants TODEV, DODEV, DO2DEV, and DO3DEV. To take advantage of rfchannel, remove statements, such as power and offset, that use these constants.
and replace them with the corresponding statements, such as \texttt{obspower} and \texttt{decoffset}, that do not contain the constants.

Standard pulse sequences have been edited to take advantage of the rf channel independence afforded by the \texttt{rfchannel} parameter. This parameter makes it a simple matter to redirect, for example, the d\textsubscript{n} nucleus to use the third or fourth rf channel.

Appropriate changes to the cabling of the probe are usually required if \texttt{rfchannel} is used.

The \texttt{MERCURYplus/-Vx} systems have two channels and the software automatically determines which channel is observe or decouple based on t\textsubscript{n} and d\textsubscript{n}.

**Pulsing a Non-Observe Transmitter**

Statements related to non-observe pulsing are \texttt{decpulse}, \texttt{decrgpulse}, \texttt{idecpulse}, \texttt{idecrgpulse}, \texttt{dec2rgpulse}, and \texttt{dec3rgpulse}. Table 6 summarizes these statements.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{decpulse}</td>
<td>Pulse decoupler transmitter with amp. gating</td>
</tr>
<tr>
<td>\texttt{decrgpulse}</td>
<td>Pulse first decoupler with amplifier gating</td>
</tr>
<tr>
<td>\texttt{dec2rgpulse}</td>
<td>Pulse second decoupler with amplifier gating</td>
</tr>
<tr>
<td>\texttt{dec3rgpulse}</td>
<td>Pulse third decoupler with amplifier gating</td>
</tr>
<tr>
<td>\texttt{dec4rgpulse}</td>
<td>Pulse deuterium decoupler with amplifier gating</td>
</tr>
<tr>
<td>\texttt{idecpulse}</td>
<td>Pulse first decoupler transmitter with IPA</td>
</tr>
<tr>
<td>\texttt{idecrgpulse}</td>
<td>Pulse first decoupler with amplifier gating and IPA</td>
</tr>
</tbody>
</table>

Use \texttt{decpulse(width,phase)} to pulse channel 2 in the pulse sequence at its current power level. \texttt{width} is the time of the pulse, in seconds, and \texttt{phase} is a real-time variable for the phase of the pulse (e.g., \texttt{decpulse(pp,v3)}).

*The amplifier is gated on during decoupler pulses as it is during observe pulses.* The amplifier gating times (see \texttt{RG1} and \texttt{RG2} for \texttt{decrgpulse} below) are internally set to zero. The decoupler modulation mode parameter \texttt{dmm} should be 'c' during any period of time in which decoupler pulses occur.

To pulse the decoupler at its current power level and have user-settable amplifier gating times, use \texttt{decrgpulse(width,phase,RG1,RG2)}, where \texttt{width} and \texttt{phase} are the same as used with \texttt{decpulse}, and \texttt{RG1} and \texttt{RG2} are the same as used with the \texttt{rgpulse} statement for observe transmitter pulses. In fact, \texttt{decrgpulse} is syntactically equivalent to \texttt{rgpulse} and functionally equivalent with two exceptions:

- The decoupler is pulsed at its current power level (instead of the transmitter).
- If \texttt{homo='n'}, the slow gate (100 ns switching time on \texttt{UNITY}, on the decoupler board is always open and therefore need not be switched open during \texttt{RG1}. In contrast, if \texttt{homo='y'}, the slow gate on the decoupler board is normally closed and must therefore be allowed sufficient time during \texttt{RG1} to switch open (\texttt{homo} is not used on the \texttt{MERCURYplus/-Vx}).

For systems with linear amplifiers, \texttt{RG1} for a decoupler pulse is important from the standpoint of amplifier stabilization under either of the following conditions:

- When t\textsubscript{n} and d\textsubscript{n} both equal \textsuperscript{3}H, \textsuperscript{1}H, or \textsuperscript{19}F (high-band nuclei).
2.3 Spectrometer Control

- When \( t_n \) and \( d_n \) are less than or equal to \( ^{31}\text{P} \) (low-band nuclei).

For these conditions, the “decoupler” amplifier module is placed in the pulse mode, in which it remains blanked between pulses. In this mode, \( RG_1 \) must be sufficiently long to allow the amplifier to stabilize after blanking is removed: 5 \( \mu \text{s} \) is typically right.

When the \( t_n \) nucleus and the \( d_n \) nucleus are in different bands, such as \( t_n \) is \( ^1\text{H} \) and \( d_n \) is \( ^{13}\text{C} \), the “decoupler” amplifier module is placed in the continuous wave (CW) mode, in which it is always unblanked regardless of the state of the receiver. In this mode, \( RG_1 \) is unimportant with respect to amplifier stabilization prior to the decoupler pulse, but with respect to phase setting, it must be set.

The remaining decoupler transmitter pulse-related statements are variations of \texttt{decpulse} and \texttt{decrgrpulse}:

- To pulse the decoupler the same as \texttt{decpulse} or \texttt{decrgrpulse}, but generate interactive parameter adjustment (IPA) information when \texttt{gf} or \texttt{go('acqi')} is entered, use \texttt{idecpulse(width,phase,string)} or \texttt{idecrgrpulse(width,phase,\textit{RG1},\textit{RG2},string)}, respectively, where \textit{string} is used as a label in \texttt{acqi}. If \texttt{go} is entered instead, the IPA information is not generated. For details on IPA, see “Using Interactive Parameter Adjustment,” page 98.

IPA is not available on \texttt{MERCURYplus/-Vx} systems.

- Use the following to pulse the second decoupler (channel 2):
  \texttt{dec2rgpulse(width,phase,\textit{RG1},\textit{RG2})}.

- Use the following to pulse the third decoupler (channel 4):
  \texttt{dec3rgpulse(width,phase,\textit{RG1},\textit{RG2})}.

- Use the following to pulse \texttt{UNITY} \texttt{INOVA} systems with a deuterium decoupler installed as the fifth channel, a fourth decoupler (channel 5):
  \texttt{dec4rgpulse(width,phase,\textit{RG1},\textit{RG2})}.

- The \texttt{width}, \texttt{phase}, \texttt{RG1}, and \texttt{RG2} arguments have the same meaning as used with \texttt{decrgrpulse} and \texttt{rgpulse}. The \texttt{homo} parameter has no effect on the gating on the second decoupler board. On \texttt{UNITY} \texttt{INOVA} systems only, \texttt{homo} controls the homodecoupler gating of the second decoupler, \texttt{homo} does the same on the third decoupler, and \texttt{homo} does the same on the fourth decoupler when it is used as a deuterium channel (on the \texttt{MERCURYplus/-Vx}, \texttt{dec2rgpulse}, \texttt{dec3rgpulse}, and \texttt{dec4rgpulse} have no meaning and \texttt{homo} is not used).

### Pulsing Channels Simultaneously

Statements for controlling simultaneous, non-shaped pulses are \texttt{simpulse}, \texttt{sim3pulse}, and \texttt{sim4pulse}. Table 7 summarizes these statements. Simultaneous pulses statements using shaped pulses are covered in a later section.

**Table 7. Simultaneous Pulses Statements**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{simpulse*}</td>
<td>Pulse observe and decoupler channels simultaneously</td>
</tr>
<tr>
<td>\texttt{sim3pulse*}</td>
<td>Pulse simultaneously on two or three rf channels</td>
</tr>
<tr>
<td>\texttt{sim4pulse*}</td>
<td>Simultaneous pulse on four channels</td>
</tr>
<tr>
<td>\texttt{sim3pulse(pw1,pw2,pw3,phase1,phase2,phase3,\textit{RG1},\textit{RG2})}</td>
<td>Simultaneous pulse on two or three rf channels</td>
</tr>
<tr>
<td>\texttt{simpulse(pw1,pw2,pw3,phase1,phase2,phase3,\textit{RG1},\textit{RG2})}</td>
<td>Simultaneous pulse on two or three rf channels</td>
</tr>
<tr>
<td>\texttt{sim4pulse(pw1,pw2,pw3,phase1,phase2,phase3,phase4,\textit{RG1},\textit{RG2})}</td>
<td>Simultaneous pulse on four channels</td>
</tr>
</tbody>
</table>

Use \texttt{simpulse(\textit{obswidth},\textit{decwidth},\textit{obsphase},\textit{decphase},\textit{RG1},\textit{RG2})} to simultaneously pulse the observe and first decoupler rf channels with amplifier gating (e.g., \texttt{simpulse(pw,pp,v1,v2,0.0,rof2)}).
Figure 2 illustrates the action of simpulse.

The shorter of the two pulses is centered on the longer pulse, while the amplifier gating occurs before the start of the longer pulse (even if it is the decoupler pulse) and after the end of the longer pulse. The absolute difference in the two pulse widths must be greater than or equal to the following values:

- **UNITYINOVA** systems: 0.2 $\mu$s
- on the **MERCURYplus/-Vx** systems: 0.4 $\mu$s

otherwise, a timed event of less than the following minimum values:

- **UNITYINOVA** systems: 0.1 $\mu$s
- **MERCURYplus/-Vx** systems: 0.2 $\mu$s

would be produced. In such cases, a short time (0.2 $\mu$s to 0.4 $\mu$s) is added to the longer of the two pulse widths to remedy the problem, or the pulses are made the same if the difference is less than half the minimum (less than 0.05 $\mu$s on **UNITYINOVA**, less than 0.2 $\mu$s on **MERCURYplus/-Vx** systems).

`simpulse(pw1,pw2,pw3,phase1,phase2,phase3,RG1,RG2)` performs a simultaneous, three-pulse pulse on three independent rf channels, where `pw1`, `pw2`, and `pw3` are the pulse durations on the observe transmitter, first decoupler, and second decoupler, respectively. `phase1`, `phase2`, and `phase3` are real-time variables for the phases of the corresponding pulses, for example, `simpulse(pw,p1,p2,oph,v10,v1,rof1,rof2)`.

A simultaneous, two-pulse pulse on the observe transmitter and the second decoupler can be achieved by setting the pulse length for the first decoupler to 0.0; for example, `simpulse(pw,0.0,p2,oph,v10,v1,rof1,rof2)`. The `simpulse` statement has no meaning on **MERCURYplus/-Vx**.

Use `simpulse(pw1,pw2,pw3,pw4,phase1,phase2,phase3,phase4, RG1,RG2)` to perform simultaneous pulses on as many as four different rf channels. Except for the added arguments `pw4` and `phase4` for a third decoupler, the arguments in `simpulse` are defined the same as `simpulse`. If any pulse is set to 0.0, no pulse is
executed on that channel. The `sim4pulse` statement has no meaning on `MERCURYplus/-Vx`.

### Setting Transmitter Quadrature Phase Shifts

The statements `txphase`, `decphase`, `dec2phase`, `dec3phase`, `dec4phase` control transmitter quadrature phase (multiple of 90°). Table 8 summarizes these statements.

**Table 8. Transmitter Quadrature Phase Control Statements**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>decphase</code></td>
<td>Set quadrature phase of first decoupler</td>
</tr>
<tr>
<td><code>dec2phase</code></td>
<td>Set quadrature phase of second decoupler</td>
</tr>
<tr>
<td><code>dec3phase</code></td>
<td>Set quadrature phase of third decoupler</td>
</tr>
<tr>
<td><code>dec4phase</code></td>
<td>Set quadrature phase of fourth decoupler</td>
</tr>
<tr>
<td><code>txphase</code></td>
<td>Set quadrature phase of observe transmitter</td>
</tr>
</tbody>
</table>

To set the transmitter phase, use `txphase(phase)`, where `phase` is a real-time variable (v1 to v14, etc.) or a real-time constant (zero, one, etc.) that references the desired phase. This enables changing the transmitter phase independently from a pulse.

For example, knowing that the transmitter phase takes a finite time to shift (about 1 μs on a `MERCURYplus/-Vx`, 200 ns for `UNITY/NOVA`), “preset” the transmitter phase at the beginning of a delay that precedes a particular pulse. The “normal” pulse sequences use an `rof1` time preceding the pulse to change the transmitter phase, and it is not necessary to “preset” the phase. The phase change will occur at the start of the next event in the pulse sequence.

The other phase control statements are variations of `txphase`:

- To set the decoupler phase, use `decphase(phase)`. The `decphase` statement is syntactically and functionally equivalent to `txphase`. `decphase` is useful for a decoupler pulse in all cases where `txphase` is useful for a transmitter pulse.
- To set the quadrature phase of the second decoupler rf or third decoupler rf, use `dec2phase(phase)` or `dec3phase(phase)`, respectively.

The hardware WALTZ decoupling lines are XORed with the decoupler phase control. The performance of the WALTZ decoupling should not be affected by the decoupler phase setting.

When using pulse sequences with implicit acquisition, the decoupler phase is controlled by the relevant shapelib file used, e.g., WALTZ16. Set to 0 automatically (within the `test4acq` procedure in the module `hwlooping.c` in `/vnmr/psg`), so under most circumstances no problems are seen. But if you are using explicit acquisition or if you are trying to perform WALTZ decoupling during a period other than acquisition, you must use a `decphase(zero)` statement in the pulse sequence before the relevant time period.

### Setting Small-Angle Phase Shifts

Setting the small-angle phase of rf pulses is implemented by three different methods:

- Fixed 90° settings
- Direct synthesis hardware control
- Phase-pulse phase shifting

The statements related to these methods are summarized in Table 9.
Chapter 2. Pulse Sequence Programming

### Fixed 90° Settings

The first method is the hardwired 90° (or quadrature) phase setting. Both the observe and the decoupler transmitters invoke phases of 0°, 90°, 180°, and 270° instantaneously using the \texttt{obspulse}, \texttt{pulse}, \texttt{rgpulse}, \texttt{simpulse}, \texttt{decpulse}, \texttt{decrgpulse}, \texttt{dec2rgpulse}, \texttt{dec3rgpulse}, \texttt{dec4rgpulse}, \texttt{txphase}, \texttt{decrphase}, \texttt{dec2phase}, \texttt{dec3phase}, and \texttt{dec4phase} statements.

### Small-Angle Phase Shifts

A second method of small-angle phase selection is implemented only on spectrometers with direct synthesis. The hardware sets transmitter phase in the following increments:

- \textit{UNITY INOVA} systems: 0.25°
- on \textit{MERCURYplus/-Vx} systems: 1.41°

independently of the phase of the receiver. This method is an absolute technique (e.g., if a phase of 60° is invoked twice, the second phase selection does nothing).

The \texttt{obsstepsize(base)} statement sets the step size of the small-angle phase increment to \texttt{base} for the observe transmitter. Similarly, \texttt{decsstepsize(base)}, \texttt{dec2stepsize(base)}, and \texttt{dec3stepsize(base)} set the step size of the small-angle phase increment to \texttt{base} for the first decoupler, second decoupler, and third decoupler, respectively (assuming that system is equipped with appropriate hardware). The \texttt{base} argument is a real number or variable.

The base phase shift selected is active only for the \texttt{xmtrphase(multiplier)} statement if the transmitter is the requested device, only for the \texttt{dclrpulse} statement if the decoupler is the requested device, only for the \texttt{dclr2phase} statement if the second decoupler is the requested device, or only for the \texttt{dclr3phase} if the third decoupler is the required device, that is, every transmitter has its own “base” phase shift. Phase information into \texttt{pulse}, \texttt{rgpulse}, \texttt{decpulse}, \texttt{decrgpulse}, \texttt{dec2rgpulse}, \texttt{dec3rgpulse}, \texttt{txphase}, \texttt{decphase}, \texttt{dec2phase}, \texttt{dec3phase}, and \texttt{dec4phase} statements.

### Table 9. Phase Shift Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{dcplrpulse(multiplier)}</td>
<td>Set small-angle phase of first decoupler</td>
</tr>
<tr>
<td>\texttt{dcplr2pulse(multiplier)}</td>
<td>Set small-angle phase of second decoupler</td>
</tr>
<tr>
<td>\texttt{dcplr3pulse(multiplier)}</td>
<td>Set small-angle phase of third decoupler</td>
</tr>
<tr>
<td>\texttt{decsstepsize(base)}</td>
<td>Set step size of first decoupler</td>
</tr>
<tr>
<td>\texttt{decsstepsize(base)}</td>
<td>Set step size of second decoupler</td>
</tr>
<tr>
<td>\texttt{decsstepsize(base)}</td>
<td>Set step size of third decoupler</td>
</tr>
<tr>
<td>\texttt{obsstepsize(base)}</td>
<td>Set step size of observe transmitter</td>
</tr>
<tr>
<td>\texttt{phaseshift*}</td>
<td>Set phase-pulse technique, rf type A or B</td>
</tr>
<tr>
<td>\texttt{stepsise(base,device)}</td>
<td>Set small-angle phase step size</td>
</tr>
<tr>
<td>\texttt{xmtrphase(multiplier)}</td>
<td>Set small-angle phase of observe transmitter, rf type C</td>
</tr>
</tbody>
</table>

* \texttt{phaseshift(base,multiplier,device)}
The decstepsize, dec2stepsize, dec3stepsize, and obsstepsize statements are similar to the stepsize statement but have the channel selection fixed. Each of the following pairs of statements are functionally the same:

- obsstepsize(base) and stepsize(base,OBSch).
- decstepsize(base) and stepsize(base,DECch).
- dec2stepsize(base) and stepsize(base,DEC2ch).
- dec3stepsize(base) and stepsize(base,DEC3ch).

On systems with Output boards only, if the product of the base and multiplier is greater than 90°, the sub-90° part is set by the xmtrphase, dclplrphase, dclplr2phase, or dclplr3phase statements. Carryovers that are multiples of 90° are automatically saved and added in at the time of the next 90° phase selection (e.g., at the time of the next pulse or decpulse). This is true even if stepsize has not been used and base is at its default value of 90°. The following example may help you to understand this question of “carryovers”:

```plaintext
obsstepsize(60.0); /* set 60° step size for obs. xmtr*/
initval(6.0,v1); modn(ct,v1,v2); /* v2=012345012345 */
xmtrphase(v2); /* phase=0,60,120,180,240,300 */
/* small-angle part=0,60,30,0,60,30 */
/* carry-over=0,0,90,180,180,270 */
```

mod4(ct,v3); pulse(pw,v3); /* specified phase=0,90,180,270 */
/* 90° phase shift actually used */
/* = 0,90,270,450,180,360 */
/* = specified + carry-over */

If xmtrphase, dclplrphase, dclplr2phase, or dclplr3phase is used to set the phase for some pulses in a pulse sequence, it is often necessary to use xmtrphase(zero), dclplrphase(zero), dclplr2phase(zero), or dclplr3phase(zero) preceding other pulses to ensure that the phase specified by a previous xmtrphase, dclplrphase, dclplr2phase, or dclplr3phase does not carry over into an unwanted pulse or decpulse statement.

Phases specified in txphase, pulse, rgpulse, decphase, decpulse, decrgpulse, dec2phase, dec2rgpulse, dec3rgpulse, and dec4rgpulse statements change the 90° portion of the phase shift only. This feature provides a separation between the small-angle phase shift and the 90° phase shifts and facilitates programming phase cycles or additional coherence transfer selective phase cycling “on top of” small-angle phase shifts.

Be sure to distinguish xmtrphase from txphase. txphase is optional and needed if the gating time RG1 is set to zero in pulse statements; xmtrphase is needed any time the transmitter phase shift is to be set to a value not a multiple of 90°. The same distinction can be made between dclplrphase and decphase, dclplr2phase and dec2phase, and dclplr3phase and dec3phase.

### Controlling the Offset Frequency

Statements for frequency control are decoffset, dec2offset, dec3offset, dec4offset, obsoffset, offset, and ioffset. Table 10 summarizes these statements.

The main statement to set the offset frequency of the observe transmitter (parameter tof), first decoupler (dof), second decoupler (dof2), or third decoupler (dof3) is the statement offset(frequency,device), where frequency is the new value of the
appropriate parameter and device is OBSch (observe transmitter), DECch (first decoupler), DEC2ch (second decoupler), or DEC3ch (third decoupler). However, for clarity, use obsoffset, decoffset, etc. in actual coding. For example, use offset(to2, OBSch) to set the observe transmitter offset frequency. DEC2ch can be used only on systems with three rf channels. Likewise, DEC3ch is used only on systems with four rf channels.

- For systems with rf type D, the frequency shift time is 14.95 μs (latching with or without over-range). No 100-μs delay is inserted into the sequence by the offset statement. Offset frequencies are not returned automatically to their “normal” values before acquisition; this must be done explicitly, as in the example below.

- The frequency shift time is 4 μs for UNITY INOVA systems.

- The setup time is 86.4 μs and the shift time is 1 μs for MERCURYplus/-Vx systems.

Other frequency control statements are variations of offset:

- To set the offset frequency of the observe transmitter the same as offset but generate interactive parameter adjustment (IPA) information when gf or go('acqi') is entered, use ioffset(frequency, device, string), where string is used as a label for the slider in acqi. If go is entered instead, the IPA information is not generated. For details on IPA, see “Using Interactive Parameter Adjustment,” page 98. IPA is not available on MERCURYplus/-Vx systems.

- To set the offset frequency of the observe transmitter (parameter tof), use obsoffset(frequency), which functions the same as offset(frequency, OBSch).

- To set the offset frequency of the first decoupler (parameter dof), use decoffset(frequency), which functions the same as offset(frequency, DECch).

- To set the offset frequency of the second decoupler (parameter dof2), use dec2offset(frequency), which functions the same as offset(frequency, DEC2ch).

- To set the offset frequency of the third decoupler (parameter dof3), use dec3offset(frequency), which functions the same as offset(frequency, DEC3ch).

- To set the offset frequency of the fourth decoupler used as the fifth channel (parameter dof4), use dec4offset(frequency), which functions the same as offset(frequency, DEC4ch).

### Controlling Observe and Decoupler Transmitter Power

Statements to control power by adjusting the coarse attenuators on linear amplifier systems are power, obspower, decpower, dec2power, dec3power, and dec4power.
2.3 Spectrometer Control

Statements to control fine power are \texttt{pwrf}, \texttt{pwrm}, \texttt{rlpwrn}, \texttt{obspwr}, \texttt{decpwr}, \texttt{dec2pwr}, and \texttt{dec3pwr}. Statements to control decoupler power level switching are \texttt{declvlon}, \texttt{declvloff}, and \texttt{decpwr}. Table 11 summarizes these statements.

Table 11. Power Control Statements

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{declvloff()}</td>
<td>Return first decoupler back to “normal” power</td>
</tr>
<tr>
<td>\texttt{declvlon()}</td>
<td>Turn on first decoupler to full power</td>
</tr>
<tr>
<td>\texttt{decpower(value)}</td>
<td>Change first decoupler power, linear amplifier</td>
</tr>
<tr>
<td>\texttt{dec2power(value)}</td>
<td>Change second decoupler power, linear amplifier</td>
</tr>
<tr>
<td>\texttt{dec3power(value)}</td>
<td>Change third decoupler power, linear amplifier</td>
</tr>
<tr>
<td>\texttt{dec4power(value)}</td>
<td>Change deuterium decoupler power, linear amplifier</td>
</tr>
<tr>
<td>\texttt{decpwr(value)}</td>
<td>Set decoupler high-power level, class C amplifier</td>
</tr>
<tr>
<td>\texttt{decpwrf(value)}</td>
<td>Set first decoupler fine power</td>
</tr>
<tr>
<td>\texttt{dec2pwrf(value)}</td>
<td>Set second decoupler fine power</td>
</tr>
<tr>
<td>\texttt{dec3pwrf(value)}</td>
<td>Set third decoupler fine power</td>
</tr>
<tr>
<td>\texttt{obspower(value)}</td>
<td>Change observe transmitter power, linear amplifier</td>
</tr>
<tr>
<td>\texttt{obspwrf(value)}</td>
<td>Set observe transmitter fine power</td>
</tr>
<tr>
<td>\texttt{power(value,device)}</td>
<td>Change transmitter or decoupler power, linear amplifier</td>
</tr>
<tr>
<td>\texttt{pwrf(value,device)}</td>
<td>Change transmitter or decoupler linear mod. power</td>
</tr>
<tr>
<td>\texttt{pwrn(value,device)}</td>
<td>Change transmitter or decoupler fine power</td>
</tr>
<tr>
<td>\texttt{rlpwrn(rlvalue,device)}</td>
<td>Set transmitter or decoupler linear mod. power</td>
</tr>
</tbody>
</table>

Coarse Attenuator Control

\textit{UNITY/INOVA} systems with linear amplifiers use the lower-level statement \texttt{power(value,device)} to change transmitter or decoupler power by adjusting the coarse attenuators from 0 (minimum power) to 63 (maximum power) on channels with a 63-dB attenuator, or from –16 (minimum power) to 63 (maximum power) on channels with a 79-dB attenuator.

- Value must be stored in a real-time variable such as \texttt{v2} for this form of control; the actual value cannot be placed directly in the \texttt{power} statement. This allows the attenuators to be changed in real-time or from pulse to pulse.

- device is \texttt{OBSch} to change the transmitter power, \texttt{DECch} to change the first decoupler power, \texttt{DEC2ch} to change the second decoupler power, or \texttt{DEC3ch} to change the third decoupler power (e.g., \texttt{power(v2,OBSch)}).

To avoid using a real-time variable, the fixed-channel statements \texttt{obspower(value)}, \texttt{decpower(value)}, \texttt{dec2power(value)}, and \texttt{dec3power(value)} should be used in place of the power statement, for example, \texttt{obspower(63.0)}. For all of these statements, value is either a real number or a variable. These statements are typically used in most sequences.

These power and associated fixed-channel statements allow configurations such as the use of the transmitter at a low power level for presaturation followed by a higher power for uniform excitation. The phase of the transmitter is specified as being constant to within 5° over the whole range of transmitter power. Therefore, pulsing at low power with a certain phase and later at high power with the same phase, the two phases are the “same” to within 5° (at any one power level, the phase is constant to considerably better than 0.5°). The time of the power change is specified in Table 33.

While no psg delay is associated with the coarse power control, the device itself takes about 2 microseconds to stabilize at the new value. This will happen in parallel with the next psg event in the program. This stabilization time is inconsequential except for back-to-back power statements.
On systems with an Output board only, the power and associated statements are preceded internally by a 0.2 μs delay by default (see the apovrride pulse statement for more details).

**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 safety maximum 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.

**Fine-Power Control**

To change the fine power of a transmitter or decoupler by adjusting the optional linear fine attenuators, use `pwrf(value,device)` or `pwrm(value,device)`. The value argument is real-time variable, which means it cannot be placed directly in the `pwrf` or `pwrm` statement, and can range from 0 to 4095 (60 dB on UNITY INOVA, about 6 dB on other systems). The device is OBSch (for the observe transmitter), or DECch (first decoupler), (on UNITY INOVA only), device can also be DEC2ch (second decoupler), or DEC3ch (third decoupler). MERCURYplus/-Fx systems do not support `pwrf` and `pwrm` with real-time parameters but support all other parameters.

The fixed-channel statements `obspwrf(value)`, `decpwrf(value)`, `dec2pwrf(value)`, and `dec3pwrf`, or `rlpwrm(value,device)` to avoid arguments using real-time variables and are the preferred usage. These statements change transmitter or decoupler power on systems with linear amplifiers and value is either a real number or a variable and is stored in a C variable of type double.

The `ipwrf(value,device,string)` and `ipwrm(value,device,string)` statement changes interactively the transmitter or decoupler fine power or linear modulators by adjusting the optional fine attenuators. The value and device arguments are the same as `pwrf`. string can be any string; the first six letters are used in acqi. This statement will generate interactive parameter adjustment (IPA) information only when the command `gf` or `go('acqi')` is typed. When the command `go` is typed, this statement is ignored by the pulse sequence. Use the `pwrf` pulse statement for this purpose. Do not execute `pwrf` and `ipwrf` in the same pulse sequence, as they cancel each other’s effect.

On systems with an Output board only, a 0.2 μs delay internally precedes the AP (analog port) bus statements power, obspower, decpower, and dec2power. The apovrride() statement prevents this 0.2 μs delay from being inserted prior to the next (and only the next) occurrence of one of the these AP bus statements.

**Decoupler Power-Level Switching**

On UNITY INOVA systems with class C or linear amplifiers, `declvlon()` and `declvloff()` switch the decoupler power level between the power level set by the high-power parameter(s) to the full output of the decoupler. The statement `declvlon()` gives full power on the decoupler channel; `declvloff` switches the decoupler to the power level set by the appropriate parameters defined by the amplifier type: dhp for class C amplifiers or dpwr for a linear amplifiers. If dhp='n', these statements do not have any effect on systems with class C amplifiers, but still function for systems with linear amplifiers.
If `declvlon` is used, make sure `declvloff` is used prior to time periods in which normal, controllable power levels are desired, for example, prior to acquisition. Full decoupler power should only be used for decoupler pulses or for solids applications. *MERCU*ryplus/-Vx systems do not use `declvlon` or `declvloff`.

**Status and Gating**

Statements to control decoupler and homospoil status are `status` and `setstatus`. Explicit transmitter and receiver gating control statements are `xmtroff`, `xmtron`, `decoff`, `decon`, `dec2off`, `dec2on`, `dec3off`, `dec3on`, `rcvroff`, and `rcvron`. Statements for amplifier blanking and unblanking are `obsblank`, `obsunblank`, `decblank`, `decunblank`, `dec2blank`, `dec2unblank`, `dec3blank`, `dec3unblank`, `blankingoff`, and `blankingon`. Finally, statements for user-dedicated lines are `sp#off` and `sp#on (#=1, 2, or 3). Table 12 summarizes these statements.

**Table 12. Gating Control Statements**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>blankingoff()</code></td>
<td>Unblank amplifier channels and turn amplifiers on</td>
</tr>
<tr>
<td><code>blankingon()</code></td>
<td>Blank amplifier channels and turn amplifiers off</td>
</tr>
<tr>
<td><code>decblank()</code></td>
<td>Blank amplifier associated with the 1st decoupler</td>
</tr>
<tr>
<td><code>dec2blank()</code></td>
<td>Blank amplifier associated with the 2nd decoupler</td>
</tr>
<tr>
<td><code>dec3blank()</code></td>
<td>Blank amplifier associated with the 3rd decoupler</td>
</tr>
<tr>
<td><code>decoff()</code></td>
<td>Turn off first decoupler</td>
</tr>
<tr>
<td><code>dec2off()</code></td>
<td>Turn off second decoupler</td>
</tr>
<tr>
<td><code>dec3off()</code></td>
<td>Turn off third decoupler</td>
</tr>
<tr>
<td><code>decon()</code></td>
<td>Turn on first decoupler</td>
</tr>
<tr>
<td><code>dec2on()</code></td>
<td>Turn on second decoupler</td>
</tr>
<tr>
<td><code>dec3on()</code></td>
<td>Turn on third decouper</td>
</tr>
<tr>
<td><code>decunblank()</code></td>
<td>Unblank amplifier associated with the 1st decoupler</td>
</tr>
<tr>
<td><code>dec2unblank()</code></td>
<td>Unblank amplifier associated with the 2nd decoupler</td>
</tr>
<tr>
<td><code>dec3unblank()</code></td>
<td>Unblank amplifier associated with the 3rd decoupler</td>
</tr>
<tr>
<td>`dhpflag=TRUE</td>
<td>FALSE`</td>
</tr>
<tr>
<td><code>initparms_sis()</code></td>
<td>Initialize parameters for spectroscopy imaging sequences</td>
</tr>
<tr>
<td><code>obsblank()</code></td>
<td>Blank amplifier associated with observe transmitter</td>
</tr>
<tr>
<td><code>obsunblank()</code></td>
<td>Explicitly enables the amplifier for the observe transmitter</td>
</tr>
<tr>
<td><code>rcvroff()</code></td>
<td>Turn off receiver gate and amplifier blanking gate</td>
</tr>
<tr>
<td><code>rcvron()</code></td>
<td>Turn on receiver gate and amplifier blanking gate</td>
</tr>
<tr>
<td><code>recoff()</code></td>
<td>Turn off receiver gate only</td>
</tr>
<tr>
<td><code>recon()</code></td>
<td>Turn on receiver gate only</td>
</tr>
<tr>
<td><code>setstatus*</code></td>
<td>Set status of observe transmitter or decoupler transmitter</td>
</tr>
<tr>
<td><code>status(state)</code></td>
<td>Change status of decoupler and homospoil</td>
</tr>
<tr>
<td><code>statusdelay(state,time)</code></td>
<td>Execute status statement with given delay time</td>
</tr>
<tr>
<td><code>xmtroff()</code></td>
<td>Turn off observe transmitter</td>
</tr>
<tr>
<td><code>xmtron()</code></td>
<td>Turn on observe transmitter</td>
</tr>
</tbody>
</table>

**Gating States**

Use `status(state)` to control decoupler and homospoil gating in a pulse sequence, where `state` is A to Z (e.g., `status(A)` or `status(B)`). Parameters controlled by `status` are `dm` (first decoupler mode), `dmm` (first decoupler modulation mode), and `hs`...
(homospoil). For systems with a third or fourth rf channel, dm2 and dm3 (second and third decoupler modes) and dmm2 and dmm3 (second and third decoupler modulation mode) are also under status control. For systems with a deuterium decoupler channel as the fourth decoupler, dm4 and dmm4 are under status control.

Each of these parameters can have multiple states: status(A) sets each parameter to the state described by the first letter of its value; status(B) uses the second letter, etc. If a pulse sequence has more status statements than there are status modes for a particular parameter, control reverts to the last letter of the parameter value. Thus, if dm='ny', status(C) will look for the third letter, find none, and then use the second letter (y) and turn the decoupler on.

Use setstatus(channel, on, mode, sync, mod_freq) to control decoupler gating as well as decoupler modulation modes (GARP, CW, WALTZ, etc.). channel is OBSch, DECch, DEC2ch, or DEC3ch, on is TRUE or FALSE, mode is a decoupler mode (’c’, ’g’, ’p’, etc.), sync is TRUE or FALSE, and mod_freq is the modulation frequency (e.g., setstatus(DECch, TRUE, ’w’, FALSE, dmf)). The setstatus statement is not available on the MERCURYplus/-Vx.

setstatus provides a way to set transmitters independent of the parameters, one channel at a time. For example, setstatus(OBSch, TRUE, ’g’, TRUE, obs_mf), turns the observe transmitter (OBSch) on (TRUE), using GARP modulation (’g’) in synchronized mode (TRUE) with a modulation frequency of obs_mf. (The obs_mf parameter will need to be calculated from a parameter set with an appropriate getval statement.)

Note: Be sure to set the power to a safe level before calling setstatus.

Timing for setstatus is the same as for status except that only one channel needs to be taken into account. To ensure that the timing is constant for the status, use the statusdelay statement (e.g., statusdelay(A, 2.0e-5))

Homospoil gating is treated somewhat differently than decoupler gating. If a particular homospoil code letter is ’y’, delays coded as hsdelay that occur when the status corresponds to that code letter will begin with a homospoil pulse, the duration of which is determined by the parameter hst. Thus if hs=’ny’, all hsdelay delays that occur during status(B) will begin with a homospoil pulse. The final status always occurs during acquisition, at which time a homospoil pulse is not permitted. Thus, if a particular pulse sequence uses status(A), status(B), and status(C), dm and other decoupler parameters may have up to three letters, but hs will only have two, since hs=’y’ during status(C) would be meaningless and is ignored.

Transmitter Gating

On all systems, transmitter gating is handled as follows:

- Explicit transmitter gating in the pulse sequence is provided by xmtroff() and xmtron(). Transmitter gating is handled automatically by obspulse, pulse, rgpulse, simpulse, sim3pulse, shaped_pulse, simshaped_pulse, sim3shaped_pulse, and spinlock. The obsprgon statement should generally be enabled with an explicit xmtron statement, followed by xmtroff.
- Explicit gating of the first decoupler in the pulse sequence is provided by decoff() and decon(). First decoupler gating is handled automatically by decpulse, decr gpulse, declvlon, declvloff, simpulse, sim3pulse, decshaped_pulse, simshaped_pulse, sim3shaped_pulse, and decspinlock. The decprgon function should generally be enabled with explicit decon statement and followed by a decoff call.
2.3 Spectrometer Control

• Explicit gating of the second decoupler in the pulse sequence is provided by `dec2off` and `dec2on`. Second decoupler gating is handled automatically by `dec2pulse`, `dec2rgpulse`, `sim3pulse`, `dec2shaped_pulse`, `sim3shaped_pulse`, and `dec2spinlock`. The `dec2prgon` function should generally be enabled with an explicit `dec2con` statement, followed by `dec2off`.

• Likewise, explicit gating of the third decoupler in the pulse sequence is provided by `dec3off` and `dec3on`. Third decoupler gating is handled automatically by `dec3pulse`, `dec3rgpulse`, `dec3shaped_pulse`, and `dec3spinlock`. The `dec3prgon` function should generally be enabled with an explicit `dec3con` statement, followed by `dec3off`.

Receiver Gating

Explicit receiver gating in the pulse sequence is provided by the `rcvroff()`, `rcvron()`, `recoff()`, and `recon()` statements. These statements control the receiver gates except when pulsing the observe channel (in which case the receiver is off) or during acquisition (in which case the receiver is on). The `recoff` and `recon` statements (available only on `UNITY INOVA` systems) affect the receiver gate only and do not affect the amplifier blanking gate, which is the role of `rcvroff` and `rcvron`.

• The receiver is on only during acquisition, except for certain imaging pulse sequences on `UNITY INOVA`, that have explicit acquires (such as SEMS, MEMS, and FLASH), and for the `initparma_sis()` statement that defaults the receiver gate to on.

• On `MERCURYplus/-Vx`, receiver gating is tied to the amplifier blanking and is normally controlled automatically by the pulse statements `rgpulse`, `pulse`, `obspulse`, `decrgpulse`, `decpulse`, and `dec2rgpulse`.

Amplifier Channel Blanking and Unblanking

Amplifier channel blanking and unblanking methods depend on the system.

• The receiver and amplifiers are not linked on `UNITY INOVA`. To explicitly blank and unblank amplifiers, the following statements are provided:

  For the amplifier associated with the observe transmitter:
  `obsblank()` and `obsunblank()`.

  For the amplifiers associated with the first, second, and third decouplers:
  `decblank()` and `decunblank()`, `dec2blank()` and `dec2unblank()`, and `dec3blank()` and `dec3unblank()`, respectively.

  These statements replace `blankon` and `blankoff`, no longer in VnmrJ.

• On `MERCURYplus/-Vx`, the receiver and amplifier are linked. At the end of each pulse statement, the receiver is automatically turned back on and the amplifier blanked. Immediately prior to data acquisition, the receiver is implicitly turned back on.

Interfacing to External User Devices

The `sp#on` and `sp#off` statements are used for interfacing with external user devices.

User-Dedicated Spare Lines

One or more user-dedicated spare lines are available for high-speed device control:

• `UNITY INOVA` consoles have five spare lines in the Breakout panel on the rear of the left cabinet. Each spare line is a BNC connector. The `sp#on()` and `sp#off()` statements control specified SPARE lines (#= spare line 1, 2, 3, 4, or 5).
User AP (Analog Port) Lines

UNITY/INOVA consoles have two 24-pin user AP connectors, J8212 and J8213, in the Breakout panel on the rear of the left cabinet. Each connector has 16 user-controllable lines coinciding with two 8-bit AP bus registers. All four of the AP bus registers are writeable but only one register is readable.

Table 13 shows the mapping of the user AP lines. On both connectors, lines 17 to 25 are ground lines.

User AP lines allow the synchronous access by users to external services while running a pulse sequence. The statements

```
setuserap(value,reg), vsetuserap(rtvar,reg), and readuserap(rtvar)
```

provide access to these lines.

The `setuserap` and `vsetuserap` statements enable writing 8-bit information to one of four registers. Each write takes one AP bus cycle, which is 0.5 μs for the UNITY/INOVA. The only difference between `setuserap` and `vsetuserap` is that `vsetuserap` uses a real-time variable to set the value.

The `readuserap` statement lets you read 8-bit information from the register into a real-time variable. You can then act on this information using real-time math and real-time control statements while the pulse sequence is running; however, because the system has to wait for the data to be read before it can continue parsing and stuffing the FIFO, a significant amount of overhead is involved in servicing the read and refilling the FIFO. The `readuserap` statement takes 500 μs to execute. The `readuserap` statement puts in a 500 μs delay immediately after reading the user AP lines in order for the parser to parse and stuff more words into the FIFO before it underflows. However, this time may not be long enough and you may want to pad this time with a delay immediately following the `readuserap` statement to avoid FIFO underflow. Depending on the actions in the pulse sequence, your delay may need to be a number of milliseconds. If there is an error in the read, a warning message is sent to the host and a –1 is returned to the real-time variable.

### Table 13. Mapping of User AP Lines

<table>
<thead>
<tr>
<th>Register</th>
<th>Connector</th>
<th>Lines</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>J8213</td>
<td>9 to 16</td>
<td>output</td>
</tr>
<tr>
<td>1</td>
<td>J8213</td>
<td>1 to 8</td>
<td>output</td>
</tr>
<tr>
<td>2</td>
<td>J8212</td>
<td>9 to 16</td>
<td>output</td>
</tr>
<tr>
<td>3</td>
<td>J8212</td>
<td>1 to 8</td>
<td>input/output</td>
</tr>
</tbody>
</table>

2.4 Pulse Sequence Statements: Phase and Sequence Control

- “Real-Time Variables and Constants,” page 77
- “Calculating in Real-Time Using Integer Mathematics,” page 78
- “Controlling a Sequence Using Real-Time Variables,” page 79
- “Real-Time vs. Run-Time—When Do Things Happen?,” page 80
- “Manipulating Acquisition Variables,” page 80
- “Intertransient and Interincrement Delays,” page 81
- “Controlling Pulse Sequence Graphical Display,” page 82

A series of internal variables, named v1, v2, ..., v14, are provided to perform calculations during “real-time” (while the pulse sequence is executing). All real-time variables are pointers to particular memory locations in the controller memory. A real-time variable does not change, rather the value in the memory location to which that real-time variable points is changed.
2.4 Pulse Sequence Statements: Phase and Sequence Control

For example, when we speak of \( v_1 \) being set equal to 1, what we really mean is that the value in the memory location pointed to by the real-time variable \( v_1 \) is 1. The actual value of \( v_1 \), a pointer, is not changed. The two ideas are interchangeable as long as we recognize exactly what is happening at the level of the controller memory.

These internal, real-time variables can be used for a number of purposes, but the two most important are control of the pulse sequence execution (for looping and conditional execution, for example) and calculation of phases. For each pulse in the sequence, the phase is calculated dynamically (at the start of each transient) rather than entirely at the start of this experiment. This allows phase cycles to attain essentially unlimited length, because only one number must be calculated for each phase during each transient. By contrast, attempting to calculate in advance a phase cycle with a cycle of 256 transients and different phases for each of 5 different pulses would require storing \( 256 \times 5 \) or 1280 different phases.

Real-Time Variables and Constants

The following variables and constants can be used for real-time calculations:

- **\( v_1 \) to \( v_{14} \)**: Real-time variables are used for calculations of loops, phases, etc. They are at the complete disposal of the user. The variables point to 16-bit integers, which can hold values of \(-32768\) to \(+32767\).
- **\( ct \)**: Completed transient counter, which 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, ...) is the basis for most calculations. Steady-state transients, invoked by the \( ss \) parameter, do not change \( ct \).
- **\( bsctr \)**: Block size counter, which points to a 16-bit integer that is decremented from \( bs \) to 1 during each block of transients. After completing the last transient in the block, \( bsctr \) is set back to a value of \( bs \). Thus if \( bs=8 \), \( bsctr \) has successive values of 8,7,6,5,4,3,2,1,8,7, ... .
- **\( oph \)**: Real-time variable that controls the phase of the receiver in 90° increments \((0°, 1=90°, 2=180°, \text{and} 3=270°)\). Prior to the execution of the pulse sequence itself, \( oph \) is set to 0 if parameter \( cp \) is set to 'n', or to the successive values 0, 1, 2, 3, 0, 1, 2, 3, ... if \( cp \) is set to 'y'. The value of \( oph \) can be changed explicitly in the pulse sequence by any of the real-time math statements described in the next section (assign, add, etc.) and is also changed by the setreceiver statement.
- **zero, one, two, three** Points to constants set to select constant phases of 0°, 90°, 180°, and 270°. They cannot be replaced by numbers 0, 1, 2, and 3.
- **ssval, ssctr, bsval** Real-time variables described in “Manipulating Acquisition Variables,” page 80.
- **id2, id3, id4** Pointers (or indexes) to constants identifying the current increment in multidimensional experiments. \( id2 \) is the current \( d2 \) increment. Its value ranges from 0 to the size of the \( d2 \) array minus 1, which is typically 0 to \((ni-1)\). \( id3 \) corresponds to current index of the \( d3 \) array in a 3D experiment. Its range is 0 to \((ni2-1)\). \( id4 \) corresponds to the current index of the \( d4 \) array. Its range is 0 to \((ni3-1)\). Only MERCURYplus/-Vx support \( id2 \).
Chapter 2. Pulse Sequence Programming

Calculating in Real-Time Using Integer Mathematics

A series of special integer mathematical statements are provided that are fast enough to execute in real-time: add, assign, dbl, decr, divn, hlv, incr, mod2, mod4, modn, mult, and sub. These statements are summarized in Table 14.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(vi,vj,vk)</td>
<td>Add integer values: set vk equal to vi + vj</td>
</tr>
<tr>
<td>assign(vi,vj)</td>
<td>Assign integer values: set vj equal to vi</td>
</tr>
<tr>
<td>dbl(vi,vj)</td>
<td>Double an integer value: set vj equal to 2*vi</td>
</tr>
<tr>
<td>decr(vi)</td>
<td>Decrement an integer value: set vi equal to vi - 1</td>
</tr>
<tr>
<td>divn(vi,vj,vk)</td>
<td>Divide integer values: set vk equal to vi div vj</td>
</tr>
<tr>
<td>hlv(vi,vj)</td>
<td>Find half the value of an integer: set vj to integer part of 0.5*vi</td>
</tr>
<tr>
<td>incr(vi)</td>
<td>Increment an integer value: set vi equal to vi + 1</td>
</tr>
<tr>
<td>mod2(vi,vj)</td>
<td>Find integer value modulo 2: set vj equal to vi modulo 2</td>
</tr>
<tr>
<td>mod4(vi,vj)</td>
<td>Find integer value modulo 4: set vj equal to vi modulo 4</td>
</tr>
<tr>
<td>modn(vi,vj,vk)</td>
<td>Find integer value modulo n: set vk equal to vi modulo vj</td>
</tr>
<tr>
<td>mult(vi,vj,vk)</td>
<td>Multiply integer values: set vk equal to vi*vj</td>
</tr>
<tr>
<td>sub(vi,vj,vk)</td>
<td>Subtract integer values: set vk equal to vi – vj</td>
</tr>
</tbody>
</table>

Remember that integer mathematics does not include fractions. If a fraction appears in a result, the value is truncated; thus, one-half of 3 is 1, not 1.5.

Integer statements also use the modulo, which is the number that remains after the modulo number is divided into the original number. For example, the value of 8 modulo 2 (often abbreviated “8 mod 2”) is found by dividing 2 into 8, giving an answer of 4 with a remainder of 0, so 8 mod 2 is 0. Similarly, 9 mod 2 is 1, since 2 into 9 gives 4 with a remainder of 1. The modulus of a negative number is not defined in VnmrJ software and should not be used.

Each statement performs one calculation at a time. For example, hlv(ct,v1) takes half the current value of ct and places it in the variable v1. Before each transient, ct has a given value (e.g., 7), and after this calculation, v1 has a certain value (e.g., 3 if ct was 7).

To visualize the action of a statement over the course of a number of transients, pulse sequences typically document this action explicitly as part of their comments. The comment v1=0,0,1,1,... (or v1=001122...) means that v1 assumes a value of 0 during the first transient, 0 during the second, 1 during the third, etc.

The following series of examples illustrates the action of integer mathematics statements and how comments are typically used:

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

mod2(ct,v2);   /* v2=010101... */
dbl(v2,v3);    /* v3=020202... */
    /* v1=00112233... */
hlv(v1,v2);    /* v2=00001111.... */
dbl(v1,v1);    /* v1=00224466.... */
add(v1,v2,v3); /* v3=00225577.... */
mod4(v3,oph);  /* oph=00221133...,receiver phase cycle */
```

Note that the same variable can be used as the input and output of a particular statement (e.g., dbl(v1,v1) is fine so it is not necessary to use dbl(v1,v2)). Note also that
although the mod4 statement is used in several cases, it is never necessary to include it, even if appropriate, because an implicit modulo 4 is always performed on all phases (except when setting small-angle phase shifts).

The division provided by the divn statement is integer division, thus remainders are ignored. \( \texttt{v} \) in each case must be a real-time variable and not a real number (like 6.0) or even an integer constant (like 6). To perform, for example, a modulo 6 operation, something like the following is required:

\[
\text{initval}(6.0,v1); \\
\text{modn}(v2,v1,v7);  \\
\text{endloop}(v3);
\]

Controlling a Sequence Using Real-Time Variables

In addition to being used for phase calculations, real-time variables can also be used for pulse sequence control. Table 15 lists pulse sequence control statements.

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{elsenz(vi)}</td>
<td>Execute succeeding statements if argument is nonzero</td>
</tr>
<tr>
<td>\texttt{endif(vi)}</td>
<td>End ifzero statement</td>
</tr>
<tr>
<td>\texttt{endloop(index)}</td>
<td>End loop</td>
</tr>
<tr>
<td>\texttt{ifzero(vi)}</td>
<td>Execute succeeding statements if argument is zero</td>
</tr>
<tr>
<td>\texttt{initval(realnumber,vi)}</td>
<td>Initialize a real-time variable to specified value</td>
</tr>
<tr>
<td>\texttt{loop(count,index)}</td>
<td>Start loop</td>
</tr>
</tbody>
</table>

By placing pulse sequence statements between a \texttt{loop(count,index)} statement and an \texttt{endloop(index)} statement, the enclosed statements can be executed repeatedly. The \texttt{count} argument used with \texttt{loop} is a real-time variable that specifies the number of times to execute the enclosed statements. \texttt{count} can be any positive number, including zero. \texttt{index} is a real-time variable used as a temporary counter to keep track of the number of times through the enclosed statements and must not be altered by any of the statements. An example of using \texttt{loop} and \texttt{endloop} is the following:

\[
\text{mod4}(ct,v5);  \\
\text{loop}(v5,v3);  \\
\text{delay}(d3);  \\
\text{endloop}(v3);
\]

Statements within the pulse sequence can be executed conditionally by being enclosed within \texttt{ifzero(vi)}, \texttt{elsenz(vi)}, and \texttt{endif(vi)} statements. \texttt{vi} is a real-time variable used as a test variable, to be tested for either being zero or non-zero. The \texttt{elsenz} statement may be omitted if it is not desired. It is also not necessary for any statements to appear between the \texttt{ifzero} and the \texttt{elsenz} or the \texttt{elsenz} and the \texttt{endif} statements. The following code is an example of a conditional construction:

\[
\text{ifzero}(v1);  \\
\text{pulse}(pw,v2);  \\
\text{delay}(d3);  \\
\text{else}(v1);  \\
\text{pulse}(2.0*pw,v2);  \\
\text{delay}(d3/2.0);  \\
\text{endif}(v1);
\]

If numbers other than those easily accessible in integer math (such as \texttt{ct}, \texttt{oph}, \texttt{three}) are needed, any variable can be initialized to a value with the \texttt{initval(number,vi)} statement (e.g., \texttt{initval(4.0,v9)}. The real number input is rounded off and placed in the variable \texttt{v}. This statement, unlike the statements such as \texttt{add} and \texttt{sub} described
above, is executed once and only once at the start of a non-arrayed 1D experiment or at the start of each increment in a 2D experiment or an arrayed 1D experiment, not at the start of each transient.

**Real-Time vs. Run-Time—When Do Things Happen?**

It may help to explain the pulse sequence execution process in more detail. When an experiment begins, the go program is executed. This program looks up the various parameters, finds the name of the current pulse sequence, and looks in seqlib for a file of that name. The file in seqlib is a compiled C program, which was compiled with the seqgen command. This program, which is run by the go program, combines the parameters supplied to it by go together with a series of instructions that form the pulse sequence.

The output of the pulse sequence program in seqlib is a table of numbers, known as the code table (generally referred to as Acodes or Acquisition codes), which contains instructions for executing a pulse sequence in a special language. The pulse sequence program sends a message to the acquisition computer to begin operation, informing it where the code table is stored. This code table is downloaded into the acquisition computer and processed by an interpreter, which is executing in the acquisition computer and which controls operation during acquisition. If after entering go or su, etc., the message that PSG aborted abnormally appears, run the psg macro to help identify the problem.

A pulse sequence can intermix statements involving C, such as \(d2 = \frac{1.0}{2.0*J}\), with special statements, such as hlv\((ct,v2)\). These two statements are fundamentally different kinds of operations. Entering go causes the evaluation of all higher-level expressions once for each increment. Thus in \(d2 = \frac{1.0}{2.0*J}\), the value of \(J\) is looked up, \(d2\) is calculated as one divided by \(2*J\), and the value of \(d2\) is fixed. Statements in this category are called run-time, since they are executed when go is run. The hlv statement, however, is executed every transient. Before each transient, the system examines the current value of \(ct\), performs the integer hlv operation, and sets the variable \(v2\) (used for phases, etc.) to that value. On successive transients, \(v2\) has values of 0,0,1,1,2,2, etc. Statements like these are called real-time, because they execute during the real-time operation of the pulse sequence.

Run-time statements are statements that are evaluated and executed in the host computer by the pulse sequence program in seqlib when go is entered. Real-time statements are statements that are repeatedly (every transient) executed by the code program run in a specific controller. Therefore, it is not possible to include a statement like \(d2 = 1.0 + 0.33*ct\). The variable \(ct\) is a real-time variable (it is actually an integer pointer variable), while “C-type” mathematics are a run-time operation. Only the special real-time statements included in this section can be executed on a transient-by-transient basis.

**Manipulating Acquisition Variables**

Certain acquisition parameters, such as ss (steady-state pulses) and bs (block size), cannot be changed in a pulse sequence with a simple C statement. The reason is that by the time the pulse sequence function is executed, the values of these variables are already stored in a region of the host computer memory that will subsequently be part of the “low-core” portion of the acquisition code in the acquisition computer. These memory locations can be accessed and modified, however, by using real-time math functions with the appropriate real-time variables.

The value of ss in low core is associated with real-time variables ssval and ssctr:
2.4 Pulse Sequence Statements: Phase and Sequence Control

- `ssval` is never modified by the acquisition computer unless specifically instructed by statements within the pulse sequence.
- `ssctr` is automatically initialized to `ssval`.

For the first increment only, if `ssval` is greater than zero, or else before every increment in an arrayed 1D or 2D experiment, `ssctr` is decremented after each steady-state transient until it reaches 0. When `ssctr` is 0, all subsequent transients are collected as data.

The value of `bs` in low core is associated with real-time variables `bsval` and `bsctr`:
- `bsval` is never modified by the acquisition computer unless specifically instructed by statements within the pulse sequence.
- `bsctr` is automatically initialized to `bsval` after each block of transients has been completed.

During the acquisition of a block of transients, `bsctr` is decremented after each transient. If `bsval` is non-zero, a zero value for `bsctr` signals that the block of transients is complete.

The ability within a pulse sequence to modify the values of these low core acquisition variables can be used to add various capabilities to pulse sequences. As an example, the following pulse sequence illustrates the cycling of pulse and receiver phases during steady-state pulses:

```c
#include <standard.h>
pulsesquence()
{
    /* Implement steady-state phase cycling */
    sub(ct, ssctr, v10);
    initval(16.0, v9);
    add(v10, v9, v10);
    /* Phase calculation statements follow, using v10 in place of ct as the starting point */
    /* Actual pulse sequence goes here */
}
```

Intertransient and Interincrement Delays

When running arrayed or multidimensional experiments (using `ni`, `ni2`, etc.), certain operations are done preceding and following the pulse sequence for every array element and every transient. These overhead operations take up time must be accounted for when running a pulse sequence. This might be especially important if the repetition time of a pulse sequence has to be maintained across every element and every scan during an arrayed or multidimensional experiment.

These overhead times between increments (array elements) and transients are deterministic (i.e., both known and constant); however, the time between increments, which we will call `x`, is longer than the time between transients, which we will call `y`. Also, the time between increments will change depending on the number of rf channels.

To maintain a constant repetition time, a parameter called `d0` (for d-zero) can be created so that `x = y + d0`. Because the interincrement overhead time will differ with different system configurations—and to keep the `d0` delay consistent across systems—if `d0` is set greater than the overhead delay, the inter-FID delay `x` is padded such that `y + d0 = x + (d0 - (x - y))`. In other words, `d0` is used to set a standard delay so the interincrement delay and the intertransient delay are the same when executing transient scans within an array element. The delay is inserted at the beginning of each scan of a FID after the first scan has

01-999379-00 A 0808

VnmrJ 2.2D User Programming 81
completed. The \( d0 \) delay can be set by the user or computed by PSG (if \( d0 \) is set to 'n'). When \( d0 \) does not exist, no delay is inserted.

Another factor to consider when keeping a consistent timing in the pulse sequence is the `status` statement. The timing of this statement varies depending on the number of channels and the type of decoupler modulation. To keep this timing constant, the pulse sequence statement `statusdelay` allows the user to set a constant delay time for changing the status. For this to work, the delay time has to be longer than the time it takes to set the status. For timing and more information, see the description of `statusdelay` in Chapter 3.

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 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.

### Controlling Pulse Sequence Graphical Display

The `dps_off`, `dps_on`, `dps_skip`, and `dps_show` statements, summarized in Table 16, can be inserted into a pulse sequence to control the graphical display of the pulse sequence statements by the `dps` command:

- Insert `dps_off()` into the sequence to turn off `dps` display of statements. All pulse sequences following `dps_off` will not be shown.
- Insert `dps_on()` into the sequence to turn on `dps` display of statements. All pulse sequences following `dps_on` will be shown.
- Insert `dps_skip()` into the sequence to skip `dps` display of the next statement. The next pulse sequence statement will not be displayed.
- Insert `dps_show(options)` statements into the pulse sequence to draw pulses for `dps` display. The pulses will appear in the graphical display of the sequence. Many options to `dps_show` are available. These options enable drawing a line to represent a delay, drawing a pulse picture and displaying the channel name below the picture, drawing shaped pulses with labels, drawing observe and decoupler pulses at the same time, and much more. Refer to Chapter 3, “Pulse Sequence Statement Reference,” for a full description of `dps_show`, including examples.

<table>
<thead>
<tr>
<th><code>dps_off()</code></th>
<th>Turn off graphical display of statements</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dps_on()</code></td>
<td>Turn on graphical display of statements</td>
</tr>
<tr>
<td><code>dps_show(options)</code>*</td>
<td>Draw delay or pulses in a sequence for graphical display</td>
</tr>
<tr>
<td><code>dps_skip()</code></td>
<td>Skip graphical display of next statement</td>
</tr>
</tbody>
</table>

* `dps_show` has many options. See Chapter 3, “Pulse Sequence Statement Reference,” for the syntax and examples of use.
2.5 Real-Time AP Tables

- “Loading AP Table Statements from Linux Text Files,” page 83
- “Table Names and Statements,” page 84
- “AP Table Notation,” page 84
- “Handling AP Tables,” page 85
- “Examples of Using AP Tables,” page 87
- “Using Internal Phase Tables,” page 87

Real-time acquisition phase (AP) tables can be created under pulse sequence control. These tables can store phase cycles, an array of attenuator values, etc. In the pulse sequence, the tables are associated with variables $t_1$, $t_2$, ..., $t_{60}$.

The following pulse sequence statements accept the table variables $t_1$ to $t_{60}$ at any place where a simple AP variable, such as $v_1$, can be used:

```plaintext
pulse             rgpulse          decpulse
decrg pulse       dec2rg pulse    dec3rg pulse
sim pulse         tx phase         dec phase
dec2 phase        dec3 phase      xmr phase
dcpl phase        dcpl2 phase     dcpl3 phase
phaseshift        spinlock        decspinlock
dec2spinlock      dec3spinlock    shaped pulse
decshaped pulse   dec2shaped pulse dec3shaped pulse
simshaped pulse   sim3shaped pulse power
pwrf
```

For example, the statement `rgpulse(pw, t1, rof1, rof2)` performs an observe transmitter pulse whose phase is specified by a particular statement in the real-time AP table $t_1$, whereas `rgpulse(pw, v1, rof1, rof2)` performs the same pulse whose phase is specified by the real-time variable $v_1$. The real-time math functions `add()`, `assign()`, etc. listed in Table 14 cannot be used with tables $t_1$-$t_{60}$. The appropriate functions to use are given in Table 17.

Statements using a table can occur anywhere in a pulse sequence except in the statements enclosed by an `ifzero-endif` pair.

Loading AP Table Statements from Linux Text Files

Table statements can be loaded from an external Linux text file with the `loadtable` statement or can be set directly within the pulse sequence with the `settable` statement. The values stored must be integral and must lie within the 16-bit integer range of $-32768$ to $32767$.

The AP table file must be placed in the user’s private directory `tablib`, which might be, for example, `/home/vnmr1/vnmrsys/tablib`, or in the system directory for table files, `/vnmr/tablib`. The software looks first in the user's personal `tablib` directory for a table of the specified name, then in the system directory. The format for the table file is quite flexible, comments are allowed, and several special notations are available.
Chapter 2. Pulse Sequence Programming

Table Names and Statements

Entries in the table file are referred to as table names. Each table name must come from the set t1 to t60 (e.g., t14 is a table name). A table name may be used only once within the table file. If a table name is used twice within the table file, an error message is displayed and pulse sequence generation (PSG) aborts.

Each table statement must be written as an integer number and separated from the next statement by some form of “white” space, such as a blank space, tab, or carriage return. The table name is separated from the table statements by an = or a += sign (the += sign is explained on page 84), and there must be a space between the table name and either of these two signs. For example, if a table file contains the table name t1 with statements 0, 1, 2, 3, 2, 3, 0, 1, it would be written as t1 = 0 1 2 3 2 3 0 1.

The index into a table can range from 0 to 1 less than the number of statements in the table. Note that an index of 0 will access the first statement in the table. Unless the autoincrement attribute (described on page 84) is imparted to the table, the index into the table is given by ct, the completed transient counter.

If the number of transients exceeds the length of the table, access to the table begins again at the beginning of the table. Thus, given a table of length n with statements numbered 0 through n–1 (this numbering is strictly a way to think about the numbering and does not imply the statements are actually numbered), then when the transient number is ct, the number of the statement of the table that will be used is ct mod n (remember that ct starts at 0 on the first transient, since ct represents the number of completed transients).

AP Table Notation

Special notation is available within the table file to simplify entering the table statements and to impart specific attributes to any table within that file:

(...)# Indicates the table segment within the parentheses is to be replicated in its entirety # times (where # ranges from 1 to 64) before preceding to any succeeding statements or segments. Do not include any space after “)”. For example: t1=(0 1 2)3 /* t1 table=012012012 */.

[...]# Indicates each statement in the table segment within square brackets is to be replicated # times (where # ranges from 1 to 64) before going to the next statement in that segment. Do not include any space after “]”. For example: t1=[0 1 2]3 /* t1 table=000111222 */.

{...}# Imparts the “divn-return” attribute to the table and indicates that the actual index into the table is to be the index divided by the number # (where # ranges from 1 to 64). # is called the divn factor and can be explicitly set within a sequence for any table (see setdivnfactor). This attribute provides a #-fold level of table compaction to the acquisition processor. The {} notation must enclose all of the table statements for a given table. This notation should not be used if this table will be subject to table operations such as ttadd (see page 86)—in this case use [ ] #, which is equivalent except for table compression. In entering the { } # notation, do not include any space after “}”.

+= Indicates that the index into the table starts at 0 for each new FID in an array or 2D experiment, is incremented after each access of the table and is therefore independent of ct. This is the autoincrement attribute, which can delimit the table name from the table statements. It can be explicitly set within a pulse sequence for any table (see setautoincrement).
The {...}# and [...]# notations are expanded by PSG at run-time and, therefore, offer no degree of table compaction to the acquisition processor. Nesting of {...} and [...] expressions is not allowed. The autoincrement += attribute can be used in conjunction with the divn-return attribute and with the {...} and [...] notations.

Multiple {...} expressions within one table are not allowed, but {...} and [...] expressions can be placed within a {...} expression.

The following examples illustrate combining the notation:

\[
\begin{align*}
\text{t2} &= [0 \ 1 \ 2 \ 3]^4 (0 \ 0 \ 2 \ 2)^4 \\
&= \{0 \ 1 \ (0 \ 2) \ 2 \ 0 \ 2 \ [3 \ 1]\}^4 \\
&= \{0 \ 1 \ 2 \ 3\}^8 \\
\text{t3} &= \{0 \ 1 \ (0 \ 2) \ 2 \ 0 \ 2 \ [3 \ 1]\}^4 \\
&= \{0 \ 1 \ 2 \ 3\}^8 \\
\text{t4} &= \{0 \ 1 \ 2 \ 3\}^8 \\
\end{align*}
\]

\[
\begin{align*}
/* \text{t2 table} = 0000111122223333002200220022 */ \\
/* \text{t3 table} = 0102020233331111 \text{ with divn-factor} = 4; \\
\text{i.e.,} \quad 000111100022220002220002222 \ldots */ \\
/* \text{t4 table with autoincrement and divn-factor} = 8 \\
\text{i.e.,} \quad 00000000111111112222222233333333 \text{ with index} \\
\text{incremented at each reference to table, not at each ct */}
\end{align*}
\]

### Handling AP Tables

Table 17 lists statements for handling AP tables.

#### Table 17. Statements for Handling AP Tables

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getelem</code></td>
<td>Retrieve an element from a AP table</td>
</tr>
<tr>
<td><code>loadtable(file)</code></td>
<td>Load AP table elements from table text file</td>
</tr>
<tr>
<td><code>setautoincrement</code></td>
<td>Set autoincrement attribute for a table</td>
</tr>
<tr>
<td><code>setdivnfactor</code></td>
<td>Set divn-return attribute and divn-factor</td>
</tr>
<tr>
<td><code>setreceiver</code></td>
<td>Associate rcvr. phase cycle with AP table</td>
</tr>
<tr>
<td><code>settable</code></td>
<td>Store array of integers in real-time AP table</td>
</tr>
<tr>
<td><code>tsadd</code></td>
<td>Add an integer to AP table elements</td>
</tr>
<tr>
<td><code>tsdiv</code></td>
<td>Divide a table into a second table</td>
</tr>
<tr>
<td><code>tsmult</code></td>
<td>Multiply an integer with AP table elements</td>
</tr>
<tr>
<td><code>tssub</code></td>
<td>Subtract an integer from AP table elements</td>
</tr>
<tr>
<td><code>ttadd</code></td>
<td>Add a table to a second table</td>
</tr>
<tr>
<td><code>ttdiv</code></td>
<td>Divide a table into a second table</td>
</tr>
<tr>
<td><code>ttmult</code></td>
<td>Multiply a table by a second table</td>
</tr>
<tr>
<td><code>ttsub</code></td>
<td>Subtract a table from a second table</td>
</tr>
<tr>
<td><code>* settable</code></td>
<td>Store array of integers in real-time AP table</td>
</tr>
<tr>
<td><code>ttadd</code></td>
<td>Add a table to a second table</td>
</tr>
<tr>
<td><code>ttdiv</code></td>
<td>Divide a table into a second table</td>
</tr>
<tr>
<td><code>ttmult</code></td>
<td>Multiply a table by a second table</td>
</tr>
<tr>
<td><code>ttsub</code></td>
<td>Subtract a table from a second table</td>
</tr>
</tbody>
</table>

The `loadtable(file)` statement loads AP table statements from table text file. file specifies the name of the table file (a UNIX text file) in the user's personal `tablib` directory or in the VnmrJ system `tablib` directory. `loadtable` can be called multiple times within a pulse sequence. Care should be taken to ensure that the same table name is not used more than once by the pulse sequence.

The `settable` (tablename, numelements, intarray) statement stores an array of integers in a real-time AP table. `tablename` specifies the name of the table (t1 to t60). numelements specifies the size of the table. intarray is a C array that contains the table elements. These elements can range from –32768 to 32767. The user must
predefine and predimension this array in the pulse sequence using C language statements prior to calling settable.

The _getelem(tablename, APindex, APdest)_ statement retrieves an element from a table. _tablename_ specifies the name of the Table (_t1_ to _t60_). _APindex_ is an AP variable (_v1_ to _v14_, _oph_, _ct_, _bsctr_, or _ssctr_) that contains the index of the desired table element. Note that the first element of a table has an index of 0. _APdest_ is also an AP variable (_v1_ to _v14_ and _oph_) into which the retrieved table element is placed. For tables for which the autoincrement feature is set, _APindex_, the second argument to _getelem_, is ignored and can be set to any AP variable name; each element in such a table is by definition always accessed sequentially.

The _setautoincrement(tablename)_ statement sets the autoincrement attribute for a table. _tablename_ specifies the name of the table (_t1_ to _t60_). The index into the table is set to 0 at the start of an FID acquisition and is incremented after each access into the table. Tables using the autoincrement feature cannot be accessed within a hardware loop.

The _setdivnfactor(tablename, divnfactor)_ statement sets the divn-return attribute and the divn-factor for a table. _tablename_ specifies the name of the table (_t1_ to _t60_). The actual index into the table is now set to (index/divnfactor). \{0 1\}2 is therefore translated by the acquisition processor, not by pulse sequence generation (PSG), into 0 0 1 1. The divn-return attribute results in a divn-factor-fold compression of the AP table at the level of the acquisition processor.

The _setreceiver(tablename)_ statement assigns the _ct_ th element of the AP 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 math statements such as _assign_, _add_, etc., the last value of _oph_ prior to the acquisition of data determines the value of the receiver phase.

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

- _tsadd(tablename, scalarval, moduloval)_
- _tssub(tablename, scalarval, moduloval)_
- _tsmult(tablename, scalarval, moduloval)_
- _tsdiv(tablename, scalarval, moduloval)_

where _tablename_ specifies the name of the table (_t1_ to _t60_) and _scalarval_ is 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). _tsdiv_ requires that _scalarval_ is not equal to 0; otherwise, an error is displayed and PSG aborts.

To perform run-time vector operations of one AP 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)_

where _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_. _ttdiv_ requires that no element in _tablenamemod_ equal 0.
## Examples of Using AP Tables

This section contains a two-pulse sequence and a homonuclear J-resolved experiment as examples of using AP tables.

### Two-Pulse Sequence

Listing 3 is the contents of the files /home/vnmr1/vnmsys/psglib/t2pul.c and /home/vnmr1/vnmsys/tablib/t2pul associated with a hypothetical two-pulse sequence T2PUL.

**Listing 3. Two-Pulse Sequence t2pul.c with Phase Tables**

```c
#include <standard.h>
pulsesquence()
{
    loadtable("t2pul");
    status(A);
    hsdelay(d1);
    status(B);
    pulse(p1,t1);
    hsdelay(d2);
    status(C);
    pulse(pw,t2);
    setreceiver(t3);
}
```

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
<td>0</td>
</tr>
<tr>
<td>t2</td>
<td>0 2 1 3</td>
</tr>
<tr>
<td>t3</td>
<td>0 2 1 3</td>
</tr>
</tbody>
</table>

Notice that t2 and t3 are identical. The pulse sequence could have used just one phase for both the observe pulse and the receiver, but using two separate phases in this way provides more flexibility for allowing run-time modification of all phases independently (e.g., a cancellation experiment can be run by changing line 2 in the tablib file to t2 = 0 or by changing line 3 to t3 = 0).

### Homonuclear J-Resolved Experiment

Listing 4 lists files /home/vnmr1/vnmsys/psglib/hom2dj.c and /home/vnmr1/vnmsys/tablib/hom2dj associated with a hypothetical homonuclear J-resolved sequence HOM2DJ.

This sequence uses “conventional” phase cycling, completely different than the pulse cycling in the standard HOM2DJ sequence found in psglib. The phase cycling, contained here in t4, is added to the phases by the pulse sequence itself with the series of three `ttadd` statements. This can also be done in the table itself, for example, by replacing the t2 line in the tablib file with \( t2 = 1 2 3 0 3 0 1 2 3 0 1 0 1 2 3 \), which is the completely “spelled out” phase cycle for the second pulse.

When using a table to be referenced with a `ttadd` statement, the table cannot be compressed by using \( t4 = \{0 2 1 3\}4 \). Square brackets, which are exactly equivalent to the curly brackets but without achieving table compression at the level of the acquisition processor must be used.

### Using Internal Phase Tables

Another use of tables is to internally declare them and convert them to `t` variables. This has the advantage of giving internal documentation and having independence of external tables.
Listing 4. Homonuclear J-Resolved Sequence hom2djt.c with Phase Tables

```c
#include <standard.h>
pulsesquence()
{
    loadtable("hom2djt");
    ttadd(t1,t4,4);
    ttadd(t2,t4,4);
    ttadd(t3,t4,4);
    status(A);
    hsdelay(d1);
    status(B);
    pulse(pw,t1);
    delay(d2/2);
    pulse(p1,t2);
    delay(d2/2);
    status(C);
    setreceiver(t3);
}
```

Listing 5. Example of Internal Phase Tables

```c
#include<standard.h>
static int phi1[2] = {3,1,3,1}, /*phase for pulse 1 */
    phi2[4]={0,0,2,2}, /*phase for pulse 2 */
    rec[4]={0,2,2,0}; /*receiver phase */
pulsesquence()
{
    ...
    /*psg statements */
    settable(t1,4,phi1); /*set t variables */
    settable(t2,4,phi2);
    settable(t3,4,rec);
    if (phase1=2) tsadd(t1,1,4); /*for states-haberkorn */
    ...
    rgpulse(pw,t1,rof1,rof1); /*phase increment for phase=2 */
    delay(d2); /*2D evolution time */
    ...
    /*psg statements */
    rgpulse(pw,t2,rof1,rof1);
    ...
    setreceiver(t3);
}
```

(which might have been modified). Listing 5 shows code for a generic phase-sensitive 2D experiment that also illustrates the use of tsadd and handles the case of phase=2.
2.6 Accessing Parameters

- “Categories of Parameters,” page 89
- “Looking Up Parameter Values,” page 96
- “Using Parameters in a Pulse Sequence,” page 96

The `getval` and `getstr` statements look up the value of parameters, providing access to parameters. Table 18 summarizes these statements.

**Table 18. Parameter Value Lookup Statements**

| `getstr(parametername, internlname)` | Look up value of string parameter |
| `internalname=getval(parametername)` | Look up value of numeric parameter |

Parameters are defined by the user in particular experiment files (exp1, exp2, etc.) in which the operation is occurring. These parameters are not the same as the parameters that are accessible to the pulse sequence during its execution, although they are at least potentially the same.

**Categories of Parameters**

Parameters can be divided into three categories:

- Parameters used in a pulse sequence exactly as in the parameter set; in other words, the name of the parameter (d1, for example) is the same in both places. Thus, a statement like `delay(d1)` is legitimate. Table 19 lists VnmrJ parameter names and corresponding pulse sequence generation (PSG) variable names and types. Table 19 is for quick reference only. For the most current listing, go to `/vnmr/psg/acqparms.h` (unity INOVA) or `/vnmr/psg/acqparms2.h` (Mercuryplus/Vx). Table 20 summarizes VnmrJ parameter names used primarily for imaging. Parameters in this category do not need to be declared as specific types (e.g., `char`) or require `getval` or `getstr`.

- Parameters used in the pulse sequence derived from those in the parameter set.

- Parameters unknown to the pulse sequence. This includes parameters created by the user for a particular pulse sequence (such as `J` or `mix`) as well as a few surprises, such as `at`, the acquisition time (the pulse sequence does not know this). The statements `getval` and `getstr` are provided for this category.

**Table 19. Global PSG Parameters**

<table>
<thead>
<tr>
<th>Acquisition</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern char il[MAXSTR] interleaved acquisition parameter, 'y', 'n'</td>
</tr>
<tr>
<td>extern double inc2D t1 dwell time in a 3D/4D experiment</td>
</tr>
<tr>
<td>extern double inc3D t2 dwell time in a 3D/4D experiment</td>
</tr>
<tr>
<td>extern double sw spectral width</td>
</tr>
<tr>
<td>extern double nf Number of FIDs in pulse sequence</td>
</tr>
<tr>
<td>extern double np Number of data points to acquire (real)</td>
</tr>
<tr>
<td>extern double nt Number of transients</td>
</tr>
<tr>
<td>extern double sfrq Observe frequency MHz</td>
</tr>
<tr>
<td>extern double dfrq Decoupler frequency MHz</td>
</tr>
<tr>
<td>extern double dfrq2 2nd decoupler frequency MHz</td>
</tr>
<tr>
<td>extern double dfrq3 3rd decoupler frequency MHz</td>
</tr>
</tbody>
</table>
### Table 19. Global PSG Parameters (continued)

<table>
<thead>
<tr>
<th>extern</th>
<th>double</th>
<th>dfreq4</th>
<th>4th decoupler frequency MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern</td>
<td>double</td>
<td>bs</td>
<td>Block size</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>tof</td>
<td>Observe transmitter offset</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>dof</td>
<td>Decoupler offset</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>dof2</td>
<td>2nd decoupler offset</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>dof3</td>
<td>3rd decoupler offset</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>dof4</td>
<td>4th decoupler offset</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>gain</td>
<td>Receiver gain value, or 'n' for autogain</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>dlp</td>
<td>Decoupler low power value</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>tpwr</td>
<td>Transmitter pulse power</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>tpwrf</td>
<td>Transmitter fine linear attenuator for pulse</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>dpwr</td>
<td>Decoupler pulse power</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>dpwrf</td>
<td>Decoupler fine linear attenuator for pulse</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>dpwr2</td>
<td>2nd decoupler fine linear attenuator</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>dpwr3</td>
<td>3rd decoupler fine linear attenuator</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>dpwr4</td>
<td>4th decoupler fine linear attenuator</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>filter</td>
<td>Pulse amp filter setting</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>xmf</td>
<td>Observe transmitter pulse width</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>dmf</td>
<td>Decoupler modulation frequency</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>dmf2</td>
<td>Decoupler modulation frequency</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dmf3</td>
<td>Decoupler modulation frequency</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dmf4</td>
<td>Filter bandwidth</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>vttemp</td>
<td>VT temperature setting</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>vtwait</td>
<td>VT temperature time-out setting</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>vtc</td>
<td>VT temperature cooling gas setting</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>cpflag</td>
<td>Phase cycling: 1=no cycling, 0=quad detect</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>dhpflag</td>
<td>Decoupler high power flag</td>
</tr>
</tbody>
</table>

**Pulse Widths**

<table>
<thead>
<tr>
<th>extern</th>
<th>double</th>
<th>pw</th>
<th>Transmitter modulation frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern</td>
<td>double</td>
<td>p1</td>
<td>A pulse width</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>pw90</td>
<td>90° pulse width</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>hst</td>
<td>Time homospoil is active</td>
</tr>
</tbody>
</table>

**Delays**

<table>
<thead>
<tr>
<th>extern</th>
<th>double</th>
<th>alfa</th>
<th>Time after receiver is turned on that acquisition begins</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern</td>
<td>double</td>
<td>beta</td>
<td>Audio filter time constant</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>d1</td>
<td>Delay</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>d2</td>
<td>An auto incremental delay, used in 2D experiments</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>d3</td>
<td>An auto incremental delay, used in 3D experiments</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>d4</td>
<td>An auto incremental delay, used in 4D experiments</td>
</tr>
</tbody>
</table>
### 2.6 Accessing Parameters

#### Table 19. Global PSG Parameters (continued)

<table>
<thead>
<tr>
<th>Extern Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern</td>
<td>pad</td>
<td>Preacquisition delay</td>
</tr>
<tr>
<td>extern</td>
<td>padactive</td>
<td>Preacquisition delay active parameter flag</td>
</tr>
<tr>
<td>extern</td>
<td>rof1</td>
<td>Amplifier unblanking delay before pulse</td>
</tr>
<tr>
<td>extern</td>
<td>rof2</td>
<td>Amplifier blanking delay</td>
</tr>
<tr>
<td>extern</td>
<td>totaltime</td>
<td>Total timer events for an experiment duration estimate</td>
</tr>
<tr>
<td>extern</td>
<td>phase1</td>
<td>Used for 2D acquisition</td>
</tr>
<tr>
<td>extern</td>
<td>phase2</td>
<td>Used for 3D acquisition</td>
</tr>
<tr>
<td>extern</td>
<td>phase3</td>
<td>Used for 4D acquisition</td>
</tr>
<tr>
<td>extern</td>
<td>d2_index</td>
<td>d2 increment (from 0 to ni-1)</td>
</tr>
<tr>
<td>extern</td>
<td>d3_index</td>
<td>d3 increment (from 0 to ni2-1)</td>
</tr>
<tr>
<td>extern</td>
<td>d4_index</td>
<td>d4 increment (from 0 to ni3-1)</td>
</tr>
<tr>
<td>extern</td>
<td>xseq</td>
<td>Digit resolution prg dec</td>
</tr>
<tr>
<td>extern</td>
<td>dseq</td>
<td>Digit resolution prg dec</td>
</tr>
<tr>
<td>extern</td>
<td>dseq2</td>
<td>Digit resolution prg dec</td>
</tr>
<tr>
<td>extern</td>
<td>dseq3</td>
<td>Digit resolution prg dec</td>
</tr>
<tr>
<td>extern</td>
<td>dseq4</td>
<td>Digit resolution prg dec</td>
</tr>
<tr>
<td>extern</td>
<td>xmsize</td>
<td>Number of characters in xm</td>
</tr>
<tr>
<td>extern</td>
<td>xmmsize</td>
<td>Number of characters in xmm</td>
</tr>
<tr>
<td>extern</td>
<td>dmsize</td>
<td>Number of characters in dm</td>
</tr>
<tr>
<td>extern</td>
<td>dmmsize</td>
<td>Number of characters in dmm</td>
</tr>
</tbody>
</table>

#### Programmable Decoupling Sequences

<table>
<thead>
<tr>
<th>Extern Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern char</td>
<td>xseq</td>
<td>Digit resolution prg dec</td>
</tr>
<tr>
<td>extern char</td>
<td>dseq</td>
<td>Digit resolution prg dec</td>
</tr>
<tr>
<td>extern char</td>
<td>dseq2</td>
<td>Digit resolution prg dec</td>
</tr>
<tr>
<td>extern char</td>
<td>dseq3</td>
<td>Digit resolution prg dec</td>
</tr>
<tr>
<td>extern char</td>
<td>dseq4</td>
<td>Digit resolution prg dec</td>
</tr>
</tbody>
</table>

#### Status Control

<table>
<thead>
<tr>
<th>Extern Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern char</td>
<td>xm</td>
<td>Transmitter status control</td>
</tr>
<tr>
<td>extern char</td>
<td>xmm</td>
<td>Transmitter modulation type control</td>
</tr>
<tr>
<td>extern char</td>
<td>dm</td>
<td>1st decoupler status control</td>
</tr>
<tr>
<td>extern char</td>
<td>dmm</td>
<td>1st decoupler modulation type control</td>
</tr>
<tr>
<td>extern char</td>
<td>dm2</td>
<td>2nd decoupler status control</td>
</tr>
<tr>
<td>extern char</td>
<td>dmm2</td>
<td>2nd decoupler modulation type control</td>
</tr>
<tr>
<td>extern char</td>
<td>dm3</td>
<td>3rd decoupler status control</td>
</tr>
<tr>
<td>extern char</td>
<td>dmm3</td>
<td>3rd decoupler modulation type control</td>
</tr>
<tr>
<td>extern char</td>
<td>dm4</td>
<td>4th decoupler status control</td>
</tr>
<tr>
<td>extern char</td>
<td>dmm4</td>
<td>4th decoupler modulation type control</td>
</tr>
<tr>
<td>extern char</td>
<td>homo</td>
<td>1st decoupler homo mode control</td>
</tr>
<tr>
<td>extern char</td>
<td>homo2</td>
<td>2nd decoupler homo mode control</td>
</tr>
<tr>
<td>extern char</td>
<td>homo3</td>
<td>3rd decoupler homo mode control</td>
</tr>
<tr>
<td>extern char</td>
<td>homo4</td>
<td>4th decoupler homo mode control</td>
</tr>
</tbody>
</table>
Chapter 2. Pulse Sequence Programming

**Table 19.** Global PSG Parameters (continued)

<table>
<thead>
<tr>
<th>extern</th>
<th>int</th>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern</td>
<td>int</td>
<td>dm2size</td>
<td>Number of characters in dm2</td>
</tr>
<tr>
<td>extern</td>
<td>int</td>
<td>dmm2size</td>
<td>Number of characters in dmm2</td>
</tr>
<tr>
<td>extern</td>
<td>int</td>
<td>dm3msize</td>
<td>Number of characters in dm3</td>
</tr>
<tr>
<td>extern</td>
<td>int</td>
<td>dmm3msize</td>
<td>Number of characters in dmm3</td>
</tr>
<tr>
<td>extern</td>
<td>int</td>
<td>dm4size</td>
<td>Number of characters in dm4</td>
</tr>
<tr>
<td>extern</td>
<td>int</td>
<td>dmm4size</td>
<td>Number of characters in dmm4</td>
</tr>
<tr>
<td>extern</td>
<td>int</td>
<td>homo2size</td>
<td>Number of characters in homo2</td>
</tr>
<tr>
<td>extern</td>
<td>int</td>
<td>homo3size</td>
<td>Number of characters in homo3</td>
</tr>
<tr>
<td>extern</td>
<td>int</td>
<td>homo4size</td>
<td>Number of characters in homo4</td>
</tr>
<tr>
<td>extern</td>
<td>int</td>
<td>hssize</td>
<td>Number of characters in hs</td>
</tr>
</tbody>
</table>

**Table 20.** Imaging and Other Variables

**RF Pulses**

<table>
<thead>
<tr>
<th>extern</th>
<th>double</th>
<th>p2</th>
<th>Pulse length</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern</td>
<td>double</td>
<td>p3</td>
<td>Pulse length</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>p4</td>
<td>Pulse length</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>p5</td>
<td>Pulse length</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>pi</td>
<td>Inversion pulse length</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>psat</td>
<td>Saturation pulse length</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>pmt</td>
<td>Magnetization transfer pulse length</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>pwx</td>
<td>X-nucleus pulse length</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>pwx2</td>
<td>X-nucleus pulse length</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>ps1</td>
<td>Spin-lock pulse length</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>pwpat [MAXSTR]</td>
<td>Pattern for pw, tpwr</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>pw1pat [MAXSTR]</td>
<td>Pattern for p1, tpwr1</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>pw2pat [MAXSTR]</td>
<td>Pattern for p2, tpwr2</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>pw3pat [MAXSTR]</td>
<td>Pattern for pw3, tpwr3</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>pw4pat [MAXSTR]</td>
<td>Pattern for pw4, tpwr4</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>pw5pat [MAXSTR]</td>
<td>Pattern for pw5, tpwr5</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>pipat [MAXSTR]</td>
<td>Pattern for pi, tpwri</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>satpat [MAXSTR]</td>
<td>Pattern for pw, tpwr</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>mtpat [MAXSTR]</td>
<td>Pattern for psat, satpat</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>ps1pat [MAXSTR]</td>
<td>Pattern for spin-lock</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>tpwr1</td>
<td>Transmitter pulse power</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>tpwr2</td>
<td>Transmitter pulse power</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>tpwr3</td>
<td>Transmitter pulse power</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>tpwr4</td>
<td>Transmitter pulse power</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>tpwr5</td>
<td>Transmitter pulse power</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>tpwri</td>
<td>Inversion pulse power</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>satpwr</td>
<td>Saturation pulse power</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>mtpwr</td>
<td>Magnetization transfer pulse power</td>
</tr>
</tbody>
</table>
### 2.6 Accessing Parameters

#### RF Decoupler Pulses

<table>
<thead>
<tr>
<th>Type</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern char</td>
<td>decpat[MAXSTR]</td>
<td>Pattern for decoupler pulse</td>
</tr>
<tr>
<td>extern char</td>
<td>decpat1[MAXSTR]</td>
<td>Pattern for decoupler pulse</td>
</tr>
<tr>
<td>extern char</td>
<td>decpat2[MAXSTR]</td>
<td>Pattern for decoupler pulse</td>
</tr>
<tr>
<td>extern char</td>
<td>decpat3[MAXSTR]</td>
<td>Pattern for decoupler pulse</td>
</tr>
<tr>
<td>extern char</td>
<td>decpat4[MAXSTR]</td>
<td>Pattern for decoupler pulse</td>
</tr>
<tr>
<td>extern char</td>
<td>decpat5[MAXSTR]</td>
<td>Pattern for decoupler pulse</td>
</tr>
<tr>
<td>extern char</td>
<td>dpwr1</td>
<td>Decoupler pulse power</td>
</tr>
<tr>
<td>extern char</td>
<td>dpwr4</td>
<td>Decoupler pulse power</td>
</tr>
<tr>
<td>extern char</td>
<td>dpwr5</td>
<td>Decoupler pulse power</td>
</tr>
</tbody>
</table>

#### Gradients

<table>
<thead>
<tr>
<th>Type</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern double</td>
<td>gro, gro2, gro3</td>
<td>Readout gradient strength</td>
</tr>
<tr>
<td>extern double</td>
<td>gpe, gpe2, gpe3</td>
<td>Phase encode for 2D, 3D, and 4D</td>
</tr>
<tr>
<td>extern double</td>
<td>gss, gss2, gss3</td>
<td>Slice-select gradients</td>
</tr>
<tr>
<td>extern double</td>
<td>gror</td>
<td>Readout focus</td>
</tr>
<tr>
<td>extern double</td>
<td>gssr</td>
<td>Slice-select refocus</td>
</tr>
<tr>
<td>extern double</td>
<td>grof</td>
<td>Readout refocus fraction</td>
</tr>
<tr>
<td>extern double</td>
<td>gssf</td>
<td>Slice-select refocus fraction</td>
</tr>
<tr>
<td>extern double</td>
<td>g0, g1, ..., g9</td>
<td>Numbered levels</td>
</tr>
<tr>
<td>extern double</td>
<td>gx, gy, gz</td>
<td>X, Y, and Z levels</td>
</tr>
<tr>
<td>extern double</td>
<td>gvox1, gvox2, gvox3</td>
<td>Voxel selection</td>
</tr>
<tr>
<td>extern double</td>
<td>gdiff</td>
<td>Diffusion encode</td>
</tr>
<tr>
<td>extern double</td>
<td>gflow</td>
<td>Flow encode</td>
</tr>
<tr>
<td>extern double</td>
<td>gspoil, gspoil2</td>
<td>Spoiler gradient levels</td>
</tr>
<tr>
<td>extern double</td>
<td>gcrush, gcrush2</td>
<td>Crusher gradient levels</td>
</tr>
<tr>
<td>extern double</td>
<td>gtrim, gtrim2</td>
<td>Trim gradient levels</td>
</tr>
<tr>
<td>extern double</td>
<td>gramp, gramp2</td>
<td>Ramp gradient levels</td>
</tr>
<tr>
<td>extern double</td>
<td>gpemult</td>
<td>Shaped phase encode multiplier</td>
</tr>
<tr>
<td>extern double</td>
<td>gradstepsz</td>
<td>Positive steps in the gradient DAC</td>
</tr>
<tr>
<td>extern double</td>
<td>gradunit</td>
<td>Dimensional conversion factor</td>
</tr>
<tr>
<td>extern double</td>
<td>gmax</td>
<td>Maximum gradient value (G/cm)</td>
</tr>
<tr>
<td>extern double</td>
<td>gxmax</td>
<td>X maximum gradient value (G/cm)</td>
</tr>
<tr>
<td>extern double</td>
<td>gymax</td>
<td>Y maximum gradient value (G/cm)</td>
</tr>
<tr>
<td>extern double</td>
<td>gzmax</td>
<td>Z maximum gradient value (G/cm)</td>
</tr>
<tr>
<td>extern double</td>
<td>gtotlimit</td>
<td>Limit combined gradient values (G/cm)</td>
</tr>
<tr>
<td>extern double</td>
<td>gxlimit</td>
<td>Safety limit for X gradient (G/cm)</td>
</tr>
<tr>
<td>extern double</td>
<td>gylimit</td>
<td>Safety limit for Y gradient (G/cm)</td>
</tr>
<tr>
<td>extern double</td>
<td>gzlimit</td>
<td>Safety limit for Z gradient (G/cm)</td>
</tr>
<tr>
<td>extern double</td>
<td>gxscale</td>
<td>X scaling factor for gmax</td>
</tr>
<tr>
<td>extern double</td>
<td>gyscale</td>
<td>Y scaling factor for gmax</td>
</tr>
</tbody>
</table>

---

Table 20. Imaging and Other Variables (continued)
### Table 20. Imaging and Other Variables (continued)

<table>
<thead>
<tr>
<th>extern</th>
<th>type</th>
<th>variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern</td>
<td>char</td>
<td>gzscale</td>
<td>Z scaling factor for gmax</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>gpatup [MAXSTR]</td>
<td>Gradient ramp-up pattern</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>gpatdown [MAXSTR]</td>
<td>Gradient ramp-down pattern</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>gropat [MAXSTR]</td>
<td>Readout gradient pattern</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>gepat [MAXSTR]</td>
<td>Phase encode gradient pattern</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>gsspat [MAXSTR]</td>
<td>Slice gradient pattern</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>gpat1 [MAXSTR]</td>
<td>General gradient pattern</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>gpat2 [MAXSTR]</td>
<td>General gradient pattern</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>gpat3 [MAXSTR]</td>
<td>General gradient pattern</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>gpat4 [MAXSTR]</td>
<td>General gradient pattern</td>
</tr>
<tr>
<td>extern</td>
<td>char</td>
<td>gpat5 [MAXSTR]</td>
<td>General gradient pattern</td>
</tr>
</tbody>
</table>

### Delays

<table>
<thead>
<tr>
<th>extern</th>
<th>type</th>
<th>variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern</td>
<td>double</td>
<td>tr</td>
<td>Repetition time per scan</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>te</td>
<td>Primary echo time</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>ti</td>
<td>Inversion time</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>tm</td>
<td>Mid-delay for STE</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>at</td>
<td>Acquisition time</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>tpe, tpe2, tpe3</td>
<td>Phase encode durations for 2D to 4D</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>tcrush</td>
<td>Crusher gradient duration</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>tdiff</td>
<td>Diffusion encode duration</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>tdelta</td>
<td>Diffusion encode duration</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>tDELT A</td>
<td>Diffusion gradient separation</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>tflow</td>
<td>Flow encode duration</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>tspoil</td>
<td>Spoiler duration</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>hold</td>
<td>Physiological trigger hold off</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>trise</td>
<td>Gradient coil rise time: sec</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>satdly</td>
<td>Saturation time</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>satfrq</td>
<td>Saturation offset</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>mtfreq</td>
<td>Magnetization transfer offset</td>
</tr>
</tbody>
</table>

### Frequencies

<table>
<thead>
<tr>
<th>extern</th>
<th>type</th>
<th>variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern</td>
<td>double</td>
<td>resto</td>
<td>Reference frequency offset</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>wsfrq</td>
<td>Water suppression offset</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>chessfrq</td>
<td>Chemical shift selection offset</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>satfrq</td>
<td>Saturation offset</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>mtfreq</td>
<td>Magnetization transfer offset</td>
</tr>
</tbody>
</table>

### Physical Sizes and Positions (for slices, voxels, and FOV)

<table>
<thead>
<tr>
<th>extern</th>
<th>type</th>
<th>variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern</td>
<td>double</td>
<td>pro</td>
<td>FOV position in readout</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>ppe, ppe2, ppe3</td>
<td>FOV position in phase encode</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>pos1, pos2, pos3</td>
<td>Voxel position</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>pss [MAXSLICE]</td>
<td>Slice position array</td>
</tr>
<tr>
<td>extern</td>
<td>double</td>
<td>lro</td>
<td>Readout FOV</td>
</tr>
</tbody>
</table>
### Table 20. Imaging and Other Variables (continued)

<table>
<thead>
<tr>
<th>Extern Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern double</td>
<td>lpe, lpe2, lpe3</td>
<td>Phase encode FOV</td>
</tr>
<tr>
<td>extern double</td>
<td>lss</td>
<td>Dimension of multislice range</td>
</tr>
<tr>
<td>extern double</td>
<td>vox1, vox2, vox3</td>
<td>Voxel size</td>
</tr>
<tr>
<td>extern double</td>
<td>thk</td>
<td>Slice or slab thickness</td>
</tr>
<tr>
<td>extern double</td>
<td>lpe, lpe2, lpe3</td>
<td>Phase encode FOV</td>
</tr>
<tr>
<td>extern double</td>
<td>fovunit</td>
<td>Dimensional conversion factor</td>
</tr>
<tr>
<td>extern double</td>
<td>thkunit</td>
<td>Dimensional conversion factor</td>
</tr>
</tbody>
</table>

#### Bandwidths

<table>
<thead>
<tr>
<th>Extern Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern double</td>
<td>sw1, sw2, sw3</td>
<td>Phase encode bandwidths / spectral widths</td>
</tr>
</tbody>
</table>

#### Counts and Flags

<table>
<thead>
<tr>
<th>Extern Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern double</td>
<td>nd</td>
<td>Experiment dimensionality</td>
</tr>
<tr>
<td>extern double</td>
<td>ns</td>
<td>Number of slices</td>
</tr>
<tr>
<td>extern double</td>
<td>ne</td>
<td>Number of echoes</td>
</tr>
<tr>
<td>extern double</td>
<td>ni</td>
<td>Number of standard increments</td>
</tr>
<tr>
<td>extern double</td>
<td>nv, nv2, nv3</td>
<td>Number phase encode views</td>
</tr>
<tr>
<td>extern double</td>
<td>ssc</td>
<td>Compressed ss transients</td>
</tr>
<tr>
<td>extern double</td>
<td>ticks</td>
<td>External trigger counter</td>
</tr>
<tr>
<td>extern char</td>
<td>ir [MAXSTR]</td>
<td>Inversion recovery flag</td>
</tr>
<tr>
<td>extern char</td>
<td>ws [MAXSTR]</td>
<td>Water suppression flag</td>
</tr>
<tr>
<td>extern char</td>
<td>mt [MAXSTR]</td>
<td>Magnetization flag</td>
</tr>
<tr>
<td>extern char</td>
<td>pilot [MAXSTR]</td>
<td>Auto gradient balance flag</td>
</tr>
<tr>
<td>extern char</td>
<td>seqcon [MAXSTR]</td>
<td>Acquisition loop control flag</td>
</tr>
<tr>
<td>extern char</td>
<td>petable [MAXSTR]</td>
<td>Name for phase encode table</td>
</tr>
<tr>
<td>extern char</td>
<td>acqtype [MAXSTR]</td>
<td>Example: “full” or “half” echo</td>
</tr>
<tr>
<td>extern char</td>
<td>exptype [MAXSTR]</td>
<td>Example: “se” or “fid” in CSI</td>
</tr>
<tr>
<td>extern char</td>
<td>apptype [MAXSTR]</td>
<td>Keyword for parameter init, e.g, “imaging”</td>
</tr>
<tr>
<td>extern char</td>
<td>seqfile [MAXSTR]</td>
<td>Pulse sequence name</td>
</tr>
<tr>
<td>extern char</td>
<td>rfspoil [MAXSTR]</td>
<td>rf spoiling flag</td>
</tr>
<tr>
<td>extern char</td>
<td>satmode [MAXSTR]</td>
<td>Presentation mode</td>
</tr>
<tr>
<td>extern char</td>
<td>verbose [MAXSTR]</td>
<td>Verbode mode for sequences and psg</td>
</tr>
</tbody>
</table>

#### Miscellaneous

<table>
<thead>
<tr>
<th>Extern Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>extern double</td>
<td>rfphase</td>
<td>rf phase shift</td>
</tr>
<tr>
<td>extern double</td>
<td>B0</td>
<td>Static magnetic field level</td>
</tr>
<tr>
<td>extern double</td>
<td>slcto</td>
<td>Slice selection offset</td>
</tr>
<tr>
<td>extern double</td>
<td>delto</td>
<td>Slice spacing frequency</td>
</tr>
<tr>
<td>extern double</td>
<td>tox</td>
<td>Transmitter offset</td>
</tr>
<tr>
<td>extern double</td>
<td>toy</td>
<td>Transmitter offset</td>
</tr>
<tr>
<td>extern double</td>
<td>toz</td>
<td>Transmitter offset</td>
</tr>
<tr>
<td>extern double</td>
<td>griserate</td>
<td>Gradient rise rate</td>
</tr>
</tbody>
</table>
Looking Up Parameter Values

The statement `internalname=getval(parametername)` allows the pulse sequence to look up the value of any numeric parameter that it otherwise does not know (`parametername`) and introduce it into the pulse sequence in the variable `internalname`. `internalname` can be any legitimate C variable name that has been defined as type `double` at the beginning of the pulse sequence (even if it is created as type `integer`). If `parametername` is not found in the current experiment parameter list, `internalname` is set to zero, and PSG produces a warning message. For example,

```c
double j;
...
j=getval("j");
```

Or simply

```c
double j=getval("j");
```

The `getstr(parametername,internalname)` statement is used to look up the value of the string parameter `parametername` in the current experiment parameter list and introduce it into the pulse sequence in the variable `internalname`. `internalname` can be any legitimate C variable name that has been defined as array of type `char` with dimension MAXSTR at the beginning of the pulse sequence. If the string parameter `parametername` is not found in the current experiment parameter list, `internalname` is set to the null string, and PSG produces a warning message. For example:

```c
char coil[MAXSTR];
...
getstr("sysgcoil",coil);
```

Using Parameters in a Pulse Sequence

As an example of using parameters in a pulse sequence, create a new pulse sequence with new variable names and have it fully functional from VnmrJ. Usually, the best way to compose a new pulse sequence is to start from a known good pulse sequence and from a known good parameter set. For many pulse sequences, `s2pul.c` in `/vnmr/psglib` and `s2pul.par` in `/vnmr/parlib` are a good place to start.

Create a new pulse sequence similar to `s2pul` but with new variable names and using a shaped pulse as follows:

1. Open a shell window.
2. Enter `cd ~/vnmrsys/psglib`.
3. Use a text editor such as `vi` to create the file `newpul.c` shown in Listing 6.
4. Save the file `newpul.c`.
5. Enter `seqgen newpul` after `newpul.c` is created.
   The following lines are displayed during pulse sequence generation:
   ```
   Beginning Pulse Sequence Generation Process...
   Adding DPS extensions to Pulse Sequence...
   Lint Check of Sequence...
   Compiling Sequence...
   Link Loading...
   Done! Pulse sequence newpul now ready to use.
   ```
6. To use the pulse sequence in VnmrJ, add new parameters starting from a known good parameter set (e.g. `s2pul.par`) by entering from the VnmrJ command line:
2.6 Accessing Parameters

Listing 6. File newpul.c for a New Pulse Sequence

/* newpul.c - new pulse sequence */
#include <standard.h>

static int ph2[4] = {0,1,2,3};

pulsesequence()
{
    double d1new, d2new, p1new, pwnew;
    char patnew[MAXSTR];
    d1new = getval("d1new");
    d2new = getval("d2new");
    p1new = getval("p1new");
    pwnew = getval("pwnew");
    getstr("patnew",patnew);
    assign(zero,v1);
    settable(t2,4,ph2);
    getelem(t2,ct,v2);

    /* equilibrium period */
    status(A);
    hsdelay(d1new);

    /* --- tau delay --- */
    status(B);
    pulse(p1new,v1);
    hsdelay(d2new);

    /* --- observe period --- */
    status(C);
    shaped_pulse(patnew,pwnew,v2,rof1,rof2);
}

s2pul
seqfill='newpul'
create('d1new','delay') d1new=1
create('d2new','delay') d2new=.001
create('p1new','pulse') p1new=0
create('pwnew','pulse') pwnew=40
create('patnew','string') patnew='square'

7. The parameters need to be saved as newpul.par in parlib so they can be easily retrieve them the next time the pulse sequence is run. Enter:
cd
cd('vnmrsys/parlib')
svp('newpul')

8. Create a macro by entering to access the new parameters and pulse sequence, for example:
macrovi('newpul')

9. In the pop-up editor window, enter the insert mode and add the line:
psgset('newpul','array','dg','d1new','d2new','p1new','pwnew','patnew')

Save the macro and exit. This macro requires the file newpul.par to be present in parlib.
Enter `newpul` in the VnmrJ command line any time the new pulse sequence is needed. Most of the pulse sequences in `/vnmr/psglib` are set up in a similar fashion, and so are easily accessible.

The `newpul.c` pulse sequence also contains examples of phase cycling. There are two basic ways to perform arbitrary user-defined phase cycling:

- Use the real-time variables `v1-v14`, `oph`, `zero`, `one`, `two`, and `three`, and perform math integer operations on them using functions in Table 14.
- Use the real-time AP tables `t1-t60`, which may be assigned either by static variable declarations and using `settable()`, or by loading in a table from `tablib` using `loadtable()` (see Table 17).

An example of using the real-time variable `v1` is given in `newpul.c` used by `assign()` and `pulse()`. An example of using real-time AP tables is given using `ph2` and `t2`. We could also replace `v2` with `t2` in the `shaped_pulse()` statement in this particular pulse sequence. In some cases, however, it is necessary to perform further integer math operations on the phase cycle, which is easier to perform on real-time variables than on AP tables, so we give the example using `getelem()` to assign the table `t2` to variable `v2`. For other examples of phase cycling calculations, see the pulse sequences in `/vnmr/psglib`.

To add 2D parameters to the `newpul.c` pulse sequence, make the following changes:

- In step 2, change `d2new` to `d2`.
- In step 4, enter `par2d set2d('newpul') p1new=40`.
- In step 7, add `par2d set2d('newpul')` to the `newpul` macro after the `psgset` line.

Also, see the `cosyps.c` pulse sequence in `/vnmr/psglib`, section 2.14 “Multidimensional NMR,” page 122, and the chapter on Multidimensional NMR in the `NMR Spectroscopy, User Guide` manual.

### 2.7 Using Interactive Parameter Adjustment

The section “Spectrometer Control,” page 60 included statements for interactive parameter adjustment (IPA). Such routines start with the letter `i` (e.g., `idelay`, `irgpulse`). For users who need added flexibility in programming, this section explains IPA and these routines in more detail. IPA is available on all systems except `MERCURYplus/-Vx`.

#### General Routines

In addition to the statements previously described, PSG has four general routines:

- `G_Pulse` for generic pulse control
- `G_Offset` for adjustment of the offset frequency
- `G_Delay` for generic delay control
- `G_Power` for fine power control.

Each of these routines is called with an argument list (see page 100) specified with attribute-value pairs, terminated by a mandatory zero. The terminating zero is mandatory. If the zero is left out, the results are unpredictable and can include a core dump of PSG.
Each attribute has a default value—a pulse can be specified simply as G_Pulse(0), which would produce a transmitter pulse of size pw with rof1 and rof2 set the same as the experiment parameters and phase cycled with the parameter oph.

The attribute SLIDER_LABEL determines whether output is generated for the Acquisition window (opened by the acqi command). If no label is specified, no IPA information is generated by the subroutine. The use of the SLIDER_LABEL with the same value for delays or pulses allows multiple delays or pulses to be controlled via one slider. This is covered later in this section.

As an example of a pulse sequence using the general routines, Listing 7 shows the source code of i2pul.c, which can be compiled and run like S2PUL, but when go('acqi') is typed, IPA information is generated in /vnmr/acqqueue/acqi.IPA.

**Listing 7. Pulse Sequence Listing of File i2pul.c**

```c
/* I2PUL - interactive two-pulse sequence */
#include <standard.h>
static int phasecycle[4]={0,2,1,3};
pulsesquence()
{
    /* equilibrium period */
    settable(t1,4,phasecycle);
    status(A);
    hsdelay(d1);
    /* --- tau delay --- */
    status(B);
    ipulse(p1,zero,"p1");
    /*
     * This ipulse statement is equivalent to
     * the following general pulse statement.
     * G_Pulse(PULSE_WIDTH, p1,
     *        PULSE_PHASE,  zero,
     *        SLIDER_LABEL, "p1",
     *        0);
     */
    G_Delay(DELAY_TIME, d2,
            SLIDER_LABEL, "d2",
            SLIDER_MAX, 10,
            0);
    /* --- observe period --- */
    status(C);
    ipulse(pw,t1,"pw");
    setreceiver(t1);
}
```

The command acqi can be used to adjust the pulses and delays in the sequence. Note that G_Pulse covers the statements obspulse, pulse, decpulse, etc.

Macro definitions have been written to cover these:

```c
#define obspulse G_Pulse(0)
#define decpulse(decpulse,phaseptr) \
        G_Pulse (PULSE_DEVICE, DODEV, \ 
                  PULSE_WIDTH, decpulse, \ 
                  PULSE_PHASE, phaseptr, \ 
                  PULSE_PRE_ROFF, 0.0, \ 
```

See the file /vmnr/psg/macros.h for a complete list. This file is automatically included when the file standard.h is included in a pulse sequence. Note also that the same pulse sequence can be used to execute go as well as go('acqi'); however, IPA information is only generated when go('acqi') is used.

Interactive adjustment of simultaneous pulses is not supported. A limit of 10 has been set on the number of calls with a label. This limits the number of parameters that can be adjusted within one pulse sequence. Note that a subroutine call within a hardware loop is still only one label.

Parameters are adjusted at the end of a sweep. Since this takes a finite amount of time, steady state may be affected. Of course, changing any parameter value also affects the steady state, so this should be of little or no consequence.

### Generic Pulse Routine

The \texttt{G\_Pulse} generic pulse routine has the following syntax:

\begin{verbatim}
G\_Pulse\( \) PULSE\_WIDTH, pw, \\
  PULSE\_PRE\_ROFF, rof1, \\
  PULSE\_POST\_ROFF, rof2, \\
  PULSE\_DEVICE, TODEV, \\
  SLIDER\_LABEL, NULL, \\
  SLIDER\_SCALE, 1, \\
  SLIDER\_MAX, 1000, \\
  SLIDER\_MIN, 0, \\
  SLIDER\_UNITS, 1e–6, \\
  PULSE\_PHASE, oph, \\
  0);
\end{verbatim}

The following table describes the attributes used with \texttt{G\_Pulse}:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PULSE_WIDTH</td>
<td>double</td>
<td>pw</td>
<td>As specified in parameter set</td>
</tr>
<tr>
<td>PULSE_PRE_ROFF</td>
<td>double</td>
<td>rof1</td>
<td>As specified in parameter set</td>
</tr>
<tr>
<td>PULSE_POST_ROFF</td>
<td>double</td>
<td>rof2</td>
<td>As specified in parameter set</td>
</tr>
<tr>
<td>PULSE_DEVICE</td>
<td>int</td>
<td>TODEV</td>
<td>TODEV for observe channel or DODEV for 1st decoupler. Also DO2DEV or DO3DEV for 2nd/3rd decoupler</td>
</tr>
<tr>
<td>SLIDER_LABEL</td>
<td>char*</td>
<td>NULL</td>
<td>Label (1-6 characters) for acqi or NULL for no output to acqi.</td>
</tr>
<tr>
<td>SLIDER_SCALE</td>
<td>int</td>
<td>1</td>
<td>Decimal places (0 to 3) on slider</td>
</tr>
<tr>
<td>SLIDER_MAX</td>
<td>int</td>
<td>100</td>
<td>Maximum value on the slider</td>
</tr>
<tr>
<td>SLIDER_MIN</td>
<td>int</td>
<td>0</td>
<td>Minimum value on the slider</td>
</tr>
<tr>
<td>SLIDER_UNITS</td>
<td>double</td>
<td>1e–6</td>
<td>Pulses are in μs, scale factor</td>
</tr>
<tr>
<td>PULSE_PHASE</td>
<td>int</td>
<td>oph</td>
<td>Real-time variable</td>
</tr>
</tbody>
</table>

Examples of using \texttt{G\_Pulse}:

\begin{verbatim}
G\_Pulse\(\) /* equals obspulse(); */
G\_Pulse\(\) PULSE\_WIDTH, pw, /* equals pulse(pw,v1); */
\end{verbatim}
### Frequency Offset Subroutine

The `G_Offset` routine adjusts the offset frequency. It has the following syntax:

```c
G_Offset(OFFSET_DEVICE, TODEV,
    OFFSET_FREQ, tof,
    SLIDER_LABEL, NULL,
    SLIDER_SCALE, 0,
    SLIDER_MAX, 1000,
    SLIDER_MIN, -1000,
    SLIDER_UNITS, 0,
    0);
```

The following table describes the attributes used with `G_Offset`:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
</table>
| OFFSET_DEVICE   | int      | none    | Device (or rf channel) to receive frequency offset. *This is required! Thus,*
|                 |          |         | `G_Offset(0)` not allowed. TODEV for transmitter channel or DODEV for first |
|                 |          |         | decoupler channel. On UNITYplus, D02DEV for 2nd decoupler channel, or D03DEV |
|                 |          |         | for 3rd decoupler channel.                                                 |
| OFFSET_FREQ     | double   | *       | Offset frequency for selected channel. Default is offset frequency parameter |
|                 |          |         | `(tof, dof, dof2, dof3)` of associated channel.                             |
| SLIDER_LABEL    | char     | NULL    | If no slider label selected, offset cannot be changed in acqi. Otherwise,  |
|                 |          |         | becomes the label (1-6 characters) in acqi.                               |
| SLIDER_SCALE    | int      | 0       | Number of decimal places displayed in acqi. Default is 0 because default  |
|                 |          |         | range is 2000 Hz, so a resolution finer than 1 Hz is not necessary.      |
| SLIDER_MAX      | int      | *       | Maximum value on the slider. Default is 1000 Hz more than the offset  |
|                 |          |         | frequency.                                                                 |
| SLIDER_MIN      | int      | *       | Minimum value on the slider. Default is 1000 Hz less than the offset  |
|                 |          |         | frequency.                                                                 |
| SLIDER_UNITS    | double   | 1.0     | Frequencies are in Hz.                                                    |

* Default value is described in the description column for this attribute.

Examples of using `G_Offset`:

```c
G_Offset(OFFSET_DEVICE, TODEV, /* equivalent to */
    OFFSET_FREQ, tof, /* offset(tof,TODEV); */
    0);                /* required terminating zero */
```

```c
G_Offset(OFFSET_DEVICE, TODEV, /* basic interactive */
    OFFSET_FREQ, tof, /* offset statement */
    SLIDER_LABEL, "TOF",/* for fine adjustment of */
    0);                /* transmitter frequency */
```
Generic Delay Routine

The G_Delay generic delay routine has the following syntax:

```
G_Delay(DELAY_TIME, d1,
    SLIDER_LABEL, NULL,
    SLIDER_SCALE, 1,
    SLIDER_MAX, 60,
    SLIDER_MIN, 0,
    SLIDER_UNITS, 1.0,
    0);
```

The following table describes the attributes used with G_Delay:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELAY_TIME</td>
<td>double</td>
<td>d1</td>
<td>As specified in parameter set.</td>
</tr>
<tr>
<td>SLIDER_LABEL</td>
<td>char *</td>
<td>NULL</td>
<td>Label (1 to 6 characters) for acqi or NULL for no output to acqi.</td>
</tr>
<tr>
<td>SLIDER_SCALE</td>
<td>int</td>
<td>1</td>
<td>Decimal places (0 to 3) displayed.</td>
</tr>
<tr>
<td>SLIDER_MAX</td>
<td>int</td>
<td>60</td>
<td>Maximum value on the slider.</td>
</tr>
<tr>
<td>SLIDER_MIN</td>
<td>int</td>
<td>0</td>
<td>Minimum value on the slider.</td>
</tr>
<tr>
<td>SLIDER_UNITS</td>
<td>double</td>
<td>1.0</td>
<td>Delays are in seconds.</td>
</tr>
</tbody>
</table>

Examples of using G_Delay:

```
G_Delay(0); /* equals delay(d1); */

G_Delay(DELAY_TIME, d2, /* equals delay(d2); */
0); /* required terminating zero */
```

IPA allows one slider to control more than one delay or pulse. The maximum number of delays or pulses a slider can control is 32. This multiple control is obtained whenever multiple calls to G_Pulse or G_Delay have the same value for the SLIDER_LABEL attribute.

The first call to G_Pulse in a pulse sequence sets the initial value, the maximum and minimum of the slider, and the scale. Later calls to G_Pulse within that pulse sequence do not alter these. The SLIDER_UNITS attribute are unique to each call to G_Pulse. This allows changing the value seen by a particular event by some multiplication factor. For example, the following two statements create a single slider in the Acquisition window (opened by the acqi command) labeled PW that will control two separate pulses.

```
G_Pulse(PULSE_DEVICE, TODEV,
    PULSE_WIDTH, pw,
    SLIDER_LABEL, "PW",
    SLIDER_SCALE, 1,
    SLIDER_MAX, 1000,
    SLIDER_MIN, 0,
    SLIDER_UNITS, 1.0e-6,
    0);

G_Pulse(PULSE_DEVICE, TODEV,
    PULSE_WIDTH, pw*2.0,
    SLIDER_LABEL, "PW",
    SLIDER_UNITS, 2.0e-6,
    0);
```
The width of the first pulse will initially be $pw$, as set by the `PULSE_WIDTH` attribute for the first `G_Pulse` call. The width of the second pulse will initially be $pw \times 2.0$, as set by the `PULSE_WIDTH` attribute for the second `G_Pulse` call.

When the slider is changed in `acqi`, the amount that the actual pulse width changes is determined by the product of the slider change and the respective multiplicative factors specified by the attribute `SLIDER_UNITS`. For example, if the slider increased by 3 units, the first pulse width would be increased by $3 \times 1.0 \times 10^{-6}$ seconds and the second pulse would be increased by $3 \times 2.0 \times 10^{-6}$ seconds. In this way, the initial 1 to 2 ratio in pulse widths is maintained while the slider is changed.

**Fine Power Subroutine**

The `G_Power` subroutine is used on systems with the optional linear fine attenuators. It has the following syntax:

```c
G_Power(Power_VALUE, tpwr, 
POWER_DEVICE, TODEV, 
SLIDER_LABEL, NULL, 
SLIDER_SCALE, 1, 
SLIDER_MAX, 4095, 
SLIDER_MIN, 0, 
SLIDER_UNITS, 1.0, 
0);
```

The following table describes the attributes used with `G_Power`:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POWER_VALUE</td>
<td>double</td>
<td>tpwr</td>
<td>As specified in parameter set.</td>
</tr>
<tr>
<td>POWER_DEVICE</td>
<td>int</td>
<td>TODEV</td>
<td>TODEV for transmitter channel or DODEV for decoupler channel. On UNITYplus also D02DEV and D03DEV for 2nd and 3rd decoupler channels.</td>
</tr>
<tr>
<td>SLIDER_LABEL</td>
<td>char *</td>
<td>NULL</td>
<td>Label (1 to 6 characters) for acqi or NULL for no output to acqi.</td>
</tr>
<tr>
<td>SLIDER_SCALE</td>
<td>int</td>
<td>1</td>
<td>Decimal places (0 to 3) on slider.</td>
</tr>
<tr>
<td>SLIDER_MAX</td>
<td>int</td>
<td>4095</td>
<td>Maximum value on the slider.</td>
</tr>
<tr>
<td>SLIDER_MIN</td>
<td>int</td>
<td>0</td>
<td>Minimum value on the slider.</td>
</tr>
<tr>
<td>SLIDER_UNITS</td>
<td>double</td>
<td>1.0</td>
<td>Power in arbitrary units.</td>
</tr>
</tbody>
</table>

Examples of using `G_Power`:

```
G_Power(0);
```

```
G_Power(Power_VALUE, dpwr, 
POWER_DEVICE, DODEV, 
0); /* required terminating zero */
```

2.8 Hardware Looping and Explicit Acquisition

- “Receiver Phase For Explicit Acquisitions,” page 107
- “Multiple FID Acquisition,” page 107
Chapter 2. Pulse Sequence Programming

The loop and endloop statements described previously generate a soft loop, which means that they force the acquisition computer to repeatedly place the information contained within the loop into the pulse program buffer (a FIFO). If this loop must run extremely fast, a condition may arise in which the acquisition computer is not able to provide input to the pulse program buffer as fast as the sequence is required to operate, and this technique does not work.

Because of this problem, a different mode of looping known as hardware looping is supported in current systems. In this mode, the pulse program buffer provides its own looping, and the speed can be at the maximum possible rate, with the only limitation being the number of events that can occur during each repetition of the loop. Table 21 lists statements related to hardware looping.

Table 21. Hardware Looping Related Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire(num_points, sampling_interval)</td>
<td>Explicitly acquire data</td>
</tr>
<tr>
<td>clearapdatatable()</td>
<td>Zero data in acquisition processor memory</td>
</tr>
<tr>
<td>endhardloop()</td>
<td>End hardware loop</td>
</tr>
<tr>
<td>starthardloop(num_repetitions)</td>
<td>Start hardware loop</td>
</tr>
</tbody>
</table>

Controlling Hardware Looping

Use the starthardloop(numrepetitions) and endhardloop() statements start and end a hardware loop. The numrepetitions argument to starthardloop must be a real-time integer variable, such as v2, and not a regular integer, a real number, or a variable. The number of repetitions of the hardware loop must be two or more. If the number of repetitions is 1, the hardware looping feature itself is not activated. A hardware loop with a count equal to 0 is not permitted and will generate an error. Depending on the pulse sequence, additional code may be needed to trap for this condition and skip the starthardloop and endhardloop statements if the count is 0.

Only instructions that require no further intervention by the acquisition computer (pulses, delays, acquires, and other scattered instructions) are allowed in a hard loop. Most notably, no real-time math statements are allowed, thereby precluding any phase cycle calculations. Also, no AP table with the autoincrement feature set can be used within a hard loop.

The number of events included in the hard loop, including the total number of data points if acquisition is performed, must be as follows:

2048 or less for the MERCURYplus/-Vx STM/Output board, or Data Acquisition Controller board.

In all cases, the number of events must be greater than 1. No nesting of hard loops is allowed.

For MERCURYplus/-Vx STM/Output boards, Data Acquisition Controller boards, there are no timing restrictions between multiple, back-to-back hard loops. There is one subtle restriction placed on the actual duration of a hard loop if back-to-back hard loops are encountered: the duration of the $i$th hard loop must be $N(i+1) * 0.4$ ms, where $N(i+1)$ is the number of events occurring in the $(i+1)$th hard loop.

Number of Events in Hardware Loops

An event is a single activation of the timing circuitry. Pulses, delays, phase shifts, etc., set or reset various gate lines to turn on and off pulses, phase shift lines, etc. but activate the timing circuitry in the same way. Timing is accomplished as follows:

- The Data Acquisition Controller board uses one time base of 12.5 ns.
2.8 Hardware Looping and Explicit Acquisition

- MERCURYplus/-Vx systems use two time bases: 0.1 μs and 1 ms. As many events as needed are used. Delays greater than 96 seconds use a hard loop.

Therefore, larger timer words may produce multiple events. The final point to understand is that some things that look like one event may actually be more. Consider, for example, the statement \( \text{rgpulse}(\text{pw}, \text{v1}, \text{rof1}, \text{rof2}) \). Does this generate a single event? No, it generates at least three (or more depending on the length of the events). That is because we generate first a time of \( \text{rof1} \) with the amplifier unblanked but transmitter off, then a time of \( \text{pw} \) with the transmitter on, and then a time \( \text{rof2} \) with the transmitter off but the amplifier unblanked. Times that are zero generate no events, however. For example, \( \text{rgpulse}(5.0 \times 10^{-6}, \text{v1}, 0.0, 0.0) \) generates only a single event.

Although pulses, delays, and data point acquisitions are the most common things to be in a hardware loop, other choices are possible. Table 22 lists the number of events that may be generated by each statement.

On MERCURYplus/-Vx systems, any delay (pulse, delay, decrgpulse, etc.) is limited to 96 seconds within a hardware loop. In practice, this is not a restriction.

Explicit Acquisition

Closely related to hardware looping is the explicit acquisition feature—the acquisition of one or more pairs of data points explicitly by the pulse sequence. This feature enables interspersing of pulses and data acquisition and allows coding pulse sequences that acquire multiple FIDs during the course of a pulse sequence (such as COCONOSY). It also allows pulse sequences that acquire a single FID one or more points at a time (such as MREV-type sequences).

The \( \text{acquire(number_points, sampling_interval)} \) statement explicitly acquires data points at the specified sampling interval, where the sequence of events is acquire a pair of points for 200 ns, delay for \( \text{sampling_interval} \) less 200 ns, then repeat \( \text{number_points}/2 \) times. For example, acquiring an FID would use \( \text{acquire(np, 1.0/sw)} \).

Both arguments to the \( \text{acquire} \) statement must be real numbers or variables. The number of complex points to be acquired must be a multiple of 2 for Data Acquisition Controller, and STM/Output boards. Inside a hardware loop, Data Acquisition Controller and STM/Output boards can accept a maximum of 2048 complex points, \( \text{number_points} \) must be a multiple of 2, because only pairs of points can be acquired.

NMR spectrometer systems include small overhead delays before and after the \( \text{acquire} \) statement. The pre-\( \text{acquire} \) delay takes into account setting the receiver phase (oph) and enabling data overflow detection. Disabling data overflow detection creates a post-\( \text{acquire} \) delay. These overhead delays and associated functions are placed outside the loop when \( \text{acquire} \) statements are within a loop, and before the first \( \text{acquire} \) and after the last \( \text{acquire} \), when more than one \( \text{acquire} \) statement is used to acquire a FID.

Once an explicit acquisition is invoked, even if for one pair of data points, the standard “implicit” acquisition is turned off, and the user is responsible for acquiring the full number of data points. Failure to acquire the correct number of data points before the end of the pulse sequence generates an error. The total number of data points acquired before the end of the sequence must equal the specified number (np). An example of the programming necessary to program a simple explicit acquisition, analogous to the normal implicit acquisition, would look like this:

\[
\text{rcvron()};
\text{txphase(zero)};
\]
decphase(zero);
delay(alfa);
acquire(np, 1.0/sw);

Although generally not needed, the `clearapdatatable()` statement is available to zero the acquired data table at times other than at the start of the execution of a pulse sequence, when the data table is automatically zeroed.

The limitation that multiple hardloops cannot be nested has consequences for the use of the `acquire` statement inside a hardloop. Depending on its arguments and how it is built into the...
2.8 Hardware Looping and Explicit Acquisition

A pulse sequence, the acquire statement may internally be done as a hardloop by itself. However, a construct like the following does not work:

```vnmrj
initval(np/2.0, v14);
starthardloop(v14);
    acquire(2.0, 1.0/sw);
endhardloop();
```

A loop that consists of a single acquire call is not permitted, but such constructs are not needed because a single statement can be used instead:

```vnmrj
acquire(np,1.0/sw);
```

This statement is not equivalent to the first construct because the acquire statement will sample more than just two points (i.e., a complex data point) per loop cycle, thus allowing for np greater than \(2.0 \times \text{maximum number of loop cycles}\). Note that the loop uses a 16-bit loop counter. Therefore, the maximum number of cycles is 32767 (the largest possible 16-bit number).

On the other hand, a hardloop that contains acquire together with other pulse sequence events works fine as long as the number of complex points to be acquired plus the number of extra FIFO words per loop cycle does not exceed the total number of words in the loop FIFO:

```vnmrj
initval(np/2.0, v14);
loop(v14);
    acquire(2.0, 1.0/sw - (rof1 + pw + rof2));
    rgpulse(pw, v1, rof1, rofr2);
endloop;
```

Explicit hardloops with acquire calls are a standard feature in multipulse solids sequences.

### Receiver Phase For Explicit Acquisitions

Receiver phase can be changed for explicit acquisitions, the same as for implicit acquisitions, by changing oph or by using the setreceiver statement. The value of oph at the time of the acquisition of the first data point is the value that determines the receiver phase setting for the duration of that particular “scan”—the receiver cannot be changed after acquiring some data points and before acquiring the rest.

### Multiple FID Acquisition

Explicit acquisition of data can also be used to acquire more than one FID per pulse sequence (simultaneous COSY-NOESY for example). This can be done for 1D or 2D experiments. The parameter nf, for number of FIDs, controls this if it is created and set. To perform such an experiment, enter `create('nf','integer')` to create nf and then set nf equal to an integer such as 2 and a second new parameter cf (current FID).

Once the data have been acquired cf (current FID) is used to identify the FID to manipulate. Setting cf=2, for example, recognizes the second FID in the COSY-NOESY experiment (and produces a NOESY spectrum after Fourier transformation). Note that this is distinct from the standard array capability and is compatible with the standard arrays. The acquisition is an array of ten experiments, with each consisting of three FIDs that are generated during each pulse sequence. To display the second FID of the seventh experiment, for example, type `cf=2 dfid(7)`.
2.9 Pulse Sequence Synchronization

- “External Time Base,” page 108
- “Controlling Rotor Synchronization,” page 108

A pulse sequence is just a set of accurately timed delays that turn the hardware on and off.

External Time Base

An external timebase halts the pulse sequence until the number of external events in the count field have occurred for purposes of synchronization. The source of events or ticks of this external timebase is up to the user. This feature is not available on MERCURYplus/-Vx systems.

Controlling Rotor Synchronization

Statements for rotor control on NMR systems with solids rotor synchronization hardware are rotorperiod, roto르sync, and xgate. Table 23 summarizes these statements.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rotorperiod(period)</td>
<td>Obtain rotor period of high-speed rotor</td>
</tr>
<tr>
<td>roto르sync(rotations)</td>
<td>Gated pulse sequence delay from MAS rotor position</td>
</tr>
<tr>
<td>xgate(events)</td>
<td>Gate pulse sequence from an external event</td>
</tr>
</tbody>
</table>

- Use rotorperiod(period) to obtain the rotor period, where period is a real-time variable into which is the rotor period is placed (e.g., rotorperiod(v5)). The period is placed into the referenced variable as an integer in units of 50 ns.
- Use roto르sync(rotations) to insert a variable-length delay, where rotations is a real-time variable that points to the number of rotations to delay, for example, roto르sync(v6). The delay allows synchronizing the execution of the pulse sequence with a particular orientation of the sample rotor. When the roto르sync statement is encountered, the pulse sequence is stopped until the number of rotor rotations has occurred as referenced by the real-time variable given.
- Use xgate(events) to halt the pulse sequence from an external event, where events is a double variable (e.g., xgate(2.0)). When the number of external events has occurred, the pulse sequence continues.

Both roto르sync and xgate can be used, but there is a very important distinction between the two—roto르sync synchronizes to the exact position of the rotor, whereas xgate synchronizes to the zero degree position of rotation. For example, if the rotor is at 90°, then for xgate(1.0), the pulse sequence will begin when the rotor is at zero degrees, a rotation of 270°; however, for the equivalent roto르sync, the pulse sequence will begin when the rotor is at 90°, or 360° rotation.

2.10 Pulse Shaping

- “File Specifications,” page 109
- “Performing Shaped Pulses,” page 112
- “Programmable Transmitter Control,” page 114
- “Setting Spin Lock Waveform Control,” page 115
RF pulse shapes can be executed on one or more rf channels, along with programmed decoupling patterns, and gradient shapes for imaging applications. *MERCURYplus/-V* shapes are Dante style pulses. Shaped decoupling is not possible on *MERCURYplus/-Vx* systems. For pulse shaping programming using Pbox, see the manual *NMR Spectroscopy, User Guide*. All VnmrS system waveforms must have 4μsec resolution.

Pulse control of the waveform generators consists of two separate parts:

- A text file describing the shape of a waveform.
- A pulse sequence statement applying that waveform in an appropriate manner.

The power of rf shape or decoupler pattern is controlled by the standard power and fine power control statements for that rf channel. For example, `obspower` and `obspwr.rf` will scale the overall power of a shape on the observe channel. *MERCURYplus/-Vx* uses only coarse power.

## File Specifications

The macro `sh2pul` sets up a shaped two-pulse (SH2PUL) experiment. This sequence behaves like the standard two-pulse sequence S2PUL except that the normal hard pulses are changed into shaped pulses from the waveform generator.

To find pulse shape definitions, the pulse sequence generation (PSG) software looks in a user's `vnmr.sys/shapelib` directory and then in the system's `shapelib`. Each `shapelib` directory contains files specifying the defined shapes for rf pulses, decoupling, and gradient waveforms. To differentiate the files in a `shapelib` directory, each type uses a different suffix, Table 24:

<table>
<thead>
<tr>
<th>Pattern Type</th>
<th>Suffix</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>rf pulses</td>
<td>.RF</td>
<td>gauss.RF</td>
</tr>
<tr>
<td>decoupling</td>
<td>.DEC</td>
<td>mlev16.DEC</td>
</tr>
<tr>
<td>gradient</td>
<td>.GRD</td>
<td>hard.GRD</td>
</tr>
</tbody>
</table>

Each pattern file is a set of element specifications with one element per line. Therefore, a 67 element pattern contains 67 lines. Any blank lines and comments (characters after a # sign on a line) are ignored.

Shapes can be created by macro, by programs, or by hand. The specifications for each kind of pattern are listed in the following table (if a field is not specified, the default given is used). As an example, a slightly modified excerpt from a file in the system directory `shapelib` is also shown.
RF Patterns

For example, the first 8 elements (after the comment lines) of the file sinc.RF:

0.000 0.000 1.000000
0.000 8.000 1.000000
0.000 16.000 1.000000
0.000 24.000 1.000000
0.000 32.000 1.000000
0.000 40.000 1.000000
0.000 48.000 1.000000
0.000 56.000 1.000000

In using the .RF patterns, the actual values for the amplitude are treated as relative values, not as absolute values. All of the amplitudes in the rf shape file are divided by the largest amplitude in the shape file and then multiplied by 4095.0. The net result is that shapes with values of the amplitudes between 0 to 10.0, or between 0 to 4095.0, or between 0 to 100000.0, are effectively all the same shape.

To implement .RF patterns with absolute values for amplitudes, use a shape element with 0 duration to fix the scaling factor for the shape. Here is a simple example:

A shape with elements

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>10.0</td>
</tr>
<tr>
<td>0.00</td>
<td>100.0</td>
</tr>
<tr>
<td>0.00</td>
<td>20.0</td>
</tr>
</tbody>
</table>

will result in an actual shape of

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>4095.0*10.0/100.0</td>
</tr>
<tr>
<td>0.00</td>
<td>4095.0*100.0/100.0</td>
</tr>
<tr>
<td>0.00</td>
<td>4095.0*20.0/100.0</td>
</tr>
</tbody>
</table>

A shape with elements

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>4095.0</td>
</tr>
<tr>
<td>0.00</td>
<td>10.0</td>
</tr>
<tr>
<td>0.00</td>
<td>100.0</td>
</tr>
<tr>
<td>0.00</td>
<td>20.0</td>
</tr>
</tbody>
</table>

will result in an actual shape of

<table>
<thead>
<tr>
<th>Element</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>4095.0*10.0/4095.0</td>
</tr>
<tr>
<td>0.00</td>
<td>4095.0*100.0/4095.0</td>
</tr>
<tr>
<td>0.00</td>
<td>4095.0*20.0/4095.0</td>
</tr>
</tbody>
</table>
Decoupler Patterns

Table 26. Decoupler Patterns

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Limits</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Tip angle per element (in degrees)</td>
<td>0° to 500°, 1° resolution</td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td>Phase limits</td>
<td>No limit on magnitude</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>RF phase (in degrees)</td>
<td>0.043° resolution</td>
<td>Required</td>
</tr>
<tr>
<td>3</td>
<td>Amplitude</td>
<td>0 to scalable max</td>
<td>max</td>
</tr>
<tr>
<td>4</td>
<td>Transmitter gate</td>
<td>0, 1</td>
<td>0 (gate off)</td>
</tr>
</tbody>
</table>

For example, the first 8 elements (after the comment lines) of the file waltz16.DEC:

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

In using the gate field in .DEC patterns, note the following:

- The waveform controller gate is ORed with the controller gate. This means that any time the controller gate is on, the transmitter is on, irrespective of any waveform generator gate.

- If a decoupler pattern is activated under status control (using dmm='p'), an implicit controller gate statement is added. In this situation, any 0s or 1s in the gate field of the .DEC pattern are irrelevant because they are overridden (as indicated above).

- If a decoupler pattern is activated by the decprgon statement, the unity INOVA waveform generator gate is the controlling factor. If this gate is specified as 0s or 1s in the .DEC file, that gating will occur. If there is no gate field in the .DEC file, the default occurs—the gate is set to 0 and the decoupler is off. An alternate is to follow the decprgon statement with some kind of gate statement (e.g., decon) to turn on the controller gate (overriding the default of the gate set to 0 from the controller) and to proceed the decprgoff statement with a statement to turn the gate off (for example, decoff).

Gradient Patterns

Table 27. Gradient Patterns

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
<th>Limits</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Output amplitude</td>
<td>−32767 to 32767, 1 unit resolution</td>
<td>Required</td>
</tr>
<tr>
<td>2</td>
<td>Relative duration</td>
<td>1 to 255</td>
<td>1</td>
</tr>
</tbody>
</table>

For example, the first 8 elements (after the comment lines) of the file trap.GRD:

1024 1
2048 1
3072 1
4096 1
5120 1
Performing Shaped Pulses

Statements to perform shaped pulses are listed in Table 28.

Table 28. Shaped Pulse Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>decshaped_pulse*</td>
<td>Perform shaped pulse on first decoupler</td>
</tr>
<tr>
<td>dec2shaped_pulse*</td>
<td>Perform shaped pulse on second decoupler</td>
</tr>
<tr>
<td>dec3shaped_pulse*</td>
<td>Perform shaped pulse on third decoupler</td>
</tr>
<tr>
<td>shaped_pulse*</td>
<td>Perform shaped pulse on observe transmitter</td>
</tr>
<tr>
<td>simshaped_pulse*</td>
<td>Perform simultaneous two-pulse shaped pulse</td>
</tr>
<tr>
<td>sim3shaped_pulse*</td>
<td>Perform a simultaneous three-pulse shaped pulse</td>
</tr>
</tbody>
</table>

* 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, obsshape, decphase, RG1, RG2)
  sim3shaped_pulse(obsshape, decshape, dec2shape, obswidth, decwidth, dec2width, obsshape, decphase, dec2phase, RG1, RG2)

Shaped Pulse on Observe Transmitter or Decouplers

Use shaped_pulse(shape, width, phase, RG1, RG2) to perform a shaped pulse on the observe transmitter where shape is the name of a text file in shapelib that stores the rf pattern (leave off the .RF file extension), width is the duration of the pulse; phase is the phase of the pulse (it must be a real-time variable); RG1 is the delay between unblanking the amplifier and gating on the transmitter (the phase shift occurs at the beginning of this delay); and RG2 is the delay between gating off the transmitter and blanking the amplifier (e.g., shaped_pulse("gauss", pw, v1, rof1, rof2)).

The statements shaped_pulse, decshaped_pulse, and dec2shaped_pulse provide pulse shaping through the linear attenuator and the small-angle phase shifter on the AP bus if a rf channel does not have a waveform generator. AP tables for the attenuation and phase values are created on the fly, and the real-time variables v12 and v13 are used to control the execution of the shape. This pulse shaping through the AP bus was exclusively controlled by the statements apshaped_pulse, apshaped_decpulse, and apshaped_dec2pulse on previous versions of VNMR.

For shaped pulses, the minimum pulse length is:

- 0.2 μs on UNITY/INOVA waveform generator control.

The overhead at the beginning and end of the shaped pulse is:

- UNITY/INOVA: starts at 0.95 μs falling to 0 at the end.
- Acquisition Controller board systems: starts at 10.75 μs falling to 4.3 μs at the end.
- Output board systems: starts at 10.95 μs falling to 4.5 μs at the end.

If the length is less than 0.2 μs on UNITY/INOVA the pulse is not executed and there is no overhead.

The decshaped_pulse, dec2shaped_pulse, and dec3shaped_pulse statements allow a shaped pulse to be performed on the first, second, and third decoupler,
respectively. The arguments and overhead used for each is the same as \texttt{shaped\_pulse}, except they apply to the decoupler controlled by the statement.

\textbf{Simultaneous Two-Pulse Shaped Pulse}

\texttt{simshaped\_pulse (obsshape, decshape, obswidth, decwidth, obsphase, decphase, RG1, RG2)} performs a simultaneous, two-pulse shaped pulse on the observe transmitter and the first decoupler under waveform generator control.

- \texttt{obsshape} is the name of the text file that contains the rf pattern to be executed on the observe transmitter;
- \texttt{decshape} is the name of the text file that contains the rf pattern to be executed on the first decoupler;
- \texttt{obswidth} is the duration of the pulse on the observe transmitter;
- \texttt{decwidth} is the duration of the pulse on the first decoupler;
- \texttt{obsphase} is the phase of the pulse on the observe transmitter (it must be a real-time variable);
- \texttt{decphase} is the phase of the pulse on the first decoupler (it must be a real-time variable);
- \texttt{RG1} is the delay between unblanking the amplifier and gating on the first rf transmitter (all phase shifts occur at the beginning of this delay); and
- \texttt{RG2} is the delay between gating off the final rf transmitter and blanking the amplifier; for example:

\begin{verbatim}
  simshaped\_pulse("gauss","hrm180",pw,p1,v2,v5,rof1,rof2)
\end{verbatim}

The overhead at the beginning and end of the simultaneous two-pulse shaped pulse is:

- \texttt{UNITY INOVA}: starts at 1.45 \(\mu\)s falling to 0 at the end.
- Acquisition Controller board systems: starts at 21.5 \(\mu\)s falling to 8.6 \(\mu\)s at the end.
- Output board systems: starts at 21.7 \(\mu\)s falling to 8.8 \(\mu\)s at the end.

These values hold regardless of the values for \texttt{obswidth} and \texttt{decwidth}.

If either \texttt{obswidth} or \texttt{decwidth} is 0.0, no pulse occurs on the corresponding channel. If both \texttt{obswidth} and \texttt{decwidth} are non-zero and either \texttt{obsshape} or \texttt{decshape} is set to the null string (""), then a pulse occurs on the channel with the null shape name. If either the pulse width is zero or the shape name is the null string, then a waveform generator is not required on that channel.

\textbf{Simultaneous Three-Pulse Shaped Pulse}

The \texttt{sim3shaped\_pulse} statement performs a simultaneous, three-pulse shaped pulse under waveform control on three independent rf channels. The arguments to \texttt{sim3shaped\_pulse} are the same as defined previously for \texttt{simshaped\_pulse}, except that \texttt{dec2shape} is the name of the text file that contains the rf pattern to be executed on the second decoupler, \texttt{dec2width} is the duration of the pulse on the second decoupler, and \texttt{dec2phase} is the phase (a real-time variable) of the pulse on the second decoupler (e.g.,

\begin{verbatim}
  sim3shaped\_pulse("gauss","hrm180","sinc",pw,p1,v2,v5,v6,rof1,rof2))
\end{verbatim}

The overhead at the beginning and end of the simultaneous three-pulse shaped pulse is:

- \texttt{UNITY INOVA}: starts at 1.95 \(\mu\)s falling to 0 at the end.
- Acquisition Controller board systems: starts at 32.25 \(\mu\)s falling to 12.9 \(\mu\)s at the end.
- Output board systems: starts at 32.45 \(\mu\)s falling to 13.1 \(\mu\)s at the end.

These values hold regardless of the values for \texttt{obswidth}, \texttt{decwidth}, and \texttt{dec2width}.

Setting one of the pulse lengths to the value 0.0, \texttt{sim3shaped\_pulse} also performs a simultaneous two-pulse shaped pulse on any combination of three rf channels. (e.g., to perform simultaneous shaped pulses on the first decoupler and second decoupler, but not the observe transmitter, set the \texttt{obswidth} argument to 0.0).
If any of the shape names are set to the null string (''), a normal rectangular pulse occurs on the channel with the null shape name. If either the pulse width is zero or the shape name is the null string, a waveform generator is not required on that channel.

**Programmable Transmitter Control**

Statements related to programmable transmitter control are listed in Table 29.

<table>
<thead>
<tr>
<th>Programmable Control Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>decprgoff()</td>
</tr>
<tr>
<td>dec2prgoff()</td>
</tr>
<tr>
<td>dec3prgoff()</td>
</tr>
<tr>
<td>decprgon*</td>
</tr>
<tr>
<td>dec2prgon*</td>
</tr>
<tr>
<td>dec3prgon*</td>
</tr>
<tr>
<td>obsprgoff()</td>
</tr>
<tr>
<td>obsprgon*</td>
</tr>
</tbody>
</table>

* decprgon(name,90_pulselength,tipangle_resoln)
  dec2prgon(name,90_pulselength,tipangle_resoln)
  dec3prgon(name,90_pulselength,tipangle_resoln)
  obsprgon(name,90_pulselength,tipangle_resoln)

**Programmable Control of Observe Transmitter**

Use `obsprgon(name, 90_pulselength, tipangle_resoln)` to set programmable phase and amplitude control of the observe transmitter. `name` is the name of the file in `shapelib` that stores the decoupling pattern, `90_pulselength` is the pulse duration for a 90° tip angle, and `tipangle_resoln` is the resolution in tip-angle degrees to which the decoupling pattern is stored in the waveform generator (e.g., `obsprgon("waltz16", pw90, 90.0)`).

The `obsprgon` statement returns the number of 50-ns ticks (as an integer value) in one cycle of the decoupling pattern. Explicit gating of the observe transmitter with `xmtrown` and `xmtroff` is generally required.

To terminate any programmable phase and amplitude control on the observe transmitter under waveform control, use `obsprgoff()`.

**Programmable Control of Decouplers**

The `decprgon`, `dec2prgon`, and `dec3prgon` statements set programming decoupling on the first, second, and third decouplers, respectively. The arguments for each statement are the same as `obsprgon`, except they apply to the decoupler controlled by the statement. Each statement returns the number of 50 ns ticks (as an integer value) in one cycle of the decoupling pattern. Similarly, explicit gating of the selected decoupler is generally required, and termination of the control is done by the `decprgoff()`, `dec2prgoff()`, and `dec3prgoff()` statements, respectively.

Arguments to `obsprgon`, `decprgon`, `dec2prgon`, and `dec3prgon` can be variables (which need the appropriate `getval` and `getstr` statements) to permit changes via parameters.

The macro `pwsadj(shape_file,pulse_parameter)` adjusts the pulse interval time so that the pulse interval for the shape specified by `shape_file` (a file from `shapelib`) is an integral multiple of 100 ns. This eliminates a time truncation error in the
execution of the shaped pulse by the programmable pulse modulators.
pulse_parameter is a string containing the adjusted pulse interval time.

Setting Spin Lock Waveform Control

Statements for spin lock control on systems with optional waveform generators are listed in Table 30.

Table 30. Spin Lock Control Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>decspinlock*</td>
<td>Set spin lock waveform control on first decoupler</td>
</tr>
<tr>
<td>dec2spinlock*</td>
<td>Set spin lock waveform control on second decoupler</td>
</tr>
<tr>
<td>dec3spinlock*</td>
<td>Set spin lock waveform control on third decoupler</td>
</tr>
<tr>
<td><em>spinlock</em></td>
<td>Set spin lock waveform control on observe transmitter</td>
</tr>
</tbody>
</table>

* decspinlock(name, 90_pulselength, tipangle_resoln, phase, ncycles)
dec2spinlock(name, 90_pulselength, tipangle_resoln, phase, ncycles)
dec3spinlock(name, 90_pulselength, tipangle_resoln, phase, ncycles)
spinlock(name, 90_pulselength, tipangle_resoln, phase, ncycles)

Spin Lock Waveform Control on Observe Transmitter

To execute a waveform-controlled spin lock on the observe transmitter, use
spinlock(name, 90_pulselength, tipangle_resoln, phase, ncycles),
name is the name of the file in shapelib that stores the decoupling pattern (leave off the .DEC file extension); 90_pulselength is the pulse duration for a 90° tip angle;
tipangle_resoln is the resolution in tip-angle degrees to which the decoupling pattern is stored in the waveform generator; phase is the phase angle of the spin lock (it must be a real-time variable); and ncycles is the number of times that the spin-lock pattern is to be executed (e.g., spinlock('mlev16', pw90, 90.0, v1, ncyc)).

Both rf gating and the mixing delay are handled within this statement.

Spin Lock Waveform Control on Decouplers

The decspinlock, dec2spinlock, and dec3spinlock set spin lock waveform control on the first, second, and third decouplers, respectively. The arguments are the same as used with spinlock, except that 90_pulselength is the pulse duration for a 90° tip angle on the decoupler controlled by the statement.

Arguments to spinlock, decspinlock, dec2spinlock, and dec3spinlock can be variables (which would need the appropriate getval and getstr statements) to permit changes via parameters.

Shaped Pulse Calibration

Macros bandinfo and pulseinfo can be run interactively (without arguments) to give a table with shaped pulse information for calibration. bandinfo takes the name of any legal shape and the bandwidth desired for the pulse and gives a table containing the duration of that pulse and a predicted 90° pulse power setting. pulseinfo takes the name of the shape and the duration of the pulse and gives the bandwidth of that pulse and a predicted 90° pulse power setting. Both macros can also be called from another macro. For more information, refer to the Command and Parameter Reference.
2.11 Shaped Pulses Using Attenuators

- “Controlling Shaped Pulses Using Attenuators,” page 117
- “Controlling Attenuation,” page 118

UNITY INOVA and MERCURYplus/-Vx systems are equipped with computer-controlled attenuators (0 dB to 79 dB on UNITY INOVA, or 0 dB to 63 dB on MERCURYplus/-Vx) on the observe and decouple channels, linear amplifiers, and T/R (transmit/receive) switch preamplifiers that allow low-level transmitter signals to be generated and pass unperturbed into the probe. The combination of these elements means that the capability for performing shaped pulse experiments is inherent in the systems and does not require the more sophisticated waveform generation capability of the optional waveform generators.

Hardware differences must be considered between systems, with and without the waveform generators. The attenuators have more limited dynamic range, slower switching time, and fewer pulse programming steps available. Nonetheless, the capability still allows significant experiments using only attenuators.

Three issues affect all shaped pulses, but particularly attenuator-based pulses:

- **Number of steps** – The more steps used, the closer the shape approximates a continuous shape. At what level does this become overkill? For the most common shape, Gaussian, as few as 19 steps have been shown to be completely acceptable.

- **Dynamic range** – How much dynamic range is required within a shape for proper results. For a Gaussian shape it has been shown that 33 dB is a useful limit; little or no improvement is achieved with more. With a single 63-dB attenuator, then, a Gaussian pulse with 33 dB dynamic range can be superimposed on a level ranging from 0- to 30-dB, more with a 79-dB attenuator.

- **Overall power level of the shape** – A Gaussian pulse has an effective power approximately 8 dB lower than a rectangular pulse with an identical peak power. This means that given a full-power rectangular pulse of, say, 25 kHz, a Gaussian pulse with the same peak power has approximately a 10 kHz strength. Using instead a Gaussian pulse with only 33 dB dynamic range and a peak power 30 dB lower results in a shaped pulse of approximately 312 Hz, which is useful for some applications, like exciting the NH region of a spectrum, but too strong for others.

To increase the dynamic range (and decrease the strength of the shaped pulse) further, we can use one of three approaches:

- Replace the 63-dB attenuator with a 79-dB unit. This adds 16 dB of dynamic range, producing shaped pulses in the range of 50 Hz, suitable for multiplet excitation.

- Add an additional 63-dB attenuator in series with the first. If you use the entire 63 dB of the second attenuator to control the level of the pulse and use the first attenuator only for the shape, you still produce a pulse whose power is (for a Gaussian) 71 dB (63 + 8) below that of the hard pulse. This would produce a 7 Hz pulse, about as weak a pulse as one ever needs (and which could be reduced 30 dB further by only using 33 dB of the first attenuator for the shape). It is possible to use this control to create shaped pulses without a waveform generator.

- Use a time-sharing or “DANTE” approach, applying the shaped pulse in such a way that it is switched on and off with a particular duty cycle during the course of the shape. A 10% duty cycle, for example, reduces the power by a factor of ten.

Both the phase and linear attenuator on each transmitter can be controlled through pulse sequence statements (see pwrf, obspwrf, decpwrf, dec2pwrf, dec3pwrf, pwrn, rlpwrn, and dclplphase) so it is possible to create shaped pulses without a waveform generator.
2.11 Shaped Pulses Using Attenuators

AP Bus Delay Constants

Table 31 lists the most important AP bus delay “constants” (C macros). The list is incomplete, but a complete list can be found at the bottom of the text file /vmnr/psg/apdelay.h.

The constants OFFSET_DELAY and OFFSET_LTCH_DELAY are applicable only to systems that use PTS synthesizers with latching on the input. Although the constants are identical, use only OFFSET_DELAY on these systems.

Table 31. AP Bus Delay Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Indicates Duration of</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACQUIRE_START_DELAY*</td>
<td>Overhead at start of acquisition</td>
</tr>
<tr>
<td>ACQUIRE_STOP_DELAY*</td>
<td>Overhead at end of acquisition</td>
</tr>
<tr>
<td>DECMODFREQ_DELAY</td>
<td>Overhead for setting modulator frequency</td>
</tr>
<tr>
<td>GRADIENT_DELAY</td>
<td>rgradient, zgradpulse (two times)</td>
</tr>
<tr>
<td>OBLIQUEGRADIENT_DELAY</td>
<td>oblique_gradient (applicable only to imaging)</td>
</tr>
<tr>
<td>OFFSET_DELAY**</td>
<td>decoffset, dec2offset, obsoffset, offset</td>
</tr>
<tr>
<td>OFFSET_LTCH_DELAY***</td>
<td>decoffset, dec2offset, obsoffset, offset</td>
</tr>
<tr>
<td>POWER_DELAY</td>
<td>decpower, dec2power, obspower, power, rlpower, etc.</td>
</tr>
<tr>
<td>PRG_OFFSET_DELAY</td>
<td>Time shift of WFG output with obsprgon, etc.</td>
</tr>
<tr>
<td>PRG_START_DELAY</td>
<td>decprgon, dec2prgon, obsprgon, etc.</td>
</tr>
<tr>
<td>PRG_STOP_DELAY</td>
<td>decprgoff, dec2prgoff, obsprgoff, etc.</td>
</tr>
<tr>
<td>PWRF_DELAY</td>
<td>decpwrf, dec2pwrf, obspwrf, pwrf</td>
</tr>
<tr>
<td>SAPS_DELAY</td>
<td>dcplrphase, dcplr2phase, dcplr3phase, xmtrphase</td>
</tr>
<tr>
<td>SEFDECMOD_DELAY</td>
<td>Overhead for setting modulator mode</td>
</tr>
<tr>
<td>SPNLCK_START_DELAY</td>
<td>Overhead at start of decspinlock, spinlock, etc.</td>
</tr>
<tr>
<td>SPNLCK_STOP_DELAY</td>
<td>Overhead at end of decspinlock, spinlock, etc.</td>
</tr>
<tr>
<td>VAGRADIENT_DELAY</td>
<td>vagradpulse (two times)</td>
</tr>
<tr>
<td>WFG_OFFSET_DELAY</td>
<td>Time shift of WFG output</td>
</tr>
<tr>
<td>WFG_START_DELAY</td>
<td>Overhead at start of decshaped_pulse, shaped_pulse</td>
</tr>
<tr>
<td>WFG_STOP_DELAY</td>
<td>Overhead at end of decshaped_pulse, shaped_pulse</td>
</tr>
<tr>
<td>WFG2_START_DELAY</td>
<td>Overhead at start of simshaped_pulse, etc.</td>
</tr>
<tr>
<td>WFG2_STOP_DELAY</td>
<td>Overhead at end of simshaped_pulse, etc.</td>
</tr>
<tr>
<td>WFG3_START_DELAY</td>
<td>Overhead at start of sim3shaped_pulse, etc.</td>
</tr>
<tr>
<td>WFG3_STOP_DELAY</td>
<td>Overhead at end of sim3shaped_pulse, etc.</td>
</tr>
</tbody>
</table>

* On UNITY/INOVA systems; on other systems, this constant is zero (no support for FSQ).
** Use OFFSET_DELAY only on UNITY/INOVA systems.
*** Only on systems that use PTS synthesizers with latching.

Controlling Shaped Pulses Using Attenuators

The statements power, obspower, decpower, dec2power, dec3power, and (optionally) pwrf, obspwrf, decpwrf, dec2pwrf, dec3pwrf, pwr, and rlpower
are used to change the attenuation (and hence the power level) of either the transmitter or decouplers. A pulse sequence in which one of these statements is placed in a loop and repeatedly executed with different values for the amount of attenuation therefore results in a shaped pulse. This can be a C loop or a “soft” loop (using the loop statement), but not a “hard” loop. The successive values for the power may be calculated in real-time, read from a table (assuming that only positive numbers are involved), or set up from a static C variable. Although no standard pulse sequences exist that implement this feature, several contributions to the user library provide excellent examples of how to do this.

The statements shaped_pulse, decshaped_pulse, and dec2shaped_pulse provide fine-grained “waveform generator-type” pulse shaping through the AP bus. If an rf channel does not have a waveform generator configured, this is the same type of pulse shaping that statements apshaped_pulse, apshaped_decpulse, and apshaped_dec2pulse provide, and is a simpler implementation.

The apshaped_pulse, apshaped_decpulse, and apshaped_dec2pulse pulse statements use table variables to define the amplitude and phase tables, whereas the standard shaped_pulse, decshaped_pulse, and dec2shaped_pulse statements create and use these tables on the fly. Both types of AP bus waveshaping statements use real-time variables v12 and v13 to control shape execution. Table 32 summarizes the statements described in this section.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>apshaped_decpulse*</td>
<td>First decoupler pulse shaping via the AP bus</td>
</tr>
<tr>
<td>apshaped_dec2pulse*</td>
<td>Second decoupler pulse shaping via the AP bus</td>
</tr>
<tr>
<td>apshaped_pulse*</td>
<td>Observe transmitter pulse shaping via the AP bus</td>
</tr>
<tr>
<td>decshaped_pulse*</td>
<td>Perform shaped pulse on first decoupler</td>
</tr>
<tr>
<td>dec2shaped_pulse*</td>
<td>Perform shaped pulse on second decoupler</td>
</tr>
<tr>
<td>shaped_pulse*</td>
<td>Perform shaped pulse on observe transmitter</td>
</tr>
</tbody>
</table>

* apshaped_decpulse(shape,pulse_width,pulse_phase, power_table,phase_table,RG1,RG2)
  apshaped_decpulse(shape,pulse_width,pulse_phase, power_table,phase_table,RG1,RG2)
  apshaped_pulse(shape,pulse_width,pulse_phase,power_table, phase_table,RG1,RG2)
  decshaped_pulse(shape,width,phase,RG1,RG2)
  dec2shaped_pulse(shape,width,phase,RG1,RG2)
  dec3shaped_pulse(shape,width,phase,RG1,RG2)
  shaped_pulse(shape,width,phase,RG1,RG2)

**MERCU plus/-Vx** systems support the shaped_pulse and decshaped_pulse. However, shapes are created using DANTE style pulses, not using a waveform generator. Furthermore, the apshaped_pulse is supported. However, only power level is controlled, not phase, which makes gauss.RF the only usable shape.

### Controlling Attenuation

On systems with two attenuators, connect the two existing attenuators in series, leaving one channel without computer-controlled attenuation. This is often acceptable in homonuclear experiments, while in heteronuclear experiments and some homonuclear experiments it may be desirable to insert a simple fixed attenuator in-line in the channel that isn’t being shaped.

If you take this approach, the tpwr and dpwr parameters (or, equivalently, the power (...OBSch) and power (...DECch) pulse sequence statements) control the two attenuators. The simplest approach is to use one of the two attenuators to control the shape,
while using the second to set the overall level of the pulse. Assuming that there are also hard pulses in the pulse sequence, you’ll also need to remember to write your pulse sequence to return both attenuators to values suitable for the hard pulse.

### 2.12 Internal Hardware Delays

- “Delays from Changing Attenuation,” page 119
- “Delays from Changing Status,” page 120
- “Waveform Generator High-Speed Line Trigger,” page 121

Many pulse sequence statements result in “hidden” delays. These delays are not intrinsic to pulse sequence generation (PSG) software but are rather internal to the hardware.

Each AP bus instruction is considered a FIFO event and incurs the following delay, which is the time it takes to set the hardware on the AP bus:

- On \textit{UNITY} \textit{INOVA}, 0.5-\mu s delay (except PFG, which has a 1.0-\mu s delay).
- On \textit{MERCURYplus/-Vx}, 1.2 \mu s delay.

#### Delays from Changing Attenuation

The pulse sequence statement \texttt{power}, which is used to change the level of attenuation produced by a 63-dB rf attenuator in the system, leads to the following values:

- On \textit{UNITY} \textit{INOVA}, 1 AP bus instruction, 0.5-\mu s concomitant internal delay (WFG start takes 1 AP bus instructions at 0.5 \mu s and extra board delay of 0.75 \mu s, total 1.25 \mu s).
- On \textit{MERCURYplus/-Vx}, 4 AP bus instructions, 4.8-\mu s concomitant internal delay.

\textbf{Table 33} lists all pulse sequence statements that lead to an internal delay and the magnitude of this delay. Similar information to the table is contained in the PSG header file \texttt{apdelay.h}, which resides in the VnmrJ system PSG directory.

\begin{center}
\textbf{Table 33.} AP Bus Overhead Delays
\end{center}

\begin{tabular}{llll}
\hline
\textbf{Pulse Sequence Statements} & \textbf{UNITY} \textit{INOVA} & \textbf{MERCURYplus/-Vx} & \textbf{Output Board} \\
\hline
\texttt{acquire} & 1.0 pre & — & — \\
 & 0.5 post & — & — \\
\texttt{xmtrphase} & 0.5 & 7.2 & 2.35 \\
\texttt{dcphase} & & & \\
\texttt{dclrphase} & 0.5 & 4.8 & 4.5 \\
\texttt{dcl2r2phase} & & & \\
\texttt{dcl3r3phase} & 0.5 & 4.8 & 4.5 \\
\texttt{power, obspower} & & & \\
\texttt{decpower} & & & \\
\texttt{dec2power} & & & \\
\texttt{dec3power} & & & \\
\texttt{pwrf, obspwrf} & & & \\
\texttt{decpwrf} & & & \\
\texttt{dec2pwrf} & & & \\
\texttt{dec3pwrf} & 0.5 & — & — \\
\texttt{offset} (S=standard L=latching) & 4.0 & 86.4 & 15.25 S \\
 & & & 21.7 L \\
\hline
\end{tabular}
Chapter 2. Pulse Sequence Programming

Table 33. AP Bus Overhead Delays

<table>
<thead>
<tr>
<th>Pulse Sequence Statements</th>
<th>UNITY</th>
<th>INOVA</th>
<th>MERCURYplus/-Vx</th>
<th>Output Board</th>
</tr>
</thead>
<tbody>
<tr>
<td>shaped_pulse</td>
<td>1.25</td>
<td>pre</td>
<td>—</td>
<td>15.45</td>
</tr>
<tr>
<td>decshaped_pulse</td>
<td>0.5</td>
<td>post</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>dec2shaped_pulse</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dec3shaped_pulse</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>simshaped_pulse</td>
<td>*</td>
<td></td>
<td>30.50</td>
<td></td>
</tr>
<tr>
<td>sim3shaped_pulse</td>
<td>**</td>
<td></td>
<td>45.55</td>
<td></td>
</tr>
<tr>
<td>obsprgon</td>
<td>1.25</td>
<td></td>
<td>—</td>
<td>10.95</td>
</tr>
<tr>
<td>decprgon</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dec2prgon</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dec3prgon</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>obsprgoff</td>
<td>0.5</td>
<td></td>
<td>4.5</td>
<td></td>
</tr>
<tr>
<td>decprgoff</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dec2prgoff</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dec3prgoff</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>spinlock</td>
<td>1.25</td>
<td>pre</td>
<td>—</td>
<td>15.45</td>
</tr>
<tr>
<td>decspinlock</td>
<td>0.5</td>
<td>post</td>
<td>—</td>
<td></td>
</tr>
<tr>
<td>dec2spinlock</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dec3spinlock</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rgradient and vgradient with gradtype='p'</td>
<td>4.0</td>
<td></td>
<td>—</td>
<td>Not an option</td>
</tr>
<tr>
<td>rgradient and vgradient with gradtype='w'</td>
<td>0.5</td>
<td></td>
<td>—</td>
<td>Not an option</td>
</tr>
<tr>
<td>zgradpulse</td>
<td>delay</td>
<td></td>
<td>—</td>
<td>Not an option</td>
</tr>
<tr>
<td>gradtype='p'</td>
<td>+ 8.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>zgradpulse</td>
<td>delay</td>
<td></td>
<td>—</td>
<td>Not an option</td>
</tr>
<tr>
<td>gradtype='w'</td>
<td>+ 1.0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* simshaped_pulse: 1.75 pre, 0.5 post
** sim3shaped_pulse: 2.25 pre, 0.5 post

On systems with the Output board, Table 33 indicates that the pulse sequence statement power incurs a 4.5 μs internal delay, not a 4.3 μs delay as previously stated. Of the 4.5 μs delay, 0.2 μs is to allow any high-speed line, (for example, the transmitter gate control line) that has been turned off in PSG at the end of the preceding delay to actually turn off in hardware before the AP bus instructions have been issued from the FIFO. Otherwise, any such high-speed line would not be turned off in hardware until the end of the series of AP bus instructions. This extra 0.2 μs delay can be avoided with the apovrride statement.

Delays from Changing Status

Other delays can be incurred with the status and setstatus statements. The first occurrence of the status statement always incurs the full delay. On subsequent occurrences of status, the delay depends on values of the parameters dmm, dmm2, and dmm3. There are three parts that contribute to this delay:

- Modulation mode – If the modulation mode changes, 1.0 μs is added to the delay, and the first occurrence of 's' in the dm string (or dmm2 or dmm3) adds an extra 1.0 μs. On systems with apinterface=3, if and only if the modulation mode changes. Note that the waveform generator (mode 'p') needs CW modulation (mode 'c').
• **Waveform generator** – Starting a waveform generator adds 1.25 μs. Stopping a waveform generator adds 1 μs on UNITY/INOVA and 4.3 μsec on other systems. (The modulation mode is to or from ‘p’.). The waveform generator also has an offset or propagation delay, which is discussed on page 121.

• **Modulation frequency** – If the modulation frequency changes, 1 μs is added on UNITY/INOVA and 6.45 μsec on other systems. Note that for the UNITY/INOVA, this is different for a shaped pulse. The modulation frequency can change if the statement setstatus is called with a modulation frequency different from the parameter corresponding to the transmitter set, or if the modulation mode changes to or from ‘g’ and ‘r’. If the change is to ‘g’ and ‘r’, the modulation frequency is internally scaled, changing the frequency.

Finally, these delays are added up for each channel, and this becomes the delay incurred for status or setstatus. For example, if dm='nnns', dmm='cpfwp', and dm2='y', then dmm2='ccccpc', Table 34 summarizes the internal intervals, assuming status(A) is the initial state.

### Table 34. Example of AP Bus Overhead Delays for status Statement

<table>
<thead>
<tr>
<th>Statement</th>
<th>Delay (μs)</th>
<th>Delay (μs) apinterface=3</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>status(B)</td>
<td>0</td>
<td>0</td>
<td>cmm from ‘c’ to ‘p’, WFG not started because dm='n' in B</td>
</tr>
<tr>
<td>status(C)</td>
<td>1.0</td>
<td>4.3</td>
<td>cmm from ‘p’ to ‘f’, no WFG to stop</td>
</tr>
<tr>
<td>status(D)</td>
<td>1.0+1.25</td>
<td>4.3+10.75</td>
<td>cmm from ‘f’ to ‘w’, synchronize, cmm2 from ‘c’ to ‘p’</td>
</tr>
<tr>
<td>status(E)</td>
<td>1.75+0.5</td>
<td>15.05+4.3</td>
<td>cmm from ‘w’ to ‘p’ (= ‘c’) and start WFG, cmm2 from ‘p’ to ‘c’, only stop WFG</td>
</tr>
</tbody>
</table>

To keep the status timing constant, use the statusdelay statement. This statement allows the user to specify a defined period of time for the status statement to execute. For example, if statusdelay(‘B’, 2.0e-5) is used, as long as the time it takes to execute status for state B is less than 20 microseconds, the statement will always take 20 microseconds. If the time to execute state B is greater than 20 microseconds, the statement still executes, but a warning message is generated.

### Waveform Generator High-Speed Line Trigger

Along with the AP bus overhead delay, the waveform generator has an offset delay as a result of high-speed line (WFG) propagation delay. This shifts the rf pattern beyond the AP bus delay. Figure 3 illustrates the delay. The time overhead for the AP bus is 1.25 μs (this includes a 0.5-μs AP bus delay and a 0.75-μs board delay). The offset delay is an additional 0.45 μs, for a total delay of 1.70 μs. The WFG also has a post pulse overhead delay.

Note that if the shaped pulse is followed by a delay, say d3, then the end of the delay is at 1.7+pshape+0.5+d3. To obtain the proper offset delay, available in apdelay.h are macros WFG_OFFSET_DELAY, WFG2_OFFSET_DELAY, and WFG3_OFFSET_DELAY.

At the end of data collection, 3.5 ms is inserted to give the acquisition computer time to check lock, temperature, spin, etc. The system has a 0.004-ms delay at the start of a transient to initialize the data collection hardware, and a 2.006-ms delay at the end of a transient for data collection error detection. For systems with gradients, the end of scan delays do not include the times to turn off gradients, which is done at the end of every scan.
2.13 Indirect Detection on Fixed-Frequency Channel

Indirect detection experiments, in which the observe nucleus is $^1$H and the decouple nucleus is a low-frequency nucleus, usually $^{13}$C, are easily done on systems with two broadband channels. Systems with a fixed-frequency decoupler depend on the type of system.

**Fixed-Frequency Decoupler**

A UNITY\textsubscript{NOVA} system with the label Type of RF set to U+ H1 Only in the CONFIG window, or any MERCURY\textsubscript{plus/-Vx} broadband system, can use the same parameter sets and pulse sequences as a dual-broadband system (e.g., HMQC) as long as the pulse statements in a sequence do not use the channel identifiers \texttt{TODEV}, \texttt{DODEV}, \texttt{DO2DEV}, and \texttt{DO3DEV}. This restriction is negligible because statements \texttt{obspower}, \texttt{decpower}, \texttt{dec2power}, and \texttt{dec3power} are available that specify an rf channel without requiring the these channel identifiers. Each of these statements require only the power level and can be remapped to different rf channels. The \texttt{rfchannel} parameter enables remapping rf channel selection. Refer to the description of \texttt{rfchannel} in the *Command and Parameter Reference* for details.

MERCURY\textsubscript{plus/-Vx} support automatic channel swapping as well.

2.14 Multidimensional NMR

- “Hypercomplex 2D,” page 123
- “Real Mode Phased 2D: TPPI,” page 124

A standard feature of all pulse sequences is the ability to array acquisition parameters and automatically acquire an array of the corresponding FIDs. For example, arraying the \texttt{pw} parameter and viewing the resulting array of spectra is one way to estimate the 90-degree pulse width. This explicit array feature is automatic, whenever a parameter is set to multiple values, such as \texttt{pw=5, 6, 7, 8, 9, 10}.

A separate type of arrayed variable is used for 2D, 3D, and 4D experiments. The distinguishing feature of this type of data set is that the arrayed element has a uniform, automatically calculated increment between values. The \texttt{ni} parameter is set to the number
of increments desired in the first indirect dimension of a multidimensional data set. The inverse of the parameter $sw_1$ defines the increment in successive values of the implicitly arrayed delay $d_2$. For example, if $ni=8$, an implicit $d_2$ array with values $d_2=0, 1/sw_1, 2/sw_1, 3/sw_1, 4/sw_1, 5/sw_1, 6/sw_1, 7/sw_1$ is generated. Eight FIDs, each using the corresponding $d_2$ delay, will be acquired.

For the second indirect dimension, the analogous parameters are $ni_2$, $sw_2$, and $d_3$. For the third indirect dimension, the analogous parameters are $ni_3$, $sw_3$, and $d_4$.

When creating a new 2D pulse sequence in standard form, the pulse sequence should contain a $d_2$ delay. To create the appropriate parameters, use the $par2d$ macro. It is usually convenient to call $par2d$ from within the macro used to set up the pulse sequence and to set the parameters to appropriate values with the $set2d$ macro. Examples of 2D pulse sequences are given in the standard software in `/vnmr/psglib` and `/vnmr/maclib`.

When creating a new 3D pulse sequence in standard form, the pulse sequence should contain the delays $d_2$ and $d_3$, and parameters can be created with the $par3d$ macro. Similarly, a 4D pulse sequence should contain the delays $d_2$, $d_3$, and $d_4$, with parameters created by the $par4d$ macro.

Each indirect dimension of data can be acquired in a phase-sensitive mode. Examples of this include the hypercomplex method and the TPPI method (see the chapter on multidimensional NMR in *NMR Spectroscopy, User Guide* manual for more details).

For each indirect dimension, a phase parameter selects the type of acquisition. For the first indirect dimension, the corresponding phase parameter is $phase$. For the second indirect dimension, the parameter is $phase2$. For the third indirect dimension, the parameter is $phase3$. The total number of FIDs in a given multidimensional data set is stored in the parameter $arraydim$. For a 2D experiment, $arraydim$ is equal to $ni*(number of elements of the phase parameter)$.

When programming the multidimensional pulse sequences, it is convenient to have access to the current increment in a particular indirect dimension, and to know what the phase element is. Table 35 lists these PSG variables (see Table 19 for the full list of Vnmr parameters and their corresponding PSG variable names and types).

Some pulse sequences, such as heteronuclear 2D-J (HET2DJ), can be used “as is” for phase-sensitive 2D NMR; however, the hypercomplex and TPPI experiments require more information compared to “normal” pulse sequences, see below.

**Hypercomplex 2D**

Hypercomplex 2D (States, Haberkorn, Ruben) requires only that a pulse sequence be run using an arrayed parameter that generates the two required experiments. While this can be any parameter, for consistency we recommend the use of a parameter $phase$, which can be set by the user to 0 (to give a non-phase-sensitive experiment) or to an array (as in $phase=1,2$) to generate the two desired experiments. The parameter $phase$ is automatically made available to a pulse sequence as the integer $phase1$. Typical code as part of the pulse sequence might look like this:

```c
pulsesquence()
{
    if (phase1==0)
    { /* Phase calculation for */
        ...
        /* non-phase-sensitive experiment */
    }
    else if (phase1==1)
```
Chapter 2. Pulse Sequence Programming

Table 35. Multidimensional PSG Variables

<table>
<thead>
<tr>
<th>PSG Variable</th>
<th>PSG type</th>
<th>VnmrJ parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>d2_index</td>
<td>int</td>
<td>0 to (ni-1)</td>
<td>Current index of the d2 array</td>
</tr>
<tr>
<td>id2</td>
<td>real-time</td>
<td>0 to (ni-1)</td>
<td>Current real-time index of the d2 array</td>
</tr>
<tr>
<td>inc2D</td>
<td>double</td>
<td>1.0/sw1</td>
<td>Dwell time for first indirect dimension</td>
</tr>
<tr>
<td>phase1</td>
<td>int</td>
<td>phase</td>
<td>Acquisition mode for first indirect dimension</td>
</tr>
<tr>
<td>d3_index</td>
<td>int</td>
<td>0 to (ni-2-1)</td>
<td>Current index of the d3 array</td>
</tr>
<tr>
<td>id3</td>
<td>real-time</td>
<td>0 to (ni-2-1)</td>
<td>Current real-time index of the d3 array</td>
</tr>
<tr>
<td>inc3D</td>
<td>double</td>
<td>1.0/sw2</td>
<td>Dwell time for second indirect dimension</td>
</tr>
<tr>
<td>phase2</td>
<td>int</td>
<td>phase2</td>
<td>Acquisition mode for second indirect dimension</td>
</tr>
<tr>
<td>d4_index</td>
<td>int</td>
<td>0 to (ni-3-1)</td>
<td>Current index of the d4 array</td>
</tr>
<tr>
<td>id4</td>
<td>real-time</td>
<td>0 to (ni-3-1)</td>
<td>Current real-time index of the d4 array</td>
</tr>
<tr>
<td>inc4D</td>
<td>double</td>
<td>1.0/sw3</td>
<td>Dwell time for third indirect dimension</td>
</tr>
<tr>
<td>phase3</td>
<td>int</td>
<td>phase3</td>
<td>Acquisition mode for third indirect dimension</td>
</tr>
<tr>
<td>ix</td>
<td>int</td>
<td>1 to arraydim</td>
<td>Current element of an arrayed experiment</td>
</tr>
</tbody>
</table>

```c
{
    /* Phase calculation for */
    ...
    /* first of two arrays */
}
else if (phase1==2)
    {
        /* Phase calculation for */
        ...
        /* second of two arrays */
    }
}
```

This code usually can be condensed because the phases are obviously related in the three experiments, and three separate phase calculations are not needed. One possibility is to write down the phase cycle for the entire experiment, interspersing the “real” and “imaginary” experiments, then generate an “effective transient counter” as follows:

```c
if (phase==0) assign(ct,v10); /* v10=01234... */
else
    {
        if (phase==1) dbl(ct,v10); /* v10=02468... */
        else
            incr(v10); /* v10=13579... */
    }
```

Now a single phase cycle can be derived from v10 instead of from ct. If phase1=0, each element of this phase cycle is selected. If phase1=1, only the odd elements are selected (the first, third, fifth, etc. transients for which ct=0, 2, 4,...). If phase1=2, the even elements only are selected (ct odd).

**Real Mode Phased 2D: TPPI**

For TPPI experiments, the increment index is typically needed at some point in the phase calculation. The simplest way to obtain the index is to use the built-in real-time constant id2. This can be used in a construction such as

```c
if (phase1==3)
```
add(v11, id2, v11);

which adds the increment value (which starts at 0) to the phase contained in v11.

## 2.15 Gradient Control for PFG and Imaging

- “Setting the Gradient Current Amplifier Level,” page 126
- “Generating a Gradient Pulse,” page 127
- “Controlling Lock Field Correction Circuitry,” page 127
- “Programming Microimaging Pulse Sequences,” page 127

Varian, Inc. NMR systems support gradient control for applications using the optional pulsed field gradient (PFG) and imaging. The configuration parameter gradtype, set by the config program, specifies the presence of gradient hardware and its capabilities. The available gradient control statements are listed in Table 36.

**Table 36. Gradient Control Statements**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lk_hold()</td>
<td>Set lock field correction circuitry to hold</td>
</tr>
<tr>
<td>lk_sample()</td>
<td>Set lock field correction circuitry to sample</td>
</tr>
<tr>
<td>obl_gradient*</td>
<td>Execute an oblique gradient</td>
</tr>
<tr>
<td>oblique_gradient*</td>
<td>Execute an oblique gradient</td>
</tr>
<tr>
<td>obl_shapedgradient*</td>
<td>Execute a shaped oblique gradient</td>
</tr>
<tr>
<td>oblique_shapedgradient*</td>
<td>Execute a shaped oblique gradient</td>
</tr>
<tr>
<td>pe_gradient*</td>
<td>Oblique gradient with PE in 1 axis</td>
</tr>
<tr>
<td>pe2_gradient*</td>
<td>Oblique gradient with PE in 2 axes</td>
</tr>
<tr>
<td>pe3_gradient*</td>
<td>Oblique gradient with PE in 3 axes</td>
</tr>
<tr>
<td>pe2_shapedgradient*</td>
<td>Oblique shaped gradient with PE in 1 axis</td>
</tr>
<tr>
<td>pe3_shapedgradient*</td>
<td>Oblique shaped gradient with PE in 2 axes</td>
</tr>
<tr>
<td>phase_encode_gradient*</td>
<td>Oblique gradient with PE in 3 axes</td>
</tr>
<tr>
<td>phase_encode3_gradient*</td>
<td>Oblique gradient with PE in 3 axes</td>
</tr>
<tr>
<td>phase_encode_shapedgradient*</td>
<td>Oblique shaped gradient with PE in 1 axis</td>
</tr>
<tr>
<td>phase_encode3_shapedgradient*</td>
<td>Oblique shaped gradient with PE in 3 axes</td>
</tr>
<tr>
<td>rgradient(channel, value)</td>
<td>Set gradient to specified level</td>
</tr>
<tr>
<td>shapedgradient*</td>
<td>Shaped gradient pulse</td>
</tr>
<tr>
<td>shaped2Dgradient*</td>
<td>Arrayed shaped gradient function</td>
</tr>
<tr>
<td>shapedincgradient*</td>
<td>Dynamic variable gradient function</td>
</tr>
<tr>
<td>shapedvgradient*</td>
<td>Dynamic variable shaped gradient function</td>
</tr>
<tr>
<td>vgradient*</td>
<td>Set gradient to level determined by real-time math</td>
</tr>
<tr>
<td>vgradient*</td>
<td>Variable angle gradient</td>
</tr>
<tr>
<td>vagradient*</td>
<td>Pulse controlled variable angle gradient</td>
</tr>
<tr>
<td>vagrpulse*</td>
<td>Variable angle shaped gradient</td>
</tr>
<tr>
<td>vashapedgradient*</td>
<td>Variable angle pulse controlled shaped gradient</td>
</tr>
<tr>
<td>zgradpulse(value, delay)</td>
<td>Create a gradient pulse on the z channel</td>
</tr>
<tr>
<td>zero_all_gradients*</td>
<td>Set all gradients to zero</td>
</tr>
</tbody>
</table>

* For the argument list, refer to the statement reference in Chapter 3

**MERCURxplus/-Vx systems use rgradient and vgradient, and the lk_sample and lk_hold statements.**
Table 37 lists delays for shaped gradient statements on systems with gradient waveform generators (gradtype='w' or gradtype='q'). The times for the three-axis gradient statements (obl_gradient, oblique_gradient, pe_gradient, phase_encode_gradient (gradtype='p', gradtype='q')) are the overhead times for setting all three gradients. The gradients are always set in sequential 'x', 'y', 'z' order.

Some gradient statements use DAC values to set the gradient levels and others use values in gauss/cm. The lower level gradient statements (gradient, rgradient, zgradpulse, shapedgradient, etc.) use DAC values, and the obliquing and variable-angle gradient statements use gauss/cm. The gradient statements associated with DAC values are used in single-axis PFG pulse sequences and microimaging pulse sequences, while the gradient statements associated with gauss/cm are used in imaging pulse sequences and triple-axis PFG pulse sequences.

### Setting the Gradient Current Amplifier Level

To set the gradient current amplifier level, use `rgradient(channel, value)`, where channel is 'X', 'x', 'Y', 'y', 'Z', or 'z' (only 'Z' or 'z' is supported on MERCURYplus/-Vx) and value is a real number for the amplifier level (e.g., `rgradient('z',1327.0)`). For the Performa I PFG module, value must be from 2048 to 2047; for Performa II, III, or IV, value must be from –32768.0 to 32767.0.

To set the gradient current amplifier level but determine the value instead by real-time math, use `vgradient(channel, intercept, slope, rtval)`, where channel is used the same as in `rgradient`, and amplifier level is determined by `intercept + slope * rtval` (e.g., `vgradient('z',-5000.0,2500.0,v10)`). This statement not available on the Performa I PFG module.
2.16 Programming the Performa XYZ PFG Module

Generating a Gradient Pulse

To create a gradient pulse on the z channel with given amplitude and duration, use `zgradpulse(value,delay)`, where `value` is used the same as in `rgradient` and `delay` is any delay parameter (e.g., `zgradpulse(1234.0,d2)`).

`shapedgradient(pattern,width,amp,channel,loops,wait)` generates a shaped gradient, where `pattern` is a file in `shapelib`, `width` is the pulse length, `amp` is a value that scales the amplitude of the pulse, `channel` is the same as used with `rgradient`, `loops` is the number of times (1 to 255) to loop the waveform, and `wait` is WAIT or NOWAIT for whether or not a delay is inserted to wait until the gradient is completed before executing the next statement.

Example:

```plaintext
shapedvgradient("hsine",d3,amplitude,igpe,v5,gphase,v1,NOWAIT,1);
```

This statement is only available on the Perform II, III, or IV PFG module.

Controlling Lock Field Correction Circuitry

On Varian, Inc. NMR systems, `lk_sample()` and `lk_hold()` are provided to control the lock field correction circuitry. If during the course of a pulse sequence the lock signal is disturbed—for instance, with a gradient pulse or pulses at the $^2$H frequency—the lock circuitry might not be able to hold on to the lock. When this is the case, the correction added in the feedback loop that holds the lock can be held constant by calling `lk_hold()`. At some time after the disturbance has passed (how long depends on the type of disturbance), the statement `lk_sample()` should be called to allow the circuitry to correct for disturbances external to the experiment.

Programming Microimaging Pulse Sequences

The procedures for programming microimaging pulse sequences are the same as those used in the programming of spectroscopy sequences, with the exception that additional pulse sequence statements have been added to define the amplitude and timing of the gradient pulses and the shaped rf pulses. For example, in the statement `rgradient(name,value)` to set a gradient, the argument `name` is either X, Y, or Z (or alternatively with the connection through the parameter `orient`, `gread`, `gphase`, or `gslice`) and `value` is the desired gradient strength in DAC units at the time the statement is to be implemented.

The basic imaging sequences included with the VnmrJ software are sequences for which the image data can be acquired, processed, and displayed with essentially the same software tools that are used with 2D spectra. These sequences have been written in a form that provides a great deal of flexibility in adapting them to the different modes of imaging and includes the capabilities of multislice and multiecho imaging. Many of the spectroscopic preparation pulse sequences can be linked to the standard imaging sequences to limit the spin population type that is imaged, to provide greater contrast in the image, or to remove artifacts from the image.
The Performa XYZ pulsed field gradient (PFG) module adds new capabilities to high-resolution liquids experiments on Varian spectrometers. The module applies gradients in B₀ along three distinct axes at different times during the course of the pulse sequence. These gradients can perform many functions, including solvent suppression and coherence pathway selection. This section describes pulse sequence programming of the module.

Creating Gradient Tables

In order for the software to have the necessary information on all three axes to convert between gauss/cm and DAC values, the XYZ PFG probe and amplifier combination can be calibrated using the creategtable macro and a gradient table made in /vnmr/imaging/gradtables.

The macro first prompts the user to see if the gradient axes are set to the same gradient strength (horizontal-bore imaging system) or if the axes have different gradient strengths (vertical-bore PFG gradients). Next, the user is prompted for a name for the gradient coil, and that name is then used in the gcoil and sysgcoil parameters in order to correctly translate between DAC and gauss/cm values. Finally, the macro prompts the user for the boresize of the magnet (51 mm), the gradient rise time (40 μs), and the maximum gradient strength obtainable for each axis. Note that the gradient strengths are not equal and the amplifier does not limit the combined output.

If the parameter gcoil does not exist in a parameter set and must be created, set the protection bit that causes the macro _gcoil to be executed must be set when the value for gcoil is changed. Set the protection bit using either of these procedures:

- Use the macro updtgcoil, which will create the gcoil parameter if it does not exist.
- Create gcoil with the following commands:
  ```
  create('gcoil','string')
  setprotect('gcoil','set',9)
  ```

In an experiment that uses gradient coils, the sysgcoil parameter can be set to the coil name specified with the creategtable macro and then the updtgcoil macro can be run to update the local gcoil parameter from the global sysgcoil parameter. When the local gcoil parameter is updated, the local gxmax, gymax, gzmax, trise and boresize parameters are also updated. Refer to the Command and Parameter Reference and the VnmrJ Imaging, User Guide for additional information about creategtable.

Pulse Sequence Programming

Table 38 lists the pulse sequence statements related to the XYZ PFG module. The system can be programmed by using the statements rgradient(channel,value) and zgradpulse(value,delay). Pulse sequences g2pul.c and profile.c in /vnmr/psglib are examples of using the gradaxis parameter and the rgradient statement.

To produce a gradient at any angle by the combination of two or more gradients, the vagradpulse(gradlvl,gradtime,theta,phi) statement can be used, and to produce three equal and simultaneous gradients, such that an effective gradient is produced at the magic angle, the magradpulse(gradlvl,gradtime) statement is available. The statements vagradpulse and magradpulse are structured so that the software does all of the calculations to produce the effective gradient desired. Both statements take the argument for the gradient level (gradlvl) in gauss/cm. This is distinctly different
from the \texttt{rgradient} and \texttt{zgradpulse} statements, which take the argument for the gradient level (\texttt{value}) in DAC.

With these statements, the \texttt{gcoil} and \texttt{sysgcoil} parameters are required for the software to calculate the correct DAC value for each channel in order to produce the requested effective gradient. After the gradients have each been calibrated and a \texttt{gradtable} has been constructed with the \texttt{creategtable} macro, as described above, then the \texttt{sysgcoil} parameter can be set to that coil name used. The \texttt{updtgcoil} macro can then update the local \texttt{gcoil} parameter from the global \texttt{sysgcoil} parameter.

The \texttt{vagradpulse} statement uses the theta and phi angles to produce an effective gradient at any arbitrary angle. For example, using \texttt{vagradpulse with theta=54.7 and phi=0.0}, an effective gradient is produced at the magic angle by the correct combination of the Z gradient and the Y gradient. Whereas, if \texttt{theta=54.7 and phi=90}, an effective gradient is produced at the magic angle by the correct combination of the Z gradient and the X gradient. Variations on the \texttt{vagradpulse} statement include the capability of shaping the gradient waveform with the \texttt{vashapedgradient} and the \texttt{vashapedgradpulse} statements. For more information about these statements, see their descriptions in Chapter 3.

In addition, the \texttt{magradpulse} statement produces equal and simultaneous gradients on all three axes in order to produce an effective gradient at the magic angle. Variations on the \texttt{magradpulse} statement include the capability of shaping the gradient waveform with the \texttt{mashapedgradient} and the \texttt{mashapedgradpulse} statements. Again, for more information, refer to Chapter 3.

### Table 38. Performa XYZ PFG Module Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>magradient(gradlvl)</code></td>
<td>Simultaneous gradient at the magic angle</td>
</tr>
<tr>
<td><code>magradpulse(gradlvl,gradtime)</code></td>
<td>Simultaneous gradient pulse at the magic angle</td>
</tr>
<tr>
<td><code>mashapedgradient</code></td>
<td>Simultaneous shaped gradient at the magic angle</td>
</tr>
<tr>
<td><code>mashapedgradpulse</code></td>
<td>Simultaneous shaped gradient pulse at the magic angle</td>
</tr>
<tr>
<td><code>rgradient(axis,value)</code></td>
<td>Set gradient to specified level</td>
</tr>
<tr>
<td><code>vagradpulse</code></td>
<td>Variable angle gradient pulse</td>
</tr>
<tr>
<td><code>vashapedgradient</code></td>
<td>Variable angle shaped gradient</td>
</tr>
<tr>
<td><code>vashapedgradpulse</code></td>
<td>Variable angle shaped gradient</td>
</tr>
<tr>
<td><code>zgradpulse(value,delay)</code></td>
<td>Create a gradient pulse on the z channel</td>
</tr>
</tbody>
</table>

* Variations are available with additional parameters*
Chapter 2. Pulse Sequence Programming

Table 39. Imaging-Related Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create_delay_list*</td>
<td>Create table of delays</td>
</tr>
<tr>
<td>create_freq_list*</td>
<td>Create table of frequencies</td>
</tr>
<tr>
<td>create_offset_list*</td>
<td>Create table of frequency offsets</td>
</tr>
<tr>
<td>endmsloop*/endpeloop*</td>
<td>Ends a loop started by the msloop/peloop</td>
</tr>
<tr>
<td>getarray*</td>
<td>Retrieves all values of arrayed parameter</td>
</tr>
<tr>
<td>getorientation*</td>
<td>Read image plane orientation</td>
</tr>
<tr>
<td>incgradient*</td>
<td>Dynamic variable gradient function</td>
</tr>
<tr>
<td>init_rfpattern*</td>
<td>Create rf pattern file</td>
</tr>
<tr>
<td>init_gradpattern*</td>
<td>Create gradient pattern file</td>
</tr>
<tr>
<td>init_vscan*</td>
<td>Initialize real-time variable for vscan</td>
</tr>
<tr>
<td>obl_gradient*</td>
<td>Execute an oblique gradient</td>
</tr>
<tr>
<td>oblique_gradient*</td>
<td>Execute an oblique gradient</td>
</tr>
<tr>
<td>obl_shapedgradient*</td>
<td>Execute a shaped oblique gradient</td>
</tr>
<tr>
<td>oblique_shapedgradient*</td>
<td>Execute a shaped oblique gradient</td>
</tr>
<tr>
<td>msloop*/peloop*</td>
<td>Provides a sequence-switchable loop</td>
</tr>
<tr>
<td>pe_gradient*</td>
<td>Oblique gradient with PE in 1 axis</td>
</tr>
<tr>
<td>pe2_gradient*</td>
<td>Oblique gradient with PE in 2 axes</td>
</tr>
<tr>
<td>pe3_gradient*</td>
<td>Oblique gradient with PE in 3 axes</td>
</tr>
<tr>
<td>pe_shapedgradient*</td>
<td>Oblique shaped gradient with PE in 1 axis</td>
</tr>
<tr>
<td>pe2_shapedgradient*</td>
<td>Oblique shaped gradient with PE in 2 axes</td>
</tr>
<tr>
<td>pe3_shapedgradient*</td>
<td>Oblique shaped gradient with PE in 3 axes</td>
</tr>
<tr>
<td>phase_encode_gradient*</td>
<td>Oblique gradient with PE in 1 axis</td>
</tr>
<tr>
<td>phase_encode3_gradient*</td>
<td>Oblique gradient with PE in 3 axes</td>
</tr>
<tr>
<td>phase_encode_shapedgradient*</td>
<td>Oblique shaped gradient with PE in 1 axis</td>
</tr>
<tr>
<td>phase_encode3_shapedgradient*</td>
<td>Oblique shaped gradient with PE in 3 axes</td>
</tr>
<tr>
<td>poffset*/position_offset*</td>
<td>Set frequency based on position</td>
</tr>
<tr>
<td>poffset_list*</td>
<td>Set frequency from position list</td>
</tr>
<tr>
<td>position_offset_list*</td>
<td>Set frequency from position list</td>
</tr>
<tr>
<td>shapedgradient*</td>
<td>Provide shaped gradient pulse</td>
</tr>
<tr>
<td>shaped2Dgradient*</td>
<td>Arrayed shaped gradient function</td>
</tr>
<tr>
<td>shapedincgradient*</td>
<td>Dynamic variable gradient function</td>
</tr>
<tr>
<td>shapedvgradient*</td>
<td>Dynamic variable shaped gradient function</td>
</tr>
<tr>
<td>sli*</td>
<td>Set SLI lines</td>
</tr>
<tr>
<td>vagradient*</td>
<td>Variable angle gradient</td>
</tr>
<tr>
<td>vagradpulse*</td>
<td>Pulse controlled variable angle gradient</td>
</tr>
<tr>
<td>vashapedgradient*</td>
<td>Variable angle shaped gradient</td>
</tr>
<tr>
<td>vashapedgradpulse*</td>
<td>Variable angle pulse controlled shaped gradient</td>
</tr>
<tr>
<td>vdelay*</td>
<td>Select delay from table</td>
</tr>
<tr>
<td>vdelay_list*</td>
<td>Get delay value from delay list with real-time index</td>
</tr>
<tr>
<td>vfreq*</td>
<td>Select frequency from table</td>
</tr>
<tr>
<td>vgradient*</td>
<td>Dynamic variable gradient</td>
</tr>
<tr>
<td>voffset*</td>
<td>Select frequency offset from table</td>
</tr>
<tr>
<td>vscan*</td>
<td>Dynamic variable scan function</td>
</tr>
<tr>
<td>vsli*</td>
<td>Set SLI lines from real-time variable</td>
</tr>
<tr>
<td>zero_all_gradients*</td>
<td>Sets all gradients to zero</td>
</tr>
</tbody>
</table>

* For the argument list, refer to the statement reference in Chapter 3

The sli and vsli statements are not supported on UNITY/INOVA. UNITY/INOVA support for interfacing to an external device is included in the AP User interface.

Support the use of real-time variables and loops when constructing imaging sequences. Using real-time variables and loops resulting in “compressed” acquisitions, instead of
standard acquisition arrays, reduces the number of acodes sets needed to run the 
experiment, cutting down significantly on the start-up time of the experiment and removing 
any inter-FID and intertransient overhead delays. This is not really a problem, because its 
small overhead delays and $d_0$ parameter make the inter-FID and intertransient delays 
consistent, but may make a difference in some applications. All VnmrS system waveforms 
must have $4\mu$s resolution.

Real-time Gradient Statements

Real-time gradient statements consist of additions to the standard gradient and 
shapedgradient statements, which provide real-time variable control for the gradient 
amplitudes. Real-time statements include shapedvgradient, which provides real-time 
control on one axis and incgradient and shapedincgradient, which support 
real-time control over three axes. The vgradient statement also belongs to this group.

Oblique Gradient Statements

To support oblique imaging and the imaging interface, oblique gradient statements include 
oblique_gradient, phase_encode_gradient, pe_gradient and all of their 
variations. The inputs to these statements are amplitudes and phases. Amplitudes are 
expressed in gauss/cm and correspond to the read-out, phase-encode, and slice-select axis 
in the logical frame. Phase angles correspond to Euler angles $\psi$, $\phi$, and $\theta$ and 
describe the coordinate rotation applied to the input amplitudes. For more information on 
use, see the manual VnmrJ Imaging, User Guide.

Global List and Position Statements

The global list statements support real-time selection of frequencies, offsets, and delays. 
Global lists are different from AP tables in that the lists are sent down to the acquisition 
console when the experiment starts up and remain accessible until the experiments 
completes. The lists can be arrayed parameters (with a protection bit set to prevent an 
arrayed acquisition) read into the pulse sequence using the getarray statement or 
standard C language arrays calculated within the pulsesquence. The lists are initialized 
with the statements create_freq_list, create_offset_list, and 
create_delay_list, and then selected and set using the vfreq, voffset, and 
vdelay_list statements, which use a real-time parameter as an index into the list.

The position statements set the rf frequency from a given position or an array of positions. 
These statements are poffset, poffset_list, position_offset, and 
position_offset_list. The position list statements use global lists, which initialize 
the list and select and set the position in a single statement.

When creating global list parameters, create them as acquisition parameters and set 
protection bit 8 (value 256) or else PSG tries to array them as standard arrayed acquisitions.

Looping Statements

The looping statements msloop and peoffset define multislice and phase encode loops 
when creating imaging pulse sequences. The looping statements also allow selection of a 
standard “arrayed” acquisition or a “compressed” acquisition using the seqcon parameter.
Waveform Initialization Statements

The waveform initialization statements `init_rfpattern` and `init_gradpattern` are available to all configurations and allow the user to calculate and create gradient and rf patterns in PSG.

2.18 User-Customized Pulse Sequence Generation

The complete pulse sequence generation (PSG) source code is supplied in the VnmrJ system `psg` directory. This code enables users to create their own `libpsglib.so` PSG directory for link loading with the pulse sequence object file `pulsesequence.o`.

The shell script `setuserpsg` in the system directory creates the directory `vnmrsys/psg` for a user, if it does not already exist, and initializes this user PSG directory with the appropriate object libraries from the system PSG directory. The script `setuserpsg` should only have to be run once by each separate user. `setuserpsg` places the file `libpsglib.a` in the user's `psg` directory.

The shell script `psggen` compiles files in the user PSG object directory and places the files in the user PSG directory. When executed, `seqgen` looks first for the user PSG library `~/vnmrsys/psg` in the user PSG directory, and then in the system library directory `/vnmr/lib`.

Modifying a PSG source file and subsequently recompiling the user PSG object directory is done as follows:

1. Enter `setuserpsg` from a shell (done only once).
   Typical output from this command is as follows:
   Creating user PSG directory...
   Copying User PSG library from system directory...

2. Copy the desired PSG source file(s) from `$vnmrsystem/psg` to `$vnmruser/psg`.

3. Modify the PSG source files in the user PSG directory.

4. Enter `psggen` from a shell or from within Vnmr.
   Typical output from this command is as follows:
   Creating additional source links...
   Compiling PSG Library...
   PSG Library Complete.
Chapter 3. Pulse Sequence Statement Reference

This chapter is a reference for the statements used in VnmrJ pulse sequence programming.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort_message</td>
<td>Send an error to VnmrJ and abort the PSG process</td>
</tr>
<tr>
<td>acquire</td>
<td>Explicitly acquire data</td>
</tr>
<tr>
<td>add</td>
<td>Add integer values</td>
</tr>
<tr>
<td>apovrride</td>
<td>Override internal software AP bus delay</td>
</tr>
<tr>
<td>apshaped_decpulse</td>
<td>First decoupler pulse shaping via AP bus</td>
</tr>
<tr>
<td>apshaped_dec2pulse</td>
<td>Second decoupler pulse shaping via AP bus</td>
</tr>
<tr>
<td>apshaped_pulse</td>
<td>Observe transmitter pulse shaping via AP bus</td>
</tr>
<tr>
<td>assign</td>
<td>Assign integer values</td>
</tr>
<tr>
<td>blankingoff</td>
<td>Unblank amplifier channels and turn amplifiers on</td>
</tr>
<tr>
<td>blankingon</td>
<td>Blank amplifier channels and turn amplifiers off</td>
</tr>
<tr>
<td>blankoff</td>
<td>Stop blanking observe or decoupler amplifier (obsolete)</td>
</tr>
<tr>
<td>blankon</td>
<td>Start blanking observe or decoupler amplifier (obsolete)</td>
</tr>
<tr>
<td>clearapdatatable</td>
<td>Zero all data in acquisition processor memory</td>
</tr>
<tr>
<td>create_delay_list</td>
<td>Create table of delays</td>
</tr>
<tr>
<td>create_freq_list</td>
<td>Create table of frequencies</td>
</tr>
<tr>
<td>create_offset_list</td>
<td>Create table of frequency offsets</td>
</tr>
<tr>
<td>dbl</td>
<td>Double an integer value</td>
</tr>
<tr>
<td>dcplrphase</td>
<td>Set small-angle phase of 1st decoupler,</td>
</tr>
<tr>
<td>dcplr2phase</td>
<td>Set small-angle phase of 2nd decoupler,</td>
</tr>
<tr>
<td>dcplr3phase</td>
<td>Set small-angle phase of 3rd decoupler,</td>
</tr>
<tr>
<td>decblank</td>
<td>Blank amplifier associated with first decoupler</td>
</tr>
<tr>
<td>dec2blank</td>
<td>Blank amplifier associated with second decoupler</td>
</tr>
<tr>
<td>dec3blank</td>
<td>Blank amplifier associated with third decoupler</td>
</tr>
<tr>
<td>declvloff</td>
<td>Return first decoupler back to “normal” power</td>
</tr>
<tr>
<td>declvlon</td>
<td>Turn on first decoupler to full power</td>
</tr>
<tr>
<td>decoff</td>
<td>Turn off first decoupler</td>
</tr>
<tr>
<td>dec2off</td>
<td>Turn off second decoupler</td>
</tr>
<tr>
<td>dec3off</td>
<td>Turn off third decoupler</td>
</tr>
<tr>
<td>decoffset</td>
<td>Change offset frequency of first decoupler</td>
</tr>
<tr>
<td>dec2offset</td>
<td>Change offset frequency of second decoupler</td>
</tr>
</tbody>
</table>
**Chapter 3. Pulse Sequence Statement Reference**

- `dec3offset`: Change offset frequency of third decoupler
- `dec4offset`: Change offset frequency of fourth decoupler
- `decon`: Turn on first decoupler
- `dec2on`: Turn on second decoupler
- `dec3on`: Turn on third decoupler
- `decpower`: Change first decoupler power level, linear amp. systems
- `dec2power`: Change second decoupler power level, linear amp. systems
- `dec3power`: Change third decoupler power level, linear amp. systems
- `dec4power`: Change fourth decoupler power level, linear amp. systems
- `decprgoff`: End programmable decoupling on first decoupler
- `dec2prgoff`: End programmable decoupling on second decoupler
- `dec3prgoff`: End programmable decoupling on third decoupler
- `decprgon`: Start programmable decoupling on first decoupler
- `dec2prgon`: Start programmable decoupling on second decoupler
- `dec3prgon`: Start programmable decoupling on third decoupler
- `decpulse`: Pulse first decoupler transmitter with amplifier gating
- `decpwr`: Set first decoupler high-power level, class C amplifier
- `decpwrf`: Set first decoupler fine power
- `dec2pwrf`: Set second decoupler fine power
- `dec3pwrf`: Set third decoupler fine power
- `decr`: Decrement an integer value
- `decrpulse`: Pulse first decoupler with amplifier gating
- `dec2rgpulse`: Pulse second decoupler with amplifier gating
- `dec3rgpulse`: Pulse third decoupler with amplifier gating
- `dec4rgpulse`: Pulse fourth decoupler with amplifier gating
- `decshaped_pulse`: Perform shaped pulse on first decoupler
- `dec2shaped_pulse`: Perform shaped pulse on second decoupler
- `dec3shaped_pulse`: Perform shaped pulse on third decoupler
- `decspinlock`: Set spin lock waveform control on first decoupler
- `dec2spinlock`: Set spin lock waveform control on second decoupler
- `dec3spinlock`: Set spin lock waveform control on third decoupler
- `decstepsize`: Set step size for first decoupler
- `dec2stepsize`: Set step size for second decoupler
- `dec3stepsize`: Set step size for third decoupler
- `decunblank`: Unblank amplifier associated with first decoupler
- `dec2unblank`: Unblank amplifier associated with second decoupler
- `dec3unblank`: Unblank amplifier associated with third decoupler
- `delay`: Delay for a specified time
- `dhpflag`: Switch decoupling from low-power to high-power
divn              Divide integer values

dps_off          Turn off graphical display of statements

dps_on           Turn on graphical display of statements

dps_show         Draw delay or pulses in a sequence for graphical display

dps_skip         Skip graphical display of next statement

elsenz           Execute succeeding statements if argument is nonzero

endhardloop      End hardware loop

endif            End execution started by ifzero or elsenz

endloop          End loop

endmsloop        End multislice loop

endpeloop        End phase-encode loop

gate             Device gating (obsolete)

getarray         Get arrayed parameter values

getelem          Retrieve an element from a table

getorientation   Read image plane orientation

getstr           Look up value of string parameter

getval           Look up value of numeric parameter

G_Delay          Generic delay routine

G_Offset         Frequency offset routine

G_Power          Fine power routine

G_Pulse          Generic pulse routine

hdwshiminit      Initialize next delay for hardware shimming

hlv              Find half the value of an integer

hsdelay          Delay specified time with possible homospoil pulse

idecpulse        Pulse first decoupler transmitter with IPA

idecrgpulse      Pulse first decoupler with amplifier gating and IPA

idelay           Delay for a specified time with IPA

ifzero           Execute succeeding statements if argument is zero

incdelay         Set real-time incremental delay

incgradient      Generate dynamic variable gradient pulse

incr             Increment an integer value

init_rfpattern   Create rf pattern file

init_gradpattern Create gradient pattern file

init_vscan       Initialize real-time variable for vscan statement

initdelay        Initialize incremental delay

initparms_sis    Initialize parameters for spectroscopy imaging sequences

initval          Initialize a real-time variable to specified value

iobspulse        Pulse observe transmitter with IPA

ioffset          Change offset frequency with IPA

ipulse           Pulse observe transmitter with IPA

ipwrf            Change transmitter or decoupler fine power with IPA

ipwrm            Change transmitter or decoupler lin. mod. power with IPA

irgpulse         Pulse observe transmitter with IPA
lk_hold
Set lock correction circuitry to hold correction

lk_sample
Set lock correction circuitry to sample lock signal

loadtable
Load AP table elements from table text file

loop
Start loop

loop_check
Check that number of FIDs is consistent with number of slices, etc.

magradient
Simultaneous gradient at the magic angle

magradpulse
Gradient pulse at the magic angle

mashapedgradient
Simultaneous shaped gradient at the magic angle

mashapedgradpulse
Simultaneous shaped gradient pulse at the magic angle

mod2
Find integer value modulo 2

mod4
Find integer value modulo 4

modn
Find integer value modulo n

msloop
Multislice loop

mult
Multiply integer values

obl_gradient
Execute an oblique gradient

oblique_gradient
Execute an oblique gradient

obl_shapedgradient
Execute a shaped oblique gradient

obl_shaped3gradient
Execute a shaped oblique gradient

oblique_shapedgradient
Execute a shaped oblique gradient

obl_blank
Blank amplifier associated with observe transmitter

obsoffset
Change offset frequency of observe transmitter

obspower
Change observe transmitter power level, lin. amp. systems

obsprgoff
End programmable control of observe transmitter

obsprgon
Start programmable control of observe transmitter

obspulse
Pulse observe transmitter with amplifier gating

obspwrf
Set observe transmitter fine power

obstsize
Set step size for observe transmitter

obsunblank
Unblank amplifier associated with observe transmitter

offset
Change offset frequency of transmitter or decoupler

pbox_ad180
Generate Hadamard encoded adiabatic 180 deg. shapes using Pbox

pbox_mix
Generate Hadamard encoded mixing shapes using Pbox.

pboxHT_F1
Generate arbitrary shapes in F1 using Pbox

pboxHT_F1e
Generate Hadamard encoded excitation shapes in F1 using Pbox

pboxHT_F1i
Generate Hadamard encoded inversion shapes in F1 using Pbox

pboxHT_F1s
Generate Hadamard encoded sequential inversion shapes

pboxHT_F1r
Generate Hadamard encoded refocusing shapes in F1 using Pbox

pe_gradient
Oblique gradient with phase encode in one axis

pe2_gradient
Oblique gradient with phase encode in two axes

pe3_gradient
Oblique gradient with phase encode in three axes

pe_shapedgradient
Oblique shaped gradient with phase encode in one axis

pe2_shapedgradient
Oblique shaped gradient with phase encode in two axes

pe3_shapedgradient
Oblique shaped gradient with phase encode in three axes

pe3_shaped3gradient
Oblique shaped gradient with phase encode in three axes
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>peloop</td>
<td>Phase-encode loop</td>
</tr>
<tr>
<td>phase_encode_gradient</td>
<td>Oblique gradient with phase encode in one axis</td>
</tr>
<tr>
<td>phase_encode3_gradient</td>
<td>Oblique gradient with phase encode in three axes</td>
</tr>
<tr>
<td>phase_encode3_shapedgradient</td>
<td>Oblique shaped gradient with PE in one axis</td>
</tr>
<tr>
<td>phasedloop</td>
<td>Oblique shaped gradient with PE in three axes</td>
</tr>
<tr>
<td>phaseshift</td>
<td>Set phase-pulse technique, rf type A or B</td>
</tr>
<tr>
<td>poffset (Inova system)</td>
<td>Set frequency based on position</td>
</tr>
<tr>
<td>poffset_list</td>
<td>Set frequency from position list</td>
</tr>
<tr>
<td>position_offset</td>
<td>Set frequency based on position</td>
</tr>
<tr>
<td>position_offset_list</td>
<td>Set frequency from position list</td>
</tr>
<tr>
<td>power</td>
<td>Change power level</td>
</tr>
<tr>
<td>psg_abort</td>
<td>Abort the PSG process</td>
</tr>
<tr>
<td>pulse</td>
<td>Pulse observe transmitter with amplifier gating</td>
</tr>
<tr>
<td>putCmd</td>
<td>Send a command to VnmrJ form a pulse sequence</td>
</tr>
<tr>
<td>pwrf</td>
<td>Change transmitter or decoupler fine power</td>
</tr>
<tr>
<td>pwrerm</td>
<td>Change transmitter or decoupler linear modulator power</td>
</tr>
<tr>
<td>rcvroff</td>
<td>Turn off receiver gate and amplifier blanking gate</td>
</tr>
<tr>
<td>rcvron</td>
<td>Turn on receiver gate and amplifier blanking gate</td>
</tr>
<tr>
<td>readuserap</td>
<td>Read input from user AP register</td>
</tr>
<tr>
<td>reconf</td>
<td>Turn off receiver gate only</td>
</tr>
<tr>
<td>recon</td>
<td>Turn on receiver gate only</td>
</tr>
<tr>
<td>rgpulse</td>
<td>Pulse observe transmitter with amplifier gating</td>
</tr>
<tr>
<td>rgradient</td>
<td>Set gradient to specified level</td>
</tr>
<tr>
<td>rlpower</td>
<td>Change power level</td>
</tr>
<tr>
<td>rlpwrf</td>
<td>Set transmitter or decoupler fine power (obsolete)</td>
</tr>
<tr>
<td>rlpwrm</td>
<td>Set transmitter or decoupler linear modulator power</td>
</tr>
<tr>
<td>rotate</td>
<td>Sets the standard oblique rotation angles</td>
</tr>
<tr>
<td>rot_angle</td>
<td>Sets user defined oblique rotation angles</td>
</tr>
<tr>
<td>rotorperiod</td>
<td>Obtain rotor period of MAS rotor</td>
</tr>
<tr>
<td>rotorsync</td>
<td>Gated pulse sequence delay from MAS rotor position</td>
</tr>
<tr>
<td>setautoincrement</td>
<td>Set autoincrement attribute for a table</td>
</tr>
<tr>
<td>setdivnfactor</td>
<td>Set divn-return attribute and divn-factor for AP table</td>
</tr>
<tr>
<td>setreceiver</td>
<td>Associate the receiver phase cycle with a table</td>
</tr>
<tr>
<td>setstatus</td>
<td>Set status of observe transmitter or decoupler transmitter</td>
</tr>
<tr>
<td>settable</td>
<td>Store an array of integers in a real-time AP table</td>
</tr>
<tr>
<td>setuserap</td>
<td>Set user AP register</td>
</tr>
<tr>
<td>shapedpulse</td>
<td>Perform shaped pulse on observe transmitter</td>
</tr>
<tr>
<td>shaped_pulse</td>
<td>Perform shaped pulse on observe transmitter</td>
</tr>
<tr>
<td>shapedgradient</td>
<td>Generate shaped gradient pulse</td>
</tr>
<tr>
<td>shaped2Dgradient</td>
<td>Generate arrayed shaped gradient pulse</td>
</tr>
<tr>
<td>shapedincgradient</td>
<td>Generate dynamic variable gradient pulse</td>
</tr>
<tr>
<td>Statement</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------------------------------</td>
</tr>
<tr>
<td>shapedvgradient</td>
<td>Generate dynamic variable shaped gradient pulse</td>
</tr>
<tr>
<td>simpulse</td>
<td>Pulse observe and decouple channels simultaneously</td>
</tr>
<tr>
<td>sim3pulse</td>
<td>Pulse simultaneously on 2 or 3 rf channels</td>
</tr>
<tr>
<td>sim4pulse</td>
<td>Simultaneous pulse on four channels</td>
</tr>
<tr>
<td>simshaped_pulse</td>
<td>Perform simultaneous two-pulse shaped pulse</td>
</tr>
<tr>
<td>sim3shaped_pulse</td>
<td>Perform a simultaneous three-pulse shaped pulse</td>
</tr>
<tr>
<td>sli</td>
<td>Set SLI lines</td>
</tr>
<tr>
<td>sp#off</td>
<td>Turn off specified spare line (Inova #=1 to 5)</td>
</tr>
<tr>
<td>sp#on</td>
<td>Turn on specified spare line (Inova #=1 to 5)</td>
</tr>
<tr>
<td>spinlock</td>
<td>Control spin lock on observe transmitter</td>
</tr>
<tr>
<td>starthardloop</td>
<td>Start hardware loop</td>
</tr>
<tr>
<td>status</td>
<td>Change status of decoupler and homospoil</td>
</tr>
<tr>
<td>statusdelay</td>
<td>Execute the status statement with a given delay time</td>
</tr>
<tr>
<td>stepsize</td>
<td>Set small-angle phase step size,</td>
</tr>
<tr>
<td>sub</td>
<td>Subtract integer values</td>
</tr>
<tr>
<td>text_error</td>
<td>Send a text error message to VnmrJ</td>
</tr>
<tr>
<td>text_message</td>
<td>Send a message to VnmrJ</td>
</tr>
<tr>
<td>tsadd</td>
<td>Add an integer to AP table elements</td>
</tr>
<tr>
<td>tsdiv</td>
<td>Divide an integer into AP table elements</td>
</tr>
<tr>
<td>tsmult</td>
<td>Multiply an integer with AP table elements</td>
</tr>
<tr>
<td>tssub</td>
<td>Subtract an integer from AP table elements</td>
</tr>
<tr>
<td>ttadd</td>
<td>Add a table to a second table</td>
</tr>
<tr>
<td>ttdiv</td>
<td>Divide a table into a second table</td>
</tr>
<tr>
<td>ttmult</td>
<td>Multiply a table by a second table</td>
</tr>
<tr>
<td>ttsub</td>
<td>Subtract a table from a second table</td>
</tr>
<tr>
<td>txphase</td>
<td>Set quadrature phase of observe transmitter</td>
</tr>
<tr>
<td>vagradient</td>
<td>Variable angle gradient</td>
</tr>
<tr>
<td>vagradpulse</td>
<td>Variable angle gradient pulse</td>
</tr>
<tr>
<td>var_active</td>
<td>Checks if the parameter is being used</td>
</tr>
<tr>
<td>vashapedgradient</td>
<td>Variable angle shaped gradient</td>
</tr>
<tr>
<td>vashapedgradpulse</td>
<td>Variable angle shaped gradient pulse</td>
</tr>
<tr>
<td>vdelay</td>
<td>Set delay with fixed timebase and real-time count</td>
</tr>
<tr>
<td>vdelay_list</td>
<td>Get delay value from delay list with real-time index</td>
</tr>
<tr>
<td>vfreq</td>
<td>Select frequency from table</td>
</tr>
<tr>
<td>vgradient</td>
<td>Set gradient to a level determined by real-time math</td>
</tr>
<tr>
<td>voffset</td>
<td>Select frequency offset from table</td>
</tr>
<tr>
<td>vscan</td>
<td>Provide dynamic variable scan</td>
</tr>
<tr>
<td>vsetuserap</td>
<td>Set user AP register using real-time variable</td>
</tr>
<tr>
<td>vsli</td>
<td>Set SLI lines from real-time variable</td>
</tr>
<tr>
<td>warn_message</td>
<td>Send a warning message to VnmrJ</td>
</tr>
<tr>
<td>xgate</td>
<td>Gate pulse sequence from an external event</td>
</tr>
<tr>
<td>xmtroff</td>
<td>Turn off observe transmitter</td>
</tr>
<tr>
<td>xmtron</td>
<td>Turn on observe transmitter</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>xmtrphase</code></td>
<td>Set transmitter small-angle phase, rf type C, D</td>
</tr>
<tr>
<td><code>zero_all_gradients</code></td>
<td>Zero all gradients</td>
</tr>
<tr>
<td><code>zgradpulse</code></td>
<td>Create a gradient pulse on the z channel</td>
</tr>
</tbody>
</table>
### Chapter 3. Pulse Sequence Statement Reference

#### abort_message
Send an error to VnmrJ and abort the PSG process

#### acquire
Explicitly acquire data

#### add
Add integer values

#### apovrride
Override internal software AP bus delay

#### apshaped_decpulse
First decoupler pulse shaping via AP bus

#### apshaped_dec2pulse
Second decoupler pulse shaping via AP bus

#### apshaped_pulse
Observe transmitter pulse shaping via AP bus

#### assign
Assign integer values

---

**Syntax:** `abort_message(char *format, ...)`

**Description:** `abort_message` sends the specified error message to VnmrJ and then aborts the PSG process.

**acquire**  **Explicitly acquire data**

**Applicability:**

**UNITY/NOVA**

For **UNITY/NOVA** systems, there are small overhead delays before and after the acquire. The pre-acquire delay takes into account setting the receiver phase with `oph` and enabling data overflow detection. The post-acquire delay is for disabling data overflow detection. When using acquire statements within a hardware loop these overhead delays and the functions associated with them are placed outside the hardware loop. When using multiple acquire statements outside a hardware loop in a pulse sequence setting, the phase and enabling data overflow detection is done before the first acquire statement. Disabling overflow detection is done after the last acquire, so there is no overhead time between acquire statements.

If an `acquire` statement occurs outside a hardware loop, the number of complex points to be acquired must be a multiple of 2 on systems with a Digital Acquisition Controller board, an Acquisition Controller board, or a Pulse Sequence Controller board, or must be a multiple of 32 on systems with a Output board (see page 140 for descriptions of each board).

Inside a hardware loop, systems with a Digital Acquisition Controller board or a Pulse Sequence Controller board can accept a maximum of 2048 complex points, systems with an Acquisition Controller board can accept a maximum of 1024 complex points, and systems with an Output board can accept a maximum of 63 complex points.

The following list identifies the acquisition controller boards used on Varian NMR spectrometer systems:
- **Data Acquisition Controller boards, Part No. 01-902010-00.** Started shipping in mid-1995 with the introduction of the UNITYINOVA system.
- **Pulse Sequence Controller boards, Part No. 00-992560-00.** Started shipping in early 1993 with the introduction of the UNITYplus system.
- **Acquisition Controller boards, Part No. 00-969204-00 or 00-990640-00.** Started shipping 00-969204-00 in late 1988 as a replacement for the Output boards. Part No. 00-990640-00 replaced 00-969204-00 in mid-1990.
- **Output boards, Part No. 00-953520-0#, where # is an integer.** Shipped with systems prior to 1988.

Examples: `acquire(np,1.0/sw);`

Related:
- `endhardloop` End hardware loop
- `starthardloop` Start hardware loop

### add

**Add integer values**

**Syntax:**
```
add(vi,vj,vk)
codeint vi; /* real-time variable vi for addend */
codeint vj; /* real-time variable vj for addend */
codeint vk; /* real-time variable vk for sum */
```

**Description:** Sets \( vk \) equal to the sum of integer values of \( vi \) and \( vj \).

**Arguments:** \( vi, vj, \) and \( vk \) are real-time variables (v1 to v14, oph, etc.).

**Examples:** `add(v1,v2,v3);`

Related:
- `assign` Assign integer values
- `dbl` Double an integer value
- `decr` Decrement an integer value
- `divn` Divide integer values
- `hlv` Half the value of an integer
- `incr` Increment an integer value
- `mod2` Find integer value modulo 2
- `mod4` Find integer value modulo 4
- `modn` Find integer value modulo \( n \)
- `mult` Multiply integer values
- `sub` Subtract integer values

### apovrride

**Override internal software AP bus delay**

**Applicability:**
- UNITYINOVA systems
- Systems with the 63-step Output board (Part No. 00-953520-0#, where # is an integer). This board shipped prior to 1988.

**Syntax:** `apovrride()`

**Description:** Systems with the 63-step Output board can use this statement to prevent a delay of 0.2 \( \mu s \) from being inserted prior to the next (and only the next) occurrence of one of the AP (analog port) bus statements `dcplrphase`, `dcplr2phase`, `dcplr3phase`, `decprgoff`, `dec2prgoff`, `dec3prgoff`, `decprgon`, `dec2prgon`, `dec3prgon`, `decshaped_pulse`, `dec2shaped_pulse`, `dec3shaped_pulse`, `decspinlock`, `dec2spinlock`, `dec3spinlock`, `obsprgoff`, `obsprgon`, `power`, `rlpower`, `shaped_pulse`, `simshaped_pulse`, `sim3shaped_pulse`, `spinlock`, and `xmtrphase`. 
**Chapter 3. Pulse Sequence Statement Reference**

**apshaped_decpulse**

**First decoupler pulse shaping via AP bus**

Applicability: **UNITY/INOVA** systems and **MERCURYplus/-Vx**.

**MERCURYplus/-Vx** only supports shapes with no phase shifts.

Syntax:

```c
apshaped_decpulse(shape, pulse_width, pulse_phase,
                    power_table, phase_table, RG1, RG2)
```

- `char *shape;` /* name of .RF shape file */
- `double pulse_width;` /* pulse width in sec */
- `codeint pulse_phase;` /* real-time phase of pulse */
- `codeint power_table;` /* table variable to store power */
- `codeint phase_table;` /* table variable to store phase */
- `double RG1;` /* gating time before pulse in sec */
- `double RG2;` /* gating time after pulse in sec */

Description: Provides first decoupler fine-grained “waveform generator-type” pulse shaping through the AP bus. A pulse shape file for the waveform generator (`/vnmr/shapelib/*.RF`) is used. This statement overrides any existing small-angle phase shifting (i.e., a preceding `dcplrphase`) and step size setting on the first decoupler channel. After `apshaped_decpulse`, first decoupler channel small-angle phase shifting is reset to zero and the step size is set to 0.25 degrees.

`apshaped_decpulse` capability is now integrated into the statement `decshaped_pulse`. The `decshaped_pulse` statement calls `apshaped_decpulse` without table variables if a waveform generator is not configured on the decoupler channel. `decshaped_pulse` creates AP tables on the fly for amplitude and phase, and does not use the AP tables allocated for users. It still uses real-time variables `v12` and `v13`.

Arguments:
- `shape` is a shape file (without the `.RF` extension) in `/vnmr/shapelib` or in `~/vnmrshapelib`. The amplitude and phase fields of the shape file are used. The relative duration field (field 3) should be left at the default value of 1.0 or at least small numbers, and the gate field (field 4) is currently not used because the transmitter is switched on throughout the shape. On **MERCURYplus/-Vx** systems, no phase is changed or set.

- `pulse_width` is the total pulse width, in seconds, excluding the amplifier gating delays around the pulse.

- `pulse_phase` is the 90° phase shift of the pulse. For small-angle phase shifting, note that `apshaped_decpulse` sets the phase step size to the minimum on the one channel that is used.

- `power_table` and `phase_table` are two table variables (t1 to t60) used as intermediate storage addresses for the amplitude and phase tables, respectively. If `apshaped_decpulse` is called more than once, different table names should be used in each call.

- `RG1` is the amplifier gating time, in seconds, before the pulse.

- `RG2` is the amplifier gating time, in seconds, after the pulse.

Examples:

```c
apshaped_decpulse("gauss", pw, v1, rof1, rof2);
```

Related:
- `apshaped_dec2pulse` Second decoupler pulse shaping via the AP bus
- `apshaped_pulse` Observe transmitter pulse shaping via the AP bus
- `dcplrphase` Set small-angle phase of first decoupler,
- `decshaped_pulse` Perform shaped pulse on first decoupler

**apshaped_dec2pulse**

**Second decoupler pulse shaping via AP bus**

Applicability: **UNITY/INOVA** systems.
Syntax:  `apshaped_dec2pulse(shape, pulse_width, pulse_phase, power_table, phase_table, RG1, RG2)`

char *shape; /* name of .RF shape file */
double pulse_width; /* pulse width in sec */
codeint pulse_phase; /* real-time phase of pulse */
codeint power_table; /* table variable to store power */
codeint phase_table; /* table variable to store phase */
double RG1; /* gating time before pulse in sec */
double RG2; /* gating time after pulse in sec */

Description: Provides second decoupler fine-grained “waveform generator-type” pulse shaping through the AP bus. A pulse shape file for the waveform generator (/vnmr/shapelib/*.RF) is used. Note that the real-time variables v12 and v13 are used by this statement. `apshaped_dec2pulse` overrides any existing small-angle phase shifting (i.e., a preceding `dcplr2phase`) and step size setting on the second decoupler channel.

After `apshaped_dec2pulse`, second decoupler channel small-angle phase shifting is reset to zero and the step size is set to 0.25 degrees.

`apshaped_dec2pulse` capability is now integrated into the statement `dec2shaped_pulse`. The `dec2shaped_pulse` statement calls `apshaped_dec2pulse` without table variables if a waveform generator is not configured on the decoupler channel. `dec2shaped_pulse` creates AP tables on the fly for amplitude and phase, and does not use the AP tables allocated for users. It still uses real-time variables v12 and v13.

Arguments: `shape` is a shape file (without the .RF extension) in /vnmr/shapelib or in ~/vnmrsys/shapelib. The amplitude and phase fields of the shape file are used. The relative duration field (field 3) should be left at the default value of 1.0 or at least small numbers, and the gate field (field 4) is currently not used because the transmitter is switched on throughout the shape.

`pulse_width` is the total pulse width, in seconds, excluding the amplifier gating delays around the pulse.

`pulse_phase` is the 90° phase shift of the pulse. For small-angle phase shifting, note that `apshaped_dec2pulse` sets the phase step size to the minimum on the one channel that is used.

`power_table` and `phase_table` are two table variables (t1 to t60) used as intermediate storage addresses for the amplitude and phase tables, respectively. If `apshaped_dec2pulse` is called more than once, different table names should be used in each call.

`RG1` is the amplifier gating time, in seconds, before the pulse.

`RG2` is the amplifier gating time, in seconds, after the pulse.

Examples:  `apshaped_dec2pulse("gauss", pw, v1, t10, t11, rof1, rof2);`

Related:  `apshaped_decpulse` First decoupler pulse shaping via the AP bus
          `apshaped_pulse` Observe transmitter pulse shaping via the AP bus
          `dcplr2phase` Set small-angle phase of 2nd decoupler,
          `dec2shaped_pulse` Perform shaped pulse on second decoupler

**apshaped_pulse** Observe transmitter pulse shaping via AP bus

Applicability:  **UNITY** INOVA systems and **MERCURYplus-Vx**.

**MERCURYplus-Vx** only supports shapes with no phase shifts.
Chapter 3. Pulse Sequence Statement Reference

Syntax: apshaped_pulse(shape,pulse_width,pulse_phase, 
    power_table,phase_table,RG1,RG2)
    char *shape;  /* name of .RF shape file */
    double pulse_width;  /* pulse width in sec */
    codeint pulse_phase;  /* real-time phase of pulse */
    codeint power_table;  /* table variable to store power */
    codeint phase_table;  /* table variable to store phase */
    double RG1;  /* gating time before pulse in sec */
    double RG2;  /* gating time after pulse in sec */

Description: Provides observe transmitter fine-grained “waveform generator-type” pulse 
shaping through the AP bus. A pulse shape file for the waveform generator (/vmnr/shapelib/*.RF) is used. This statement overrides any existing 
small-angle phase shifting (i.e., a preceding xmtrphase) and step size setting 
on the observe transmitter channel. After apshaped_pulse, observe 
transmitter channel small-angle phase shifting is reset to zero and the step size 
is set to 0.25 degrees.
apshaped_pulse capability is now integrated into the shaped_pulse 
statement. The shaped_pulse statement calls apshaped_pulse without 
table variables if a waveform generator is not configured on the decoupler 
channel. shaped_pulse creates AP tables on the fly for amplitude and phase, 
and does not use the AP tables allocated for users. It still uses real-time variables 
v12 and v13.

Arguments: pattern is a shape file (without the .RF extension) in /vmnr/shapelib 
or in ~/vmnrsys/shapelib. The amplitude and phase fields of the shape 
file are used. The relative duration field (field 3) should be left at the default 
value of 1.0 or at least small numbers, and the gate field (field 4) is currently not 
used because the transmitter is switched on throughout the shape. On 
MERCURYplus/-Vx systems, no phase is changed or set.
pulse_width is the total pulse width, in seconds, excluding amplifier gating 
delays around the pulse.
pulse_phase is the 90° phase shift of the pulse. For small-angle phase 
shifting, note that apshaped_pulse sets the phase step size to the minimum 
on the one channel that is used.
power_table and phase_table are two table variables (t1 to t60) used 
as intermediate storage addresses for the amplitude and phase tables, 
respectively. If apshaped_pulse is called more than once, different table 
names should be used in each call.
RG1 is the amplifier gating time, in seconds, before the pulse.
RG2 is the amplifier gating time, in seconds, after the pulse.

Examples: apshaped_pulse("gauss",pw,v1,rof1,rof2);

Related: apshaped_decpulse First decoupler pulse shaping via the AP bus 
apshaped_decpulse Second decoupler pulse shaping via the AP bus 
shaped_pulse Perform shaped pulse on observe transmitter 
xmtrphase Set small-angle phase of observe transmitter, rf C or D

assign Assign integer values

Syntax: assign(vi,vj)
    codeint vi;  /* real-time variable for starting value */
    codeint vj;  /* real-time variable for assigned value */

Description: Sets vj equal to the integer value vi.
Arguments: \( v_i \) and \( v_j \) are real-time variables (\( v_1 \) to \( v_{14} \), \( \text{oph} \), etc.).

Examples: \( \text{assign}(v_3,v_2) \);

Related: 
- `add` Add integer values
- `dbl` Double an integer value
- `decr` Decrement an integer value
- `divn` Divide integer values
- `hlv` Half the value of an integer
- `incr` Increment an integer value
- `mod2` Find integer value modulo 2
- `mod4` Find integer value modulo 4
- `modn` Find integer value modulo \( n \)
- `mult` Multiply integer values
- `sub` Subtract integer values

### B

#### Top

A B C D E G H I L M O P R S T V W X Z

#### blankingoff

Unblank amplifier channels and turn amplifiers on

Applicability: \( MERCURY^{\text{plus/-}}Vx \) systems only.

Syntax: `blankingoff()`

Description: Unblanks, or enables, both amplifier channels.

Related: `blankingon` Blank amplifier channels and turn amplifiers off

#### blankingon

Blank amplifier channels and turn amplifiers off

Applicability: \( MERCURY^{\text{plus/-}}Vx \) systems only.

Syntax: `blankingon()`

Description: Blanks, or disables, both amplifier channels.

Related: `blankingoff` Unblank amplifier channels and turn amplifiers on

#### blankoff

Stop blanking observe or decoupler amplifier (obsolete)

Description: No longer in VnmrJ. The `blankoff` statement is replaced by the statements `obsunblank`, `decunblank`, `dec2unblank`, and `dec3unblank`.

Related: 
- `decunblank` Unblank amplifier associated with first decoupler
- `dec2unblank` Unblank amplifier associated with second decoupler
null
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, 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_delay_list` to create the list when it equals 1 (as shown in the example).

**Arguments:**
- `list` is a pointer to a list of delays.
- `nvals` is the number of values in the list.
- `list_number` –1 or a unique number from 0 to 255 for each list.

**Examples:**
```
pulsesequence()
{
    /* Declare static to save between calls */
    static int list1, list2;
    int i, n;
    double delay1[1024], delay2[1024];

    n = 1024;
    if (ix == 1) {
        for (i=0; i<n; i++) {
            ... /* Initialize delay1 & delay2 arrays */
        }
        /* First, list1 is set to 0 */
        list1 = create_delay_list(delay1,n,0);
        /* This is list #1 */
        create_freq_list(freqs,nfreqs,OBSch,1);
        /* This is list #2 */
        create_offset_list(freqs,nfreqs,OBSch,2);
        /* Next, list2 is set to 3 */
        list2 = create_delay_list(delay2,n,-1);
    }
    ...
    vdelay_list(list2,v5); /* Use v5 from list2 */
    vfreq(1,v2); /* Use v2 from list #1 */
    voffset(2,v1); /* Use v1 from list #2 */
    vdelay_list(list1,v1); /* Use v1 from list1 */
    ...}
```

**Related:**
- `create_freq_list` Create table of frequencies
- `create_offset_list` Create table of frequency offsets
- `delay` Delay for a specified time
- `getarray` Retrieves all values of an arrayed parameter
- `vdelay` Select delay from table
create_freq_list

Create table of frequencies

Applicability: \textit{UNITY/INOVA} systems.

Syntax: 
\begin{verbatim}
create_freq_list(list,nvals,device,list_number)
\end{verbatim}

declare list; /* pointer to list of frequencies */
int nvals; /* number of values in list */
int device; /* OBSch, DECch, DEC2ch, or DEC3ch */
int list_number; /* number 0-255 for each list */
\end{verbatim}

Description: Stores global lists of frequencies that can be accessed with a real-time variable or table element for dynamic setting of frequencies. Frequency lists use frequencies in MHz (such as from \texttt{sfrq}, \texttt{dfrq}). The lists need to be created in order starting from 0 using the \texttt{list_number} argument, or by setting the \texttt{list_number} argument to \texttt{-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 \texttt{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_freq_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 \texttt{-1} in the \texttt{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 \texttt{getarray} statement. To ensure that the list is only created once, check the global array counter variable \texttt{ix}, and only call \texttt{create_freq_list} to create the list when it equals 1. An example is shown in the entry for the \texttt{create_delay_list} statement.

Arguments: 
- \texttt{list} is a pointer to a list of frequencies.
- \texttt{nvals} is the number of values in the list.
- \texttt{device} is OBSch (observe transmitter), DECch (first decoupler), DEC2ch (second decoupler), or DEC3ch (third decoupler).
- \texttt{list_number} is \texttt{-1} or a unique number from 0 to 255 for each list created.

Examples: See the example for the \texttt{create_delay_list} statement.

Related: 
- \texttt{create_delay_list} Create table of delays
- \texttt{create_offset_list} Create table of frequency offsets
- \texttt{getarray} Retrieves all values of an arrayed parameter
- \texttt{delay} Delay for a specified time
- \texttt{vfreq} Select frequency from table

create_offset_list

Create table of frequency offsets

Applicability: \textit{UNITY/INOVA} systems.

Syntax: 
\begin{verbatim}
create_offset_list(list,nvals,device,list_number)
\end{verbatim}

declare 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 */
\end{verbatim}
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.

Arguments: list is a pointer to a list of frequency offsets.

Arguments: list is a pointer to a list of frequency offsets.

Arguments: list is a pointer to a list of frequency offsets.

Arguments: list is a pointer to a list of frequency offsets.

Arguments: list is a pointer to a list of frequency offsets.

Examples: See the example for the create_delay_list statement.

Related: create_delay_list Create table of delays
create_freq_list Create table of frequencies
getarray Retrieves all values of an arrayed parameter
delay Delay for a specified time
voffset Select frequency offset from table

dbl

dcplhrphase

dcplr2phase

dcplr3phase

decblank

Double an integer value
Set small-angle phase of 1st decoupler,
Set small-angle phase of 2nd decoupler,
Set small-angle phase of 3rd decoupler
Blank amplifier associated with first decoupler
Chapter 3. Pulse Sequence Statement Reference

- `dec2blank`: Blank amplifier associated with second decoupler
- `dec3blank`: Blank amplifier associated with third decoupler
- `declvloff`: Return first decoupler back to “normal” power
- `declvlon`: Turn on first decoupler to full power
- `decoff`: Turn off first decoupler
- `dec2off`: Turn off second decoupler
- `dec3off`: Turn off third decoupler
- `decoffset`: Change offset frequency of first decoupler
- `dec2offset`: Change offset frequency of second decoupler
- `dec3offset`: Change offset frequency of third decoupler
- `dec4offset`: Change offset frequency of fourth decoupler
- `decon`: Turn on first decoupler
- `dec2on`: Turn on second decoupler
- `dec3on`: Turn on third 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`: Change first decoupler power level, linear amp. systems
- `dec2power`: Change second decoupler power level, linear amp. systems
- `dec3power`: Change third decoupler power level, linear amp. systems
- `dec4power`: Change fourth decoupler power level, linear amp. systems
- `decprgoff`: End programmable decoupling on first decoupler
- `dec2prgoff`: End programmable decoupling on second decoupler
- `dec3prgoff`: End programmable decoupling on third decoupler
- `decprgon`: Start programmable decoupling on first decoupler
- `dec2prgon`: Start programmable decoupling on second decoupler
- `dec3prgon`: Start programmable decoupling on third decoupler
- `decpulse`: Pulse first decoupler transmitter with amplifier gating
- `decpwr`: Set first decoupler high-power level, class C amplifier
- `decpwrf`: Set first decoupler fine power
- `dec2pwrf`: Set second decoupler fine power
- `dec3pwrf`: Set third decoupler fine power
- `decr`: Decrement an integer value
- `decr2rgpulse`: Pulse first decoupler with amplifier gating
- `decr2rgpulse`: Pulse second decoupler with amplifier gating
- `decr3rgpulse`: Pulse third decoupler with amplifier gating
- `decr4rgpulse`: Pulse fourth decoupler with amplifier gating
- `decshaped_pulse`: Perform shaped pulse on first decoupler
- `dec2shaped_pulse`: Perform shaped pulse on second decoupler
- `dec3shaped_pulse`: Perform shaped pulse on third decoupler
- `decspinlock`: Set spin lock waveform control on first decoupler
- `dec2spinlock`: Set spin lock waveform control on second decoupler
**dbl**

*Double an integer value*

**Syntax:**
```c
dbl(vi, vj)
```
```c
codeint vi;  /* variable for starting value */
codeint vj;  /* variable for twice starting value */
```

**Description:** Sets \( vj \) equal to twice the integer value of \( vi \).

**Arguments:** \( vi \) and \( vj \) are real-time variables (\( v1 \) to \( v14 \), \( cph \), etc.).

**Examples:**
```c
dbl(v1, v2);
```

**Related:**
- **add** Add integer values
- **assign** Assign integer values
- **decr** Decrement an integer value
- **divn** Divide integer values
- **hlv** Half the value of an integer
- **incr** Increment an integer value
- **mod2** Find integer value modulo 2
- **mod4** Find integer value modulo 4
- **modn** Find integer value modulo \( n \)
- **mult** Multiply integer values
- **sub** Subtract integer values

**dcplrphase**

*Set small-angle phase of 1st decoupler*

**Applicability:** Unity/INova systems using a first decoupler with rf type C or D, and MERCURYplus/-Vx.

**Syntax:**
```c
dcplrphase(multiplier)
```
```c
codeint multiplier;  /* real-time phase step multiplier */
```

**Description:** Sets first decoupler phase in step size units set by the \( \text{stepsize} \) statement. The small-angle phaseshift is a product of \( \text{multiplier} \) and the step size. If \( \text{decstepsize} \) has not been used, default step size is 90°.

If the product of the step size set by the \( \text{decstepsize} \) statement and \( \text{multiplier} \) is greater than 90°, the sub-90° part is set by \( \text{dcplrphase} \).
Only on systems with an Output board are carryovers that are multiples of 90° automatically saved and added in at the time of the next 90° phase selection (such as at the time of the next pulse or decpulse). On systems with a Data Acquisition Controller board, a Pulse Sequence Controller board, or an Acquisition Controller board, this is done by dcplrphase (see the description section of the acquire statement for further information about these boards).

Unlike decphase, dcplrphase is needed any time the first decoupler phase shift is to be set to a value not a multiple of 90°. decphase sets quadrature phase shift only, which is rarely needed.

Arguments: multiplier is a small-angle phaseshift multiplier for the first decoupler. The value must be a real-time variable (v1 to v14, oph, etc.) or real-time constant (zero, one, etc.).

Examples: dcplrphase(zero);

Related: dcplr2phase Set small-angle phase of 2nd decoupler
dcplr3phase Set small-angle phase of third decoupler
dec2phase Set quadrature phase of first decoupler
decstepsize Set small-angle phase step size
xmtrphase Set small-angle phase of obs. transmitter

dcplr2phase Set small-angle phase of 2nd decoupler

Applicability: UNITY / NOVA systems using a first decoupler with rf type C or D.

Syntax: dcplr2phase(multiplier);

codeint multiplier; /* real-time phase step multiplier */

Description: Sets second decoupler phase in step size units set by the dec2stepsize statement. The small-angle phaseshift is a product of multiplier and the step size. If dec2stepsize has not been used, the default step size is 90°.

If the product of the step size set by the stepsize statement and multiplier is greater than 90°, the sub-90° part is set by dcplr2phase.

The following apply to UNITY / NOVA systems with the specified hardware:

Output board:
carryovers that are multiples of 90° are automatically saved and added in at the time of the next 90° phase selection (such as at the time of the next pulse or dec2pulse).

Data Acquisition Controller board:
a Pulse Sequence Controller board, or an Acquisition Controller board, this is done by dcplr2phase (see the description section of the acquire statement for further information about these boards).

Unlike dec2phase, dcplr2phase is needed any time the second decoupler phase shift is to be set to a value that is not a multiple of 90°. dec2phase sets quadrature phase shift only, which is rarely need.

Arguments: multiplier is a small-angle phaseshift multiplier for the second decoupler. The value must be a real-time variable (v1 to v14, oph, etc.) or real-time constant (zero, one, etc.).

Examples: dcplr2phase(zero);

Related: dcplrphase Set small-angle phase of first decoupler,
dec2phase Set quadrature phase of second decoupler
**dcplr3phase**  
*Set small-angle phase of 3rd decoupler*

**Applicability:**  
UNITY, INOVA systems using a first decoupler with rf type C or D.

**Syntax:**  
dcplr3phase(multiplier)  
codeint multiplier; /* multiplies phase step */

**Description:**  
Sets the third decoupler phase in units set by the `dec3stepsize` statement. If `dec3stepsize` has not been used, the default step size is 90°. The small-angle phaseshift is a product of `multiplier` and the preset `stepsize`. The full small-angle phase is set by `dcplr3phase`.

Unlike `dec3phase`, `dcplr3phase` is needed any time the third decoupler phase shift is to be set to a value that is not a multiple of 90°. `dec3phase` sets quadrature phase shift only, which is rarely needed.

**Arguments:**  
`multiplier` is a small-angle phaseshift multiplier for the third decoupler. The value must be a real-time variable (v1 to v14, oph, etc.) or real-time constant (zero, one, etc.).

**Examples:**  
dcplr2phase(zero);

**Related:**  
dcplrphase  
dec3phase  
dec3stepsize  
xmtrphase  

---

**decblank**  
*Blank amplifier associated with first decoupler*

**Applicability:**  
UNITY, INOVA systems.

**Syntax:**  
decblank()

**Description:**  
Disables the amplifier for the first decoupler. This is generally used after a call to `decunblank`. See also: “Amplifier Channel Blanking and Unblanking,” page 75.

**Related:**  
decunblank  
obslank  
obsunblank  
rcvroff  
rcvron

---

**dec2blank**  
*Blank amplifier associated with second decoupler*

**Applicability:**  
UNITY, INOVA systems with linear amplifiers.

**Syntax:**  
dec2blank()

**Description:**  
Disables the amplifier for the second decoupler. This is generally used after a call to `dec2unblank`. See also: “Amplifier Channel Blanking and Unblanking,” page 75.

**Related:**  
dec2unblank  
rcvroff  
rcvron
Chapter 3. Pulse Sequence Statement Reference

**dec3blank**  
Blank amplifier associated with third decoupler  
Applicability: Unity/NOVA systems using a third decoupler.  
Syntax: dec3blank()  
Description: Disables the amplifier for the third decoupler. This is generally used after a call to dec3unblank. See also: “Amplifier Channel Blanking and Unblanking,” page 75.  
Related:  
dec3unblank  
Unblank amplifier associated with third decoupler  
rcvroff  
Turn off receiver  
rcvrons  
Turn on receiver

**declvloff**  
Return first decoupler back to “normal” power  
Syntax: declvloff()  
Description: Switches the decoupler power to the power level set by the appropriate parameters defined by the amplifier type: dhp for class C amplifiers or dpwr for linear amplifiers. If dhp='n', declvloff has no effect on systems with class C amplifiers but still functions for systems with linear amplifiers.  
Related:  
dclvlons  
Turn on first decoupler to full power  
power  
Change transmitter or decoupler power, lin. amp. sys.  
pwrfs  
Change transmitter or decoupler fine power  
r潮ler  
Change transmitter or decoupler power, lin. amp. sys.  
r潮f  
Set transmitter or decoupler fine power

**declvlon**  
Turn on first decoupler to full power  
Syntax: declvlon()  
Description: Switches the first decoupler power level between the power level set by the high-power parameter(s) to the full output of the decoupler. If dhp='n', declvlon has no effect on systems with class C amplifiers but still functions for systems with linear amplifiers.  
If declvlon is used, make sure declvloff is used prior to time periods in which normal, controllable power levels are desired, such as prior to acquisition. Use full decoupler power only for decoupler pulses or for solids applications.  
Related:  
dclvlloffs  
Return first decoupler back to “normal” power  
power  
Change transmitter or decoupler power, lin. amp. sys.  
pwrfs  
Change transmitter or decoupler fine power  
r潮ler  
Change transmitter or decoupler power, lin. amp. sys.  
r潮f  
Set transmitter or decoupler fine power

**decoff**  
Turn off first decoupler  
Syntax: decoff()  
Description: Explicitly gates off the first decoupler in the pulse sequence. Amplifier blanking state is unchanged. See also: “Amplifier Channel Blanking and Unblanking,” page 75.  
Related:  
dec  
Turn on first decoupler  
dec2off  
Turn off second decoupler  
dec3off  
Turn off third decoupler
**dec2off**  
*Turn off second decoupler*

**Applicability:** Systems with a second decoupler. Amplifier blanking state is unchanged.

**Syntax:** `dec2off()`

**Description:** Explicitly gates off the second decoupler in the pulse sequence. See also: “Amplifier Channel Blanking and Unblanking,” page 75.

**Related:** `dec2on`  
*Turn on second decoupler*

**dec3off**  
*Turn off third decoupler*

**Applicability:** *UNITY*/*NOVA* systems with a third decoupler. Amplifier blanking state is unchanged.

**Syntax:** `dec3off()`

**Description:** Explicitly gates off the third decoupler in the pulse sequence. See also: “Amplifier Channel Blanking and Unblanking,” page 75.

**Related:** `dec3on`  
*Turn on third decoupler*

**decoffset**  
*Change offset frequency of first decoupler*

**Syntax:** `decoffset(frequency)`

```c
double frequency; /* offset in Hz */
```

**Description:** Changes the offset frequency of the first decoupler (parameter `dof`). It is functionally the same as `offset(frequency, DO1DEV)`.

**Arguments:** `frequency` is the offset frequency desired, in hertz.

**Examples:**

```c
decoffset(dof1);
```

**Related:**

- `dec2offset`  
  *Change offset frequency of second decoupler*

- `dec3offset`  
  *Change offset frequency of third decoupler*

- `obsoffset`  
  *Change offset frequency of observe transmitter*

- `offset`  
  *Change offset frequency of transmitter or decoupler*

**dec2offset**  
*Change offset frequency of second decoupler*

**Syntax:** `dec2offset(frequency)`

```c
double frequency; /* offset frequency in Hz */
```

**Description:** Changes the offset frequency of the second decoupler (parameter `dof2`). It is functionally the same as `offset(frequency, DO2DEV)`.

**Arguments:** `frequency` is the offset frequency desired, in hertz.

**Examples:**

```c
dec2offset(dof2);
```

**Related:**

- `decoffset`  
  *Change offset frequency of first decoupler*

- `dec3offset`  
  *Change offset frequency of third decoupler*

- `obsoffset`  
  *Change offset frequency of observe transmitter*

- `offset`  
  *Change offset frequency of transmitter or decoupler*

**dec3offset**  
*Change offset frequency of third decoupler*

**Syntax:** `dec3offset(frequency)`

```c
double frequency; /* offset frequency in Hz */
```

**Description:**
Description: Changes the offset frequency of the third decoupler (parameter `dof3`). It is functionally the same as `offset(frequency,DO3DEV)`.

Arguments: `frequency` is the offset frequency desired, in hertz.

Examples: `dec3offset(do3);`

Related: `decoffset` Change offset frequency of first decoupler
`dec2offset` Change offset frequency of second decoupler
`obsoffset` Change offset frequency of observe transmitter
`offset` Change offset frequency of transmitter or decoupler

---

**dec4offset**  
**Change offset frequency of fourth decoupler**

Applicability: **UNITY** **INOVA** system with a deuterium decoupler channel as the fourth decoupler.

Syntax: `dec4offset(frequency)`
```c
double frequency; /* offset frequency in Hz */
```

Description: Changes the offset frequency of the fourth decoupler (parameter `dof4`). It is functionally the same as `offset(frequency,DO4DEV)`.

Arguments: `frequency` is the offset frequency desired, in hertz.

Examples: `dec4offset(do4);`

Related: `decoffset` Change offset frequency of first decoupler
`dec2offset` Change offset frequency of second decoupler
`obsoffset` Change offset frequency of observe transmitter
`offset` Change offset frequency of transmitter or decoupler
`rftype` Type of rf generation

---

**decon**  
**Turn on first decoupler**

Syntax: `decon()`

Description: Explicitly gates on the first decoupler in the pulse sequence. First decoupler gating is handled automatically by the statements `declvloff, declvlon, decpulse, decrgpulse, decshaped_pulse, decspinlock, simpulse, sim3pulse, simshaped_pulse, sim3shaped_pulse`. `decprgon` generally needs to be enabled with an explicit `decon` statement and followed by a `decoff` call. Amplifier blanking state is unchanged. See also: “Amplifier Channel Blanking and Unblanking,” page 75.

Related: `decoff` Turn off first decoupler
`dec2on` Turn on second decoupler
`dec3on` Turn on third decoupler

---

**dec2on**  
**Turn on second decoupler**

Applicability: **UNITY** **INOVA** system using a second decoupler.

Syntax: `dec2on()`

Description: Explicitly gates on the second decoupler in the pulse sequence. Second decoupler gating is handled automatically by the statements `dec2rgpulse, dec2shaped_pulse, dec2spinlock, sim3pulse`, and `sim3shaped_pulse`.

Related: `decoff` Turn off second decoupler
`dec2on` Turn on second decoupler
`dec3on` Turn on third decoupler
dec2prgon generally needs to be enabled with an explicit dec2on statement and followed by a dec2off call. Amplifier blanking state is unchanged. See also: “Amplifier Channel Blanking and Unblanking,” page 75.

Related: dec2off  Turn off second decoupler

**dec3on**  
**Turn on third decoupler**

Applicability: **UNITY INOVA** system using a third decoupler.

Syntax: `dec3on()`

Description: Explicitly gates on the third decoupler in the pulse sequence. Third decoupler gating is handled automatically by the statements **dec3rgpulse**, **dec3shaped_pulse**, and **dec3spinlock**

Related: dec3off  Turn off third decoupler

**decphase**  
**Set quadrature phase of first decoupler**

Syntax: `decphase(phase)`

```c

codeint phase; /* real-time variable for quad. phase */
```

Description: Sets quadrature phase (multiple of 90°) for the first decoupler rf. `decphase` is syntactically and functionally equivalent to **txphase** and is useful for a decoupler pulse in all cases where **txphase** is useful for a transmitter pulse.

Arguments: **phase** is the quadrature phase for the first decoupler rf. The value must be a real-time variable (v1 to v14, oph, ct, etc.).

Examples: `decphase(v4);`

Related:  
**dcplphase**  Set small-angle phase of first decoupler,
**dec2phase**  Set quadrature phase of second decoupler
**dec3phase**  Set quadrature phase of third decoupler
**txphase**  Set quadrature phase of observe transmitter

**dec2phase**  
**Set quadrature phase of second decoupler**

Applicability: **UNITY INOVA** system using a second decoupler.

Syntax: `dec2phase(phase)`

```c

codeint phase; /* real-time variable for quad. phase */
```

Description: Sets quadrature phase (multiple of 90°) for the second decoupler rf.

Arguments: **phase** is the quadrature phase for the second decoupler rf. The value must be a real-time variable (v1 to v14, oph, ct, etc.).

Examples: `dec2phase(v9);`

Related:  
**dcpl2phase**  Set small-angle phase of second decoupler,
**decphase**  Set quadrature phase of first decoupler

**dec3phase**  
**Set quadrature phase of third decoupler**

Applicability: **UNITY INOVA** system using a third decoupler.
Chapter 3. Pulse Sequence Statement Reference

Syntax: \texttt{dec3phase(phase)}
\texttt{\hspace{1em}codeint phase; /* real-time variable for quad. phase */}

Description: Sets quadrature phase (multiple of 90°) for the third decoupler rf.

Arguments: \texttt{phase} is the quadrature phase for the third decoupler rf. The value must be a real-time variable (v1 to v14, oph, ct, etc.).

Examples: \texttt{dec3phase(v9)};

Related: \texttt{dcplr3phase} Set small-angle phase of third decoupler,
\texttt{decphase} Set quadrature phase of first decoupler

\textbf{dec4phase} \hspace{1em} Set quadrature phase of fourth decoupler

Applicability: \texttt{UNITY/NOVA} system with a deuterium decoupler channel as the fourth decoupler.

Syntax: \texttt{dec4phase(phase)}
\texttt{\hspace{1em}codeint phase; /* real-time variable for quad. phase */}

Description: Sets quadrature phase (multiple of 90°) for the fourth decoupler rf.

Arguments: \texttt{phase} is the quadrature phase for the third decoupler rf. The value must be a real-time variable (v1 to v14, oph, ct, etc.).

Examples: \texttt{dec4phase(v9)};

Related: \texttt{rf-type} Type of rf generation
\texttt{decphase} Set quadrature phase of first decoupler

\textbf{decpower} \hspace{1em} Change first decoupler power level

Applicability: \texttt{UNITY/NOVA} systems with linear amplifiers.

Syntax: \texttt{decpower(power)}
\texttt{\hspace{1em}double power; /* new power level for DODEV */}

Description: Changes the first decoupler power. It is functionally the same as \texttt{rlpower(value,DODEV)}. See also: “Amplifier Channel Blanking and Unblanking,” page 75.

Arguments: \texttt{power} sets the power level by assuming values from –16 (minimum power) to 79 (maximum power) on channels with a 63-dB attenuator, or from –16 (minimum power) to 63 (maximum power) on channels with a 79-dB attenuator.

\textbf{CAUTION:} Be careful, on systems with linear amplifiers, when using values of \texttt{decpower 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. Use config to set a safety maximum for parameters \texttt{tpwr}, \texttt{dpwr}, \texttt{dpwr2}, and \texttt{dpwr3}.

Related: \texttt{dec2power} Change second decoupler power
\texttt{dec3power} Change third decoupler power
\texttt{obspower} Change observe transmitter power
\texttt{rlpower} Change power level

\textbf{dec2power} \hspace{1em} Change second decoupler power level

Applicability: \texttt{UNITY/NOVA} system using a second decoupler.

Syntax: \texttt{dec2power(power)}
double power;  /* new power level for DO2DEV */

Description: Changes the second decoupler power. It is functionally the same as
\texttt{rlpower}(value,DO2DEV). See also: "Amplifier Channel Blanking and
Unblanking," page 75.

Arguments: \texttt{power} sets the power level by assuming values from 0 (minimum power) to 63
(maximum power) on channels with a 63-dB attenuator, or from –16 (minimum
power) to 63 (maximum power) on channels with a 79-dB attenuator.

Related: \texttt{decpower} Change first decoupler power
\texttt{dec3power} Change third decoupler power
\texttt{obspower} Change observe transmitter power
\texttt{rlpower} Change power level

\texttt{dec3power} \hspace{1cm} \textbf{Change third decoupler power level}

Applicability: \texttt{UNITY/NOVA} system using a third decoupler.

Syntax: \texttt{dec3power(power)}
\begin{verbatim}
  double power;  /* new power level for DO3DEV */
\end{verbatim}

Description: Changes the third decoupler power. It is functionally the same as
\texttt{rlpower}(value,DO3DEV). See also: "Amplifier Channel Blanking and
Unblanking," page 75.

Arguments: \texttt{power} sets the power level by assuming values from 0 (minimum power) to 63
(maximum power) on channels with a 63-dB attenuator, or from –16 (minimum
power) to 63 (maximum power) on channels with a 79-dB attenuator.

Related: \texttt{decpower} Change first decoupler power
\texttt{dec2power} Change second decoupler power
\texttt{obspower} Change observe transmitter power
\texttt{rlpower} Change power level

\texttt{dec4power} \hspace{1cm} \textbf{Change fourth decoupler power level}

Applicability: \texttt{UNITY/NOVA} system with a deuterium decoupler channel as the fourth decoupler.

Syntax: \texttt{dec4power(power)}
\begin{verbatim}
  double power;  /* new power level for DO4DEV */
\end{verbatim}

Description: Changes the third decoupler power. It is functionally the same as
\texttt{rlpower}(value,DO4DEV). See also: "Amplifier Channel Blanking and
Unblanking," page 75.

Arguments: \texttt{power} sets the power level by assuming values from –16 (minimum power) to
63 (maximum power).

Related: \texttt{decpower} Change first decoupler power
\texttt{dec2power} Change second decoupler power
\texttt{obspower} Change observe transmitter power
\texttt{rlpower} Change power level
\texttt{rftype} Type of rf generation

\texttt{decprgoff} \hspace{1cm} \textbf{End programmable decoupling on first decoupler}

Applicability: \texttt{UNITY/NOVA} systems with a waveform generator on rf channel for the first
decoupler.
Chapter 3. Pulse Sequence Statement Reference

decprgoff()  
Description: Terminates any waveform-controlled programmable decoupling on the first decayer started by the decprgon statement. See also: “Amplifier Channel Blanking and Unblanking,” page 75.

Related: decprgon  Start programmable decoupling on first decoupler
dec2prgoff  End programmable decoupling on second decoupler
dec3prgoff  End programmable decoupling on third decoupler

dec2prgoff  End programmable decoupling on second decoupler
Applicability: UNITY/NOVA systems with a waveform generator on rf channel for the second decoupler.
Syntax: dec2prgoff()
Description: Terminates any waveform-generator controlled programmable decoupling on the second decoupler set by the dec2prgon statement. See also: “Amplifier Channel Blanking and Unblanking,” page 75.

Related: dec2prgon  Start programmable decoupling on second decoupler
dec3prgoff  End programmable decoupling on third decoupler

dec3prgoff  End programmable decoupling on third decoupler
Applicability: UNITY/NOVA systems with a waveform generator on rf channel with the third decoupler.
Syntax: dec3prgoff()
Description: Terminates any waveform-generator-controlled programmable decoupling on the third decoupler set by the dec3prgon statement. See also: “Amplifier Channel Blanking and Unblanking,” page 75.

Related: dec3prgon  Start programmable decoupling on third decoupler
decprgon  Start programmable decoupling on first decoupler
Applicability: UNITY/NOVA systems with a waveform generator on rf channel for the first decoupler.
Syntax: decprgon(pattern, 90_pulselength, tipangle_resoln)
char *pattern; /* name of .DEC file */
double 90_pulselength; /* 90-deg pulse length in sec */
double tipangle_resoln; /* tip-angle resolution */
Description: Executes programmable decoupling on the first decoupler under waveform generator control, and returns the number of 12.5-ns ticks (as an integer value) in one cycle of the decoupling pattern. Explicit gating of the first decoupler with decon and decoff is generally required. Arguments can be variables (which require the appropriate getval and getstr statements) to permit changes by the parameters (see the second example). See also: “Amplifier Channel Blanking and Unblanking,” page 75.

Arguments: pattern is the name of the text file in the shapelib directory that stores the decoupling pattern (leave off the .DEC file extension).
90_pulselength is the pulse duration, in seconds, for a 90° tip angle on the first decoupler.
tipangle_resoln is the resolution, in tip-angle degrees, of the decoupling pattern stored in the waveform generator.

Examples:
```
decprgon("garp1",1/dmf, 1.0);
decprgon(modtype,pwx90,dres);
ticks = decprgon("waltz16",1/dmf,90.0);
```

Related:
- decprgonoff: End programmable decoupling on first decoupler
- dec2prgon: Start programmable decoupling on second decoupler
- dec3prgon: Start programmable decoupling on third decoupler
- obsprgon: Start programmable control of obs. transmitter

**dec2prgon**  
**Start programmable decoupling on second decoupler**

**Applicability:** UNITY/NOVA systems with a waveform generator on rf channel for the second decoupler.

**Syntax:**
```
dec2prgon(pattern,90_pulselength,tipangle_resoln)
```

```c
char *pattern; /* name of .DEC text file */
double 90_pulselength; /* 90° pulse length in sec */
double tipangle_resoln; /* tip-angle resolution */
```

**Description:** Executes programmable decoupling on second decoupler under waveform generator control, and returns the number of 12.5-ns ticks (as an integer value) in one cycle of the decoupling pattern. Explicit gating of the second decoupler with `dec2on` and `dec2off` is generally required. Arguments can be variables (which require the appropriate `getval` and `getstr` statements) to permit changes by the parameters (see the second example). See also: “Amplifier Channel Blanking and Unblanking,” page 75.

**Arguments:**
- `pattern` is the name of the text file in the `shapelib` directory that stores the decoupling pattern (leave off the .DEC file extension).
- `90_pulselength` is the pulse duration, in seconds, for a 90° tip angle on the second decoupler.
- `tipangle_resoln` is the resolution, in tip-angle degrees, to which the decoupling pattern is stored in the waveform.

**Examples:**
```
(1) dec2prgon("waltz16",1/dmf2,90.0);
(2) dec2prgon(modtype,pwx290,dres2);
    ticks=dec2prgon("garp1",1/dmf2,1.0);
```

Related:
- decprgon: Start programmable decoupling on first decoupler
- dec3prgonoff: End programmable decoupling on third decoupler
- obsprgon: Start programmable control of obs. transmitter

**dec3prgon**  
**Start programmable decoupling on third decoupler**

**Applicability:** UNITY/NOVA systems with a waveform generator on rf channel for the third decoupler.

**Syntax:**
```
dec3prgon(pattern,90_pulselength,tipangle_resoln)
```

```c
char *pattern; /* name of .DEC text file */
double 90_pulselength; /* 90° pulse length in sec */
double tipangle_resoln; /* tip-angle resolution */
```

**Description:** Executes programmable decoupling on third decoupler under waveform control. It returns the number of 12.5-ns ticks (as an integer value) in one cycle
Chapter 3. Pulse Sequence Statement Reference

of the decoupling pattern. Explicit gating of the third decoupler with dec3on and dec3off is generally required. Arguments can be variables (which require the appropriate getval and getstr statements) to permit changes by parameters (see second example). See also: “Amplifier Channel Blanking and Unblanking,” page 75.

Arguments: pattern is the name of the text file in the shapelib directory that stores the decoupling pattern (leave off the .DEC file extension).

90_pulse_length is the pulse duration, in seconds, for a 90° tip angle on the third decoupler.

tip_angle_resoln is the resolution, in tip-angle degrees, to which the decoupling pattern is stored in the waveform.

Examples: (1) dec3prgon("waltz16",1/dmf3,90.0);
(2) dec3prgon(modtype,pwx390,dres3);
    ticks = dec3prgon("garp1",1/dmf3,1.0);

Related: dec3prgon Start programmable decoupling on third decoupler
dec2prgonoff End programmable decoupling on second decoupler
decclear Start programmable control of obs. transmitter

decpulse Pulse first decoupler transmitter with amplifier gating

Syntax: decpulse(width,phase)

double width; /* width of pulse in sec */

codeint phase; /* real-time variable for phase of pulse */

Description: Pulses the first decoupler at its current power level. The amplifier is gated off during decoupler pulses as it is during observe pulses. The amplifier gating times (see RG1 and RG2 for decrgpulse) are internally set to zero for this statement. dm should be set to 'c' during any period of time in which decoupler pulses occur. See also: “Amplifier Channel Blanking and Unblanking,” page 75.

Arguments: width is the duration of the pulse, in seconds.

phase is the phase of the pulse. The value must be a real-time variable (v1 to v14, etc.) or a real-time constant (zero, one, etc.).

Examples: decpulse(pp,v3);
          decpulse(2.0*pp,zero);

Related: decrgpulse Pulse decoupler transistor with amplifier gating
         rgpulse Pulse observe transistor with amplifier gating
         simpulse Pulse observe, decoupler channels simultaneously
         sim3pulse Simultaneous pulse on 2 or 3 rf channels

decpwr Set first decoupler high-power level, class C amplifier

Applicability: All systems with class C amplifiers.

Syntax: decpwr(level)

double level; /* new power level for DODEV channel */

Description: Changes the first decoupler high-power level to the value specified. To reset the power back to the “standard” dhp level, use decpwr(dhp).

Switching between low power decoupling (dhp='n') and high power decoupling (dhp='x'), as well as switching between different levels of low
power decoupling, uses relays whose switching time is about 10 ms and are not provided for in the standard pulse sequence capability. Neither function should prove necessary because extremely low levels of decoupling are provided for in dhp mode by using very small (0 to 30) values of dhp.

Arguments: level specifies the decoupler high-power level, from 0 (lowest) to 255 (full power). These values in this range increase monotonically but are neither linear nor logarithmic.

Examples: decpwr(255.0);
decpwr(level1);

deCPWRf  Set first decoupler fine power

Applicability: "UNITY/INOVA" systems with fine power control on the first decoupler.

Syntax: decpwr(power)
        double power; /* new fine power value for DODEV */

Description: Changes first decoupler fine power. It is functionally the same as rlpwr(value,DECch). See also: "Amplifier Channel blanking and Unblanking," page 75.

Arguments: power is the fine power desired.

Examples: decpwr(4.0);

Related: decpwrf  Set first decoupler fine power
dec3pwrf  Set third decoupler fine power
obspwrf  Set observe transmitter fine power
rlpwrf  Set transmitter or decoupler fine power

deC2PWRF  Set second decoupler fine power

Applicability: "UNITY/INOVA" systems with fine power control on the second decoupler.

Syntax: dec2pwrf(power)
       double power; /* new fine power value for DO2DEV */

Description: Changes the second decoupler fine power. It is functionally the same as rlpwr(value,DEC2ch). See also: "Amplifier Channel blanking and Unblanking," page 75.

Arguments: power is the fine power desired.

Examples: dec2pwrf(4.0);

Related: decpwrf  Set first decoupler fine power
dec3pwrf  Set third decoupler fine power
obspwrf  Set observe transmitter fine power
rlpwrf  Set transmitter or decoupler fine power

deC3PWRF  Set third decoupler fine power

Applicability: "UNITY/INOVA" systems with fine power control on the third decoupler.

Syntax: dec3pwrf(power)
       double power; /* new fine power value for DO3DEV */

Description: Changes third decoupler fine power. It is functionally the same as rlpwr(value,DEC3ch). See also: "Amplifier Channel blanking and Unblanking," page 75.
Chapter 3. Pulse Sequence Statement Reference

Arguments: `power` is the fine power desired.

Examples: `decpwrf(4.0);`

Related: `decpwrf` Set first decoupler fine power
        `dec2pwrf` Set second decoupler fine power
        `obspwrf` Set observe transmitter fine power
        `rlpwrf` Set transmitter or decoupler fine power

decr  **Decrement an integer value**

Syntax: `decr(vi)`

codeint vi; /* real-time variable for starting value */

Description: Decrements integer value `vi` by 1 (i.e., `vi=vi-1`).

Arguments: `vi` is a real-time variable (`v1` to `v14`, oph, etc.).

Examples: `decr(v5);`

Related: `add` Add integer values
        `assign` Assign integer values
        `dbl` Double an integer value
        `divn` Divide integer values
        `hlv` Half the value of an integer
        `incr` Increment an integer value
        `mod2` Find integer value modulo 2
        `mod4` Find integer value modulo 4
        `modn` Find integer value modulo `n`
        `mult` Multiply integer values
        `sub` Subtract integer values

decrgpulse  **Pulse first decoupler with amplifier gating**

Syntax: `decrgpulse(width,phase,RG1,RG2)`

double width; /* width of pulse in sec */

codeint phase; /* real-time variable for phase */

double RG1; /* gating delay before pulse in sec */

double RG2; /* gating delay after pulse in sec */

Description: Syntactically equivalent to `rgpulse` statement and functionally equivalent to `rgpulse` with two exceptions. First, the first decoupler (instead of the transmitter) is pulsed at its current power level. Second, if `homo='n'`, the slow gate on the first decoupler board is always open and therefore need not be switched open during `RG1`. In contrast, if `homo='y'`, the slow gate on the first decoupler board is normally closed and must therefore be allowed sufficient time during `RG1` to switch open.

For systems with linear amplifiers, `RG1` for a decoupler pulse is important from the standpoint of amplifier stabilization under the following conditions: `tn`, `dn` equal `\{^{1}H, ^{1}H, ^{19}F\}` (high-band nuclei, `^{3}H` does not apply to `MERCURYplus/-Vx` systems), or `tn`, `dn` less than or equal to `^{31}P` (low-band nuclei). For these conditions, the "decoupler" amplifier module is placed in pulse mode, in which it remains blanked as long as the receiver is on. In this mode, `RG1` must be sufficiently long to allow the amplifier to stabilize after blanking is removed: 5 to 10 μs (2 μs typical for `MERCURYplus/-Vx`) for high-band nuclei and 10 to 20 μs (2 μs typical for `MERCURYplus/-Vx`) for low-band nuclei. Solids require at
least 1.5 μs. On 500-MHz systems that use the ENI-5100 class A amplifier for low-band nuclei on the observe channel, $RG1$ should be 40–60 μs.

If the $tn$ nucleus and the $dn$ nucleus are in different bands (e.g., $tn$ is $^1H$ and $dn$ is $^{13}C$), the “decoupler” amplifier module is placed in the $cw$ mode, in which it is always unblanked regardless of the state of the state of the receiver. In this mode $RG1$ is unimportant with respect to amplifier stabilization prior to the decoupler pulse.

Arguments:
- $\text{width}$ is the duration, in seconds, of the decoupler transmitter pulse.
- $\text{phase}$ is the phase of the pulse. It must be a real-time variable ($v1$ to $v14$, etc.) or a real-time constant ($\text{zero}$, $\text{one}$, etc.).
- $RG1$ is the time, in seconds, before the start of the pulse that the amplifier is gated off.
- $RG2$ is the time, in seconds, after the end of the pulse that the amplifier is gated on.

Examples:
- $\text{decrgpulse(}pp, v3, \text{rof1, rof2)}$;
- $\text{decrgpulse(}pp, \text{zero, 1.0e-6, 0.2e-6)}$;

Related:
- $\text{decpulse}$ Pulse first decoupler with amplifier gating
- $\text{dec2rgpulse}$ Pulse second decoupler with amplifier gating
- $\text{dec3rgpulse}$ Pulse third decoupler with amplifier gating
- $\text{idecpulse}$ Pulse first decoupler transmitter with IPA
- $\text{idecrgpulse}$ Pulse first decoupler with amplifier gating and IPA
- $\text{irrgpulse}$ Pulse observe transmitter with IPA
- $\text{rgpulse}$ Pulse observe transmitter with amplifier gating
- $\text{simpulse}$ Pulse observe, decoupler channels simultaneously
- $\text{sim3pulse}$ Simultaneous pulse on 2 or 3 rf channels

$\text{dec2rgpulse}$ Pulse second decoupler with amplifier gating

Applicability: unity $\text{INOVA}$ system with a second decoupler.

Syntax:
- $\text{dec2rgpulse(width, phase, RG1, RG2)}$

  - double $\text{width}$; /* width of pulse in sec */
  - codeint $\text{phase}$; /* real-time variable for phase */
  - double $\text{RG1}$; /* gating delay before pulse in sec */
  - double $\text{RG2}$; /* gating delay after pulse in sec */

Description: Performs an explicit amplifier-gated pulse on the second decoupler (DES2ch).

Arguments:
- $\text{width}$ is the duration, in seconds, of the pulse.
- $\text{phase}$ is the phase of the pulse. It must be a real-time variable ($v1$ to $v14$, etc.) or a real-time constant ($\text{zero}$, $\text{one}$, etc.).
- $RG1$ is the delay, in seconds, between gating the amplifier on and gating the rf transmitter on (the phaseshift occurs at the beginning of this delay). $RG1$ is important for amplifier stabilization under the same conditions as described for $\text{decrgpulse}$.
- $RG2$ is the delay, in seconds, between gating the rf transmitter off and gating the amplifier off. $\text{homo}$ has no effect on the gating on the second decoupler board. $\text{homo2}$ controls gating of second decoupler rf.

Examples:
- $\text{dec2rgpulse(p1, v10, rof1, rof2)}$;
dec3rgpulse  Pulse third decoupler with amplifier gating

Applicability:  

UNITY/INOVA systems with a third decoupler.

Syntax:  

dec3rgpulse(width,phase,RG1,RG2)

double width; /* width of pulse in sec */
codeint phase; /* real-time variable for phase */
double RG1; /* gating delay before pulse in sec */
double RG2; /* gating delay after pulse in sec */

Description: Performs an explicit amplifier-gated pulse on the third decoupler (DEC3ch).

Arguments: width is the duration, in seconds, of the pulse.

phase is the phase of the pulse. It must be a real-time variable (v1 to v14, etc.) or a real-time constant (zero, one, etc.).

RG1 is the delay, in seconds, between gating the amplifier on and gating the rf transmitter on (the phaseshift occurs at the beginning of this delay). RG1 is important for amplifier stabilization under the same conditions as described for decrgpulse.

RG2 is the delay, in seconds, between gating the rf transmitter off and gating the amplifier off. homo has no effect on the gating on the third decoupler board. homo3 controls gating of third decoupler rf.

Examples:  
dec3rgpulse(p1,v10,rof1,rof2);

Related:  

decpulse  Pulse first decoupler with amplifier gating

decrgpulse  Pulse first decoupler with amplifier gating

idecpulse Pulse first decoupler with IPA

rgrpulse  Pulse observe transmitter with amplifier gating

simpulse  Pulse observe, decoupler channels simultaneously

sim3pulse Simultaneous pulse on 2 or 3 rf channels

dec4rgpulse  Pulse fourth decoupler with amplifier gating

Applicability:  

UNITY/INOVA systems with a deuterium decoupler channel as the fourth decoupler.

Syntax:  

dec4rgpulse(width,phase,RG1,RG2)

double width; /* width of pulse in sec */
codeint phase; /* real-time variable for phase */
double RG1; /* gating delay before pulse in sec */
double RG2; /* gating delay after pulse in sec */

Description: Performs an explicit amplifier-gated pulse on the fourth decoupler (DEC4ch).

Arguments: width is the duration, in seconds, of the pulse.

phase is the phase of the pulse. It must be a real-time variable (v1 to v14, etc.) or a real-time constant (zero, one, etc.).

RG1 is the delay, in seconds, between gating the amplifier on and gating the rf transmitter on (the phaseshift occurs at the beginning of this delay). RG1 is
important for amplifier stabilization under the same conditions as described for decrgpulse.

RG2 is the delay, in seconds, between gating the rf transmitter off and gating the amplifier off.

Examples: dec4rgpulse(p1,v10,rof1,rof2);

Related: decpulse  Pulse first decoupler with amplifier gating
decrgpulse  Pulse first decoupler with amplifier gating
rgpulse  Pulse observe transmitter with amplifier gating
simpulse  Pulse observe, decoupler channels simultaneously
sim3pulse  Simultaneous pulse on 2 or 3 rf channels

**decshaped_pulse**  Perform shaped pulse on first decoupler

Applicability: UNITY/INOVA systems with waveform generator on rf channel for the first decoupler.

Syntax: decshaped_pulse(pattern,width,phase,RG1,RG2)
char *pattern;  /* name of .RF text file */
double width;  /* width of pulse in sec */
codeint phase;  /* real-time variable for phase */
double RG1;  /* gating delay before pulse in sec */
double RG2;  /* gating delay after pulse in sec */

Description: Performs a shaped pulse on the first decoupler. If a waveform generator is configured on the channel, it is used; otherwise, the linear attenuator and the small-angle phase shifter are used to effectively perform an apshaped_decpulse statement.

When using the waveform generator, the shapes are downloaded into the waveshaper before the start of an experiment. When decshaped_pulse is called, the shape is addressed and started. The minimum pulse length and stepsize is 50 ns. The overhead at the start and end of the shaped pulse varies:

- **UNITY**: 1 μs (start), 0 (end)
- System with Acquisition Controller board: 10.75 μs (start), 4.3 μs (end)
- System with Output board: 10.95 μs (start), 4.5 μs (end)

INOVA: If the length is less than 50 ns, the pulse is not executed and there is no overhead.

When using the linear attenuator and the small-angle phase shifter to generate a shaped pulse, the decshaped_pulse statement creates AP tables on the fly for amplitude and phase. *It also uses the real-time variables v12 and v13 to control the execution of the shape.* It does not use AP table variables. For timing and more information, see the description of apshaped_decpulse. Note that if using AP tables with shapes that have a large number of points, the FIFO can become overloaded with words generating the pulse shape and FIFO Underflow errors can result.

Arguments: pattern is the name of a text file in the shapelib directory that stores the rf pattern (leave off the .RF file extension).

width is the duration, in seconds, of the pulse.

phase is the phase of the pulse. It must be a real-time variable (v1 to v14, etc.) or a real-time constant (zero, one, etc.)
RG1 is the delay, in seconds, between gating the amplifier on and gating the first decoupler on (the phaseshift occurs at the beginning of this delay).

RG2 is the delay, in seconds, between gating the first decoupler off and gating the amplifier off.

Examples: `decshaped_pulse("sinc",p1,v5,rof1,rof2);`

Related:
- `dec2shaped_pulse` Perform shaped pulse on second decoupler
- `dec3shaped_pulse` Perform shaped pulse on third decoupler
- `shaped_pulse` Perform shaped pulse on observe transmitter
- `simshaped_pulse` Simultaneous two-pulse shaped pulse
- `sim3shaped_pulse` Simultaneous three-pulse shaped pulse

### `dec2shaped_pulse`
Perform shaped pulse on second decoupler

**Applicability:** UNITY/INOVA systems with waveform generator on rf channel for the second decoupler.

**Syntax:**
```
dec2shaped_pulse(pattern,width,phase,RG1,RG2)
```
- `char *pattern;` /* name of .RF text file */
- `double width;` /* width of pulse in sec */
- `codeint phase;` /* real-time variable for phase */
- `double RG1;` /* gating delay before pulse in sec */
- `double RG2;` /* gating delay after pulse in sec */

**Description:** Performs a shaped pulse on the second decoupler. If a waveform generator is configured on the channel, it is used; otherwise, the linear attenuator and the small-angle phase shifter are used to effectively perform an `apshaped_dec2pulse` statement.

When using the waveform generator, the shapes are downloaded into the waveshaper before the start of an experiment. When `dec2shaped_pulse` is called, the shape is addressed and started. The minimum pulse length and stepsize is 50 ns. The overhead at the start and end of the shaped pulse varies:

- **UNITY/INOVA:** 1 μs (start), 0 (end)
- **System with Acquisition Controller board:** 10.75 μs (start), 4.3 μs (end)
- **System with Output board:** 10.95 μs (start), 4.5 μs (end)

If the length is less than 50 ns, the pulse is not executed and there is no overhead.

When using the linear attenuator and the small-angle phase shifter to generate a shaped pulse, the `dec2shaped_pulse` statement creates AP tables on the fly for amplitude and phase. It also uses the real-time variables v12 and v13 to control the execution of the shape. It does not use AP table variables. For timing and more information, see the description of `apshaped_dec2pulse`. Note that if using AP tables with shapes that have a large number of points, the FIFO can become overloaded with words generating the pulse shape and FIFO Underflow errors can result.

**Arguments:**
- `pattern` is the name of a text file in the shapelib directory that stores the rf pattern (leave off the .RF file extension).
- `width` is the duration, in seconds, of the pulse.
- `phase` is the phase of the pulse. It must be a real-time variable (v1 to v14, etc.) or a real-time constant (zero, one, etc.)
- `RG1` is the delay, in seconds, between gating the amplifier on and gating the second decoupler on (the phaseshift occurs at the beginning of this delay).
RG2 is the delay, in seconds, between gating the second decoupler off and gating the amplifier off.

Examples: \texttt{dec2shaped\_pulse("gauss",p1,v9,rof1,rof2);}

Related: \texttt{decshaped\_pulse} Perform shaped pulse on first decoupler
\texttt{shaped\_pulse} Perform shaped pulse on observe transmitter
\texttt{sim3shaped\_pulse} Simultaneous three-pulse shaped pulse

\textbf{dec3shaped\_pulse} \hspace{1cm} \textbf{Perform shaped pulse on third decoupler}

\textbf{Applicability:} UNITY/INOVA systems with waveform generator on rf channel for the third decoupler

\textbf{Syntax:} \texttt{dec3shaped\_pulse(pattern,width,phase,RG1,RG2)}
\begin{verbatim}
  char *pattern; /* name of .RF text file */
  double width; /* width of pulse in sec */
  codeint phase; /* real-time variable for phase */
  double RG1; /* gating delay before pulse in sec */
  double RG2; /* gating delay after pulse in sec */
\end{verbatim}

\textbf{Description:} Performs a shaped pulse on the third decoupler. If a waveform generator is configured on the channel, it is used; otherwise, the linear attenuator and the small-angle phase shifter are used to effectively perform an \texttt{apshaped\_dec3pulse} statement.

The shapes are downloaded into the controller before the start of an experiment. When \texttt{dec3shaped\_pulse} is called, the shape is addressed and started. The minimum pulse length and stepsize is 50 ns. The overhead at the start and end of the shaped pulse varies:

- \texttt{UNITY/INOVA}: 1 \textmu s (start), 0 (end)
- System with Acquisition Controller board: 10.75 \textmu s (start), 4.3 \textmu s (end)
- System with Output board: 10.95 \textmu s (start), 4.5 \textmu s (end)

If the length is less than 50 ns, the pulse is not executed and there is no overhead.

When using the linear attenuator and the small-angle phase shifter to generate a shaped pulse, the \texttt{dec3shaped\_pulse} statement creates AP tables on the fly for amplitude and phase. \textit{It also uses the real-time variables v12 and v13 to control the execution of the shape.} It does not use AP table variables. For timing and more information, see the description of \texttt{apshaped\_dec3pulse}. Note that if using AP tables with shapes that have a large number of points, the FIFO can become overloaded with words generating the pulse shape and FIFO Underflow errors can result.

\textbf{Arguments:}
- \texttt{pattern} is the name of a text file in the shapelib directory that stores the rf pattern (leave off the .RF file extension).
- \texttt{width} is the duration, in seconds, of the pulse.
- \texttt{phase} is the phase of the pulse. It must be a real-time variable (v1 to v14, etc.) or a real-time constant (zero, one, etc.).
- \texttt{RG1} is the delay, in seconds, between gating the amplifier on and gating the third decoupler on (the phaseshift occurs at the beginning of this delay).
- \texttt{RG2} is the delay, in seconds, between gating the third decoupler off and gating the amplifier off.

Examples: \texttt{dec3shaped\_pulse("gauss",p1,v9,rof1,rof2);}
decspinlock  Set spin lock waveform control on first decoupler

Applicability: UNITY/NOVA systems with waveform generator on rf channel for the 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; /* phase of spin lock */
int ncylces; /* number of cycles to execute */

Description: Executes a waveform-controlled spin lock on the first decoupler, handling both rf gating and the mixing delay. Arguments can be variables (which require the appropriate getval and getstr statements) to permit changes via parameters (see the second example).

Arguments: pattern is the name of the text file in the shapelib directory that stores the decoupling pattern (leave off the .DEC file extension).

90_pulselength is the pulse duration, in seconds, for a 90° tip angle.

tipangle_resoln is the resolution, in tip-angle degrees, to which the decoupling pattern is stored in the waveform generator.

phase is the phase of the spin lock. It must be a real-time variable (v1 to v14, etc.) or a real-time constant (zero, one, etc.).

ncycles is the number of times the spin-lock pattern is to be executed.

Examples: decspinlock("mlev16", p190, dres, v1, 30);
decspinlock(spinlk, pp90, dres, v1, cycles);

Related: dec2spinlock  Set spin lock waveform control on second decoupler
dec3spinlock  Set spin lock waveform control on third decoupler
spinlock  Set spin lock waveform control on obs. transmitter

dec2spinlock  Set spin lock waveform control on second decoupler

Applicability: UNITY/NOVA systems with waveform generator on rf channel for the 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 of channel */
double tipangle_resoln; /* resolution of tip angle */
codeint phase; /* phase of spin lock */
int ncylces; /* number of cycles to execute */

Description: Executes a waveform-controlled spin lock on the second decoupler. Both the rf gating and the mixing delay are handled within this function. Arguments can be variables (which require the appropriate getval and getstr statements) to permit changes via parameters (see the second example).

Arguments: pattern is the name of the text file in the shapelib directory that stores the decoupling pattern (leave off the .DEC file extension).
90_pulselength is the pulse duration, in seconds, for a 90° tip angle.
tipangle_resoln is the resolution, in tip-angle degrees, to which the
decoupling pattern is stored in the waveform generator.
phase is the phase of the spin lock. It must be a real-time variable (v1 to v14,
etc.) or a real-time constant (zero, one, etc.).
ncycles is the number of times that the spin-lock pattern is to be executed.

Examples: (1) dec2spinlock("mlev16",p290,dres2,v1,42);
(2) dec2spinlock(lock2,pwx2,dres2,v1,cycles);

Related:  decspinlock  Set spin lock waveform control on first decoupler
         spinlock               Set spin lock waveform control on obs. transmitter

dec3spinlock  Set spin lock waveform control on third decoupler
Applicability:  UNITY/INOVA systems with waveform generator on rf channel for the 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 of channel */
double tipangle_resoln; /* resolution of tip angle */
codeint phase; /* phase of spin lock */
int ncyycles; /* number of cycles to execute */

Description: Executes a waveform-controlled spin lock on the third decoupler. Both the rf
gating and the mixing delay are handled within this function. Arguments can be
variables (which would need the appropriate getval and getstr statements)
to permit changes via parameters (see the second example).

Arguments: pattern is the name of the text file in the shapelib directory that stores the
decoupling pattern (leave off the .DEC file extension).
90_pulselength is the pulse duration, in seconds, for a 90° tip angle.
tipangle_resoln is the resolution in tip-angle degrees to which the
decoupling pattern is stored in the waveform generator.
phase is the phase of the spin lock. It must be a real-time variable (v1 to v14,
etc.) or a real-time constant (zero, one, etc.).
ncycles is the number of times that the spin-lock pattern is to be executed.

Examples:  dec3spinlock("mlev16",p390,dres3,v1,42);
           dec3spinlock(lock2,pwx2,dres3,v1,cycles);

Related:  decspinlock  Set spin lock waveform control on first decoupler
         spinlock               Set spin lock waveform control on obs. transmitter

decstepsize  Set step size for first decoupler
Syntax:  decstepsize(step_size)
  double step_size;  /* phase step size */

Description: Sets the step size of the first decoupler. It is functionally the same as
stepszie(base,DECch).

Arguments: step_size is the phase step size desired and is a real number or a variable.

Examples: decstepsize(30.0);
Chapter 3. Pulse Sequence Statement Reference

Related: **dec2stepsize**  Set step size of second decoupler  
**dec3stepsize**  Set step size of third decoupler  
**obsstepsize**  Set step size of observe transmitter  
**stepsize**  Set small-angle phase step size,

**dec2stepsize**  Set step size for second decoupler  

**Applicability:** UNITY/INOVA system with a second decoupler.  
**Syntax:** 
```c
dec2stepsize(step_size)
double step_size;  /* phase step size */
```
**Description:** Sets the step size of the first decoupler. This statement is functionally the same as `stepsize(base,DEC2ch)`.  
**Arguments:** `step_size` is the phase step size desired and is a real number or a variable.  
**Examples:** 
```c
dec2stepsize(30.0);
```

**Related:**  
**decstepsize**  Set step size of first decoupler  
**dec3stepsize**  Set step size of third decoupler  
**obsstepsize**  Set step size of observe transmitter  
**stepsize**  Set small-angle phase step size,

**dec3stepsize**  Set step size for third decoupler  

**Applicability:** UNITY/INOVA system with a third decoupler.  
**Syntax:** 
```c
dec3stepsize(step_size)
double step_size;  /* phase step size */
```
**Description:** Sets the step size of the third decoupler. This statement is functionally the same as `stepsize(base,DEC3ch)`.  
**Arguments:** `step_size` is the phase step size desired and is a real number or a variable.  
**Examples:** 
```c
dec3stepsize(30.0);
```

**Related:**  
**decstepsize**  Set step size of first decoupler  
**dec2stepsize**  Set step size of second decoupler  
**obsstepsize**  Set step size of observe transmitter  
**stepsize**  Set small-angle phase step size,

**decunblank**  Unblank amplifier associated with first decoupler  

**Applicability:** UNITY/INOVA systems.  
**Syntax:** `decunblank()`  
**Description:** Explicitly enables the amplifier for the first decoupler. This overwrites the implicit blanking and unblanking of the amplifier before and after pulses. `decunblank` is generally followed by a call to `decblank`.  

**Related:**  
**decblank**  Blank amplifier associated with first decoupler  
**obsblank**  Blank amplifier associated with observe transmitter  
**obsunblank**  Unblank amplifier associated with observe transmitter  
**rcvroff**  Turn off receiver  
**rcvron**  Turn on receiver
**dec2unblank**  Unblank amplifier associated with second decoupler

Applicability: UNITY/NOVA systems with a second decoupler.

Syntax: `dec2unblank()`

Description: Explicitly enables the amplifier for the second decoupler. This overwrites the implicit blanking and unblanking of the amplifier before and after pulses. `dec2unblank` is generally followed by a call to `dec2blank`.

Related: `dec2blank`  Blank amplifier associated with second decoupler
`rcvroff`  Turn off receiver
`rcvron`  Turn on receiver

**dec3unblank**  Unblank amplifier associated with third decoupler

Applicability: UNITY/NOVA systems with a third decoupler.

Syntax: `dec3unblank()`

Description: Explicitly enables the amplifier for the third decoupler. This overwrites the implicit blanking and unblanking of the amplifier before and after pulses. `dec3unblank` is generally followed by a call to `dec3blank`.

Related: `dec3blank`  Blank amplifier associated with third decoupler
`rcvroff`  Turn off receiver
`rcvron`  Turn on receiver

**delay**  Delay for a specified time

Applicability: UNITY/NOVA systems with class C amplifiers.

Syntax: `delay(time)`

```
double time;       /* delay in sec */
```

Description: Sets a delay for a specified number of seconds.

Arguments: `time` specifies the delay, in seconds (minimum of 50 ns, minimum increment 12.5 ns)

Examples: `delay(d1);`
          `delay(d2/2.0);`

Related: `dps_show`  Draw delay or pulses in a sequence for graphical display
`hsdelay`  Delay specified time with possible homospoil pulse
`incdelay`  Real time incremental delay
`initdelay`  Initialize incremental delay
`vdelay`  Delay with fixed timebase and real time count

**dhpflag**  Switch decoupling from low-power to high-power

Applicability: All systems with class C amplifiers.

Syntax: `dhpflag`

Description: Switches the system from low-power to high-power decoupling; e.g.,
`dhpflag=TRUE` (correct use of upper and lower case letters is necessary).

Values: `TRUE`; switches the system to high-power decoupling.
`FALSE`; switches the system to low-power decoupling.

Related: `status`  Draw delay or pulses in a sequence for graphical display
### divn

**Divide 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:** Sets the integer value `vk` equal to `vi` divided by `vj`. Any remainder is ignored.

**Arguments:** `vi` is the dividend, `vj` is the divisor, and `vk` is the quotient. All three are real-time variables (`v1` to `v14`, `oph`, etc.).

**Examples:**
```
divn(v2,v3,v4);
```
char *label; /* text label selected by user */
double width; /* pulse length in sec */

Syntax: (4) dps_show("simpulse",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 in sec */
char *label_of_dec; /* text label selected by user */
double width_of_dec; /* pulse length in sec */

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 in sec */
char *label_of_dec; /* text label selected by user */
double width_of_dec; /* pulse length in sec */

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 in sec */
char *label_of_dec; /* text label selected by user */
double width_of_dec; /* pulse length in sec */
char *label_of_dec2; /* text label selected by user */
double width_of_dec2; /* pulse length in sec */

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 in sec */
char *label_of_dec; /* text label selected by user */
double width_of_dec; /* pulse length in sec */
char *label_of_dec2; /* text label selected by user */
double width_of_dec2; /* pulse length in sec */

Syntax: (8) dps_show("zgradpulse",value,delay)
double value; /* amplitude of gradient on z channel */
double delay; /* length of gradient in sec */

Syntax: (9) dps_show("rgradient",channel,value)
char channel; /* 'X', 'x', 'Y', 'y', 'Z', or 'z' */
double value; /* amplitude of gradient amplifier */

Syntax: (10) dps_show("vgradient",channel,intercept,
slope,mult)
char channel; /* gradient channel 'x', 'y' or 'z' */
int intercept; /* initial gradient level */
int slope; /* gradient increment */
codeint mult; /* real-time variable */

Syntax: (11) dps_show("shapedgradient",pattern,width,amp,
channel,loops,wait)
char *pattern; /* name of shape text file */
double width; /* length of pulse */
double amp; /* amplitude of pulse */
char channel; /* gradient channel 'x', 'y', or 'z' */
int loops; /* number of loops */
int wait; /* WAIT or NOWAIT */

Syntax: (12) dps_show("shaped2Dgradient",pattern,width,amp,
channel,loops,wait,tag)
char *pattern; /* name of shape text file */
double width; /* length of pulse */
double amp; /* amplitude of pulses */
char channel; /* gradient channel 'x', 'y', or 'z' */
int loops; /* number of loops */
int wait; /* WAIT or NOWAIT */
int tag; /* unique number for gradient element */

Description: Draws for dps graphical display the pulses, lines, and labels related to the statement (if it exists) given as the first argument.

- Syntax 1 draws a line to represent a delay.
- Syntax 2 draws a pulse picture and display a label underneath the picture.
- Syntax 3 draws the picture of a shaped pulse and displays a label underneath the picture.
- Syntax 4 draws observe and decoupler pulses at the same time.
- Syntax 5 draws a shaped pulse for observe and decoupler channels at the same time.
- Syntax 6 draws observe, decoupler, and second decoupler pulses at the same time.
- Syntax 7 draws a shaped pulse for observe, decoupler, and the second decoupler channels at the same time.
- Syntax 8 draws a pulse on the z channel.
- Syntax 9 draws a pulse on the specified channel.
- Syntax 10 draws a gradient picture.
- Syntax 11 draws a shaped pulse on a specified channel.
- Syntax 12 draws a shaped pulse on a specified channel. For an explanation of the arguments (delay, shapedpulse, etc.), see the corresponding entry in this reference.

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("shaped_pulsedpue","gauss",p1,"gauss",p2);
dps_show("shaped2D_pulse",p1","p2","p1*2","p1*2");
dps_show("shaped2D_pulse",123.0,d1);
dps_show("rgradient","x",1234.0);
dps_show("vgradient","x",0,2000,v10);
dps_show("shapedgradient","sinc",1000.0,3000.0, \ 'y',1,NOWAIT);
dps_show("shaped2Dgradient","square",1000.0, \ 3000.0,'y',0,NOWAIT,1);

Related:
delay Delay for a specified time
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 Pulse observe transmitter with amplifier gating
rgradient Set gradient to specified level
dps_skip

Skip graphical display of next statement

Syntax: dps_skip()

Description: Skips dps display of the next statement. The statement following dps_skip is not shown in the graphical display.

Related:
- dps_off Turn off graphical display of statements
- dps_on Turn on graphical display of statements
- dps_show Draw delay or pulses for graphical display of a sequence

elsenz

Execute succeeding statements if argument is nonzero

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 nonzero. The elsenz statement can be omitted if it is not desired. 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 v14, oph, etc.) tested for either being zero or non-zero.

n is the same value (1, 2, or 3) as used in the corresponding ifzero statement.

Examples: elsenz(v2);
          elsenz(1);
Chapter 3. Pulse Sequence Statement Reference

**endhardloop**  
End hardware loop

**Applicability:**  
UNITY, INOVA, and MERCURY\textit{plus}/-Vx excluding MERCURY\textit{plus}/-Vx with output boards have part numbers 00-953529-0# where # is from 0 to 4.

**Syntax:**  
`endhardloop()`

**Description:**  
Ends a hardware loop that was started by the `starthardloop` statement.

**Related:**  
`acquire` Explicitly acquire data  
`starthardloop` Start hardware loop

**endif**  
End execution started by `ifzero` or `elsenz`

**Syntax:**  
`endif(vi)`

```c
codeint vi; /* real-time variable to test if 0 or not */
```

**Description:**  
Ends conditional execution started by the `ifzero` and `elsenz` statements.

**Arguments:**  
`vi` is a real-time variable (v1 to v14, oph, etc.) that is tested for either being zero or non-zero.

`n` is the same value (1, 2, or 3) as used in the corresponding `ifzero` statement.

**Examples:**  
`endif(v4);`
`endif(2);`

**Related:**  
`elsenz` Execute succeeding statements if argument is nonzero  
`ifzero` Execute succeeding statements if argument is zero

**endloop**  
End loop

**Syntax:**  
`endloop(index)`

```c
codeint index; /* real-time variable */
```

**Description:**  
Ends a loop that was started by a `loop` statement.

**Arguments:**  
`index` is a real-time variable used as a temporary counter to keep track of the number of times through the loop. It must not be altered by any statements within the loop.

`n` is the same value (1, 2, or 3) as used in the corresponding `loop` statement.

**Examples:**  
`endloop(v2);`
`endloop(2);`

**Related:**  
`loop` Start loop

**endmsloop**  
End multislice loop

**Applicability:**  
UNITY, INOVA systems.

**Syntax:**  
`endmsloop(state, apv2)`

```c
char state; /* compressed or standard */
codeint apv2; /* current counter value */
```

**Description:**  
Ends 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 loop that it is ending.

apv2 is a real-time variable that holds the current counter value. This variable should be the same variable that was in the apv2 counter variable in the msloop loop that it is ending.

Examples:  
endmsloop(seqcon[1],v12);

Related:  
msloop Multislice loop
dloop End loop
dloop End phase-encode loop

dloop End phase-encode loop

Applicability:  UNITY/NOVA systems.

Syntax:  
deloop(state,apv2)
char state;  /* compressed or standard */
coint apv2;  /* current counter value */

Description:  
Ends 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 loop that it is ending.

apv2 is a real-time variable that holds the current counter value. This variable should be the same variable that was in the apv2 counter variable in the peloop loop that it is ending.

Examples:  
deloop(seqcon[1],v12);

Related:  
peloop Phase-encode loop
dloop End loop
dloop End multi-slice loop

gate Device gating (obsolete)
getarray Get arrayed parameter values
getelem Retrieve an element from a table
getorientation Read image plane orientation
getstr Look up value of string parameter
getval Look up value of numeric parameter
g_Delay Generic delay routine
g_Offset Frequency offset routine
Chapter 3. Pulse Sequence Statement Reference

**gate**

**Device gating (obsolete)**

**Description:** Not supported. Replace gate statements as follows:
- `gate(DECUPLR, TRUE)` by a `decon()` statement.
- `gate(DECUPLR, FALSE)` by a `decoff()` statement.
- `gate(DECUPLR2, TRUE)` by a `dec2on()` statement.
- `gate(DECUPLR2, FALSE)` by a `dec2off()` statement.
- `gate(RXOFF, TRUE)` by a `rcvroff()` statement.
- `gate(RXOFF, FALSE)` by a `rcvron()` statement.
- `gate(TXON, TRUE)` by a `xmtron()` statement.
- `gate(TXON, FALSE)` by a `xmtroff()` statement.

**getarray**

**Get arrayed parameter values**

**Applicability:** UNITY/INova systems.

**Syntax:**
```
number = getarray(parname, array)
```

**char *parname; /* parameter name */
**double array[]; /* starting address of array */

**Description:** Retrieves all values of an arrayed parameter from the parameter set. It performs `sizeof` on the array address to check for the maximum number of statements that the array can hold. The number of statements in the arrayed parameter `parname` is determined and returned by `getarray` as an integer. This statement is very useful when reading in parameter values for a global list of PSG statements such as `poffset_list` and `position_offset_list`.

When creating an acquisition parameter array that will be treated as lists, protection bit 8 (256) is set if the parameter is not to be treated as an arrayed acquisition parameter. An example of the `pss` parameter when compressing slice select portion of the acquisition is `create(pss, real)
setprotect(pss, on, 256)`

**Arguments:**
- `number` is an integer return argument that holds the number of values in `parname`.
- `parname` is a numeric parameter, either arrayed or single value.
- `array` is the starting address of an array of doubles.

**Examples:**
```
double upss[256]; /* declare array upss */
int uns;
uns = getarray(upss, upss); /* get values from upss */
poffset_list(upss, gss, uns, v12);
```

**Related:**
- `create_delay_list` Create table of delays
- `create_freq_list` Create table of frequencies
- `create_offset_list` Create table of offsets
- `poffset_list` Set frequency from position list
- `position_offset_list` Set frequency from position list

**getelem**

**Retrieve an element from a table**

**Syntax:**
```
getelem(table, index, dest)
```

**G_Power**

Fine power routine

**G_Pulse**

Generic pulse routine
Description: Gets an element from a table. The element is identified by an index.

Arguments: 
- **table** specifies the name of the table (t1 to t60).
- **index** is a variable (v1 to v14, oph, ct, bsctr, or ssctr) that contains the index of the desired table element. Note that the first element of a table has an index of 0. For tables for which the autoincrement feature is set, the index argument is ignored and can be set to any variable name; each element in such a table is by definition always accessed sequentially.
- **dest** is an variable (v1 to v14 and oph) into which the retrieved table element is placed.

Examples: `getelem(t25,ct,v1);`

Related:
- **loadtable** Load AP table elements from table text file
- **setautoincrement** Set autoincrement attribute for a table
- **setdivnfactor** Set divn-return attribute and divn-factor for AP table
- **setreceiver** Associate the receiver phase cycle with a table
- **settable** Store an array of integers in a real-time AP table

---

**getorientation** Read image plane orientation

**Applicability:** UNITY INOVA systems with PFG modules.

**Syntax:**
```c
<error_return => getorientation(&char1,
    &char2,&char3,search_string)
char *char1,*char2,*char3; /* program variable pointers */
char *search_string; /* pointer to search string */
```

**Description:** Reads 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 `search_string` and reading in the value of that parameter.
  - **error_return** 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, i.e., 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 there is no string associated with search_string or the parameter name cannot be found.

Examples: (1) 
```c
pulsesequence()
{
  ...
  char phase, read, slice;
  ...
  getorientation(&read, &phase, &slice, "orient");
  ...
}
```

(2) 
```c
pulsesequence()
{
  ...
  char rd, ph, sl;
  int error;
  ...
  error = getorientation(&rd, &ph, &sl, "ort");
  ...
}
```

Related: shapedvgradient Dynamic variable shaped gradient function
rgradient Set gradient to specified level
vgradient Dynamic variable gradient function

**getstr**

Look up value of string parameter

**Syntax:** `getstr(parameter_name, internal_name)`
```c
char *parameter_name; /* name of parameter */
char *internal_name; /* parameter value buffer name */
```

**Description:** Looks up the value of the string parameter `parameter_name` in the current experiment parameter list and introduces it into the pulse sequence in the variable `internal_name`. If `parameter_name` is not found in the current experiment parameter list, `internal_name` is set to the null string and PSG produces a warning message.

**Arguments:**

- `parameter_name` is a string parameter.
- `internal_name` is any legitimate C variable name defined at the beginning of the pulse sequence as an array of type char with dimension MAXSTR.

**Examples:**
```c
getstr("xpol", xpol);
```

Related: `getval` Look up value of numeric parameter

**getval**

Look up value of numeric parameter

**Syntax:** `internal_name = getval(parameter_name)`
```c
char *parameter_name; /* name of parameter */
```

**Description:** Looks up the value of the numeric parameter `parameter_name` in the current experiment parameter list and introduces it into the pulse sequence in the variable `internal_name`. If `parameter_name` is not found in the current experiment parameter list, `internal_name` is set to zero and PSG produces a warning message.

**Arguments:**

- `parameter_name` is a numeric parameter.
internal_name can be any legitimate C variable name that has been defined at the beginning of the pulse sequence as type double.

Examples:  
J = getval("J");
acqtime = getval("at");
delay(getval("mix"));

Related:  
getstr  Look up value of string parameter

G_Delay  
Generic delay routine

Applicability:  
UNITY/INOVA systems.

Syntax:  
G_Delay(Delay_Time,  d1,
SLIDER_LABEL,  NULL,
SLIDER_SCALE,  1,
SLIDER_MAX,  60,
SLIDER_MIN,  0,
SLIDER_UNITS,  1.0,
0);

Description:  
See the section “Generic Pulse Routine,” page 100.

G_Offset  
Frequency offset routine

Applicability:  
UNITY/INOVA systems.

Syntax:  
G_Offset(OFFSET_DEVICE, TODEV,
OFFSET_FREQ, tof,
SLIDER_LABEL,  NULL,
SLIDER_SCALE,  0,
SLIDER_MAX,  1000,
SLIDER_MIN,  -1000,
SLIDER_UNITS,  0,
0);

Description:  
See the section “Frequency Offset Subroutine,” page 101.

G_Power  
Fine power routine

Applicability:  
UNITY/INOVA systems.

Syntax:  
G_Power(POWER_VALUE, tpwr,
POWERDEVICE, TODEV,
SLIDER_LABEL,  NULL,
SLIDER_SCALE,  1,
SLIDER_MAX,  4095,
SLIDER_MIN,  0,
SLIDER_UNITS,  1.0,
0);

Description:  
See the section “Fine Power Subroutine,” page 103.

G_Pulse  
Generic pulse routine

Applicability:  
UNITY/INOVA systems.

Syntax:  
G_Pulse(PULSE_WIDTH, pw,
PULSE_PRE_ROFF, rof1,
PULSE_POST_ROFF, rof2,
PULSE_DEVICE,   TODEV,
SLIDER_LABEL,   NULL,
SLIDER_SCALE,   1,
SLIDER_MAX,     1000,
SLIDER_MIN,     0,
SLIDER_UNITS,   1e-6,
PULSE_PHASE,    oph,
0);

Description: See “Generic Pulse Routine,” page 100.

hdwshiminit
Initialize next delay for hardware shimming

Syntax: hdwshiminit()

Description: Enables hardware shimming during the following delay or during the following
presaturation pulse, defined as a power level change followed by pulse.

hdwshiminit is not necessary for the first delay or presaturation pulse in a
pulse sequence, which is automatically enabled for hardware shimming.

Examples:
hdwshiminit();
delay(d2);
    /*hardware shim during d2 if hdwshim='y'*/
hdwshiminit();
obspower(satpwr);
rgpulse(satdly,v5, rof1, rof2);
    /*hardware shim during satdly if hdwshim='p'*/

Related: delay          Delay for a specified time

hlv
Find half the value of an integer

Syntax: hlv(vi,vj)
    codeint vi;  /* real-time variable for starting value */
    codeint vj;  /* real-time variable for 1/2 starting value */

Description: Sets vj equal to the integer part of one-half of vi.

Arguments: vi is the starting value, and vj is the integer part of one-half of the starting
value. Both arguments much be real-time variables (v1 to v14, oph, etc.).

Examples: hlv(v2,v5);
hsdelay  
Delay specified time with possible homospoil pulse

Syntax:  
hsdelay(time)

double time;    /* delay in sec */

Description:  
Sets a delay for a specified number of seconds. If the homospoil parameter hs is set appropriately (see the definition of status), hsdelay inserts a homospoil pulse of length hst sec at the beginning of the delay.

Arguments:  
time specifies the length of the delay, in seconds.

Examples:  
hsdelay(d1);
hsdelay(1.5e–3);

Related:  
delay Delay for a specified time
incdelay Real time incremental delay
initdelay Initialize incremental delay
vdelay Delay with fixed timebase and real time count
Chapter 3. Pulse Sequence Statement Reference

idecpulse  Pulse first decoupler transmitter with IPA

Applicability: UNITY/INOVA systems

Syntax:    idecpulse(width, phase, label)
            double width;    /* pulse width in sec */
            codeint phase;  /* real-time variable for phase */
            char *label;    /* slider label in acqi */

Description: Functions the same as the decpulse statement but generates interactive parameter adjustment (IPA) information when gf or go('acqi') is typed. idecpulse is the same as decpulse if go is typed.

Arguments:

  width is the duration, in seconds, of the pulse.

  phase is the phase of the pulse. It must be a real-time variable (v1 to v14, oph, etc.) or a real-time constant (zero, one, etc.).

  label is the short character string to be given to the slider when displayed in the Acquisition window (acqi program).

Examples:

  idecpulse(pp,v1,"decpul");
  idecpulse(pp,v2,"pp");

Related: decpulse  Pulse the decoupler transmitter

idecrgpulse  Pulse first decoupler with amplifier gating and IPA

Applicability: UNITY/INOVA systems

Syntax:    idecrgpulse(width, phase, RG1, RG2, label)
            double width;    /* pulse width in sec */
            codeint phase;  /* real-time variable for phase */
            double RG1;     /* gating delay before pulse in sec */
            double RG2;     /* gating delay after pulse in sec */
            char *label;    /* slider label in acqi */

Description: Works similar to the deccrgpulse statement but generates interactive parameter adjustment (IPA) information when gf or go('acqi') is typed. idecrgpulse is the same as deccrgpulse if go is typed.

Arguments:

  width is the duration, in seconds, of the decoupler transmitter pulse.

  phase sets the decoupler transmitter phase. The value must be a real-time variable.

  RG1 is the time, in seconds, that the amplifier is gated on prior to the start of the pulse.

  RG2 is the time, in seconds, that the amplifier is gated off after the end of the pulse.
**idelay**  
**Delay for a specified time with IPA**  

**Applicability:** unity/NOVA systems  

**Syntax:**  
`idelay(time,label)`  
`double time; /* delay in sec */`  
`char *label; /* slider label in acqi */`  

**Description:** Works similar to the delay statement but generates interactive parameter adjustment (IPA) information when `gf` or `go('acqi')` is entered. `idelay` is the same as `delay` if `go` is entered.

**Arguments:**  
- `time` is the length of the delay, in seconds.  
- `label` is the short character string to be given to the slider when displayed in the Acquisition window (acqi program).

**Examples:**  
`idelay(d1,"delay");`  
`idelay(d1,"d1");`

**Related:**  
- `delay`  
  Delay for a specified time

**ifzero**  
**Execute succeeding statements if argument is zero**  

**Syntax:**  
`ifzero(vi)`  
`codeint vi; /* real-time variable to check for zero */`

**Description:** Executes 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 `v14`, `oph`, etc.) that is tested for being either zero or non-zero.  
- `n` is the same value (1, 2, or 3) as used in the corresponding `elsenz` or `endif` statements.

**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 succeeding statements if argument is nonzero  
- `endif`  
  End ifzero statement  
- `initval`  
  Initialize real-time variable to specified value
**incdelay**  
Set real-time incremental delay

Applicability:  
UNITY/NOVA systems.

Syntax:  
```c
incdelay(count, index)
```

codeint count; /* real-time variable */
int index; /* time increment: DELAY1, DELAY2, etc. */

Description: Enables real-time incremental delays. Before `incdelay` can be used to set a delay, an associated `initdelay` statement must be executed to initialize the time increment and delay index.

Arguments:  
- `count` is a real-time variable (ct, v1 to v14, etc.) that multiplies the time increment (initialized by the `initdelay` statement) to set the delay time.
- `index` is DELAY1, DELAY2, DELAY3, DELAY4, or DELAYS. It identifies which time increment is being multiplied by `count` to equal the delay.

Examples:  
```c
incdelay(ct, DELAY1);
incdelay(v3, DELAY2);
```

Related:  
- `delay`  
  Delay for a specified time
- `hsdelay`  
  Delay with possible homospoil pulse
- `initdelay`  
  Initialize incremental delay
- `vdelay`  
  Delay with fixed timebase and real time count

**incgradient**  
Generate dynamic variable gradient pulse

Applicability:  
UNITY/NOVA systems.

Syntax:  
```c
incgradient(channel, base, inc1, inc2, inc3, mult1, mult2, mult3)
```

char channel; /* gradient 'x', 'y', or 'z' */
int base; /* base value */
int inc1, inc2, inc3; /* increments */
codeint mult1, mult2, mult3; /* multipliers */

Description: Provides a dynamic variable gradient pulse controlled using the AP math functions. It drives the chosen gradient to the level defined by the formula:

```
level = base + inc1 * mult1 + inc2 * mult2 + inc3 * mult3
```

with increments inc1, inc2, inc3 and multipliers mult1, mult2, mult3.

The range of the gradient level is –32767 to +32767. If the requested level lies outside the legal range, it is clipped at the appropriate boundary value. Note that, while each variable in the `level` formula must fit in a 16-bit integer, partial sums and products in the calculation are done with double-precision 32-bit integers.

The action of the gradient after the use of the `incgradient` statement is controlled by the gradient power supply and optional gradient compensation boards. The gradient level is ramped at the maximum slew rate to the value requested by `incgradient`. This fact becomes a concern when using the `incgradient` statement in a loop with a delay statement to produce a modulated gradient. The delay statement should be sufficiently long so as to allow the gradient to reach the assigned value, that is,

```
delay ≥ \frac{|\text{new level} - \text{old level}| \times \text{risetime}}{\text{full scale}}
```

The following error messages are possible:

- Bad gradient specified: channel is caused by the channel character evaluating to other than 'x', 'y', or 'z'; or by being a string.
• mult[i] illegal RT variable: multiplier_i is caused by
  mult1, mult2, or mult3 having a value other than a AP math variable,
  v1 to v14.

Arguments: channel1 is an expression that evaluates to the character 'x', 'y', or 'z'.
(donot confuse characters 'x', 'y' and 'z' with strings "x", "y" and "z".)
base and inc1, inc2, inc3 are the base value and increments used in the
formula for determining the gradient level.
mult1, mult2, mult3 are the multipliers used in the gradient level formula.
These arguments should be math variables, v1 to v14. Note that tables (t1 to
t60) are not allowed in this statement.

Examples: See the program incstst.c

Related: getorientation Read image plane orientation
         rgradient Set gradient to specified level
         shapedgradient Provide shaped gradient pulse to gradient channel
         shaped2Dgradient Generate arrayed shaped gradient pulse
         shapedvgradient Generate dynamic variable shaped gradient pulse
         vgradient Generate dynamic variable gradient pulse

incr Increment an integer value

Syntax: incr(vi)
codeint vi; /* real-time variable to increment */

Description: Increments by 1 the integer value given by vi (i.e., vi=vi+1).

Arguments: vi is the integer to be incremented, It must be a real-time variable (v1 to v14,
            oph, etc.).

Examples: incr(v4);

Related: add Add integer values
         assign Assign integer values
dbl Double an integer value
decr Decrement an integer value
divn Divide integer values
hlv Half the value of an integer
mod2 Find integer value modulo 2
mod4 Find integer value modulo 4
modn Find integer value modulo n
mult Multiply integer values
sub Subtract integer values

indirect Set indirect detection

Applicability: No longer useful to any system using VNMR 5.2 or later.

Syntax: indirect()

Description: Starting with VNMR 5.2, if tn is 'H1' and dn is not 'H1', the software
            automatically uses the decoupler as the observe channel and the broadband
            channel as the decoupler channel.

init_rfpattern Create rf pattern file

Applicability: UNITY, INOVA systems.
Syntax:

```
init_rfpattern(pattern, rfpat_struct, nsteps)
```

- `pattern`: /* name of .RF text file */
- `rfpat_struct`: /* pointer to struct RFpattern */
- `nsteps`: /* number of steps in pattern */

```
typedef struct _RFpattern {
    double phase; /* phase of pattern step */
    double amp; /* amplitude of pattern step */
    double time; /* length of pattern step in sec */
} RFpattern
```

Description: Creates and defines rf patterns within a pulse sequence. The patterns can be created by any algorithm as long as each pattern step is correctly put into the `rfpat_struct` argument. The number of steps in the pattern also has to be furnished as an argument. `init_rfpattern` saves the created pattern as a pattern file (with the suffix `.RF` appended to the name) in the user’s `shapelib` directory. This statement does not have any return value.

Arguments:
- `pattern` is the name of the pattern file (without the `.RF` suffix).
- `rfpat_struct` is the rf structure that contains the pattern.
- `nsteps` is the number of steps in the pattern.

Examples:

```c
#include "standard.h"

pulsecsequence()
{
    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(p1pat,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(p1pat,p1,v1,rof1,rof1);
    ...
    shapedgradient("gpat",.01, 16000.0, 'z', 1, WAIT);
    ...
}
```

Related:
- `init_gradpattern`>Create gradient pattern file
- `pulse`>Pulse observe transmitter with amplifier gating
- `shaped_pulse`>Perform shaped pulse on observe transmitter
- `shapedgradient`>Provide shaped gradient pulse to gradient channel
- `simpulse`>Pulse observe and decouple channels simultaneously
- `simshaped_pulse`>Perform simultaneous two-pulse shaped pulse
init_gradpattern  Create gradient pattern file

Applicability:  UNITY/NOVA systems.

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: Creates and defines gradient patterns within a pulse sequence. The patterns can
be created by any algorithm as long as each pattern step is correctly put into the
gradpat_struct argument. The number of steps in the pattern also has to
be furnished as an argument. init_gradpattern saves the created pattern
as a pattern file (with a .GRD suffix is appended to the name) in the user’s
shapelib directory. This statement has no return value.

Arguments: pattern is the name of the pattern file (without the .GRD suffix).
gradpat_struct is the gradient structure that contains the pattern.
nsteps is the number of steps in the pattern.

Examples: See the example for the init_rfpattern statement.

Related:  pulse  Pulse observe transmitter with amplifier gating
        shaped_pulse  Perform shaped pulse on observe transmitter
        simpulse  Pulse observe and decouple channels simultaneously
        simshaped_pulse  Perform simultaneous two-pulse shaped pulse

initdelay  Initialize incremental delay

Applicability:  UNITY/NOVA systems

Syntax:  initdelay(time_increment,index)
          double time_increment;  /* time increment in sec */
          int index;  /* time increment: DELAY1, etc. */

Description: Initializes a time increment delay and its associated delay index. This statement
must be executed before an incdelay statement can set an incremental delay.
A maximum of five incremental delays (set by the index argument) can be
defined in one pulse sequence.

Arguments: time_increment is the time increment, in seconds, that is multiplied by the
count argument (set in the incdelay statement) for the delay time.
index is DELAY1, DELAY2, DELAY3, DELAY4, or DELAY5, and identifies
which time increment is being initialized.

Examples:  initdelay(1.0/sw,DELAY1);
            initdelay(1.0/sw1,DELAY2);

Related:  delay  Delay for a specified time
          hadelay  Delay with possible homospoil pulse
          incdelay  Real time incremental delay
          vdelay  Delay with fixed timebase and real time count
initparms_sis  Initialize parameters for spectroscopy imaging sequences

Applicability: Imaging systems; however, this statement will be obsoleted in future versions of VnmrJ.

Syntax: void initparms_sis()

Description: Sets the default state of the receiver to ON so that the receiver is enabled for explicit acquisitions. The original purpose of initparms_sis was to initialize the standard imaging parameters in imaging sequences, but starting with VNMR 5.3, initialization of these parameters has been folded into PSG.

Examples: /* To upgrade older SIS sequences for Vnmr 5.1+: */
/* insert initparms_sis() after the variable */
/* declarations and update ‘griserate’ variable. */
...
/* EXTERNAL TRIGGER */
double rcvry, hold;
initparms_sis();
griserate = trise/gradstepsz;
/**[3.2] PARAMETER READ IN FROM EXPERIMENT *******/
...

initval  Initialize a real-time variable to specified value

Syntax: initval(number, vi)

double number; /* value to use for initialization */
codeint vi; /* variable to be initialized */

Description: Initializes a real-time variable with a real number. The real number input is rounded off and placed in the variable vi. Unlike add, sub, etc., 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; this must be taken into account in pulse sequence programming, as shown in the example.

Arguments: number is the real number, from –32768.0 to 32767.0, to be placed in the real-time variable. Entering a value less than –32768.0 (after rounding off) results in using –32768, and entering a value greater than 32767.0 (after rounding off) results in using 32767.

vi is the real-time variable (v1 to v14, etc.) to be initialized

Examples: (1) initval(nt,v8);
(2) ifzero(ct);
   assign(v8,v7);
elsez(ct);
   decr(v7);
endif(ct);

Related:  elsenz  Execute succeeding statements if argument is nonzero
         ifzero  Execute succeeding statements if argument is zero
         loop  Start loop

iobspulse  Pulse observe transmitter with IPA

Applicability: UNITY/INOVA systems

Syntax: iobspulse(label)

char *label;  /* slider label in acqi */
Description: Functions the same as `obspulse` except `iobspulse` generates interactive parameter adjustment (IPA) information when `gf` or `go('acqi')` is entered. If `go` is entered, `iobspulse` is the same as `obspulse`.

Arguments: `label` is the short character string to be given to the slider when displayed in the Acquisition window (acqi program).

Examples:
```c
iobspulse("pulse");
iobspulse("pw");
```

Related: `obspulse` Pulse observe transmitter with amplifier gating

### ioffset

**Change offset frequency with IPA**

**Applicability:** `UNITY/INOVA` systems

**Syntax:**
```c
ioffset(frequency, device, label)
```

- `double frequency; /* offset frequency */`
- `int device; /* OBSch, DECch, DEC2ch, or DEC3ch */`
- `char *label; /* slider label in acqi */`

**Description:** Functions the same as `offset` except that `ioffset` generates interactive parameter adjustment (IPA) information when `gf` or `go('acqi')` is entered. If `go` is entered, `ioffset` is the same as `offset`.

**Arguments:**
- `frequency` is the new offset frequency of the device specified.
- `device` is `OBSch` (observe transmitter) or `DECch` (first decoupler). `device` can also be `DEC2ch` (second decoupler) or `DEC3ch` (third decoupler).
- `label` is the short character string to be given to the slider when displayed in the Acquisition window (acqi program).

**Examples:**
```c
ioffset(tof,OBSch,"tof");
```

**Related:** `offset` Change offset frequency of transmitter or decoupler

### ipulse

**Pulse observe transmitter with IPA**

**Applicability:** `UNITY/INOVA` systems

**Syntax:**
```c
ipulse(width, phase, label)
```

- `double width; /* pulse length in sec */`
- `codeint phase; /* real-time variable for phrase */`
- `char *label; /* slider label in acqi */`

**Description:** Functions the same as `pulse(width, phase)` statement except that `ipulse` generates interactive parameter adjustment (IPA) information when `gf` or `go('acqi')` is entered. If `go` is entered, `ipulse` is the same as `pulse`.

**Arguments:**
- `width` specifies the duration, in seconds, of the pulse.
- `phase` sets the phase of the pulse. The value must be a real-time variable (v1 to v14, oph, etc.).
- `label` is the short character string to be given to the slider when displayed in the Acquisition window (acqi program).

**Examples:**
```c
ipulse(pw,v4,"pulse");
ipulse(pw,v5,"pw");
```

**Related:** `pulse` Pulse observe transmitter with amplifier gating
**ipwrf**  
*Change transmitter or decoupler fine power with IPA*

**Applicability:** `UNITY/NOVA` systems

**Syntax:**
```c
ipwrf(power, device, label)
```

- `double power; /* new fine power level */`
- `int device; /* OBSch, DECCh, DEC2ch, DEC3ch */`
- `char *label; /* slider label in acqi */`

**Description:** Functions the same as `rlpwrf` statement except that `ipwrf` generates interactive parameter adjustment (IPA) information when `gf` or `go('acqi')` is entered. If `go` is entered, `ipwrf` is ignored by the pulse sequence; use `rlpwrf` for this purpose. Do not execute `rlpwrf` and `ipwrf` together because they cancel each other's effect.

**Arguments:**
- `power` is the new fine power level. It can range from 0.0 to 4095.0 (60 dB on `UNITY/NOVA`, about 6 dB on other systems).
- `device` is `OBSch` (observe transmitter) or `DECch` (first decoupler). For the `UNITY/NOVA` only `device` can also be `DEC2ch` (second decoupler) or `DEC3ch` (third decoupler).
- `label` is the short character string to be given to the slider when displayed in the Acquisition window (`acqi` program).

**Examples:**
- `ipwrf(power, OBSch, "fpower");`
- `ipwrf(2000.0, DECch, "dpwrf");`

**Related:** `rlpwrf`  
Set transmitter or decoupler fine power

---

**ipwrm**  
*Change transmitter or decoupler lin. mod. power with IPA*

**Applicability:** `UNITY/NOVA` systems

**Syntax:**
```c
ipwrm(value, device, label)
```

- `double value; /* new linear modulator power level */`
- `int device; /* OBSch, DECCh, DEC2ch, or DEC3ch */`
- `char *label; /* slider label in acqi */`

**Description:** Functions the same as `rlpwrm` statement except that `ipwrm` generates interactive parameter adjustment (IPA) information when `gf` or `go('acqi')` is entered. If `go` is entered, `ipwrm` is ignored by the pulse sequence; use `rlpwrm` for this purpose. Do not execute `rlpwrm` and `ipwrm` together as they cancel each other's effect.

**Arguments:**
- `value` is the new linear modulator power level. It can range from 0.0 to 4095.0 (60 dB on `UNITY/NOVA`, about 6 dB on other systems).
- `device` is `OBSch` (observe transmitter) or `DECch` (first decoupler). On the `UNITY/NOVA` only `device` can also be `DEC2ch` (second decoupler) or `DEC3ch` (third decoupler).
- `label` is the short character string to be given to the slider when displayed in the Acquisition window (`acqi` program).

**Examples:**
- `ipwrm(power, OBSch, "fpower");`
- `ipwrm(2000.0, DECch, "dpwrm");`

**Related:** `rlpwrm`  
Set transmitter or decoupler linear modulator power

---

**irgpulse**  
*Pulse observe transmitter with IPA*

**Applicability:** `UNITY/NOVA` systems

**Syntax:**
```c
irgpulse(width, phase, RG1, RG2, label)
```

**Related:** `rlpwrf`  
Set transmitter or decoupler fine power

**Related:** `rlpwrm`  
Set transmitter or decoupler linear modulator power
double width;    /* pulse length in sec */
codeint phase;  /* real-time variable for phase */
double RG1;     /* gating delay before pulse in sec */
double RG2;     /* gating delay after pulse in sec */
char *label;    /* slider label in acqi */

Description: Functions the same as the **rgpulse** statement except that **irgpulse** generates interactive parameter adjustment (IPA) information when **gf** or **go('acqi')** is entered. If **go** is entered, **irgpulse** is the same as **rgpulse**.

Arguments: **width** specifies the duration, in seconds, of the observe transmitter pulse. **phase** sets the observe transmitter phase. It must be a real-time variable. **RG1** is the time, in seconds, the amplifier is gated on prior to the start of the pulse. **RG2** is the time, in seconds, the amplifier is gated off after the end of the pulse. **label** is the short character string to be given to the slider when displayed in the Acquisition window (**acqi** program).

Examples:
- `irgpulse(pw,v3,rof1,rof2,"rgpul");`
- `irgpulse(pw,v7,rof1,rof2,"pw");`

Related: **rgpulse** Pulse observe transmitter with amplifier gating

---

**lk_hold**

Set lock correction circuitry to hold correction

Syntax: `lk_hold()`

Description: Makes the lock correction circuitry hold the correction to the z0 constant, thereby ignoring any influence on the lock signal such as gradient or pulses at $^2$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.

**lk_sample**

Set lock correction circuitry to sample lock signal

Syntax: `lk_sample()`

---

**loadtable**

Load table elements from table text file

**loop**

Start loop

**loop_check**

Check that number of FIDs is consistent with number of slices, etc.
Description: Makes 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.

loadtable  Load table elements from table text file
Syntax: loadtable(file)
char file; /* name of table file */
Description: Loads table elements from a table file (a UNIX text file). It can be called multiple times within a pulse sequence but make sure that the same table name is not used more than once within all the table files accessed by the sequence. Table values can be greater than, equal to, or less than zero.
Arguments: file is the name of a table file in a user's private tablib or in the system tablib.
Examples: loadtable("tabletest");

Related: getelem Retrieve an element from a table
setautoincrement Set autoincrement attribute for a table
setdivnfactor Set divn-return attribute and divn-factor for AP table
setreceiver Associate the receiver phase cycle with a table
settable Store an array of integers in a real-time AP table

loop  Start loop
Syntax: loop(count,index)
codeint count /* number of times to loop */
codeint index /* real-time variable to use during loop */
Description: Starts a loop to execute statements within the pulse sequence. The loop is ended by 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.
index is a real-time variable used as a temporary counter to keep track of the number of times through the loop. The value must not be altered by any statements within the loop.

n is the same value (1, 2, or 3) as used in the corresponding endloop statement.
Examples: (1) 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);
(2) loop(2,5.0,v9);

Related: initval Initialize real-time variable to specified value
endloop End loop
msloop Multislice loop
**loop_check**  
Check that number of FIDs is consistent with number of slices, etc.  

**Syntax:**  
loop_check

**Description:** Checks 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.

---

**M**

| Top | A | B | C | D | E | G | H | I | L | M | O | P | R | S | T | V | W | X | Z |
| maggradient | Simultaneous gradient at the magic angle |  |
| magradpulse | Gradient pulse at the magic angle |  |
| mashapedgradient | Simultaneous shaped gradient at the magic angle |  |
| mashapedgradpulse | Simultaneous shaped gradient pulse at the magic angle |  |
| mod2 | Find integer value modulo 2 |  |
| mod4 | Find integer value modulo 4 |  |
| modn | Find integer value modulo n |  |
| msloop | Multislice loop |  |
| mult | Multiply integer values |  |

**magradient**  
Simultaneous gradient at the magic angle

**Applicability:** UNITY/NOVA systems.

**Syntax:**  
magradient (gradlvl)
double gradlvl;  /* gradient amplitude in G/cm */

**Description:** Applies a simultaneous gradient on the x, y, and z axes at the magic angle to B0. Information from a gradient table is used to scale and set values correctly. The gradients are left at the given levels until they are turned off. To turn off the gradients, add another magradient statement with gradlvl set to zero or insert the statement zero_all_gradients.

**Arguments:** gradlvl is the gradient amplitude, in gauss/cm.

**Examples:**  
magradient (3.0);
pulse (pw, oph);
delay (0.001 - pw);
zero_all_gradients();

**Related:**  
magradpulse  Simultaneous gradient pulse at the magic angle  
mashapedgradient  Simultaneous shaped gradient at the magic angle  
mashapedgradpulse  Simultaneous shaped gradient pulse at the magic angle  
vagradient  Variable angle gradient  
vagradpulse  Variable angle gradient pulse  
vashapedgradient  Variable angle shaped gradient  
vashapedgradpulse  Variable angle shaped gradient pulse  
zero_all_gradients  Zero all gradients
**magradpulse**  
Gradient pulse at the magic angle

Applicability: UNITY/NOVA systems.

Syntax:

```c
magradpulse(gradlvl, gradtime)
```

double gradlvl; /* gradient amplitude in G/cm */
double gradtime; /* gradient time in sec */

Description: Applies a simultaneous gradient pulse on the x, y, and z axes at the magic angle to B₀. Information from a gradient table is used to scale and set values correctly.

`magradpulse` differs from `magradient` in that the gradients are turned off after `gradtime` seconds. Use `magradpulse` if there are no other actions while the gradients are on. `magradient` is used if there are actions to be performed while the gradients are on.

Arguments:
- `gradlvl` is the gradient pulse amplitude, in gauss/cm.
- `gradtime` is the time, in seconds, to apply the gradient.

Examples:

```
magradpulse(3.0, 0.001);
```

Related:
- `magradient` Simultaneous gradient at the magic angle
- `mashapedgradient` Simultaneous shaped gradient at the magic angle
- `mashapedgradpulse` Simultaneous shaped gradient pulse at the magic angle
- `vagradient` Variable angle gradient
- `vagradpulse` Variable angle gradient pulse
- `vashapedgradient` Variable angle shaped gradient
- `vashapedgradpulse` Variable angle shaped gradient pulse
- `zero_all_gradients` Zero all gradients

**mashapedgradient**  
Simultaneous shaped gradient at the magic angle

Applicability: UNITY/NOVA systems.

Syntax:

```c
mashapedgradient(pattern, gradlvl, gradtime, loops, wait)
```

c char *pattern; /* name of gradient shape text file */
double gradlvl; /* gradient amplitude in G/cm */
double gradtime; /* gradient time in seconds */
int loops; /* number of waveform loops */
int wait; /* WAIT or NOWAIT*/

Description: Applies a simultaneous gradient with shape `pattern` and amplitude `gradlvl` on the x, y, and z axes at the magic angle to B₀. Information is used from a gradient table to scale and set the values correctly.

`mashapedgradient` leaves the gradients at the given levels until they are turned off. To turn off the gradients, add another `mashapedgradient` statement with `gradlvl` set to zero or include the `zero_all_gradients` statement.

`mashapedgradpulse` differs from `mashapedgradient` in that the gradients are turned off after `gradtime` seconds. `mashapedgradient` is used if there are actions to be performed while the gradients are on. `mashapedgradpulse` is best when there are no other actions required while the gradients are on.

Arguments:
- `pattern` is the name of a text file describing the shape of the gradient. The text file is located in `$vnmrsystem/shapelib` or in the user directory `$vnmruser/shapelib`.
- `gradlvl` is the gradient amplitude, in gauss/cm.
- `gradtime` is the gradient application time, in seconds.
loops is a value from 0 to 255 to loop the selected waveform. Gradient waveforms do not use this field, and loops is set to 0.

wait is a keyword, either WAIT or NOWAIT, that selects whether or not a delay is inserted to wait until the gradient is completed before executing the next statement.

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);

Related: magradient Simultaneous gradient at the magic angle
magradpulse Simultaneous gradient pulse at the magic angle
mashapedgradpulse Simultaneous shaped gradient pulse at the magic angle
vagradient Variable angle gradient
vagradpulse Variable angle gradient pulse
vashapedgradient Variable angle shaped gradient
vashapedgradpulse Variable angle shaped gradient pulse
zero_all_gradients Zero all gradients

mashapedgradpulse Simultaneous shaped gradient pulse at the magic angle

Applicability: UNITY INOVA systems.

Syntax: mashapedgradpulse(pattern,gradlvl,gradtime,theta,ph)
char *pattern; /* name of gradient shape text file */
double gradlvl; /* gradient amplitude in G/cm */
double gradtime; /* gradient time in sec */

Description: Applies a simultaneous gradient with shape pattern and amplitude gradlvl on the x, y, and z axes at the magic angle to B0. mashapedgradpulse assumes that the gradient pattern zeroes the gradients at its end and so it does not explicitly zero the gradients. Information from a gradient table is used to scale and set values correctly.

mashapedgradpulse is used if there are no other actions required when the gradients are on. mashapedgradient is used if there are actions to be performed while the gradients are on.

Arguments: pattern is the name of a text file describing the shape of the gradient. The text file is located in $vnmr/system/shapelib or in the user directory $vnmruser/shapelib.
gradlvl is the gradient amplitude, in gauss/cm.
gradtime is the gradient application time, in seconds.

Examples: mashapedgradpulse("hsine",3.0, 0.001);

Related: magradient Simultaneous gradient at the magic angle
magradpulse Simultaneous gradient pulse at the magic angle
mashapedgradient Simultaneous shaped gradient at the magic angle
vagradient Variable angle gradient
vagradpulse Variable angle gradient pulse
vashapedgradient Variable angle shaped gradient
vashapedgradpulse Variable angle shaped gradient pulse
zero_all_gradients Zero all gradients
mod2  

Find integer value modulo 2

Syntax:  
mod2(vi,vj)

codeint vi; /* variable for starting value */
codeint vj; /* variable for result */

Description:  
Sets the value of vj equal to vi modulo 2.

Arguments:  
vi is the starting integer value and vj is the value of vi modulo 2 (the remainder after vi is divided by 2). Both arguments must be real-time variables (v1 to v14, etc.).

Examples:  
mod2(v3,v5);

Related:  
add Add integer values
assign Assign integer values
dbl Double an integer value
decr Decrement an integer value
divn Divide integer values
hlv Half the value of an integer
incr Increment an integer value
mod4 Find integer value modulo 4
modn Find integer value modulo n
mult Multiply integer values
sub Subtract integer values

mod4  

Find integer value modulo 4

Syntax:  
mod4(vi,vj)

codeint vi; /* variable for starting value */
codeint vj; /* variable for result */

Description:  
Sets the value of vj equal to vi modulo 4.

Arguments:  
vi is the starting integer value and vj is the value of vi modulo 4 (the remainder after vi is divided by 4). Both arguments must be real-time variables (v1 to v14, etc.).

Examples:  
mod4(v3,v5);

modn  

Find integer value modulo n

Syntax:  
modn(vi,vj,vk)

codeint vi; /* real-time variable for starting value */
codeint vj; /* real-time variable for modulo number */
codeint vk; /* real-time variable for result */

Description:  
Sets the value of vk equal to vi modulo vj.

Arguments:  
vi is the starting integer value, vj is the modulo value, and vk is vi modulo vj (the remainder after vi is divided by vj). All arguments must be real-time variables (v1 to v14, etc.).

Examples:  
modn(v3,v5,v4);

msloop  

Multislice loop

Applicability:  
UNITY/NOVA systems.

Syntax:  
msloop(state,max_count,apv1,apv2)

char state; /* compressed or standard */
double max_count; /* initializes apv1 */
codeint apv1;    /* maximum count */
codeint apv2;    /* current counter value */

Description: Provides a sequence-switchable loop that can use real-time variables in what is known as a compressed loop or it 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 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:  
         endmsloop     End multislice loop
         loop        Start loop
        apelooop     Phase-encode loop

**mult**  
Multiply 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: Sets the value of vk equal to the product of the integer values vi and vj.

Arguments: vi is an integer value, vj is another integer value, and vk is the product of vi and vj. All arguments must be real-time variables (v1 to v14 etc.).

Examples: mult(v3,v5,v4);

Related:  
         add         Add integer values
         assign     Assign integer values
         dbl        Double an integer value
         decr       Decrement an integer value
         divn       Divide integer values
         hlv        Half the value of an integer
         incr       Increment an integer value
         mod2       Find integer value modulo 2
         mod4       Find integer value modulo 4
         modn       Find integer value modulo n
         sub        Subtract integer values
Chapter 3. Pulse Sequence Statement Reference

obl_gradient  Execute an oblique gradient
Applicability: UNITY/INOVA systems.
Syntax: obl_gradient(level1, level2, level3)
        double level1, level2, level3; /* gradient values in G/cm */
Description: Defines an oblique gradient with respect to the magnet reference frame. This
statement is basically the same as the statement oblique_gradient except
that obl_gradient uses the parameters psi, phi, and theta in the
parameter set rather than setting them directly. It has no return value.

The pulse sequence generation aborts if the DACs on a particular gradient are
overrun after the angles and amplitude have been resolved.

Arguments: level1, level2, level3 are gradient values, in gauss/cm.

Examples: obl_gradient(0.0, 0.0, gss);
         obl_gradient(gro, 0.0, 0.0);

oblique_gradient  Execute an oblique gradient
Applicability: UNITY/INOVA systems.
Syntax: oblique_gradient(level1, level2, level3, psi, phi, theta)
        double level1, level2, level3; /* gradient values in G/cm */

double psi, phi, theta;  /* Euler angles in degrees */

Description: Defines an oblique gradient with respect to the magnet reference frame. It has no return value. The gradient amplitudes (level1, level2, level3) are put through a coordinate transformation matrix using psi, phi, and theta to determine the actual x, y, and z gradient levels. These are then converted into DAC values and set with their corresponding gradient statements. For more coordinate system information, refer to the manual *VnmrJ Imaging, User Guide*.

The pulse sequence generation aborts if the DACs on a particular gradient are overrun after the angles and amplitude have been resolved.

Arguments: level1, level2, level3 are gradient values, in gauss/cm.
psi is an Euler angle, in degrees, with a range of –90 to +90.
phi is an Euler angle, in degrees, with the range of –180 to +180.
theta is an Euler angle, in degrees, with the range –90 to +90.

Examples: oblique_gradient(gvox1, 0, 0, vpsi, vphi, vtheta);

obl_shapedgradient  Execute a shaped oblique gradient

Applicability: *UNITY* *INOVA* systems.

Syntax: *UNITY* *INOVA* Systems
obl_shapedgradient(pat1, pat2, pat3, width, lvl1, lvl2, lvl3, loops, wait)
char *pat1, *pat2, *pat3;  /* names of gradient shapes */
double width;  /* gradient length in sec */
double lvl1, lvl2, lvl3;  /* gradient values in G/cm */
int loops;  /* times to loop waveform */
int wait;  /* WAIT or NOWAIT */

Description: Defines a shaped oblique gradient with respect to the magnet reference frame. The pulse sequence generation aborts if the DACs on a particular gradient are overrun after the angles and amplitude have been resolved.

Arguments: pat, pat1, pat2, and pat3 are names of gradient shapes.
width is the length of the gradient, in seconds.
level1, level2, level3 are gradient values, in gauss/cm.
loops is the number of times, from 1 to 255, to loop the waveform.
wait is a keyword, either WAIT or NOWAIT, that selects whether or not a delay is inserted to stop until the gradient has completed before executing the next statement.

Examples: *UNITY* *INOVA* systems
obl_shapedgradient("ramp_hold","","",
trise,gro,0.0,0.0,1,NOWAIT);

oblique_shapedgradient  Execute a shaped oblique gradient

Applicability: *UNITY* *INOVA* systems.

Syntax: oblique_shapedgradient(pat1, pat2, pat3, width, lvl1, lvl2, lvl3, psi, phi, theta, loops, wait)
char *pat1, *pat2, *pat3;  /* names of gradient shapes */
double width;  /* gradient length in sec */
Chapter 3. Pulse Sequence Statement Reference

```c
double lvl1, lvl2, lvl3; /* gradient values in G/cm */
double psi, phi, theta; /* Euler angles in degrees */
int loops; /* times to loop waveform */
int wait; /* WAIT or NOWAIT */
```

Description: Defines a shaped oblique gradient with respect to the magnet reference frame. The gradient patterns (pat1, pat2, pat3) and the gradient amplitudes (lvl1, lvl2, lvl3) are put through a coordinate transformation matrix using psi, phi, and theta to determine the actual x, y, and z gradient levels. pat1 and lvl1 correspond to the logical read-out axis. pat2 and lvl2 correspond to the logical phase-encode axis. pat3 and lvl3 correspond to the logical slice-select axis. Patterns are read in; scaled according to their respective amplitudes; rotated into x, y, and z patterns; rescaled; converted to DAC values; and written out to temporary files shapedgradient_x, shapedgradient_y, and shapedgradient_z in the user’s shapelib directory; and set with their corresponding shapedgradient statements. If an axis does not have a pattern, use empty quotes ("") to indicate a null pattern. The patterns must have the same number of points, or an integral multiple number of points. The pulse sequence generation aborts if the DACs on a particular gradient are overrun after the angles and amplitude have been resolved.

Arguments: pat1, pat2, pat3 are names of gradient shapes. width is the length of the gradient, in seconds. lvl1, lvl2, lvl3 are gradient values, in gauss/cm. psi is an Euler angle, in degrees, with a range of –90 to +90. phi is an Euler angle, in degrees, with the range –180 to +180. theta is an Euler angle, in degrees, with the range –90 to +90. loops is the number of times, from 1 to 255, to loop the waveform. wait is a keyword, either WAIT or NOWAIT, that selects whether or not a delay is inserted to stop until the gradient has completed before executing the next statement. WAIT or NOWAIT adds extra pulse sequence programming flexibility for imaging experiments. It allows performing other pulse sequence events during the gradient pulse. Because oblique_shapedgradient “talks” to the x, y, and z gradient axes, NOWAIT cannot be used to produce simultaneous oblique gradient pulses, even if they are orthogonal. In the following example,

```
oblique_shapedgradient (patx, tdelta, gdiff,
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1, NOWAIT);
oblique_shapedgradient (patty, tdelta, 0.0, gdiff,
  0.0, 0.0, 0.0, 0.0, 0.0, 1, NOWAIT);
oblique_shapedgradient (patz, tdelta, 0.0, 0.0, gdiff,
  0.0, 0.0, 0.0, 0.0, 1, WAIT);
```

the first two function calls set up all three gradients. In both cases, after a few microseconds, the gradient hardware is reset by the third function call, which is the only call fully executed. Even though the third call is executed, expect negative side-effects from the first two suppressed calls.

Examples: oblique_shapedgradient("ramp_hold", "", "", trise, gvox1,
  0.0, vpsi, vphi, vtheta, 1, NOWAIT);
obsblank  Blank amplifier associated with observe transmitter
Syntax: obsblank()
Description: Disables the amplifier for the observe transmitter. This statement is generally used after a call to obsunblank.
Related: decunblank Unblank amplifier associated with first decoupler
obsunblank Unblank amplifier associated with observe transmitter
rcvroff Turn off receiver
rcvron Turn on receiver

obsoffset  Change offset frequency of observe transmitter
Syntax: obsoffset(frequency)
double frequency; /* offset frequency */
Description: Changes the offset frequency, in Hz, of the observe transmitter (parameter tof). It is functionally the same as offset(frequency,OBSch).
• Systems with rf types A or B: the frequency typically changes between 10 to 30 μs, but 100 μs is automatically inserted into the sequence by the offset statement so that the time duration of offset is constant and not frequency-dependent.
• Systems with rf type C: which necessarily have PTS frequency synthesizers, the frequency shift time is 15.05 μs for standard, non-latching synthesizers and 21.5 μs for the latching synthesizers with the overrange/under-range option.
• UNITY INOVA, the frequency shift time is 4 μs.
• MERCURYplus/-Vx, this statement inserts a 86.4-μs delay, although the actual switching of the frequency takes 1 μs.
• Systems with an Output board (and only those systems): all offset statements by default are preceded internally by a 0.2-μs delay (see the apovrride statement for more details).
Arguments: frequency is the offset frequency desired for the observe channel.
Examples: obsoffset(to);
Related: decoffset Change offset frequency of first decoupler
dec2offset Change offset frequency of second decoupler
dec3offset Change offset frequency of third decoupler
offset Change offset frequency of transmitter or decoupler

obspower  Change observe transmitter power level
Applicability: UNITY/INOVA systems with linear amplifiers.
Syntax: obspower(power)
double power; /* new coarse power level */
Description: Changes observe transmitter power. This statement is functionally the same as ripower(value,OBSch).
Arguments: power sets the power level by assuming values from 0 (minimum power) to 63 (maximum power) on channels with a 63-dB attenuator or from –16 (minimum power) to 63 (maximum power) on channels with a 79-dB attenuator.
CAUTION: Be careful when using values of obspower 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. Use config to set a safety maximum for the tpwr, dpwr, dpwr2, and dpwr3 parameters.

Related: decpower Change first decoupler power
dec2power Change second decoupler power
dec3power Change third decoupler power
rlpower Change power level

obsprgoff End programmable control of observe transmitter

Applicability: UNITY/NOVA systems with a waveform generator on the observe transmitter channel.

Syntax: obsprgoff()

Description: Terminates any programmable phase and amplitude control on the observe transmitter started by the obsprgon statement under waveform control.

obsprgon Start programmable control of observe transmitter

Applicability: UNITY/NOVA systems with a waveform generator on the observe transmitter channel.

Syntax: obsprgon(pattern, 90_pulselength, tipangle_resoln)

Description: Executes programmable phase and amplitude control on the observe transmitter under waveform control. It returns the number of 12.5-ns ticks (as an integer value) in one cycle of the decoupling pattern. Explicit gating of the observe transmitter with xmtroff and xmtron is generally required. Arguments can be variables (which requires appropriate getval and getstr statements) to permit changes via parameters (see second example).

Arguments: pattern is the name of the text file (without the .DEC file suffix) in the shapelib directory that stores the decoupling pattern. 90_pulselength is the pulse duration, in seconds, for a 90° tip angle on the observe transmitter. tipangle_resoln is the resolution in tip-angle degrees to which the decoupling pattern is stored in the waveform generator.

Examples: obsprgon("waltz16", pw90, 90.0);
obsprgon("modulation", pp90, dres);
ticks=obsprgon("waltz16", pw90, 90.0);

Related: decpgon Start programmable decoupling on first decoupler
dec2prgon Start programmable decoupling on second decoupler
obsprgoff End programmable control of observe transmitter

obspulse Pulse observe transmitter with amplifier gating

Syntax: obspulse()

Description: A special case of the rgpulse (width, phase, RG1, RG2) statement, in which width is preset to pw and phase is preset to oph. Thus, obspulse
is exactly equivalent to `rgpulse(pw, oph, rof1, rof2)`. Note that `obspulse` has nothing whatsoever to do with data acquisition, despite its name. Except in special cases, data acquisition begins at the end of the pulse sequence.

Related:
- `pulse` Pulse observe transmitter with amplifier gating
- `rgpulse` Pulse observe transmitter with amplifier gating
- `simpulse` Pulse observe, decoupler channels simultaneously
- `sim3pulse` Simultaneous pulse on 2 or 3 rf channels

### obspwrf

**Set observe transmitter fine power**

**Applicability:** `UNITY/NOVA` systems.

**Syntax:**
```c
obspwrf(power)
```

**Description:** Changes observe transmitter fine power. This statement is functionally the same as `rlpwrf(value, OBSch)`. 

**Arguments:**
- `value` is the fine power desired.

**Examples:**
```
obspwrf(4.0);
```

Related:
- `decpwrf` Set first decoupler fine power
- `dec2pwrf` Set second decoupler fine power
- `dec3pwrf` Set third decoupler fine power
- `rlpwrf` Set transmitter or decoupler fine power

### obsstepsize

**Set step size for observe transmitter**

**Syntax:**
```c
obsstepsize(step_size)
```

**Description:** Sets the step size of the observe transmitter. This statement is functionally the same as `stepsize(base, OBSch)`. 

**Arguments:** `step_size` is the phase step size desired and is a real number or a variable. 

**Examples:**
```
obsstepsize(30.0);
```

Related:
- `decsstepsize` Set step size of first decoupler
- `dec2stepsize` Set step size of second decoupler
- `dec3stepsize` Set step size of third decoupler
- `stepsize` Set small-angle phase step size,

### obsunblank

**Unblank amplifier associated with observe transmitter**

**Syntax:**
```c
obsunblank()
```

**Description:** Explicitly enables the amplifier for the observe transmitter. `obsunblank` is generally followed by a call to `obsblank`. 

**Related:**
- `declblank` Blank amplifier associated with first decoupler
- `decunblank` Unblank amplifier associated with first decoupler
- `obslblank` Blank amplifier associated with observe transmitter
- `rcvroff` Turn off receiver
- `rcvron` Turn on receiver
offset  Change offset frequency of transmitter or decoupler

Use obsoffset, decoffset, dec2offset, or dec3offset, as appropriate, in place of this statement.

Applicability:

Syntax:  offset(frequency,device)
        double frequency; /* frequency offset */
        int device;    /* OBSch, DECch, DEC2ch, or DEC3ch */

Description: Changes the offset frequency of the observe transmitter (parameter tof), first decoupler (dof), second decoupler (dof2), or third decoupler (dof3).

Arguments: frequency is the offset frequency desired.
            device is OBSch (observe transmitter) or DECch (first decoupler).
            device can also be DEC2ch (second decoupler) or DEC3ch (third decoupler).

Examples:  offset(do2,DECch);
           offset(to2,OBSch);
           delay(d2);
           offset(tof,OBSch);

Related:  decoffset  Change offset frequency of first decoupler
          dec2offset  Change offset frequency of second decoupler
          dec3offset  Change offset frequency of third decoupler
          obsoffset  Change offset frequency of observe transmitter
          ioffset  Change offset frequency with IPA

Top  A  B  C  D  E  G  H  I  L  M  O  P  R  S  T  V  W  X  Z

pbox_ad180  Generate adiabatic 180 deg. shapes using Pbox
pbox_mix  Generate mixing shapes using Pbox.
pboxHT_F1  Generate arbitrary Hadamard encoded shapes in F1 using Pbox
pboxHT_F1e  Generate Hadamard encoded excitation shapes in F1 using Pbox
pboxHT_F1i  Generate Hadamard encoded inversion shapes in F1 using Pbox
pboxHT_F1s  Generate Hadamard encoded sequential inversion shapes
pboxHT_F1r  Generate Hadamard encoded refocusing shapes in F1 using Pbox
pe_gradient  Oblique gradient with phase encode in one axis
pe2_gradient  Oblique gradient with phase encode in two axes
pe3_gradient  Oblique gradient with phase encode in three axes
pe_shapedgradient  Oblique shaped gradient with phase encode in one axis
pe2_shapedgradient  Oblique shaped gradient with phase encode in two axes
pe3_shapedgradient  Oblique shaped gradient with phase encode in three axes
pe_loop  Phase-encode loop
phase_encode_gradient  Oblique gradient with phase encode in one axis
pbox_ad180 Generate adiabatic 180 deg. shapes using Pbox

Applicability: UNITY/INOVA systems.

Syntax: pbox_ad180(waveform, ref_pw90, ref_pwr)
char *waveform;
double ref_pw90;
int ref_pwr;

Description: Generates adiabatic 180 degree pulses for Pbox experiments.

Arguments: The pulse shape is defined by the argument waveform containing the
waveform name as defined by the cawurst inversion pulse. The ref_pwr is the
reference power level in dB and ref_pw90 is the reference 90 degree pulse
duration in microseconds.

Examples: static shape ad180;
ad180 = pbox_ad180("ad180", pwx, pwxlvl);

pbox_mix Generate mixing shapes using Pbox.

Applicability: UNITY/INOVA systems.

Syntax: pbox_mix(mix_pattern, waveform, mix_pwr, ref_pw90, ref_pwr)
char *mix_pattern, *waveform;
double ref_pw90;
int mix_pwr, ref_pwr;

Description: Generates decoupling mixing pulses for Pbox experiments.

Arguments: The pulse shape is defined by the argument waveform containing the
waveform name as defined in the wavelib/mixing directory by the
mix_pattern parameter. The mix_pwr parameter is the mixing pattern
power level in dB. The ref_pwr parameter is the reference power level in dB
and ref_pw90 is the reference 90 degree pulse duration in us.

Examples: static shape hhmix;
hhmix = pbox_mix("HHmix", "DIPSII2", mixpwr, pw*compH, tpwr);
Chapter 3. Pulse Sequence Statement Reference

**pboxHT_F1**  
Generate arbitrary Hadamard encoded shapes in F1 using Pbox

Applicability: UNITY/INOVA systems.

Syntax:  
```c
pboxHT_F1(waveform, ref_pw90, ref_pwr, type)
char *waveform, type;
double ref_pw90;
int ref_pwr;
```

Description: Generates arbitrary pulses for Hadamard experiments according to the Hadamard matrix size defined by ni.

Arguments: The pulse shape is defined by the argument waveform containing the waveform name as defined in the appropriate wavelib/ directory. The ref_pwr is the reference power level in dB and ref_pw90 is the reference 90 degree pulse duration in µs. Parameter type defines the shape type and can take values of 'e' (excitation pulses), 'i' (inversion pulses), 'r' (refocusing pulses) and 's' (sequential inversion pulses).

Examples:  
```c
pboxHT_F1("rsnob", pwH*compH, pwHlvl, 'r');
```

Related:  
- **pboxHT_F1e** Generate Hadamard encoded excitation shapes in F1 using Pbox  
- **pboxHT_F1s** Generate Hadamard encoded sequential inversion shapes in F1 using Pbox  
- **pboxHT_F1r** Generate Hadamard encoded refocusing shapes in F1 using Pbox  
- **pboxHT_F1i** Generate Hadamard encoded inversion shapes in F1 using Pbox

**pboxHT_F1e**  
Generate Hadamard encoded excitation shapes in F1 using Pbox

Applicability: UNITY/INOVA systems.

Syntax:  
```c
pboxHT_F1e(waveform, ref_pw90, ref_pwr)
char *waveform;
double ref_pw90;
int ref_pwr;
```

Description: Generates excitation pulses for Hadamard experiments according to the Hadamard matrix size defined by ni. The pulse element is applied with zero phase if the Hadamard matrix element is '+' and with 180-degree phase if the Hadamard matrix element is '–'.

Arguments: The pulse shape is defined by the argument waveform containing the waveform name as defined in wavelib. The ref_pwr is the reference power level in dB and ref_pw90 is the reference 90 degree pulse duration in µs.

Examples:  
```c
pboxHT_F1e("esnob", pwH*compH, pwHlvl);
```

Related:  
- **pboxHT_F1i** Generate Hadamard encoded inversion shapes in F1 using Pbox  
- **pboxHT_F1s** Generate Hadamard encoded sequential inversion shapes in F1 using Pbox  
- **pboxHT_F1r** Generate Hadamard encoded refocusing shapes in F1 using Pbox  
- **pboxHT_F1** Generate Hadamard encoded arbitrary shapes in F1 using Pbox

**pboxHT_F1i**  
Generate Hadamard encoded inversion shapes in F1 using Pbox

Applicability: UNITY/INOVA systems.

Syntax:  
```c
pboxHT_F1i(waveform, ref_pw90, ref_pwr)
char *waveform; double ref_pw90;
int ref_pwr;
```

Description: Generates Hadamard encoded inversion shapes in F1 using Pbox.

Arguments: The pulse shape is defined by the argument waveform containing the waveform name as defined in wavelib. The ref_pwr is the reference power level in dB and ref_pw90 is the reference 90 degree pulse duration in µs.

Examples:  
```c
pboxHT_F1i("ihnob", pwH*compH, pwHlvl);
```
Description: Generates inversion pulses for Hadamard experiments according to the Hadamard matrix size defined by \( n_1 \). The pulse elements are encoded according to the ‘on/off’ principle, where the pulse element is applied if the Hadamard matrix element is ‘+’ and is not applied if the Hadamard matrix element is ‘-’.

Arguments: The pulse shape is defined by the argument waveform containing the waveform name as defined in wavelib. The \( \text{ref\_pwr} \) is the reference power level in dB and \( \text{ref\_pw90} \) is the reference 90 degree pulse duration in µs.

Examples: \( \text{pboxHT\_F1i} \("\text{gaus180}\", \text{pwC*compC, pwClvl}); \)

Related: \( \text{pboxHT\_Fle} \) Generate Hadamard encoded excitation shapes in F1 using Pbox
\( \text{pboxHT\_Fli} \) Generate Hadamard encoded inversion shapes in F1 using Pbox
\( \text{pboxHT\_Flr} \) Generate Hadamard encoded refocusing shapes in F1 using Pbox
\( \text{pboxHT\_F1} \) Generate Hadamard encoded arbitrary shapes in F1 using Pbox

\( \text{pboxHT\_F1s} \) Generate Hadamard encoded sequential inversion shapes

Applicability: \( \text{UNITY/INOVA} \) systems.

Syntax: \( \text{pboxHT\_F1s(waveform, ref\_pw90, ref\_pwr)} \)
\[
\begin{align*}
\text{char *waveform;} \\
\text{double ref\_pw90;} \\
\text{int ref\_pwr;}
\end{align*}
\]

Description: Generates inversion pulses for Hadamard experiments according to the Hadamard matrix size defined by \( n_1 \). The pulse elements are encoded sequentially (inversion of individual sites is carried out sequentially rather than simultaneously) according to the ‘on/off’ principle, where the pulse element is applied if the Hadamard matrix element is ‘+’ and is not applied if the Hadamard matrix element is ‘-’.

Arguments: The pulse shape is defined by the argument waveform containing the waveform name as defined in wavelib. The \( \text{ref\_pwr} \) is the reference power level in dB and \( \text{ref\_pw90} \) is the reference 90 degree pulse duration in µs.

Examples: \( \text{pboxHT\_F1s} \("\text{gaus180}\", \text{pwC*compC, pwClvl}); \)

Related: \( \text{pboxHT\_Fle} \) Generate Hadamard encoded excitation shapes in F1 using Pbox
\( \text{pboxHT\_Fli} \) Generate Hadamard encoded inversion shapes in F1 using Pbox
\( \text{pboxHT\_Flr} \) Generate Hadamard encoded refocusing shapes in F1 using Pbox
\( \text{pboxHT\_F1} \) Generate Hadamard encoded arbitrary shapes in F1 using Pbox

\( \text{pboxHT\_Flr} \) Generate Hadamard encoded refocusing shapes in F1 using Pbox

Applicability: \( \text{UNITY/INOVA} \) systems.

Syntax: \( \text{pboxHT\_Flr(waveform, ref\_pw90, ref\_pwr)} \)
\[
\begin{align*}
\text{char *waveform;} \\
\text{double ref\_pw90;} \\
\text{int ref\_pwr;}
\end{align*}
\]

Description: Generates refocusing pulses for Hadamard experiments according to the Hadamard matrix size defined by \( n_1 \). The pulse element is applied with zero phase if the Hadamard matrix element is ‘+’ and with 90-degree phase if the Hadamard matrix element is ‘-’.
Chapter 3. Pulse Sequence Statement Reference

The pulse shape is defined by the argument waveform containing the waveform name as defined in `wavelib/refocusing` directory. The `ref_pwr` is the reference power level in dB and `ref_pw90` is the reference 90 degree pulse duration in µs.

Examples:

```
Examples: pboxHT_F1r("rsnob", pWH*compH, pWHlvl);
```

Related:

- `pboxHT_F1e` Generate Hadamard encoded excitation shapes in F1 using Pbox
- `pboxHT_F1i` Generate Hadamard encoded inversion shapes in F1 using Pbox
- `pboxHT_F1s` Generate Hadamard encoded sequential inversion shapes in F1 using Pbox
- `pboxHT_F1` Generate Hadamard encoded arbitrary shapes in F1 using Pbox

**peGradient**  
**Oblique gradient with phase encode in one axis**

**Applicability:** `NOVA` systems.

**Syntax:**

```c
pe_gradient(stat1, stat2, stat3, step2, vmult2)
```

double stat1, stat2, stat3; /* static gradient components */
double step2; /* variable gradient stepsize */
codeint vmult2; /* real-time math variable */

**Description:** Oblique gradient levels with one phase encode. The phase encode gradient is associated with the second axis of the logical frame. This corresponds to the convention read, phase, slice for the functions of the logical frame axes.

On `UNITY`/`NOVA` systems `pe_gradient` is same as the statement `phase_encode_gradient` except the Euler angles are read from the default set for imaging. `lim2` is automatically set to half the `nv` (number of views) where `nv` is usually the number of phase encode steps.

Pulse sequence generation aborts if the DACs on a particular gradient are overrun after the angles and amplitude have been resolved.

**Arguments:**

- `stat1`, `stat2`, `stat3` are values, in gauss/cm, of the components for the static portion of the gradient in the logical reference frame.
- `step2` is the value, in gauss/cm, of the component for the step size change in the variable portion of the gradient.
- `vmult2` is a real-time math variable (`v1` to `v14`, `ct`, `zero`, `one`, `two`, `three`) or reference to tables (`t1` to `t60`), whose associated values vary dynamically in a manner controlled by the user.

**Examples:**

```
Examples: pe_gradient(0.0,-sgpe*nv/2.0,gss,sgpe,v6);
```

Related:

- `phase_encode_gradient` Oblique gradient with phase encode in 1 axis

**pe2Gradient**  
**Oblique gradient with phase encode in two axes**

**Applicability:** `UNITY`/`NOVA` systems.

**Syntax:**

```c
pe2_gradient(stat1, stat2, stat3, step2, step3, vmult2, vmult3)
```

double stat1, stat2, stat3; /* static gradient components */
double step2, step3; /* variable gradient stepsize */
codeint vmult2, vmult /* real-time math variables */

**Description:** Sets only two oblique phase encode gradients.

Pulse sequence generation aborts if the DACs on a particular gradient are overrun after the angles and amplitude have been resolved.

**Arguments:**

- `stat1`, `stat2`, `stat3` are values, in gauss/cm, of the components for the static portion of the gradient in the logical reference frame.
step2, step3 are values, in gauss/cm, of the components for the step size change in the variable portion of the gradient.

vmult2, vmult3 are real-time math variables (v1 to v14, ct, zero, one, two, three) or references to tables (t1 to t60), whose associated values vary dynamically in a manner controlled by the user.

Examples: pe2_gradient (gro, sgpe*nv/2.0, sgpe2*nv2/2.0, sgpe, sgpe2, v6, v8);

Related: pe3_gradient  Oblique gradient with phase encode in 3 axes

pe3_gradient  Oblique gradient with phase encode in three axes
Applicability: UNITY INOVA systems.
Syntax:
```
pe3_gradient(stat1,stat2,stat3,step1,step2,step3,
vmult1,vmult2,vmult3)
```

double stat1,stat2,stat3; /* static gradient components */
double step1,step2,step3; /* gradient step sizes */
codeint vmult1,vmult2,vmult3; /* real-time variables */

Description:
Three oblique phase encode gradients.

pe_gradient is same as phase_encode3_gradient except the Euler angles are read from the default set for imaging. lim1, lim2, and lim3 are set to nv/2, nv2/2, and nv3/2, respectively.

Pulse sequence generation aborts if the DACs on a particular gradient are overrun after the angles and amplitude have been resolved.

Arguments:
stat1, stat2, stat3 are values, in gauss/cm, of the components for the static portion of the gradient in the logical reference frame.
step1, step2, step3 are values, in gauss/cm, of the components for the step size change in the variable portion of the gradient.
vmult1, vmult2, vmult3 are real-time math variables (v1 to v14, ct, zero, one, two, three) or references to AP tables (t1 to t60) whose associated values vary dynamically in a manner controlled by the user.

Examples: pe3_gradient (gro, sgpe*nv/2.0, sgpe2*nv2/2.0, 0.0, sgpe, sgpe2, zero, v6, v8);

pe_shapedgradient  Oblique shaped gradient with phase encode in one axis
Applicability: UNITY INOVA systems.
Syntax:
```
pe_shapedgradient(pattern,width,
stat1,stat2,stat3,step2,vmult2,wait,tag)
```

char *pattern; /* name of gradient shape file */
double width; /* width of gradient in sec */
double stat1,stat2,stat3; /* static gradient components */
double step2; /* variable gradient step size */
codeint vmult2; /* real-time math variable */
int wait; /* WAIT or NOWAIT */
int tag; /* tag to a gradient element */

Description:
Static oblique shaped gradient one phase encode shaped gradient.

Pulse sequence generation aborts if the DACs on a particular gradient are overrun after the angles and amplitude have been resolved.
Chapter 3. Pulse Sequence Statement Reference

On UNITY INOVA systems pe_shapedgradient is same as\nphase_encode_shapedgradient except in pe_shapedgradient\nthe Euler angles are read from the default set for imaging. lim2 is\nautomatically set to \n\nArguments:\n
pattern is the name of a gradient shape file.
width is the length, in seconds, of the gradient.
stat1, stat2, stat3 are values, in gauss/cm, of the components for the\nstatic portion of the gradient in the logical reference frame.
step2 is the value, in gauss/cm, of the component for the step size change in\nthe variable portion of the gradient.
vmult2 is a real-time math variable (v1 to v14, ct, zero, one, two,\nthree) or reference to tables (t1 to t60) whose associated values vary\ndynamically in a manner controlled by the user.
wait is a keyword, either WAIT or NOWAIT, that selects whether or not a\ndelay is inserted to wait until the gradient has completed before executing the\nnext statement.
tag is a unique integer that “tags” the gradient element from any other gradient\nelements used in the sequence. These tags are used for variable amplitude\npulses.

pe2_shapedgradient Oblique shaped gradient with phase encode in two axes

Applicability: UNITY INOVA systems.

Syntax:\npe2_shapedgradient(pattern, width, stat1, stat2, stat3,\nstep2, step3, vmult2, vmult3)

char *pattern; /* name of gradient shape file */
double width; /* length of gradient in sec */
double stat1, stat2, stat3; /* static gradient components */
double step2, step3; /* variable gradient step size */
codeint vmult2, vmult3; /* real-time math variables */

description: Sets two oblique phase encode shaped gradients; otherwise, this statement is the\nsame as pe3_shapedgradient.
Pulse sequence generation aborts if the DACs on a particular gradient are\noverrun after the angles and amplitude have been resolved.

Arguments:\n
pattern is the name of a gradient shape file.
width is the length, in seconds, of the gradient.
stat1, stat2, stat3 are values, in gauss/cm, of the components for the\nstatic portion of the gradient in the logical reference frame.
step2, step3 are values, in gauss/cm, of the components for the step size\nchange in the variable portion of the gradient.
vmult2, vmult3 are real-time math variables (v1 to v14, ct, zero,\none, two, three) or references to tables (t1 to t60) whose associated\nvalues vary dynamically in a manner controlled by the user.

Related: pe3_shapedgradient Oblique shaped gradient with phase encode in 3 axes
pe3_shapedgradient  Oblique shaped gradient with phase encode in three axes

Applicability: UNITY/INOVA systems.

Syntax: pe3_shapedgradient(pattern, width, stat1, stat2, stat3, step1, step2, step3, vmult1, vmult2, vmult3)

char *pattern;  /* name of gradient shape file */
double width;    /* width of gradient in sec */
double stat1, stat2, stat3;  /* static gradient components */
double step1, step2, step3;  /* var. gradient components */
codeint vmult1, vmult2, vmult3;  /* real-time variables */

Description:

On UNITY/INOVA systems pe3_shapedgradient is same as phase_encode3_shapedgradient except the Euler angles are read from the default set for imaging. The lim1, lim2, and lim3 arguments in phase_encode3_shapedgradient are set to nv/2, nv2/2, and nv3/2, respectively.
Pulse sequence generation aborts if the DACs on a particular gradient are overrun after the angles and amplitude have been resolved.

Arguments:

pattern is the name of a gradient shape file.
width is the length, in seconds, of the gradient.
stat1, stat2, stat3 are values, in gauss/cm, of the components for the static portion of the gradient in the logical reference frame.
step1, step2, step3 are values, in gauss/cm, of the components for the step size change in the variable portion of the gradient.
vmult1, vmult2, vmult3 are real-time math variables (v1 to v14, ct, zero, one, two, three) or references to tables (t1 to t60) whose associated values vary dynamically in a manner controlled by the user.
wait is a keyword, either WAIT or NOWAIT, that selects whether or not a delay is inserted to wait until the gradient has completed before executing the next statement.

peloop  Phase-encode loop

Applicability: UNITY/INOVA systems.

Syntax: peloop(state, max_count, apv1, apv2)

char state;  /* compressed or standard */
double max_count;  /* initializes apv1 */
codeint apv1;  /* maximum count */
codeint apv2;  /* current counter value */

Description: Provides a sequence-switchable loop that can use real-time variables in what is known as a compressed loop, or it 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.
Chapter 3. Pulse Sequence Statement Reference

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
's' and max_count is greater than zero, apv2 is set to nth2D-1;
otherwise, it is set to zero.

Examples: peloop(seqcon[2],nv,v5,v6);
          msloop(seqcon[1],nv,v11,v12);
          ... 
          poffset_list(pss,gss,ns,v12):
          ... 
          pe_gradient(gror,-0.5*sgpe*nv,gssr,sgpe,v6);
          ... 
          acquire(np,1.0/sw);
          ... 
          endmsloop(seqcon[1],v12);
          endpeloop(seqcon{2},v6;

Related:  endpeloop   End phase-encode loop
          loop   Start loop
          msloop   Multislice loop

phase_encode_gradient   Oblique gradient with phase encode in one axis

Applicability: unty/NOVA systems.

Syntax:  phase_encode_gradient(stat1,stat2,stat3,step2,
          vmult2,lim2,ang1, ang2, ang3)
        double stat1,stat2,stat3; /* static gradient components */
        double step2; /* variable gradient stepsize */
        codeint vmult2; /* real-time math variable */
        double lim2; /* max. gradient value step */
        double ang1,ang2,ang3; /* Euler angles in degrees */

Description: Sets static oblique gradient levels plus one oblique phase encode gradient. The
phase encode gradient is associated with the second axis of the logical frame.
This corresponds to the convention: read, phase, slice for the functions of the
logical frame axes. It has no return value.
Pulse sequence generation aborts if the DACs on a particular gradient are
overrun after the angles and amplitude have been resolved.

Arguments: stat1,stat2,stat3 are values, in gauss/cm, of the components for the
static portion of the gradient in the logical reference frame.
step2 is the value, in gauss/cm, of the component for the step size change in
the variable portion of the gradient.
vmult2 is a real-time math variable(v1-v14,ct,zero,one,two,three)
or reference to tables (t1 to t60), whose associated values vary dynamically
in a manner controlled by the user.
lim2 is a value representing the dynamic step that will generate the maximum
gradient value for each component. This provides error checking in pulse
sequence generation and is normally nv/2.
ang1 is Euler angle psi, in degrees, with the range –90 to +90.
ang2 is Euler angle phi, in degrees, with the range –180 to +180.
ang3 is Euler angle theta, in degrees, with the range –90 to +90.
phase_encode3_gradient  Oblique gradient with phase encode in three axes

Applicability:  

Syntax:  

```c
phase_encode3_gradient(stat1,stat2,stat3,
    step1,step2,step3,vmult1,vmult2,vmult3,
    lim1,lim2,lim3,ang1,ang2,ang3)
```

- `double stat1,stat2,stat3; /* static gradient components */`
- `double step1,step2,step3; /* var. gradient stepsize */`
- `codeint vmult1,vmult2,vmult3; /* real-time variables */`
- `double lim1,lim2,lim3; /* max. gradient value steps */`
- `double ang1,ang2,ang3; /* Euler angles in degrees */`

Description:  
Sets three oblique phase encode gradients. It has no return value. 

Pulse sequence generation aborts if the DACs on a particular gradient are overlapped after the angles and amplitude have been resolved.

Arguments:  
- `stat1,stat2,stat3` are values, in gauss/cm, of the components for the static portion of the gradient in the logical reference frame.
- `step1,step2,step3` are values, in gauss/cm, of the components for the step size change in the variable portion of the gradient.
- `vmult1,vmult2,vmult3` are real-time math variables (\(v_1\) to \(v_{14}\), ct, zero, one, two, three) or references to tables (\(t_1\) to \(t_{60}\)) whose associated values vary dynamically in a manner controlled by the user.
- `lim1,lim2,lim3` are values representing the dynamic step that will generate the maximum gradient value for each component. This provides error checking in pulse sequence generation and is normally \(nv/2\).
- `ang1` is Euler angle \(\psi\), in degrees, with the range \(-90\) to \(+90\).
- `ang2` is Euler angle \(\phi\), in degrees, with the range \(-180\) to \(+180\).
- `ang3` is Euler angle \(\theta\), in degrees, with the range \(-90\) to \(+90\).

Examples:  
```
phase_encode3_gradient(0,0,0,0,0,2.0*gcrush/ne, \
    zero,zero,v12,0,0,0,psi,phi,theta);
```

Related:  
- `oblique_gradient`  
- `oblique_shapedgradient`  
- `pe_gradient`  
- `phase_encode_shapedgradient`  
- `phase_encode3_gradient`  
- `phase_encode3_shapedgradient`  
- `pe3_gradient`  
- `phase_encode_shapedgradient`  
- `phase_encode3_shapedgradient`
Chapter 3. Pulse Sequence Statement Reference

```
codeint vmult2;        /* real-time math variable */
double lim2;           /* max. gradient value steps */
double ang1,ang2,ang3; /* Euler angles in degrees */
codeint vloops;        /* number of loops */
int wait;              /* WAIT or NOWAIT */
int tag;               /* tag to a gradient element */
```

Description: Sets static oblique shaped gradients plus one oblique phase encode shaped gradient. The phase encode gradient is associated with the second axis of the logical frame. This corresponds to the convention: read, phase, slice for the functions of the logical frame axes. One gradient shape is used for all three axes. It has no return value.

Pulse sequence generation aborts if the DACs on a particular gradient are overrun after the angles and amplitude have been resolved.

Arguments:

- **pattern** is the name of a gradient shape file.
- **width** is the length, in seconds, of the gradient.
- **stat1**, **stat2**, **stat3** are values, in gauss/cm, of the components for the static portion of the gradient in the logical reference frame.
- **step2** is the value, in gauss/cm, of the component for the step size change in the variable portion of the gradient.
- **vmult2** is a real-time math variable (v1 to v14, ct, zero, one, two, three) or reference to tables (t1 to t60) whose associated values vary dynamically in a manner controlled by the user.
- **lim2** is the value representing the dynamic step that will generate the maximum gradient value for the component. This provides error checking in pulse sequence generation and is normally \( nV/2 \).
- **ang1** is the Euler angle \( \psi \), in degrees, with the range of \(-90\) to \(+90\).
- **ang2** is the Euler angle \( \phi \), in degrees, with the range of \(-180\) to \(+180\).
- **ang3** is the Euler angle \( \theta \), in degrees, with the range of \(-90\) to \(+90\).
- **vloops** is a real-time math variable (v1 to v14, ct, zero, one, two, three) or references to tables (t1 to t60) that dynamically sets the number of times to loop the waveform.
- **wait** is a keyword, either WAIT or NOWAIT, that selects whether or not a delay is inserted to wait until the gradient has completed before executing the next statement.
- **tag** is a unique integer that “tags” the gradient element from any other gradient elements used in the sequence. These tags are used for variable amplitude pulses.

Related:
- oblique_gradient
- oblique_shapedgradient
- pe_shapedgradient
- phase_encode3_shapedgradient

Syntax:
```
phase_encode3_shapedgradient(pattern,width,stat1,stat2,stat3,
step1,step2,step3,vmult1,vmult2,vmult3,
lim1,lim2,lim3,ang1,ang2,ang3,loops,wait)
```

Applicability: **UNITY**/**NOVA** systems.
char *pattern; /* name of gradient shape file */
double width; /* width of gradient in sec */
double stat1,stat2,stat3; /* static gradient components */
double step1,step2,step3; /* var. gradient step sizes */
codeint vmult1,vmult2,vmult3; /* real-time variables */
double lim1,lim2,lim3; /* max. gradient value steps */
double ang1,ang2,ang3; /* Euler angles in degrees */
int loops; /* number of times to loop */
int wait; /* WAIT or NOWAIT */

Description: Sets three oblique phase encode shaped gradient. Note that this statement has a
loops argument that is an integer, as opposed to the vloops argument in
phase_encode_shapedgradient. It has no return value.
Pulse sequence generation aborts if the DACs on a particular gradient are
overrun after the angles and amplitude have been resolved.

Arguments: pattern is the name of the gradient shape file.
width is the length, in seconds, of the gradient.
stat1, stat2, stat3 are values, in gauss/cm, of the components for the
static portion of the gradient in the logical reference frame.
step1, step2, step3 are values, in gauss/cm, of the components for the step
size change in the variable portion of the gradient.
vmult1, vmult2, vmult3 are real-time math variables (v1 to v14, ct,
zero, one, two, three) or references to tables (t1 to t60) whose associated
values vary dynamically in a manner controlled by the user.
lim1, lim2, lim3 are values representing the dynamic step that will generate
the maximum gradient value for each component. This provides error checking
in pulse sequence generation and is normally nv/2.
ang1 is the Euler angle psi, in degrees, with the range of –90 to +90.
ang2 is the Euler angle phi, in degrees, with the range of –180 to +180.
ang3 is the Euler angle theta, in degrees, with the range of –90 to +90.
loops is non-real-time integer value, from 1 to 255, that sets the number of
times to loop the waveform.
wait is a keyword, either WAIT or NOWAIT, that selects whether or not a
delay is inserted to wait until the gradient has completed before executing
the next statement.

Related: pe3_shapedgradient Oblique sh. gradient with PE in 3 axes
        phase_encode_shapedgradient Oblique sh. gradient with PE on 1 axis
        phase_encode3_gradient Oblique gradient with PE in 3 axes

phaseshift Set phase-pulse technique, rf type A or B

Applicability: Systems with rf type A or B (MERCURYplus/-Vx systems are rf type E or F).

Syntax: phaseshift (base,multiplier,device)
double base; /* base small-angle phase shift */
codeint multiplier; /* real-time variable */
int device; /* channel, TODEV or DODEV */

Description: Implements the “phase-pulse” technique.

Arguments: base is a real number, expression, or variable representing the base phase shift
in degrees. Any value is acceptable.
multiplier is a real-time variable (v1 to v14, ct, etc.). The value must be positive. The actual phase shift is \(((\text{base} \cdot \text{multiplier}) \mod 360)\).

device is TODEV (observe transmitter) or DODEV (first decoupler).

Examples:
- \text{phaseshift}(60.0,\text{ct},\text{TODEV});
- \text{phaseshift}(-30.0,\text{v1},\text{DODEV});

\textbf{poffset} \hspace{1cm} \textbf{Set frequency based on position}

\textbf{Applicability}: Unity/INOVA systems.

\textbf{Syntax}:
\begin{verbatim}
double position; /* slice position in cm */
double level;   /* gradient level in G/cm */
\end{verbatim}

\textbf{Description}: Sets the rf frequency from position and conjugate gradient values. poffset is functionally the same as \texttt{position\_offset} except that poffset takes the value of \texttt{resfrq} from the \texttt{resto} parameter and always assumes the device is the observe transmitter.

\textbf{Arguments}:
- \texttt{position} is the slice position, in cm.
- \texttt{level} is the gradient level, in gauss/cm, used in the slice selection process.

\textbf{Examples}:
- \texttt{poffset(\text{pss}[0],\text{gss});}

\textbf{Related}: \texttt{position\_offset} \hspace{1cm} \text{Set frequency based on position}

\textbf{poffset\_list} \hspace{1cm} \textbf{Set frequency from position list}

\textbf{Applicability}: Unity/INOVA systems.

\textbf{Syntax}:
\begin{verbatim}
double position_array[]; /* position values in cm */
double level;   /* gradient level in G/cm */
double nslides; /* number of slices */
codeint vi;    /* variable or table */
\end{verbatim}

\textbf{Description}: Sets the rf frequency from a position list, conjugate gradient value, and dynamic math selector. poffset\_list is functionally the same as \texttt{position\_offset\_list} except that poffset\_list takes the value of \texttt{resfrq} from the \texttt{resto} parameter, assumes the device is the observe transmitter device OBSch, and assumes that the list number is zero.

\textbf{Arguments}:
- \texttt{position\_array} is a list of position values, in cm.
- \texttt{level} is the gradient level, in gauss/cm, used in the slice selection process.
- \texttt{nslides} is the number of slices or position values.
- \texttt{vi} is a dynamic real-time variable (v1 to v14) or table (t1 to t60).

\textbf{Examples}:
- \texttt{poffset\_list(\text{pss},\text{gss},\text{ns},\text{v8});}

\textbf{Related}: \texttt{getarray} \hspace{1cm} \text{Retrieves all values of an arrayed parameter}

\texttt{position\_offset\_list} \hspace{1cm} \text{Set frequency from position list}
int device;    /* OBSch, DECch, DEC2ch, or DEC3ch */

Description: Sets the rf frequency from position and conjugate gradient values. It has no return value.

Arguments: pos is the slice position, in cm.
grad is the gradient level, in gauss/cm, used in the slice selection process.
resfrq is the resonance offset value, in Hz, for the nucleus of interest.
device is OBSch (observe transmitter) or DECch (first decoupler). device can also be DEC2ch (second decoupler) or DEC3ch (third decoupler).

Examples: position_offset(pos1,gvox1,resto,OBSch);

Related: poffset Set frequency based on position
position_offset_list Set frequency from position list

position_offset_list Set frequency from position list

Applicability: UNITY INOVA systems.

Syntax: position_offset_list(posarray,grad,nslices, \
    resfrq,device,list_number,apv1)
    double posarray[];    /* position values in cm */
    double level;    /* gradient level in G/cm */
    double nslices;    /* number of slices */
    double resfrq;    /* resonance offset in Hz */
    int device;    /* OBSch, DECch, DEC2ch, or DEC3ch */
    int list_number;    /* number for global list */
    codeint vi;    /* real-time variable or table */

Description: Sets the rf frequency from a position list, conjugate gradient value, and dynamic math selector. The dynamic math selector (apv1) holds the index for required slice offset value as stored in the array. The arrays provided to this statement must count zero up; that is, array[0] must have the first slice position and array[ns-1] the last. It has no return value.

Arguments: position_array is a list of position values, in cm.
level is the gradient level, in gauss/cm, used in the slice selection process.
nslices is the number of slices or position values.
resfrq is the resonance offset, in Hz, for the nucleus of interest.
device is OBSch (observe transmitter) or DECch (first decoupler). device can also be DEC2ch (second decoupler) or DEC3ch (third decoupler).
list_number is a value for identifying a global list. The first global list must begin at zero and each created list must be incremented by one.
vi is a dynamic real-time variable (v1 to v14) or table (t1 to t60).

Related: getarray Retrieves all values of an arrayed parameter
poffset_list Set frequency from position list
position_offset Set frequency based on position

power Change power level

Applicability: Systems with linear amplifiers. Use the statements obspower, decpower, dec2power, or dec3power, as appropriate, in preference to power.

Syntax: power(power,device)
Chapter 3. Pulse Sequence Statement Reference

int power;  // new value for coarse power control */
int device;  // OBSch, DECch, DEC2ch, or DEC3ch */

Description: Changes transmitter or decoupler power by assuming values of 0 (minimum power) to 63 (maximum power) on channels with a 63-dB attenuator or –16 (minimum power) to 63 (maximum power) on channels with a 79-dB attenuator. On systems with an Output board, by default, power statements are preceded internally by a 0.2-µs delay (see the apovrride statement for more details).

Arguments: power is the power desired. It must be stored in a real-time variable (v1-v14, etc.), which means it cannot be placed directly in the power statement. This allows the power to be changed in real-time or from pulse to pulse. Setting the power argument is most commonly done using initval (see the example). To avoid consuming a real-time variable, use the rlpower statement instead of the power statement.

device is OBSch (observe transmitter) or DECch (first decoupler). device can also be DEC2ch (second decoupler) or DEC3ch (third decoupler).

CAUTION: On systems with linear amplifiers, be careful when using values of power 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. Use config to set a safety maximum for the tpwr, dpwr, dpwr2, and dpwr3 parameters.

Examples: pulsesequence()
{   
 double newpwr;
 newpwr=getval("newpwr");
 initval(newpwr,v2);
 power(v2,OBSch);
 ...
 }

Related: decpower Change first decoupler power
dec2power Change second decoupler power
dec3power Change third decoupler power
initval Initialize a real-time variable to a specified value
obspower Change observe transmitter power
pwrf Change transmitter or decoupler fine power
rlpower Change transmitter or decoupler power, linear amplifier
rlpwrf Set transmitter or decoupler fine power

psg_abort Abort the PSG process

Syntax: psg_abort (int_error)

Description: psg_abort aborts the PSG process. The acquisition will not start. the error argument is typically 1.

pulse Pulse observe transmitter with amplifier gating

Syntax: pulse (width, phase)
            double width;    // pulse length in sec */
            codeint phase;   // real-time variable for phase */

Description: Turns on a pulse the same as the rgpulse (width, phase, RG1, RG2) statement, but with RG1 and RG2 set to the parameters rof1 and rof2,
respectively. Thus, pulse is a special case of \texttt{rgpulse} where the “hidden”
parameters rof1 and rof2 remain “hidden.”

**Arguments:**
- \texttt{width} specifies the width of the observe transmitter pulse.
- \texttt{phase} sets the phase and must be a real-time variable.

**Examples:**
\begin{verbatim}
pulse(pw,v2);
\end{verbatim}

**Related:**
- \texttt{dps\_show} Draw delay or pulses in a sequence for graphical display
- \texttt{obspulse} Pulse observe transmitter with IPA
- \texttt{irgpulse} Pulse observe transmitter with IPA
- \texttt{obspulse} Pulse observe transmitter with amplifier gating
- \texttt{rgpulse} Pulse observe transmitter with amplifier gating
- \texttt{simpulse} Pulse observe, decoupler channels simultaneously
- \texttt{sim3pulse} Simultaneous pulse on 2 or 3 rf channels
putCmd("banner('acquisition started')");
putCmd("dps");

**pwr**

**Change transmitter or decoupler fine power**

**Applicability:** *UNITY/INOVA* systems

Use `obspwrf`, `decpwrf`, `declpwrf`, or `dec3pwrf`.

**Syntax:** `pwr` `f` (`power`,`device`)  
`int power;` /* new value for fine power control */  
`int device;` /* OBSch, DECch, DEC2ch, or DEC3ch */

**Description:** Changes the fine power of the device specified by adjusting the optional fine attenuators. Do not execute `pwr` and `ipwr` together because they will cancel each other's effect.

**Arguments:** `power` is the fine power desired. It must be a real-time variable (v1 to v14, etc.), which means it cannot be placed directly in the `pwr` statement. It can range from 0 to 4095 (60 dB on *UNITY/INOVA*, about 6 dB on other systems).

`device` is OBSch (observe transmitter) or DECch (first decoupler). On the *UNITY/INOVA* only, `device` can also be DEC2ch (second decoupler) or DEC3ch (third decoupler).

**Examples:** `pwr(v1,OBSch)`;

**Related:**
- `ipwr` Change transmitter or decoupler fine power
- `power` Change transmitter or decoupler power, linear amp. system
- `rlpwr` Set transmitter or decoupler fine power

**pwr**

**Change transmitter or decoupler linear modulator power**

*UNITY/INOVA* systems only. Use of statements `obspwrf`, `decpwrf`, `dec2pwrf`, or `dec3pwrf`, as appropriate, is preferred.

**Syntax:** `pwr` `m` (`power`,`device`)  
`int power;` /* new value for fine power control */  
`int device;` /* OBSch, DECch, DEC2ch, or DEC3ch */

**Description:** Changes the linear modulator power of the device specified by adjusting the optional fine attenuators. Do not execute `pwr` and `ipwr` together because they will cancel each other's effect.

**Arguments:** `power` is the linear modulator power desired. It must be a real-time variable (v1 to v14, etc.), which means the power level as an integer cannot be placed directly in the `pwr` statement. It can range from 0 to 4095 (60 dB on *UNITY/INOVA*).

`device` is OBSch (observe transmitter) or DECch (first decoupler). `device` can also be DEC2ch (second decoupler) or DEC3ch (third decoupler).

**Examples:** `pwr(v1,OBSch)`;

**Related:**
- `decpwrf` Set first decoupler fine power
- `dec2pwrf` Set second decoupler fine power
- `dec3pwrf` Set third decoupler fine power
- `ipwr` Change transmitter or decoupler fine power with IPA
- `ipwr` Change transmitter or decoupler linear modulator power
- `obspwrf` Set observe transmitter fine power
- `rlpwr` Set transmitter or decoupler linear modulator power
### rcvroff

**Turn off receiver gate and amplifier blanking gate**

**Syntax:** rcvroff()

**Description:** The receiver is normally off during the pulse sequence and is turned on only during acquisition. The `rcvroff` statement also unblanks, or enables, the observe transmitter.

Receiver gating is normally controlled automatically by `decpulse`, `decr2rgpulse`, `decr3rgpulse`, `obspulse`, `pulse`, and `rgpulse`. At the end of each of these statements, the receiver is automatically turned back on if and only if the receiver has not been previously turned off explicitly by a `rcvroff` statement. In all cases, the receiver is implicitly turned back on immediately prior to data acquisition.

**Related:**
- `rcvron` - Turn on receiver gate and amplifier blanking gate
- `recoff` - Turn off receiver only
- `recon` - Turn on receiver only

### rcvron

**Turn on receiver gate and amplifier blanking gate**

**Syntax:** rcvron()

**Description:** The receiver is normally off during the pulse sequence. It is turned on only during acquisition. On other systems, `rcvron` provides explicit receiver gating in the pulse sequence. The `rcvron` statement also blanks, or disables, the observe transmitter.
Receiver gating is normally controlled automatically by `obspulse`, `pulse`, and `rgpulse`, `decpulse`, `decrgpulse`, `dec2rgpulse`, and `dec3rgpulse`. At the end of each of these statements, the receiver is automatically turned back on if and only if the receiver has not been previously turned off explicitly by a `rcvroff` statement. In all cases, the receiver is implicitly turned back on immediately prior to data acquisition.

The `rcvron` statement automatically executes a delay of `rof3` before turning on the receiver. If `rof3` is not defined then a delay of 2.0 \( \mu \text{s} \) is used. Usually the delay protects the receiver from being turned on immediately after an `rgpulse` statement but `rof3` can be set to zero in other circumstances where it does not immediately follow a pulse.

**Related:**
- `rcvroff` Turn off receiver gate and amplifier blanking gate
- `recoff` Turn off receiver gate only
- `recon` Turn on receiver gate only

**readuserap**  Read input from user AP register

**Applicability:** UNITY/INOVA systems

**Syntax:**
```
readuserap(vi)
```

codeint vi; /* index to value read in user AP register */

**Description:** Reads input from user AP bus register 3 to a real-time variable. The user can then act on this information using real-time math and real time control statements while the pulse sequence is running. Register 3 is lines 1 to 8 of the USER AP connector J8212 on the Breakout panel on the rear of the left console cabinet. This register interfaces to a bidirectional TTL-compatible 8-bit buffer, which has a 100-ohm series resistor for circuit protection.

readuserap stops parsing acodes (acquisition codes) until the lines in the buffer have been read and the value placed in to the specified real-time variable. In order for the parser to parse and stuff more words into the FIFO before underflowing, the readuserap statement puts in a 500 \( \mu \text{s} \) delay after reading the input. However, depending on what is to be done after reading the lines, a longer delay may be needed to avoid FIFO underflow.

If an error occurs in reading, a warning message is sent to the host and a value of –1 is returned to the real-time variable.

**Arguments:**
- `vi` is a real-time variable (v1 to v14, etc.) that indexes a signed or unsigned number read from user AP register 3.

**Examples:**
```
/* Check a value read in from input register and */
/* execute a pulse if it is the expected value. */
double testval;
testval=getval(testval) /* set value to check */
initval(testval,v2);
loop(two,v1); /* reset below makes loop go */
readuserap(v1); /* until expected value reads in */
delay(d2);
sub(v1,v2,v3);
ifzero(v3);
pulse(pw,oph);
assign(one,v1);
elseonz(v3)
    assign(zero,v1); /*reset counter*/
```
endif(v3);
endloop(v1);

Related: setuserap Set user AP register
vsetuserap Set user AP register using real-time variable

**recoff**  
**Turn off receiver gate only**

**Applicability:** *UNITY INOVA* systems.

**Syntax:** recoff()

**Description:** Receiver gating has been decoupled from amplifier blanking. The recoff statement is similar to the rcvroff statement in that it defaults the receiver off throughout the pulse sequence; however, unlike rcvroff, the recoff statement only affects the receiver gate and does not affect the amplifier blanking gate. In all cases, the receiver is turned off when applying pulses and turned on during acquisition. The default state of the receiver is off (except for whole body systems and for imaging pulses sequences that have the initparms_sis statement at the beginning).

**Related:**
- initparms_sis Initialize parameters for spectroscopy imaging sequences
- rcvroff Turn off receiver gate and amplifier blanking gate
- rcvron Turn on receiver gate and amplifier blanking gate
- recon Turn on receiver gate only

**recon**  
**Turn on receiver gate only**

**Applicability:** *UNITY INOVA* systems.

**Syntax:** recon()

**Description:** Receiver gating has been decoupled from amplifier blanking. The recoff statement is similar to the rcvron statement in that it defaults the receiver on throughout the pulse sequence; however, unlike rcvron, the recon statement only affects the receiver gate and does not affect the amplifier blanking gate. In all cases, the receiver is turned off when applying pulses and turned on during acquisition. The default state of the receiver is off (except for whole body systems and for imaging pulses sequences that have the initparms_sis statement at the beginning).

**Related:**
- initparms_sis Initialize parameters for spectroscopy imaging sequences
- rcvroff Turn off receiver gate and amplifier blanking gate
- rcvron Turn on receiver gate and amplifier blanking gate
- recoff Turn off receiver gate only

**rgpulse**  
**Pulse observe transmitter with amplifier gating**

**Syntax:** rgpulse(width,phase,RG1,RG2)
  
  double width; /* length of pulse in sec */
  codeint phase; /* real-time variable for phase */
  double RG1; /* gate delay before pulse in sec */
  double RG2; /* gate delay after pulse in sec */

**Description:** Pulses the observe transmitter with amplifier gating. The amplifier is gated on prior to the start of the pulse by RG1 sec and gated off RG2 sec after the end of the pulse. The total length of this event is therefore not simply width, but width+RG1+RG2.
The amplifier gating times RG1 and RG2 may be specified explicitly. The parameters rof1 and rof2 are often used for these times. These parameters are normally “hidden” parameters, not displayed on the screen and entered by the user. Their values can be interrogated by entering the name of the parameter followed by a question mark (e.g., rof1?).

Arguments: width specifies the duration, in seconds, of the observe transmitter pulse.
phase sets the observe transmitter phase and must be a real-time variable.
RG1 is the time, in seconds, the amplifier is gated on prior to the start of the pulse (typically 10 \( \mu \)s for \(^1\)H/\(^19\)F, 40 \( \mu \)s for other nuclei, and 2 \( \mu \)s for the MERCURYplus/-Vx).
RG2 is the time, in seconds, before the amplifier is gated off after the end of the pulse (typically 10 \( \mu \)s on the MERCURYplus/-Vx, and about 5 \( \mu \)s on other systems).

Examples:
```
rgpulse(pw,v1,rof1,rof2);
rgpulse(2.0*pw,v2,1.0e–6,0.2e–6);
```

Related: 
- obspulse: Pulse observe transmitter with amplifier gating
- pulse: Pulse observe transmitter with amplifier gating
- simpulse: Pulse observe, decoupler channels simultaneously
- sim3pulse: Simultaneous pulse on 2 or 3 rf channels

**rgradient**  
Set gradient to specified level

Applicability: Systems with imaging or PFG modules.

Syntax:
```
rggradient(channel,value)
```

char channel; /* gradient 'x', 'y', or 'z' */
double value; /* amplitude of gradient amplifier */

Description: Sets the gradient current amplifier to specified value. In imaging, rgradient sets a gradient to a specified level in DAC units.

Arguments:
- channel specifies the gradient to set. It uses one of the characters 'X', 'x', 'Y', 'y', 'Z' or 'z'. In imaging, channel can be 'gread', 'gphase', or 'gslice'.
- value specifies the gradient level by a real number (a DAC setting in imaging) 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.

Examples:
```
rggradient('z',1327.0);
```

Related: 
- dps_show: Draw delay or pulses in a sequence for graphical display
- getorientation: Read image plane orientation
- shapedgradient: Generate shaped gradient
- vgradient: Set gradient to a level determined by real-time math
- zgradpulse: Create a gradient pulse on the z channel

**rlpower**  
Change power level

Applicability: Systems with linear amplifiers. This statement is due to be eliminated in future versions of VnmrJ software. Although it is still functional, you should not write pulse sequences using it and should replace it in existing sequences with Use obspower, decpower, dec2power, or dec3power, as appropriate.

Syntax:
```
rlpower(power,device)
```
double power; /* new level for coarse power */
int device; /* OBSch, DECch, DEC2ch, or DEC3ch */

Description: Changes transmitter or decoupler power the same as the power statement but avoids consuming a real-time variable for the value. On systems with the Output board (and only on these systems), by default, rlppower statements are preceded internally by a 0.2-μs delay (see the apovrride statement for more details).

Arguments: power sets the power level by assuming values of 0 (minimum power) to 63 (maximum power) on channels with a 63-dB attenuator or –16 (minimum power) to 63 (maximum power) on channels with a 79-dB attenuator.
device is OBSch (observe transmitter) or DECch (first decoupler). device can also be DEC2ch (second decoupler) or DEC3ch (third decoupler).

CAUTION: On systems with linear amplifiers, be careful when using values of rlpower 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. Use config to set a safety maximum for the tpwr, dpwr, dpwr2, and dpwr3 parameters.

Examples: (1) pulsesquence()
{
    double satpwr;
satpwr=getval("satpwr");
    ...
    rlpower(satpwr,OBSch);
    ...
}
(2) rlpower(63.0,OBSch);

Related: decpower Change first decoupler power
dec2power Change second decoupler power
dec3power Change third decoupler power
obspower Change observe transmitter power
power Change transmitter or decoupler power, linear amp. sys.
rlpwrf Set transmitter or decoupler fine power

rlpwrf Set transmitter or decoupler fine power (obsole}e

Description: Use obspwrf, decpwrf, dec2pwrf, or dec3pwrf, as appropriate.

Description: Do not write any new pulse sequences using this statement and should replace it in existing sequences. Changes transmitter or decoupler fine power the same as the pwr statement, except rlpwrf uses a real-number variable for the power level desired instead of consuming a real-time variable for the level.

Related: decpwrf Set first decoupler fine power
dec2pwrf Set second decoupler fine power
dec3pwrf Set third decoupler fine power
ipwrf Change transmitter or decoupler fine power with IPA
obspwrf Set observe transmitter fine power
power Change transmitter or decoupler power, lin. amp. sys.
pwrf Change transmitter or decoupler fine power

rlpwrn Set transmitter or decoupler linear modulator power

Applicability: UNITY/NOVA systems.
Syntax: \texttt{rlpwrms} (power, device)

\texttt{double power; /* new level for lin. mod. power */}
\texttt{int device; /* OBSch, DECch, DEC2ch, or DEC3ch */}

Description: Changes transmitter or decoupler linear modulator power the same as the \texttt{pwrms} statement, but to avoid using real-time variables, \texttt{rlpwrms} uses a C variable of type double as the argument for the amount of change.

Arguments: \texttt{power} is the linear modulation (fine) power desired.
\texttt{device} is \texttt{OBSch} (observe transmitter), \texttt{DECch} (first decoupler), \texttt{DEC2ch} (second decoupler), or \texttt{DEC3ch} (third decoupler).

Examples: \texttt{rlpwrms} (4.0, OBSch);
Related: \texttt{ipwrms} Change transmitter or decoupler lin. mod. power with IPA
\texttt{pwrms} Change transmitter or decoupler linear modulator power

\textbf{rotate} \hspace{1cm} \textbf{Sets the standard oblique rotation angles}

Description: Sets the standard oblique rotation angles psi, phi, and theta for gradient rotation.

Syntax: \texttt{rotate()}\par
Examples: \texttt{rotate( )};

\textbf{rot_angle} \hspace{1cm} \textbf{Sets user defined oblique rotation angles}

Description: Sets user defined oblique rotation Euler angles \texttt{ang1}, \texttt{ang2}, and \texttt{ang3} for gradient rotation.

Syntax: \texttt{rot_angle(ang1, ang2, ang3)}

Arguments: \texttt{ang1}, \texttt{ang2}, and \texttt{ang3} are the user defined oblique rotation Euler angles in degrees.

Examples: \texttt{rot_angle(ang1, ang2, ang3)};

\textbf{rotorperiod} \hspace{1cm} \textbf{Obtain rotor period of MAS rotor}

Applicability: Systems with MAS (magic-angle spinning) rotor synchronization hardware.

Description: Obtains the rotor period.

Syntax: \texttt{rotorperiod(period)}\par
\texttt{codeint period; /* variable to hold rotor period */}

Arguments: \texttt{period} is a real-time variable into which is placed the rotor period as an integer in units of 100 ns. For example, for \texttt{rotorperiod(v4)}, if \texttt{v4} contains the value 1700, the rotor period is 170 μs and the rotor speed is \texttt{1E+7 / 1700 = 5882 Hz}.

Examples: \texttt{rotorperiod(v4)};

Related: \texttt{rotorsync} Gated pulse sequence delay from MAS rotor position
\texttt{xgate} Gate pulse sequence from an external event

\textbf{rotorsync} \hspace{1cm} \textbf{Gated pulse sequence delay from MAS rotor position}

Applicability: Systems with MAS (magic-angle spinning) rotor synchronization hardware.

Syntax: \texttt{rotorsync(rotations)}\par
\texttt{codeint rotations; /* variable for turns to wait */}
Description: Inserts a variable-length delay that allows synchronizing the execution of the pulse sequence with a particular orientation of the sample rotor. When the rotorsync statement is encountered, the pulse sequence is stopped until the number of rotor rotations has occurred.

Arguments: rotations is a real-time variable that specifies the number of rotor rotations to occur before restarting the pulse sequence.

Examples: rotorsync(v6);

Related: rotorperiod Obtain rotor period of MAS rotor
         xgate Gate pulse sequence from an external event

S

Top  A  B  C  D  E  G  H  I  L  M  O  P  R  S  T  V  W  X  Z

setautoincrement Set autoincrement attribute for a table
setdivnfactor Set divn-return attribute and divn-factor for AP table
setreceiver Associate the receiver phase cycle with a table
setstatus Set status of observe transmitter or decoupler transmitter
settable Store an array of integers in a real-time AP table
setuserap Set user AP register
shapedpulse Perform shaped pulse on observe transmitter
shaped_pulse Perform shaped pulse on observe transmitter
shapedgradient Generate shaped gradient pulse
shaped2Dgradient Generate arrayed shaped gradient pulse
shapedincgradient Generate dynamic variable gradient pulse
shapedvgradient Generate dynamic variable shaped gradient pulse
simpulse Pulse observe and decouple channels simultaneously
sim3pulse Pulse simultaneously on 2 or 3 rf channels
sim4pulse Simultaneous pulse on four channels
simshaped_pulse Perform simultaneous two-pulse shaped pulse
sim3shaped_pulse Perform a simultaneous three-pulse shaped pulse
sli Set SLI lines
sp#off Turn off specified spare line (Inova #=1 to 5)
sp#on Turn on specified spare line (Inova #=1 to 5)
spinlock Control spin lock on observe transmitter
starthardloop Start hardware loop
status Change status of decoupler and homospoil
statusdelay Execute the status statement with a given delay time
stepsize Set small-angle phase step size
sub Subtract integer values
setautoincrement

Set autoincrement attribute for a table

Syntax: setautoincrement (table)

codeint table;  /* real-time table variable */

description: Sets the autoincrement attribute in a table. The index into the table is set to 0 at
the start of an FID acquisition and is incremented after each access into the
table. Tables using the autoincrement feature cannot be accessed within a
hardware loop.

Arguments: table is the name of the table (t1 to t60).

Examples: setautoincrement (t9);

setdivnfactor

Set divn-return attribute and divn-factor for table

Syntax: setdivnfactor (table, divn_factor)

codeint table;  /* real-time table variable */

int divn_factor;  /* number to compress by */

description: Sets the divn-return attribute and divn-factor for a table. The actual index into
the table is now set to (index/divn-factor). {0 1}2 is therefore translated by the
controller, not by PSG (pulse sequence generation), into 0 0 1 1. The divn-return
attribute results in a divn-factor-fold compression of the table.

Arguments: table specifies the name of the table (t1 to t60).

divn_factor specifies the divn-factor for the table.

Examples: setdivnfactor (t7, 4);

setreceiver

Associate the receiver phase cycle with a table

Syntax: setreceiver (table)

codeint table;  /* real-time table variable */

description: Assigns the ctth element of a table 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 math statements such as assign, add, etc., 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);
setstatus  Set status of observe transmitter or decoupler transmitter

Applicability: UNITY/INOVA systems.

Syntax: setstatus(channel, on, mode, sync, mod_freq)

int channel; /* OBSch, DECch, DEC2ch, or DEC3ch */
int on; /* TRUE (=on) or FALSE (=off) */
char mode; /* 'c', 'w', 'g', etc. */
int sync; /* TRUE (=synchronous) or FALSE */
double mod_freq; /* modulation frequency */

Description: Sets the status of a transmitter independent of the status statement, thus overriding decoupler parameters such as dm and dmm. Since the setstatus statement is part of the pulse sequence, it has no effect when only an su command is executed. It is the only way the observe transmitter can be modulated. See also: “Amplifier Channel Blanking and Unblanking,” page 75

Arguments: channel is OBSch (observe transmitter), DECch (first decoupler), DEC2ch (second decoupler), or DEC3ch (third decoupler).

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 pulse modulation (i.e., waveform).
- 'r' sets square wave modulation.
- 'u' (UNITY/INOVA only) sets user-supplied modulation from external hardware.
- '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: status Change status of decoupler and homospoil

settable  Store an array of integers in a real-time table

Syntax: settable(tablename, numelements, intarray)

codeint tablename; /* real-time table variable */
Chapter 3. Pulse Sequence Statement Reference

```c
int numelements; /* number in array */
int *intarray; /* pointer to array of elements */
```

**Description:** Stores an integer array in a real-time table. The autoincrement or divn-return attributes can be subsequently associated with a table defined by `settable` by using `setautoincrement` and `setdivnfactor`.

**Arguments:**
- `table` is the name of the table (t1 to t60).
- `numelements` is the size of the table.
- `intarray` is a C array that contains the table elements, which can range from -32768 to 32767. Before calling `settable`, this array must be predefined and predimensioned in the pulse sequence using C statements.

**Examples:**
```
settable(t1,10,int_array);
```

**Related:**
- `getelem` Retrieve an element from a table
- `loadtable` Load AP table elements from table text file
- `setautoincrement` Set autoincrement attribute for a table
- `setdivnfactor` Set divn-return attribute and divn-factor for table
- `setreceiver` Associate the receiver phase cycle with a table

---

### setuserap

**Set user AP register**

**Applicability:** `UNITY/INOVA` systems

**Syntax:**
```
setuserap(value, register)
```

- `real value;` /* value sent to user AP register */
- `int register;` /* AP bus register number: 0, 1, 2, or 3 */

**Description:** Sets a value in one of the four 8-bit AP bus registers that provide an output interface to user devices. The outputs of these registers go to the USER AP connectors J8212 and J8213, located on the back of the left console cabinet. These outputs have a 100-ohm series resistor for circuit protection.

**Arguments:**
- `value` is a signed or unsigned number (real or integer) to output to the specified user AP register. The number is truncated to an 8-bit byte.
- `register` is the AP register number, mapped to output lines as follows:
  - Register 0 is J8213, lines 9 to 16.
  - Register 1 is J8213, lines 1 to 8.
  - Register 2 is J8212, lines 9 to 16.
  - Register 3 is J8212, lines 1 to 8.

**Examples:**
```
setuserap(127.0,0);
```

**Related:**
- `readuserap` Read input from user AP register
- `vsetuserap` Set user AP register using real-time variable

---

### shapedpulse

**Perform shaped pulse on observe transmitter**

**Applicability:** This statement is due to be eliminated in future versions of VnmrJ software. Although it is still functional, you should not write any new pulse sequences using it and should replace it in existing sequences with `shaped_pulse`, which functions exactly the same as `shapedpulse`.

---

### shaped_pulse

**Perform shaped pulse on observe transmitter**

**Applicability:** `UNITY/INOVA` systems with a waveform generator on the observe transmitter channel.
Syntax:  
\texttt{shaped\_pulse(pattern, width, phase, RG1, RG2)}

\begin{verbatim}
char *pattern;  /* name of .RF text file */
double width;  /* width of pulse in sec */
codeint phase; /* real-time variable for phase */
double RG1;   /* gating delay before pulse in sec */
double RG2;   /* gating delay after pulse in sec */
\end{verbatim}

Description: Performs a shaped pulse on the observe transmitter.

When using the waveform, the shapes are downloaded into the controller before the start of an experiment. The minimum pulse length is and stepsize is 50 ns. If the length is less than 50 ns, the pulse is not executed.

\textit{UNITY/INOVA Systems:}

These systems use the waveform generator if it is configured on the channel. Systems without a waveform generator on the channel use the linear attenuator and the small-angle phase shifter are used to effectively perform an \texttt{apshaped\_pulse} statement.\textit{UNITY/INOVA Systems} address and start the shape when \texttt{shaped\_pulse} is called. The overhead at the start and end of the shaped pulse varies with the system:

- \textit{UNITY/INOVA}: 1 \(\mu\text{s}\) (start), 0 (end)
- System with Acquisition Controller board: 10.75 \(\mu\text{s}\) (start), 4.3 \(\mu\text{s}\) (end)
- System with Output board: 10.95 \(\mu\text{s}\) (start), 4.5 \(\mu\text{s}\) (end)

\textit{UNITY/INOVA Systems}, using the linear attenuator and the small-angle phase shifter to generate a shaped pulse, create AP tables for amplitude and phase on the fly when the \texttt{shaped\_pulse} statement is called. It also uses the real-time variables \texttt{v12} and \texttt{v13} to control the execution of the shape. It does not use AP table variables. For timing and more information, see the description of \texttt{apshaped\_pulse}. Note that if using AP tables with shapes that have a large number of points, the FIFO can become overloaded with words generating the pulse shape and FIFO Underflow errors can result.

Arguments: \texttt{file} is the name of a text file in the \texttt{shapelib} directory that stores the rf pattern (leave off the .RF file extension). \texttt{width} is the duration, in seconds, of the pulse on the observe transmitter \texttt{phase} is the phase of the pulse and must be a real-time variable. \texttt{RG1} is the delay, in seconds, between gating the amplifier on and gating the observe transmitter on (the phase shift occurs at the beginning of this delay). \texttt{RG2} is the delay, in seconds, between gating the observe transmitter off and gating the amplifier off.

Examples: \texttt{shaped\_pulse("gauss",pw,v1,rof1,rof2)};

Related:  
\begin{verbatim}
\texttt{decshaped\_pulse}  \texttt{dec2shaped\_pulse}  \texttt{simshaped\_pulse}  \texttt{sim3shaped\_pulse}
\end{verbatim}

\texttt{shapedgradient} \hspace{1cm} Generate shaped gradient pulse

Applicability: Systems with waveform generation on imaging or PFG module.

Syntax:  
\begin{verbatim}
shapedgradient(pattern, width, amp, channel, loops, wait)
\end{verbatim}

\begin{verbatim}
char *pattern;  /* name of shape text file */
\end{verbatim}
double width; /* length of pulse */
double amp; /* amplitude of pulse */
char channel; /* gradient channel 'x', 'y', or 'z' */
int loops; /* number of loops */
int wait; /* WAIT or NOWAIT */

Description: Operates the selected gradient channel to provide a gradient pulse to the
selected set of gradient coils. The pulse has a pulse shape determined by the
arguments name, width, amp, and loops. Unlike the shaped rf pulses, the
shaped gradient leaves the gradients at the last value in the gradient pattern
when the pulse completes.

Arguments: pattern is the name of a text file without a.GRD extension to describe
the shape of the pulse. The text file with a.GRD extension should be located in
$vnmrsystem/shapelib or in the users directory $vnmruser/shapelib.

width is the requested length of the pulse in seconds. The pulse length is
affected by two factors: (1) the minimum time of every element in the shape
file must be at least 10 μs long, and (2) the time for every element must be a
multiple of 50 ns. If the width of the pulse is less than 10 μs times the number
of steps in the shape, a warning message is generated. The shaped gradient
software rounds each element to a multiple of 50 ns. If the requested width
differs from the actual width by more than 2%, a warning message is displayed.

amp is a value that scales the amplitude of the pulse. Only the integer portion
of the value is used and it ranges from 32767 to –32767; where 32767 is full
scale and –32767 is negative full scale.

channel selects the gradient coil channel desired and should evaluate to the
characters 'x', 'y', or 'z'. (Be sure not to confuse the characters 'x', 'y',
or 'z' with the strings "x", "y", or "z".)

loops is a value, from 1 to 255, that allows the user to loop the selected
waveform. Note that the given value is the number of loops to be executed and
that the values 0 and 1 cause the pattern to execute once.

wait is a keyword, either WAIT or NOWAIT, that selects whether or not a delay
is inserted to wait until the gradient is completed before executing the next
statement. The total time it will wait is width*loops. If loops is supplied as
0, it will be counted as 1 when determining its total time.

Examples: The line:
#define POVR 1.2e-5 /* shaped pulse overhead=12 us */
is required for UNITY/INOVA systems.

UNITY/INOVA:

shapedgradient("hsine",0.02,32767,'y',1,NOWAIT);
#include "standard.h"
#define POVR 1.2e-5 /* shaped pulse overhead=12 us */
pulsesequence()
{
...
for (i=-32000; i<=32000; i+16000)
{
  shapedgradient("hsine",pw+d3+rx1+rx2,i,'x', \
                1,NOWAIT);
  shapedpulse("sinc",pw,oph,rx1,rx2);
  delay(d3);
} /* This step sets a square gradient from a low value */ /* to a high value while executing a shaped pulse */ /* and a delay during each gradient value. */ ...

Related:  
dps_show       Draw delay or pulses in a sequence for graphical display 
grgradient     Set gradient to a specified level 
shapedgradient Provide shaped gradient pulse to gradient channel 
shaped2Dgradient Arrayed shaped gradient function 
vgradient     Set gradient to a level determined by real-time math

**shaped2Dgradient**  Generate arrayed shaped gradient pulse

Applicability: Systems with imaging or PFG module.

Syntax:  
shaped2Dgradient(pattern, width, amp, channel, 
    loops, wait, tag)
char *pattern; /* name of pulse shape text file */
double width;  /* length of pulse */
double amp;    /* amplitude of pulse */
char channel;  /* gradient channel 'x', 'y', or 'z' */
int loops;    /* number of loops */
int wait;     /* WAIT or NOWAIT */
int tag;      /* unique number for gradient element */

Description: Operates the selected gradient channel to provide a gradient pulse to the selected set of gradient coils. This statement is basically the same as the `shapedgradient` statement except that `shaped2Dgradient` is tailored to be used in pulse sequences where the amplitude is arrayed (imaging sequences).

Arguments:  
`pattern` is the name of a text file without a.GRD extension that describes the shape of the pulse. The text file with a.GRD extension should be located in `$vnmrsystem/shapelib` or in the users directory `$vnmruser/shapelib`.

`width` is the requested length of the pulse in seconds. The width of the pulse is affected by: (1) (UNITY/INNOVA only) the minimum time of every element in the shape file must be at least 200 ns long, and (2) (UNITY/INNOVA) the time for every element must be a multiple of 50 ns. If the `width` of the pulse is less than 10 μs times the number of steps in the shape, a warning message is generated. The shaped gradient software will round each element to a multiple of 50 ns. If the requested width differs from the actual width by more than 2%, a warning message is displayed.

`amp` is a value that scales the amplitude of the pulse. Only the integer portion of the value is used and it ranges from 32767 to –32767; where 32767 is full scale and –32767 is negative full scale.

`channel` selects the gradient coil channel desired and should evaluate to the characters 'x', 'y', or 'z'. (Be sure not to confuse the characters 'x', 'y', or 'z' with the strings "x", "y", or "z".)

`loops` is a value, from 1 to 255, that allows the user to loop the selected waveform. Note that the given value is the number of loops to be executed and that the values 0 and 1 cause the pattern to execute once.
UNITY INOVA only: A digital hardware bug affecting looping requires that all patterns be carefully constructed to achieve the desired results.

wait is a keyword, either WAIT or NOWAIT, that selects whether or not a delay is inserted to wait until the gradient is completed before executing the next element. The total time it will wait is width*loops.

tag is a unique integer that “tags” the gradient element from any other gradient elements used in the sequence.

Examples:  #include "standard.h"
pulsecycle()
{
  ...
  shaped2Dgradient("hsine",d3,0.0-gpe,'x',0,NOWAIT,1);
  delay(d3);
  shaped2Dgradient("hsine",d4,gpe,'y',0,NOWAIT,2);
  ...
}

Related:  dps_show Draw delay or pulses in a sequence for graphical display
  rgradient Set gradient to a specified level
  shapedgradient Provide shaped gradient pulse to gradient channel
  vgradient Set gradient to a level determined by real-time math

shapedincgradient  Generate dynamic variable gradient pulse

Applicability: Systems with imaging or PFG module.

Syntax:  shapedincgradient(channel,pattern,width, \\
         a0,a1,a2,a3,x1,x2,x3,loops,wait)
  char channel; /* gradient channel 'x', 'y', or 'z' */
  char *pattern; /* name of pulse shape text file */
  double width; /* length of pulse */
  double a0,a1,a2,a3; /* coefficients to determine level */
  codeint x1,x2,x3; /* variables to determine level */
  int loops; /* number of loops */
  int wait; /* WAIT or NOWAIT */

Description: Provides a dynamic, variable shaped gradient pulse controlled using the real-time math functions. The statement drives the chosen gradient with the specified pattern, scaled to the level defined by the formula:

level = a0 + a1*x1 + a2*x2 + a3*x3

The pulse has a pulse shape determined by the pattern, width, and loops arguments, as well as the calculation of level.

Unlike the shaped rf pulses, the shapedincgradient will leave the gradients at the last value in the gradient pattern when the pulse completes. The range of the gradient level is –32767 to +32767. If the requested level lies outside the legal range, it is clipped at the appropriate boundary value. Note that, while each variable in the calculation of level must fit in a 16-bit integer, intermediate sums and products in the calculation are done with double precision, 32-bit integers.

The following error messages are displayed if the requested shape cannot be found or if a width of zero is specified:

shapedincgradient: x[i] illegal RT variable: xi or
shapedincgradient: no match!
Arguments: channel selects the gradient coil channel desired and should evaluate to the characters 'x', 'y', or 'z'. (Be careful not to confuse the characters 'x', 'y', or 'z' with the strings "x", "y", or "z").

pattern is the name of a text file without a.GRD extension to describe the shape of the pulse. The text file with a.GRD extension should be located in $vnmrsystem/shapelib or in the users directory $vnmruser/shapelib.

width is the requested length of the pulse in seconds. The width of the pulse is affected by two factors: (1) the minimum time of every element in the shape file must be at least 10 μs, and (2) the time for every element must be a multiple of 50 ns. If the width of the pulse is less than 10 μs times the number of steps in the shape), a warning message is generated. The shapedincgradient software will round each element to a multiple of 50 ns. If the requested width differs from the actual width by more than 2%, a warning message is displayed.

a0, a1, a2, a3, x1, x2, x3 are values used in the calculation of “level.”

loops is a value, from 1 to 255, that allows the user to loop the selected waveform. Note that the given value is the number of loops to be executed and that the values 0 and 1 cause the pattern to execute once.

UNITY/INOVA only: A digital hardware bug affecting looping requires that all patterns be carefully constructed to achieve the desired results.

wait is a keyword, either WAIT or NOWAIT, that selects whether or not a delay is inserted to wait until the gradient is completed before executing the next element. The total time it will wait is width*loops. If loops is supplied as 0, it will be counted as 1 when determining its total time.

Related: getorientation Read image plane orientation
rgradient Set gradient to a specified level
shapedgradient Provide shaped gradient pulse to gradient channel
shaped2Dgradient Generate arrayed shaped gradient pulse
vgradient Set gradient to a level determined by real-time math

shapedvgradient Generate dynamic variable shaped gradient pulse

Applicability: Systems with imaging or PFG module.

Syntax: shapedvgradient(pattern,width,amp_const, amp_incr,amp_vmult,channel,vloops,wait,tag)

char *pattern; /* name of pulse shape text file */
double width; /* length of pulse */
double amp_const; /* sets amplitude of pulse */
double amp_incr; /* sets amplitude of pulse */
codeint amp_vmult; /* sets amplitude of pulse */
char channel; /* gradient channel 'x', 'y', or 'z' */
codeint vloops; /* variable for number of loops */
int wait; /* WAIT or NOWAIT */
int tag; /* unique number for gradient element */

Description: Operates the selected gradient channel to provide a shaped gradient pulse to the selected set of gradient coils. This statement is tailored to provide a dynamic variable shaped gradient level controlled using the system real-time math functions and real-time looping. The statement drives the chosen gradient shape to the level defined by the formula:

amplitude = amp_const + amp_incr*amp_vmult
Chapter 3. Pulse Sequence Statement Reference

The range of the gradient amplitude is $-32767$ to $+32767$, where 32767 is full scale and $-32767$ is negative full scale.

If the requested level lies outside this range, it is truncated to the appropriate boundary value. Note that the \texttt{vloops} argument is also controlled by a real-time AP math variable. Unlike the shaped rf pulses, the shaped gradient leaves the gradients at the last value in the gradient pattern when the pulse completes.

Arguments:

\texttt{name} is the name of a text file without a .GRD extension to describe the shape of the pulse. The text file with a .GRD extension should be located in \\
$\texttt{vnmrsystem/shapelib}$ or in the user’s directory $\texttt{vnmruser/shapelib}$.

\texttt{width} is the requested length of the pulse in seconds. The width of the pulse is affected by two factors: (1) the minimum time of every element in the shape file must be at least 10 $\mu$s, and (2) the time for every element must be a multiple of 50 ns. If \texttt{width} is less than 10 $\mu$s times the number of steps in the shape, a warning message is generated. The shaped gradient software will round each element to a multiple of 50 ns. If the requested width differs from the actual width by more than 2\%, a warning message is displayed.

\texttt{amp\_const}, \texttt{amp\_incr}, and \texttt{amp\_vmult} scale the amplitude of the pulse according to the formula above. \texttt{amp\_const} and \texttt{amp\_incr} can be values of type double or integer. \texttt{amp\_vmult} must be a real-time AP math variable (\texttt{v1 to v14}) or a table pointer (\texttt{t1 to t60}). The amplitude ranges are also given above.

\texttt{channel} selects the gradient coil channel desired and should evaluate to the characters 'x', 'y', or 'z'. (Be careful not to confuse the characters 'x', 'y', or 'z' with the strings "x", "y", or "z".)

\texttt{vloops} allows the user to loop the selected waveform. Values range from 1 to 255. This also must be a real-time math variable (\texttt{v1 to v14}) or a table pointer (\texttt{t1 to t60}). Do not use 0 for \texttt{vloops}, because this may cause inconsistencies when \texttt{WAIT} is selected for the \texttt{wait\_4\_me} argument.

\texttt{UNITY} \texttt{INOV\texttt{A}} only: A digital hardware bug affecting looping requires that all patterns be carefully constructed to achieve the desired results.

\texttt{wait} is a keyword, either \texttt{WAIT} or \texttt{NOWAIT}, that selects whether or not a delay is inserted to wait until the gradient is completed before executing the next element. The total time it will wait is \texttt{width*vloops}. It uses the \texttt{incdelay} statement when waiting for the gradient pulse to complete.

\texttt{tag} is a unique integer that “tags” this gradient statement from any other gradient statement used in the sequence.

Examples:

```c
#include "standard.h"

pulsesequence()
{
  ...
  char gphase, gread, gslice;
  ...
  amplitude=(int)(0.5*ni*gpe);
  stat=getorientation(&gread,&gphase,&gslice,"orient");
  ...
  initval(1.0,v1);
  initval(nf,v9);
  loop(v9,v5);
  ...
}```
shapedvgradient("hsine",d3,amplitude,igpe, v5,gphase,v1,NOWAIT,1);

endloop(v5);

Related: incdelay Set real-time incremental delay
gradient Set gradient to specified level
shapedgradient Generate shaped gradient pulse
shaped2Dgradient Generate arrayed shaped gradient pulse
evgradient Generate dynamic variable gradient pulse

simpulse  Pulse observe and decouple channels simultaneously

Syntax: simpulse(obswidth,decwidth,obsphase,decphase, RG1,RG2)

double obswidth, decwidth; /* pulse lengths in sec */
codeint obsphase,decphase; /* variables for phase */
double RG1; /* gating delay before pulse */
double RG2; /* gating delay after pulse */

Description: Gates the observe and decoupler channels. The shorter of the two pulses is centered on the longer pulse, while the amplifier gating occurs before the start of the longer pulse (even if it is the decoupler pulse) and after the end of the longer pulse.

The absolute difference in the two pulse widths must be greater than or equal to 0.1 μs; otherwise, a timed event of less than the minimum value (0.05 μs) would be produced:

- if the difference is less than 0.1 μs, the pulses are made equally long.
- If the difference is from 0.1 to 0.2 μs, the difference is made 0.2 μs.
- If the difference is larger than 0.2 μs, the difference is made as close as the timing resolution allows (0.0125 μs).

Excluding UNITY/NOVA systems: the minimum time is 0.2 μs; thus, the times are doubled (the difference must be 0.4 μs, resolution is 0.025 μs).

Arguments: obswidth and decwidth are the duration, in sec, of the pulse on the observe transmitter and first decoupler, respectively.

obsphase and decphase are the phase of the pulse on the observe transmitter and the first decoupler, respectively. Each must be a real-time variable.

RG1 is the delay, in seconds, between gating the amplifier on and gating the first rf transmitter on (all phase shifts occur at the beginning of this delay).

RG2 is the delay, in seconds, between gating the final rf transmitter off and gating the amplifier off.

Examples: simpulse(pw,pp,v1,v2,0.0,rof2);

Related: decpulse Pulse the decoupler transmitter
decrgpulse Pulse decoupler transmitter with amplifier gating
dps_show Draw delay or pulses in a sequence for graphical display
rgpulse Pulse observe transmitter with amplifier gating
**sim3pulse**  Simultaneous pulse on 2 or 3 rf channels

**sim4pulse**  Simultaneous pulse on four channels

---

### sim3pulse

**Pulse simultaneously on 2 or 3 rf channels**

**Applicability:** UNITY/INOVA systems with two or more independent rf channels.

**Syntax:**

```c
sim3pulse(pw1,pw2,pw3,phase1,phase2,phase3,RG1,RG2)
double pw1,pw2,pw3;  /* pulse lengths in sec */
codeint phase1,phase2,phase3;  /* variables for phases */
double RG1;  /* gating delay before pulse */
double RG2;  /* gating delay after pulse */
```

**Description:** Performs a simultaneous, three-pulse pulse on three independent rf channels. A simultaneous, two-pulse pulse on the observe transmitter and second decoupler can also be performed by setting the pulse length for the first decoupler to 0.0 (see the second example for how this is done).

Timing limitations connected with the difference in pulse widths are covered in the description of `simpulse`.

**Arguments:**

- `pw1`, `pw2`, and `pw3` are the pulse length, in seconds, of channels OBSch, DECch, and DEC2ch, respectively.
- `phase1`, `phase2`, and `phase3` are the phases of the corresponding pulses. These must be real-time variables (v1 to v14, oph, etc.).
- `RG1` is the delay, in seconds, between gating the amplifier on and gating the first rf transmitter on (all phase shifts occur at the beginning of this delay).
- `RG2` is the delay, in seconds, between gating the final rf transmitter off and gating the amplifier off.

**Examples:**

```c
sim3pulse(pw,p1,p2,oph,v10,v1,rof1,rof2);
sim3pulse(pw,0.0,p2,oph,v10,v1,rof1,rof2);
```

**Related:**
- **decpulse**  Pulse the decoupler transmitter
- **decrGPLuse**  Pulse decoupler transmitter with amplifier gating
- **dps_show**  Draw delay or pulses in a sequence for graphical display
- **rgpulse**  Pulse observe transmitter with amplifier gating
- **simpulse**  Pulse observe, decoupler channels simultaneously
- **sim4pulse**  Simultaneous pulse on four channels

---

### sim4pulse

**Simultaneous pulse on four channels**

**Applicability:** UNITY/INOVA systems with two or more independent rf channels.

**Syntax:**

```c
sim4pulse(pw1,pw2,pw3,pw4,phase1,phase2, \\
    phase3,phase4,RG1,RG2)
double pw1,pw2,pw3,pw4;  /* pulse length in sec */
codeint phase1,phase2;  /* variables for phase */
codeint phase3,phase4;  /* variables for phase */
double RG1;  /* gating delay before pulse */
double RG2;  /* gating delay after pulse */
```

**Description:** Allows for simultaneous pulses on up to four different channels. If any of the pulses are set to 0.0, no pulse is executed on that channel.

Timing limitations connected with the difference in pulse widths are covered in the description of `simpulse`.

**Arguments:**

- `pw1`, `pw2`, `pw3`, and `pw4` are the pulse length, in seconds, of channels OBSch, DECch, DEC2ch, and DEC3ch, respectively.
phase1, phase2, phase3, and phase4 are the phases of the corresponding pulses. Each must be real-time variable (v1-v14, oph, etc.)

RG1 is the delay, in seconds, between gating on the amplifier and turning on the first transmitter (all phases set at beginning of RG1, even if pw is 0.0).

RG2 is the delay, in seconds, between the final transmitter off and gating the amplifier off.

Example:

```c
sim4pulse(pw, 2*pw, pl, 2*pl, oph, v3, ZERO, TWO, rof1, rof1);
sim4pulse(pw, 0.0, 0.0, 2*pl, oph, ZERO, TWO, rof1, rof1);
```

Related:
- **rgpulse**  
  Pulse observe channel with amplifier gating
- **simpulse**  
  Pulse observe and decoupler channel simultaneously
- **sim3pulse**  
  Pulse simultaneously on 2 or 3 channels

### simshaped_pulse

**Perform simultaneous two-pulse shaped pulse**

**Applicability:** UNITY INOVA Systems with a waveform generator on two or more rf channels.

**Syntax:**
```
simshaped_pulse(obsshape, decshape, obswidth, decwidth, obsphase, decphase, RG1, RG2)
```  
```c
char *obsshape, *decshape; /* names of .RF shape files */
double obswidth, decwidth; /* pulse lengths in sec */
codeint obsphase, decphase; /* variables for phase */
double RG1; /* gating delay before pulse */
double RG2; /* gating delay after pulse */
```

**Description:** Performs a simultaneous, two-pulse shaped pulse on the observe transmitter and the first decoupler under waveform control.

If either obswidth or decwidth is 0.0, no pulse occurs on the corresponding channel. If both obswidth and decwidth are non-zero and either obsshape or decshape is set to the null string (""'), then a rectangular pulse occurs on the channel with the null shape name. If either the pulse width is zero or the shape name is the null string, then a waveform is not required on that channel.

**UNITY INOVA:**

The overhead at the start and end of the two-pulse shaped pulse varies with the system:

- **UNITY INOVA:** 1.45 μs (start), 0 (end).
- Systems with an Acquisition Controller board: 21.5 μs, 8.6 μs.
- Systems with an Output board: 21.7 μs, 8.8 μs.

These values hold regardless of the values for the arguments obswidth and decwidth.

**Arguments:**
- obsshape is the name of the text file in the shapelib directory that contains the rf pattern to be executed on the observe transmitter.
- decshape is the name of the text file in the shapelib directory that contains the rf pattern to be executed on the first decoupler.
- obswidth is the length of the pulse, in seconds, on the observe transmitter.
- decwidth is the length of the pulse, in seconds, on the first decoupler.
- obsphase is the phase of the pulse on the observe transmitter. The value must be a real-time variable (v1 to v14, oph, etc.).
decphase is the phase of the pulse on the first decoupler. The value must be a real-time variable (v1 to v14, oph, etc.).

RG1 is the delay, in seconds, between gating the amplifier on and gating the first rf transmitter on (all phase shifts occur at the beginning of this delay).

RG2 is the delay, in seconds, between gating the final rf transmitter off and gating the amplifier off.

Examples:

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

Related:
- decshaped_pulse: Shaped pulse on first decoupler
- dec2shaped_pulse: Shaped pulse on second decoupler
- shaped_pulse: Shaped pulse on observe transmitter
- sim3shaped_pulse: Simultaneous three-pulse shaped pulse

**sim3shaped_pulse**

**Perform a simultaneous three-pulse shaped pulse**

**Applicability:** UNITY/INOVA systems with a waveform generator on three or more rf channels.

**Syntax:**
```
sim3shaped_pulse(obsshape,decshape,dec2shape, \   obswidth,decwidth,dec2width,obsphase, \   decphase,dec2phase,RG1,RG2)
```

- `char *obsshape; /* name of obs .RF file */`
- `char *decshape; /* name of dec .RF file */`
- `char *dec2shape; /* name of dec2 .RF file */`
- `double obswidth; /* obs pulse length in sec */`
- `double decwidth; /* dec pulse length in sec */`
- `double dec2width; /* dec2 pulse length in sec */`
- `codeint obsphase; /* obs real-time var. for phase */`
- `codeint decphase; /* dec real-time var. for phase */`
- `codeint dec2phase; /* dec2 real-time var. for phase */`
- `double RG1; /* gating delay before pulse in sec */`
- `double RG2; /* gating delay after pulse in sec */`

**Description:** Performs a simultaneous, three-pulse shaped pulse under waveform control on three independent rf channels.

**sim3shaped_pulse** can also be used to perform a simultaneous two-pulse shaped pulse on any combination of three rf channels. This can be achieved by setting one of the pulse lengths to the value 0.0 (see the second example for an illustration of how this is done).

If any of the shape names are set to the null string (""), then a rectangular pulse occurs on the channel with the null shape name. If either the pulse width is zero or the shape name is the null string, then a waveform is not required on that channel.

**UNITY/INOVA:**

The overhead at the start and end of the shaped pulse varies:

- **UNITY/INOVA:** 1.95 μs (start), 0 (end).
- Systems with an Acquisition Controller board: 32.25 μs, 12.9 μs.
- Systems with an Output board: 32.45 μs, 13.1 μs.

These values hold regardless of the values of the arguments `obswidth`, `decwidth`, and `dec2width`.

**Arguments:** `obsshape` is the name of the text file in the `shapelib` directory that contains the rf pattern to be executed on the observe transmitter.
decshape is the name of the text file in the shapelib directory that contains the rf pattern to be executed on the first decoupler.

dec2shape is the name of the text file in the shapelib directory that contains the rf pattern to be executed on the second decoupler.

obswidth is the length of the pulse, in seconds, on the observe transmitter.

decwidth is the length of the pulse, in seconds, on the first decoupler.

dec2width is the length of the pulse, in seconds, on the second decoupler.

obsphase is the phase of the pulse on the observe transmitter. The value must be a real-time variable (v1 to v14, oph, etc.).

decphase is the phase of the pulse on the first decoupler. The value must be a real-time variable (v1 to v14, oph, etc.).

dec2phase is the phase of the pulse on the second decoupler. The value must be a real-time variable (v1 to v14, oph, etc.).

RG1 is the delay, in seconds, between gating the amplifier on and gating the first rf transmitter on (all phase shifts occur at the beginning of this delay).

RG2 is the delay, in seconds, between gating the final rf transmitter off and gating the amplifier off.

Examples:

```
sim3shaped_pulse("gauss","hrm180","sinc",pw,p1,p2, \ 
v2,v5,v6,rof1,rof2);
sim3shaped_pulse("dumy","hrm180","sinc",0.0,p1,p2, \ 
v2,v5,v6,rof1,rof2);
```

Related:

decshaped_pulse Shaped pulse on first decoupler

dec2shaped_pulse Shaped pulse on second decoupler

shaped_pulse Shaped pulse on observe transmitter

simshaped_pulse Simultaneous two-pulse shaped pulse

### sli

**Set SLI lines**

**Applicability:** UNITY/INOVA systems.

**Syntax:**
```
sli(address,mode,value)
```

```
int address; /* SLI board address */
int mode;    /* SLI_SET, SLI_OR, SLI_AND, SLI_XOR */
unsigned value; /* bit pattern */
```

**Description:** Sets lines on the SLI board. It has no return value. Systems with imaging capability and the Synchronous Line Interface (SLI) board, an option that provides an interface to custom user equipment. The board contains 32 TTL-compatible logic signals that can be set by these functions. Each line has an LED indicator and a 100-ohm series resistor for circuit protection. The lines are accessible through the 50-pin ribbon connector J4 on the front edge of the SLI board. The pin assignments are as follows:

- Pins 1 and 49 are a +5 V supply through 100-ohm series resistor (enabled by installing jumper J3L)
- Pins 3 to 10 control bits 0 to 7
- Pins 12 to 19 control bits 8 to 15
- Pins 21 to 28 control bits 16 to 23
- Pins 41 to 48 control bits 24 to 31
- Pins 2, 11, 20, 29, 40, and 50 are ground
sli has a pre-execution delay of 10.950 μs but no post-execution delay. The delay is composed of a 200-ns startup delay.

**UNITY/INOVA:** with 5 AP bus cycles (1 AP bus cycle = 2.150 μs).

The logic levels on the SLI lines are not all set simultaneously. The four bytes of the 32 bit word are set consecutively, the low-order byte first. The delay between setting of consecutive bytes is 1 AP bus cycle ±100 ns. (This 100-ns timing jitter is non-cumulative.)

The error message **Illegal mode: n** is caused by the **mode** argument not being one of **SLI_SET**, **SLI_OR**, **SLI_XOR**, or **SLI_AND**.

**Arguments:**

- **address** is the address of the SLI board in the system. It must match the address specified by jumper J7R on the board. Note that the jumpers 19-20 through -2 specify bits 2 through 11, respectively. Bits 0 and 1 are always zero. An installed jumper signifies a “one” bit, and a missing jumper a “zero”. The standard addresses for the SLI in the VME card cage:
  
  - Digital (left) side is C90 (hex) = 3216
  - Analog (right) side is 990 (hex) = 2448

- **mode** determines how to combine the specified value with the current output of the SLI to produce the new output. The four possible modes:
  
  - **SLI_SET** is to load the new value directly into the SLI
  - **SLI_OR** is to logically OR the new value with the old
  - **SLI_AND** is to logically AND the new value with the old
  - **SLI_XOR** is to logically XOR the new value with the old

- **value** (as modified by the **mode** argument) specifies the bit pattern to be set in the SLI board. This should be a non-negative number, between 0 (all lines low) and $2^{32} - 1$ (all lines high).

**Examples:**

```c
pulsetrace()
{
    ...
    int SLIaddr;  /* Address of SLI board */
    unsigned SLIbits;  /* 32 bits of SLI line settings */
    ...
    SLIbits = getval("sli");
    SLIaddr = getval("address");
    ...
    sli(SLIaddr, SLI_SET, SLIbits);
    ...
}
```

Note that **sli** and **address** are not standard parameters, but need to be created by the user if they are mentioned in a user pulse sequence (for details, see the description of the **create** command).

**Related:**

- **sp#on** Turn on specified spare line
- **sp#off** Turn off specified spare line
- **vsli** Set SLI lines from real-time variable

**sp#off**

**Turn off specified spare line (Inova #=1 to 5)**

**Applicability:** **UNITY/INOVA** systems.

**Syntax:** **sp1off()** to **sp3off()**
sp1off

Description: Turns off the specified user-dedicated spare line connector (sp1off for SPARE 1, sp2off for SPARE 2, etc.) for high-speed device control.

- UNITY INOVA has five spare lines available from the Breakout panel on the back of the left console cabinet.

Examples: sp1off(); sp3off();

sp#on

Turn on specified spare line (Inova #=1 to 5)

Applicability: UNITY INOVA systems.

Syntax: sp1on() to sp3on()

Description: Turns on the specified user-dedicated spare line connector (sp1on for SPARE 1, sp2on for SPARE 2, etc.) for high-speed device control. Each spare line changes from low to high when turned on.

- UNITY INOVA has five spare lines available from the Breakout panel on the back of the left console cabinet.

Examples: sp1on(); sp3on();

spinlock

Control spin lock on observe transmitter

Applicability: UNITY INOVA Systems with a waveform generator on the observe transmitter channel.

Syntax: spinlock(pattern, 90_pulselength, tipangle_resoln, phase, ncycles)
char *pattern; /* name of .DEC text file */
double 90_pulselength; /* 90-deg pulse length of channel */
double tipangle_resoln; /* resolution of tip angle */
codeint phase; /* phase of spin lock */
int ncycle; /* number of cycles to execute */

Description: Executes a waveform-controlled spin lock on the observe transmitter. Both the rf gating and the mixing delay are handled within this function. Arguments can be variables (which require the appropriate getval and getstr statements) to permit changes via parameters (see the second example).

Arguments: pattern is the name of the text file in the shapelib directory that stores the decoupling pattern (leave off the .DEC file extension).
90_pulselength is the pulse duration for a 90° tip angle on the observe transmitter.
tipangle_resoln is the resolution in tip-angle degrees to which the decoupling pattern is stored in the waveform generator.
phase is the phase angle of the spin lock. It must be a real-time variable (v1 to v14, oph, etc.).
ncycle 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 First decoupler spin lock waveform control
dec2spinlock Second decoupler spin lock waveform control
dec3spinlock Third decoupler spin lock waveform control
**starthardloop**  
**Start hardware loop**  
**Syntax:**  
```c
codeint vloop; /* real-time variable for loop count */
```

**Description:** Starts a hardware loop. The number of repetitions of the hardware loop must be two or more. If the number of repetitions is 1, the hardware looping feature is not activated. A hardware loop with a count equal to 0 is not permitted and generates an error. Depending on the pulse sequence, additional code may be needed to trap for this condition and skip the `starthardloop` and `endhardloop` statements if the count is 0.

Only instructions that require no further intervention by the acquisition computer (pulses, delays, acquires, and other scattered instructions) are allowed in a hard loop. Most notably, no real-time math statements are allowed, thereby precluding any phase cycle calculations. The number of events included in the hard loop, including the total number of data points if acquisition is performed, is subject to the following limitations:

- 2048 or less for the Data Acquisition Controller board, Pulse Sequence Controller board, or **MERCURYplus/-Vx STM/Output board**.
- 1024 or less for the Acquisition Controller board.
- 63 or less for the Output board (see the description section of the `acquire` statement for further information about these boards).

In all cases, the number of events must be greater than one. No nesting of hard loops is allowed.

For the Output board, a hardware loop must be preceded by some timed event other than an explicit acquisition or another hardware loop. If two hardware loops must follow one another, it will therefore be necessary to insert a statement like `delay(0.2e-6)` between the first `endhardloop` and the second `starthardloop`. With only a single hardware loop, there is no timing limitation on the length of a single cycle of the loop. With two hardware loops (such as a loop of pulses and delays followed by an implicit acquisition), the first hardware loop must have a minimum cycle length of approximately 80 μs. With three or more hardware loops, loops that are not the first or last must have a minimum cycle length of about 100 μs.

For the Data Acquisition Controller, Pulse Sequence Controller, Acquisition Controller, and **MERCURYplus/-Vx STM/Output boards**, there are no timing restrictions between multiple, back-to-back hard loops. There is one subtle restriction placed on the actual duration of a hard loop if back-to-back hard loops are encountered: the duration of the $i$th hard loop must be $N(i+1) * 0.4 \mu s$, where $N(i+1)$ is the number of events occurring in the $(i+1)$th hard loop.

**Arguments:**  
- `vloop` is the number of hardware loop repetitions. It must be a real-time variable (`v1` to `v14`, `ct`, etc.) and **not** an integer, a real number, or a regular variable.

**Examples:**  
```c
starthardloop(v2);
```

**Related:**  
- `acquire`  
  Explicitly acquire data  
- `endhardloop`  
  End hardware loop

**status**  
**Change status of decoupler and homospoil (z-shim coil)**  
**Applicability:** **UNITY/INOVA** systems.

**Syntax:**  
```c
int state; /* index: A, B, C, ..., Z */
```
Description: Controls decoupler and homospoil gating. Parameters controlled by status are \( d_m \) (first decoupler mode), \( d_{mm} \) (first decoupler modulation mode), \( d_m^2 \) (second decoupler mode), \( d_m^3 \) (third decoupler mode), \( d_{mm}^2 \) (second decoupler modulation mode), and \( d_{mm}^3 \) (third decoupler modulation mode).

Each of these parameters can have multiple states: status(\( A \)) sets each parameter to the state described by the first letter of its value, status(\( B \)) uses the second letter, etc. If a pulse sequence has more status statements than there are status modes for a particular parameter, control reverts to the last letter of the parameter value. Thus if \( d_m = 'ny' \), status(\( C \)) will look for the third letter, find none, and then use the second letter (\( y \)) and turn the decoupler on (actually, leave the decoupler on).

The states do not have to increase monotonically during a pulse sequence. It is perfectly possible to write a pulse sequence that starts with status(\( A \)), goes later to status(\( B \)), then goes back to status(\( A \)), then to status(\( C \)), etc.

Homospoil is treated slightly differently than the decoupler. If a particular homospoil code letter is 'y', delays coded as hsdelay that occur during the time the status corresponds to that code letter will begin with a homospoil pulse, the duration of which is determined by the parameter hst. Thus if \( hs = 'ny' \), all hsdelay delays that occur during status(\( B \)) will begin with a homospoil pulse. The final status always occurs during acquisition, at which time a homospoil pulse is not permitted. Thus, if a particular pulse sequence uses status(\( A \)), status(\( B \)), and status(\( C \)), \( dm \) and other decoupler parameters can have up to three letters, but \( hs \) has only two, because having \( hs = 'y' \) during status(\( C \)) is meaningless and is consequently ignored. See also: “Amplifier Channel Blanking and Unblanking,” page 75

UNITYINOVA Systems and all systems with class C amplifiers to switch from low-power to high-power decoupling, insert dhpflag=TRUE; or dhpflag=FALSE; in a pulse sequence just before a status statement.

Arguments: state sets the status mode to A, B, C, ..., or Z.

Examples: status(A);

Related: hsdelay Delay specified time with possible homospoil pulse
setstatus Set status of observe transmitter or a decoupler transmitter
statusdelay Execute the status statement with a given delay time

statusdelay **Execute the status statement with a given delay time**

Applicability: UNITYINOVA systems.

Syntax: statusdelay(state,time)

int state; /* index: A, B, C, ..., Z */
double time; /* delay time, in sec. */

Description: Executes the status statement and delays for the time provided as an argument. statusdelay allows the user to specify a defined period of time for the status statement to execute.

UNITYINOVA:

The current status statement takes a variable amount of time to execute, which depends on the number of rf channels configured in the system, the previous status state of each decoupler channel, and the new status state of each decoupler channel. This time is small (on the order of a few microseconds
without programmable decoupling to tens of microseconds with programmable decoupling) but can be significant in certain experiments.

If the amount of time given as an argument is not long enough to account for the overhead delays of status; the pulse sequence will still run, but a warning message will be generated to let the user know of the discrepancy.

The following table lists the maximum amount of time per channel for the status statement to execute is 2.5 microseconds.

<table>
<thead>
<tr>
<th>Without programmable decoupling (μs)</th>
<th>With programmable decoupling (μs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.5</td>
<td>2.5</td>
</tr>
</tbody>
</table>

Arguments:  
- state specifies the status mode as A,B,C,...,Z.
- time specifies the delay time, in seconds.

Examples:  
- statusdelay(A,d1);
- statusdelay(B,0.000010);

Related:  
- status Change status of decoupler and homospoil

**stepsize**  
Set small-angle phase step size

Applicability:  
*unary* INOVA systems with rf type C or D and MERCURYplus/-Vx. This statement is due to be eliminated in future versions of VnmrJ software. Although it is still functional, you should not write any pulse sequences using it and should replace it in existing sequences with Use obsstepsize, decstepsize, dec2stepsize, or dec3stepsize, as appropriate.

Syntax:  
```
stepsize(step_size,device)
double step_size; /* step size of phase shifter */
int device; /* OBSch, DECch, DEC2ch, or DEC3ch */
```

Description: Sets the step size of the small-angle phase increment for a particular device. The phase information into statements decpulse, decrgpulse, dec2rgpulse, dec3rgpulse, pulse, rgpulse, and simpulse is still expressed in units of 90°.

Arguments:  
- step_size is a real number or a variable for the phase step size desired.
- device is OBSch (observe transmitter) or DECch (first decoupler). device can also be DEC2ch (second decoupler) or DEC3ch (third decoupler). The step_size phase shift selected is active only for the xmtrphase statement if device is OBSch, only for the dcplrphase statement if device is DECch, only for the dcplr2phase statement if device is DEC2ch, or only for the dcplr3phase statement if the device is DEC3ch.

Examples:  
- stepsize(30.0,OBSch);
- stepsize(step,DEC2ch);

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,
- decstepsize Set step size of first decoupler
- dec2stepsize Set step size of second decoupler
- dec3stepsize Set step size of third decoupler
- obsstepsize Set step size of observe transmitter
- xmtrphase Set small-angle phase of observe transmitter, rf type C
**sub**

**Subtract 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: Sets the value of `vk` equal to `vi – vj`.

Arguments: `vi` is the integer value of the minuend, `vj` is the integer value of the subtrahend, and `vk` is the difference of `vi` and `vj`. Each argument must be a real-time variable (`v1` to `v14`, `oph`, etc.).

Examples: `sub(v2,v5,v6);`

Related:
- `add` Add integer values
- `assign` Assign integer values
- `dbl` Double an integer value
- `decr` Decrement an integer value
- `divn` Divide integer values
- `hlv` Half the value of an integer
- `incr` Increment an integer value
- `mod2` Find integer value modulo 2
- `mod4` Find integer value modulo 4
- `modn` Find integer value modulo n
- `mult` Multiply integer values

---

**T**

---

**Top**

| A | B | C | D | E | G | H | I | L | M | O | P | R | S | T | V | W | X | Z |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

- `text_error` Send a text error message to VnmrJ
- `text_message` Send a message to VnmrJ
- `tsadd` Add an integer to AP table elements
- `tsdiv` Divide an integer into AP table elements
- `tsmult` Multiply an integer with AP table elements
- `tssub` Subtract an integer from AP table elements
- `ttadd` Add a table to a second table
- `ttdiv` Divide a table into a second table
- `ttmult` Multiply a table by a second table
- `ttsub` Subtract a table from a second table
- `txphase` Set quadrature phase of observe transmitter

**text_error**

Send a text error message to VnmrJ

Syntax: `text_error(char *format, ...)`
Chapter 3. Pulse Sequence Statement Reference

Description: Sends an error message to VnmrJ and writes the message into the file userdir+'/psg.error'.

text_message  Send a message to VnmrJ
Syntax: text_message(char *format, ...)
Description: Sends a message to VnmrJ. text_message is like warn_message, except it does not cause the beep to occur.

tsadd  Add an integer to table elements
Syntax: tsadd(table,scalarval,moduloval)
codeint table; /* real-time table variable */
int scalarval; /* integer added */
int moduloval; /* modulo value of result */
Description: A run-time scalar operation that adds an integer to 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: tsdiv  Divide an integer into table elements
tsmult  Multiply an integer with table elements
tssub  Subtract an integer from table elements

tsdiv  Divide an integer into table elements
Syntax: tsdiv(table,scalarval,moduloval)
codeint table; /* real-time table variable */
int scalarval; /* integer divisor */
int moduloval; /* modulo value of result */
Description: A run-time scalar operation that divides an integer into the 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);

Related: tsadd  Add an integer to table elements
tsmult  Multiply an integer with table elements
tssub  Subtract an integer from table elements

tsmult  Multiply an integer with table elements
Syntax: tsmult(table,scalarval,moduloval)
codeint table; /* real-time table variable */
int scalarval; /* integer multiplier */
int moduloval; /* modulo value of result */
Description: A run-time scalar operation that multiplies an integer with the elements of a table.

Arguments: table specifies the name of the table (t1 to t60).
scalarval is an integer to be multiplied with 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);

Related: tsadd Add an integer to table elements
tadiv Divide an integer into table elements
ttsmult Multiply an integer with table elements

**tssub** Subtract an integer from table elements

Syntax: tssub(table,scalarval,moduloval)
codeint table; /* real-time table variable */
int scalarval; /* integer subtracted */
int moduloval; /* modulo value of result */

Description: A run-time scalar operation that subtracts an integer from the elements of a table.

Arguments: table specifies 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: tsadd Add an integer to table elements
tadiv Divide an integer into table elements

**ttadd** Add a table to a second table

Syntax: ttadd(table_dest,table_mod,moduloval)
codeint table_dest; /* real-time table variable */
codeint table_mod; /* real-time table variable */
int moduloval; /* modulo value of result */

Description: A run-time vector operation that adds one table to a second table.

Arguments: tablenamedest 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: ttdiv Divide a table into a second table
ttdiv

**Divide a table into a second table**

Syntax: `ttdiv(table_dest, table_mod, moduloval)`

codeint table_dest; /* real-time table variable */
codeint table_mod; /* real-time table variable */
int moduloval; /* modulo value of result */

Description: A run-time vector operation that divides one table into a second table.

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 equal 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: `ttadd` Add a table to a second table
`ttmult` Multiply a table by a second table
`ttsub` Subtract a table from a second table

ttmult

**Multiply a table by a second table**

Syntax: `ttmult(table_dest, table_mod, moduloval)`

codeint table_dest; /* real-time table variable */
codeint table_mod; /* real-time table variable */
int moduloval; /* modulo value of result */

Description: A run-time vector operation that multiplies one table by a second table.

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: `ttadd` Add a table to a second table
`ttdiv` Divide a table into a second table
`ttsub` Subtract a table from a second table

ttsub

**Subtract a table from a second table**

Syntax: `ttsub(table_dest, table_mod, moduloval)`

codeint table_dest; /* real-time table variable */
codeint table_mod; /* real-time table variable */
int moduloval; /* modulo value of result */
Description: A run-time vector operation that subtracts one table from a second table.

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: `ttadd` Add a table to a second table
- `ttdiv` Divide a table into a second table
- `ttmult` Multiply a table by a second table

---

txphase

Set quadrature phase of observe transmitter

Syntax: `txphase(phase)`

```c
codeint phase; /* variable for quadrature phase */
```

Description: Sets the observe transmitter quadrature phase to the value referenced by the real-time variable so that the transmitter phase is changed independently from a pulse. This may be useful to “preset” the transmitter phase at the beginning of a delay that precedes a particular pulse. For example, in the sequence `txphase(v2); delay(d2); pulse(pw,v2);`, the transmitter phase is changed at the start of the d2 delay. In a “normal” sequence, an `rof1` time precedes the pulse in which the transmitter phase is changed.

Arguments: `phase` is the quadrature phase for the observe transmitter. It must be a real-time variable (v1 to v14, oph, ct, etc.).

Examples: `txphase(v3);`

Related: `decphase` Set quadrature phase of first decoupler
- `dec2phase` Set quadrature phase of second decoupler
- `dec3phase` Set quadrature phase of third decoupler

---

V

Top  A  B  C  D  E  G  H  I  L  M  O  P  R  S  T  V  W  X  Z

- `vagradient` Variable angle gradient
- `vagradpulse` Variable angle gradient pulse
- `var_active` Checks if the parameter is being used
- `vashapedgradient` Variable angle shaped gradient
- `vashapedgradpulse` Variable angle shaped gradient pulse
- `vdelay` Set delay with fixed timebase and real-time count
vgradient  Variable angle gradient
Syntax:  
```c
vgradient(gradlvl,theta,phi)
```

Arguments:  
- `gradlvl` is the gradient amplitude, in gauss/cm.
- `theta` defines the angle, in degrees, from the z axis.
- `phi` defines the angle of rotation, in degrees, about the xy plane.

Examples:  
```c
vgradient(3.0, 54.7, 0.0);
pulse(pw,oph);
delay(0.001 - pw);
zero_all_gradients();
```

Related:  
- `magradient`  Simultaneous gradient at the magic angle
- `magradpulse`  Simultaneous gradient pulse at the magic angle
- `mashapedgradient`  Simultaneous shaped gradient at the magic angle
- `mashapedgradpulse`  Simultaneous shaped gradient pulse at the magic angle
- `vagradient`  Variable angle gradient pulse
- `vashapedgradient`  Variable angle shaped gradient
- `vashapedgradpulse`  Variable angle shaped gradient pulse
- `zero_all_gradients`  Zero all gradients

vagradpulse  Variable angle gradient pulse
Applicability:  
UNITY/INOVA systems.

Syntax:  
```c
vagradpulse(gradlvl,gradtime,theta,phi)
```

Arguments:  
- `gradlvl` is the gradient amplitude, in G/cm.
- `gradtime` is the gradient time in sec.
- `theta` defines the angle, in degrees, from the z axis.
- `phi` defines the angle of rotation, in degrees, about the xy plane.

Examples:  
```c
vagradpulse(3.0, 54.7, 0.0);
pulse(pw,oph);
delay(0.001 - pw);
zero_all_gradients();
```
double phi;       /* angle of rotation in degrees */

Description: Applies a gradient pulse of amplitude \textit{gradlvl} at an angle \textit{theta} from the \textit{z} axis and rotated about the \textit{xy} plane at an angle \textit{phi}. Information from a gradient table is used to scale and set the values correctly. The values applied to each gradient axis are as follows:

\begin{align*}
    x &= \textit{gradlvl} \times (\sin(\phi) \times \sin(\theta)) \\
    y &= \textit{gradlvl} \times (\cos(\phi) \times \sin(\theta)) \\
    z &= \textit{gradlvl} \times (\cos(\theta))
\end{align*}

The gradients are turned off after \textit{gradtime} seconds.

\texttt{vagradpulse} is simpler to use if there are no other actions while the gradients are on. \texttt{vagradient} is used if there are actions to be performed while the gradients are on.

Arguments: \textit{gradlvl} is the gradient amplitude, in gauss/cm.
\textit{gradtime} is the time, in seconds, to apply the gradient.
\textit{theta} is the angle, in degrees, from the \textit{z} axis
\textit{phi} is the angle of rotation, in degrees, about the \textit{xy} plane.

Examples: \texttt{vagradpulse(3.0, 0.001, 54.7, 0.0)};

Related: \texttt{magradient} Simultaneous gradient at the magic angle
\texttt{magradpulse} Simultaneous gradient pulse at the magic angle
\texttt{mashapedgradient} Simultaneous shaped gradient at the magic angle
\texttt{mashapedgradpulse} Simultaneous shaped gradient pulse at the magic angle
\texttt{vagradient} Variable angle gradient
\texttt{vashapedgradient} Variable angle shaped gradient
\texttt{vashapedgradpulse} Variable angle gradient pulse
\texttt{zero_all_gradients} Zero all gradients

\texttt{var\_active} Checks if the parameter is being used

Syntax: \texttt{var\_active}

Description: Checks if the parameter is active (returns 1) or inactive (returns 0). Applies to numbers, not strings. “Inactive” means that the parameter is not being used. If the parameter is a number, it can be set to 'n' to make it “inactive.” For example, setting \texttt{fn=256} or \texttt{fn='n'}. If the parameter does not exist, \texttt{var\_active} is 0.

\texttt{vashapedgradient} Variable angle shaped gradient

Applicability: \texttt{UNITY}/\texttt{NOVA} systems.

Syntax: \texttt{vashapedgradient(pattern, gradlvl, gradtime, theta, phi, loops, wait)}
\begin{verbatim}
char* pattern;       /* name of gradient shape text file */
double gradlvl;      /* gradient amplitude in G/cm */
double gradtime;     /* time to apply gradient in sec */
double theta;        /* angle from z axis in degrees */
double phi;          /* angle of rotation in degrees */
int loops;           /* number of waveform loops */
int wait;            /* WAIT or NOWAIT */
\end{verbatim}
**Description:** Applies a gradient shape pattern with an amplitude `gradlvl` at an angle `theta` from the z axis and rotated about the xy plane at an angle `phi`. Information from a gradient table is used to scale and set the values correctly. The amplitudes applied to each gradient axis are as follows:

- \[ x = gradlvl \times (\sin(\phi) \times \sin(\theta)) \]
- \[ y = gradlvl \times (\cos(\phi) \times \sin(\theta)) \]
- \[ z = gradlvl \times (\cos(\theta)) \]

`vashapedgradient` leaves the gradients at the given levels until they are turned off. To turn off the gradients, add another `vashapedgradient` statement with `gradlvl` set to zero or insert a `zero_all_gradients` statement. Note that `vashapedgradient` assumes the gradient pattern zeroes the gradients at its end, and it does not explicitly zero the gradients.

`vashapedgradient` is used if there are actions to be performed while the gradients are on.

**Arguments:**
- `pattern` is a text file that describes the shape of the gradient. The text file is located in `$vnmrsystem/shapelib` or in the users directory `$vnmruser/shapelib`.
- `gradlvl` is the gradient amplitude, in gauss/cm.
- `gradtime` is the time, in seconds, to apply the gradient.
- `theta` is the angle, in degrees, from the z axis.
- `phi` is the angle of rotation, in degrees, about the xy plane.
- `loops` is a value from 0 to 255 to loop the selected waveform. Gradient waveforms do not use this field and it should be set to 0.
- `wait` is a keyword, either `WAIT` or `NOWAIT`, that selects whether or not a delay is inserted to wait until the gradient is completed before executing the next statement.

**Examples:**

```plaintext
vashapedgradient("ramp_hold",3.0,trise,54.7, 0.0,0,NOWAIT);
pulse(pw,oph);
delay(0.001-pw-2*trise);
vashapedgradient("ramp_down",3.0,trise,54.7, 0.0,0,NOWAIT);
```

**Related:**
- `magrant` Simultaneous gradient at the magic angle
- `magrafpulse` Simultaneous gradient pulse at the magic angle
- `mashapedgradient` Simultaneous shaped gradient at the magic angle
- `mashapedgradpulse` Simultaneous shaped gradient pulse at the magic angle
- `vagrant` Variable angle gradient
- `vagradpulse` Variable angle gradient pulse
- `vashapedgradpulse` Variable angle shaped gradient pulse
- `zero_all_gradients` Zero all gradients

**vashapedgradpulse** Variable angle shaped gradient pulse

**Applicability:** `UNITY/NOVA` systems.

**Syntax:**

```plaintext
vashapedgradpulse(pattern,gradlvl,gradtime,  
theta,phi)
char *pattern;      /* gradient shape text file */
double gradlvl;     /* gradient amplitude in G/cm */
double gradtime;    /* gradient time in seconds */
```
double theta;    /* angle from z axis in degrees */
double phi;      /* angle of rotation in degrees */

Description: Applies a gradient shape pattern with an amplitude gradlvl at an angle
theta from the z axis and rotated about the xy plane at an angle phi.
Information from a gradient table is used to scale and set the values correctly.
The amplitudes applied to each gradient axis are as follows:

\[
x = \text{gradlvl} \times (\sin(\phi) \times \sin(\theta)) \\
y = \text{gradlvl} \times (\cos(\phi) \times \sin(\theta)) \\
z = \text{gradlvl} \times (\cos(\theta))
\]

The gradient are turned off after gradtime seconds. Note that
\text{vashapedgradpulse} assumes that the gradient pattern zeroes the gradients
at its end and does not explicitly zero the gradients.
\text{vashapedgradpulse} is simpler to use than the \text{vashapedgradient}
statement if there are no other actions while the gradients are on.
\text{vashapedgradient} is used when there are actions to be performed while
the gradients are on.

Arguments: pattern is a text file that describes the shape of the gradient. The text file is
located in $vnmrsystem/shapelib or in the user directory $vnmruser/shapelib.
gradlvl is the gradient amplitude, in gauss/cm.
gradtime is the time, in seconds, to apply the gradient.
theta is the angle, in degrees, from the z axis.
phi is the angle of rotation, in degrees, about the xy plane.

Examples: \text{vashapedgradpulse("hsine",3.0,0.001,54.7,0.0);} 

Related: \text{magradient} Simultaneous gradient at the magic angle
\text{magradpulse} Simultaneous gradient pulse at the magic angle
\text{mashapedgradient} Simultaneous shaped gradient at the magic angle
\text{mashapedgradpulse} Simultaneous shaped gradient pulse at the magic angle
\text{vagradient} Variable angle gradient
\text{vagradpulse} Variable angle gradient pulse
\text{vashapedgradient} Variable angle shaped gradient
\text{zero_all_gradients} Zero all gradients

\text{vdelay} Set delay with fixed timebase and real-time count

Applicability: \text{UNITY/NOVA} systems.

Syntax: \text{vdelay(timebase,count)}

\begin{verbatim}
int timebase;    /* NSEC, USEC, MSEC, or SEC */
codeint count;   /* real-time variable for count */
\end{verbatim}

Description: Sets a delay for a time period equal to the product of the specified timebase
and the count.

Arguments: timebase is one of the four defined time bases: NSEC (described below),
USEC (microseconds), MSEC (milliseconds), or SEC (seconds).
count is a real-time variable (v1 to v14). For predictable acquisition, the real-
time variable should have a value of 2 or more.
If timebase is set to NSEC, the delay depends on which acquisition controller
board is used on the system (see the description section of the \text{acquire}
statement for further information about these boards):
• Systems with a Data Acquisition Controller board:
  
The minimum delay is a count of 0 (100 ns), and a count of \( n \) corresponds to a delay of \( (100 + (12.5* n)) \) ns. For example, \( \text{vdelay}(\text{NSEC}, v1) \), when \( v1 = 4 \), gives a delay of \( (100 + (12.5*4)) \) ns or 150 ns.

• Systems with a Pulse Sequence Controller board or an Acquisition Controller board:
  
The minimum delay is a count of 2 (200 ns). A count greater than 2 is the minimum delay plus the resolution (25 ns) of the board. For example, \( \text{vdelay}(\text{NSEC}, v1) \), when \( v1 = 4 \), gives a delay of \( (200 + 25) \) ns or 225 ns.

• Systems with Output boards
  
The minimum delay is a count of 2 (200 ns). A count greater than 2 is the minimum delay plus the resolution (100 ns) of the board. For example, \( \text{vdelay}(\text{NSEC}, v1) \), when \( v1 = 4 \), gives a delay of \( (200 + 100) \) ns or 300 ns.

Examples: \( \text{vdelay}(\text{USEC}, v3) \);

Related: \( \text{create\_delay\_list} \) Create table of delays
  
\( \text{delay} \) Delay for a specified time
  
\( \text{hsdelay} \) Delay specified time with possible homospoil pulse
  
\( \text{incdelay} \) Real time incremental delay
  
\( \text{initdelay} \) Initialize incremental delay
  
\( \text{vfreq} \) Select frequency from table
  
\( \text{voffset} \) Select frequency offset from table
  
\( \text{vdelay\_list} \) Get delay value from delay list with real-time index

\( \text{vdelay\_list}(\text{list\_number}, \text{vindex}) \) Get delay value from delay list with real-time index

Applicability: UNITY/INOVA systems.

Syntax: \( \text{vdelay\_list}(\text{list\_number}, \text{vindex}) \)
  
\( \text{int list\_number;} \) /* same index as create\_delay\_list */
  
\( \text{codeint vindex;} \) /* 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 \( \text{create\_delay\_list} \) statement. It has no return value.

Arguments: \( \text{tlist\_number} \) is the number between 0 and 255 for each list. This number must match the \( \text{list\_number} \) used when creating the table.
  
\( \text{vindex} \) is a real-time variable (\( v1 \) to \( v14 \)) or a table (\( t1 \) to \( t60 \)).

Examples: \( \text{pulsequence()} \)
  
\{...
  \text{int noffset, ndelay, listnum;}
  \text{double offsets1[256], offsets2[256], delay[256];}
  ...
  \text{/* initialize offset and delay lists */}
  \text{create\_offset\_list(offsets1, noffset, OBSch, 0);}
  \text{create\_delay\_list(delay, ndelay, 1);}
  \text{create\_offset\_list(offsets2, noffset, DECCh, 2);}
  ...
  \text{voffset(0, v4); /* get v4 from observe offset list */} \}
vdelay_list(1,v5); /* get v5 from delay list */
voffset(2,v4); /* get v4 from decouple offset list */
...

Related: create_delay_list Create table of delays
delay Delay for a specified time
hsdelay Delay specified time with possible homospoil pulse
incdelay Real time incremental delay
initdelay Initialize incremental delay
vfreq Select frequency from table
voffset Select frequency offset from table
vdelay Set delay with fixed timebase and real-time count

vfreq Select frequency from table

Applicability: UNITY/INOVA systems.

Syntax: vfreq(list_number,vindex)
int list_number; /* same index as for create_freq_list */
codeint vindex; /* real-time variable */

Description: Provides a means of indexing into previously created frequency 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 frequency table must have been created with the create_freq_list statement. It has no return value.

Arguments: list_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: See the example for the vdelay statement.

Related: create_freq_list Create table of frequencies
vdelay Select delay from table
voffset Select frequency offset from table

vgradient Set gradient to a level determined by real-time math

Applicability: UNITY/INOVA systems with PFG modules. Not applicable to MERCURYplus/-Vx.

Syntax: vgradient(channel,intercept,slope,mult)
char channel; /* gradient channel 'x', 'y' or 'z' */
int intercept; /* initial gradient level */
int slope; /* gradient increment */
codeint mult; /* real-time variable */

Description: Provides a dynamic variable gradient controlled using the real-time math functions. It has no return value. The statement drives the chosen gradient to the level defined by the formula:

\[
\text{level} = \text{intercept} + \text{slope} \times \text{mult}.
\]

The gradient level ranges from \(-2047\) to \(+2047\) for systems with 12-bit DACs, or from \(-32767\) to \(+32767\) for gradients using the waveform, which have 16-bit DACs. If the requested level lies outside this range, it is rounded to the appropriate boundary value.

After vgradient, the action of the gradient is controlled by the gradient power supply. The gradient level is ramped at the preset slew rate (2047 DAC...
units per millisecond) to the value requested by vgradient. This fact becomes a concern when using vgradient in a loop with a delay element, in order to produce a modulated gradient. The delay element should be sufficiently long so as to allow the gradient to reach the assigned value:
\[
\text{delay} \geq \frac{\text{new_level} - \text{old_level}}{2047} \times \text{risetime}
\]

Arguments:  channel specifies the gradient to be set and is one of the characters 'X', 'x', 'Y', 'y', 'Z', or 'z'. In imaging, channel can also be 'gread', 'gphase', or 'gslice'.
intercept and slope are integers. In imaging, intercept is the initial gradient DAC setting and slope is the gradient DAC increment.
mult is a real-time variable (v1 to v14, etc.). In imaging, mult is set so that intercept+slope*mult is the output.

Examples:

(1) mod2(ct,v10); /* v10 is 0,1,0,1,0,1,... */
vgradient('z',0,2000,v10); /* z gradient is 0,2000,0,2000,... */
delay(d2); /* delay for duration d2 */
grgradient('z',0.0); /* gradient turned off */

(2) mod4(ct,v10); /* v10 is 0,1,2,3,4,0,1,2,3,4,... */
vgradient('z',-5000.0,2500.0,v10); /* z is –5000,–2500,0,2500 */

(3) pulsesequence()
{
...
char gphase, gread, gslice;
int amplitude, igpe, stat;
double gpe;
...
gpe = getval("gpe");
amplitude = (int)(0.5*ni*gpe);
ingpe = (int)gpe;
stat = getorientation(&gread,&gphase,&gslice,"orient");
...
initval(nf,v9);
loop(v9,v5);
...
vgradient(gphase,amplitude,igpe,v5);
...
endloop(v5);
...
}

Related:
- dps_show: Draw delay or pulses in a sequence for graphical display
- getorientation: Read image plane orientation
- grgradient: Set gradient to specified level
- shapedgradient: Provide shaped gradient pulse to gradient channel
- shaped2Dgradient: Generate arrayed shaped gradient pulse
- shapedvgradient: Generate dynamic variable shaped gradient pulse
- zgradpulse: Create a gradient pulse on the z channel
voffset  Select frequency offset from table

Applicability: UNITY/NOVA systems.

Syntax: voffset(list_number,vindex)
        int list_number; /* number of list */
        codeint vindex; /* real-time or table variable */

Description: Provides a means of indexing into previously created frequency offset 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 offset table has to have been
              created with the create_offset_list statement. It has no return value.

Arguments: list_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: See the example for the vdelay statement.

Related: create_offset_list  Create table of frequency offsets
         vdelay         Select delay from table
         vfreq         Select frequency from table

vsetuserap  Set user AP register using real-time variable

Applicability: UNITY/NOVA systems.

Syntax: vsetuserap(vi,register)
         codeint vi; /* variable output to AP bus register */
         int register; /* AP bus register: 0, 1, 2, or 3 */

Description: Sets one of the four 8-bit AP bus registers that provide an output interface to
              custom user equipment. The outputs of these registers go the USER AP
              connectors J8212 and J8213, located on the back of the left console cabinet. The
              outputs have a 100-ohm series resistor for circuit protection.

Arguments: vi is an index to a real-time variable that contains a signed or unsigned real
            number or integer to output to the specified user AP register.
            register is the AP register number, mapped to output lines as follows:
            • Register 0 is J8213, lines 9 to 16.
            • Register 1 is J8213, lines 1 to 8.
            • Register 2 is J8212, lines 9 to 16.
            • Register 3 is J8212, lines 1 to 8.

Examples: vsetuserap(v1,1);

Related: readuserap  Read input from user AP register
         setuserap    Set user AP register

---

W
Chapter 3. Pulse Sequence Statement Reference

**warn_message**  
Send a warning message to VnmrJ

**warn_message**  
Send a warning message to VnmrJ  
Syntax: `warn_message(char *format, ...)`  
Description: Sends an warning message to VnmrJ and causes a beep.

---

**x**

---

### Top  A  B  C  D  E  G  H  I  L  M  O  P  R  S  T  V  W  X  Z

- **xgate**  
  Gate pulse sequence from an external event  
  Applicability: UNITY INOVA systems.  
  Syntax: `xgate(events)`  
  ```c
  double events; /* number of external events */
  ```  
  Description: Halts the pulse sequence. When the number of external events have occurred, the pulse sequence continues.  
  Arguments: `events` is the number of external events.  
  Examples:  
  ```c
  xgate(2.0);
  xgate(events);
  ```
  Related:  
  - `rotorperiod`  
  - `rotorsync`  

- **xmtroff**  
  Turn off observe transmitter  
  Syntax: `xmtroff()`  
  Description: Explicitly gates off the observe transmitter in the pulse sequence.

  Related:  
  - `xmtron`  

- **xmtron**  
  Turn on observe transmitter  
  Syntax: `xmtron()`
Description: Explicitly gates on the observe transmitter in the pulse sequence. Transmitter gating is handled automatically by the statements `obspulse`, `pulse`, `rgpulse`, `shaped_pulse`, `simpulse`, `sim3pulse`, `simshaped_pulse`, `sim3shaped_pulse`, and `spinlock`. The `obsprgon` statement generally needs to be enabled with an explicit `xmtron` statement and followed by a `xmtroff` call.

**Related:** `xmtroff` Turn on observe transmitter

---

**xmtrphase** Set transmitter small-angle phase

**Syntax:**
```
xmtrphase(multiplier)
```

**Codeint** multiplier; /* real-time AP variable */

**Description:** Sets the phase of transmitter in units set by the `obsstepsize` statement. The small-angle phaseshift is a product of `multiplier` and the preset step size for the transmitter. If `stepsize` has not been used, the default step size is 90°.

If the product of the step size set by the `stepsize` statement and `multiplier` is greater than 90°, the sub-90° part is set by `xmtrphase`. Carryovers that are multiples of 90° are automatically saved and added in at the time of the next 90° phase selection (such as at the time of the next `pulse` or `decpulse`).

`xmtrphase` should be distinguished from `txphase`. `xmtrphase` is needed any time the transmitter phase shift is to be set to a value that is not a multiple of 90°. `txphase` is optional and rarely is needed.

**Arguments:** `multiplier` is a small-angle phaseshift multiplier and must be a real-time variable.

**Examples:**
```
xmtrphase(v1);
```

**Related:**
- `dcplr1phase` Set small-angle phase of first decoupler
- `dcplr2phase` Set small-angle phase of second decoupler
- `dcplr3phase` Set small-angle phase of third decoupler
- `stepsize` Set small-angle phase step size

---

**Z**

---

**zero_all_gradients** Zero all gradients

**Syntax:**
```
zero_all_gradients()
```

**Description:** Sets the gradients in the x, y, and z axes to zero.

**Related:**
- `zgradpulse` Create a gradient pulse on the z channel
Examples: 

\[ \text{vagradient}(3.0, 54.7, 0.0); \]
\[ \text{delay}(0.001); \]
\[ \text{zero_all_gradients}(); \]

Related: 

- \text{vagradient} Variable angle gradient
- \text{vagradpulse} Variable angle gradient pulse
- \text{vashapedgradient} Variable angle shaped gradient
- \text{vashapedgradpulse} Variable angle shaped gradient pulse

\text{zgradpulse} \quad \text{Create a gradient pulse on the z channel}

Applicability: \text{UNITY/NOVA} systems with PFG module.

Syntax: 

\[ \text{zgradpulse}(\text{value}, \text{delay}) \]
\[ \text{double value;} \quad /* \text{amplitude of gradient on z channel} */ \]
\[ \text{double delay;} \quad /* \text{length of gradient in sec} */ \]

Description: Creates a gradient pulse on the z channel with amplitude and duration given by the arguments. At the end of the pulse, the gradient is set to 0.

Arguments: 

- \text{value} is the amplitude of the pulse. It is a real number between \(-32768\) and \(32767\).
- \text{delay} is any delay parameter, such as \text{d2}.

Examples: 

\[ \text{zgradpulse}(1234.0, \text{d2}); \]

Related: 

- \text{dps_show} Draw delay or pulses for graphical display of a sequence
- \text{rgradient} Set gradient to specified level
- \text{vgradient} Set gradient to level determined by real-time math
Chapter 4. Linux Level Programming

Sections in this chapter:

- 4.1 “Linux and VnmrJ,” this page
- 4.2 “Linux Reference Guide,” page 267
- 4.3 “Linux Commands Accessible from VnmrJ,” page 270
- 4.4 “Background VNMR,” page 270
- 4.5 “Shell Programming,” page 272

Hundreds of books 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.

4.1 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 on line help provided with Linux and other published third party references to learn about these tools.

4.2 Linux Reference Guide

- “Command Entry,” page 268
- “File Names,” page 268
- “File Handling Commands,” page 268
- “Directory Names,” page 268
- “Directory Handling Commands,” page 268
- “Text Commands,” page 269
- “Other Commands,” page 269
- “Special Characters,” page 269

This is a brief overview of the operating system and its associated commands.
Chapter 4. Linux Level Programming

Command Entry

Single command entry
Command names
Multiple command separator
Arguments

commandname
Generally lowercase, case-sensitive
; (semicolon) or new line
commandname arg1 arg2

File Names

Typical (shorthand names usually used)
Level separator
Individual filenames
Characters in filenames
First character in filename

/vnmr/fidlib/fid1d
/ (forward slash)
Any number of characters (256 unique)
Underline, period often used
First character unrestricted

File Handling Commands

Delete (unlink) a file(s)
Copy a file
Rename a file
Make an alias (link)
Sort files
Tape backup
Package files

rm filenames
cp filename newfilename
mv filename newfilename
ln target linkname
sort filenames
tar
zip

Directory Names

Home directory for each user
Working directory
Shorthand for current directory
Shorthand for parent directory
Shorthand for home directory
Root directory

Directory assigned by administrator
Current directory user is in
.(single period)
.. (two periods)
~ (tilde character)
/ (forward slash)

Directory Handling Commands

Create (or make) a directory
Rename a directory
Remove an empty directory
Delete directory and all files in it
List files in a directory, short list
List files in a directory, long list
Copy file(s) into a directory
Move file(s) into a directory

mkdir directoryname
mv dirname newdirname
rmdir directoryname
rm -r directoryname
ls directoryname
ls -l directoryname
cp filenames directoryname
mv filenames directoryname
4.2 Linux Reference Guide

Show current directory  
Change current directory 

Text Commands

- Edit a text file using vi editor: vi filename
- Edit a text file using ed editor: ed filename
- Edit a text file using textedit editor: textedit filename
- Display first part of a file: head filename
- Display last part of a file: tail filename
- Concatenate and display files: cat filenames
- Compare two files: cmp filename1 filename2
- Compare two files differentially: diff filename1 filename2
- Display first part of a file: head filename
- Display last part of a file: tail filename
- Concatenate and display files: cat filenames
- Compare two files: cmp filename1 filename2
- Compare two files differentially: diff filename1 filename2
- Print file(s) on line printer: lp filenames
- Search file(s) for a pattern: grep expression filenames
- Find spelling errors: spell filename

Other Commands

- Pattern scanning and processing: awk pattern filename
- Change file protection mode: chmod newmode filename
- Display current date and time: date
- Summarize disk usage: du -k
- Report free disk space: df -k filesystem
- Kill a background process: kill process-id
- Sign onto system: login username
- Send mail to other users: mail
- Print out UNIX manual entry: man commandname
- Process status: ps
- Convert quantities to another scale: units
- Who is on the system: w
- System identification: uname -a

Special Characters

- Send output into named file: > filename
- Append output into named file: >> filename
- Take input from named file: < filename
- Send output from first command to input of second command (pipe): | (vertical bar)
- Wildcard character for a single character in filename operations: ?
- Wildcard character for multiple characters in filename operations: *
- Run program in background: &
- Abort the current process: Control-C
- Logout or end of file: Control-D
4.3 Linux Commands Accessible from VnmrJ

- “Opening a Text Editor from VnmrJ,” page 270
- “Opening a Shell from VnmrJ,” page 270

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 most widely used Linux and UNIX text editors because of its many 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(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:

```sh
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.

4.4 Background VNMR

- “Running VNMR Command as a Linux Background Task,” page 270
- “Running VNMR Processing in the Background,” page 271

Running VNMR commands and processing as a Linux or UNIX background task are possible using `vbg` commands from Linux or UNIX.

**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 \(-m\)back is a keyword (entered exactly as shown), \(-n\#\) sets 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 \(Vnmr -m\)back \(wft2da\) &. Usually it is a good idea to use redirection (> or >>) with background processing:
\(Vnmr -m\)back \(-n3\) \(wft2da\) > \(vnmroutput\) &

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 multiuser 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 use other experiments, however.
- Pressing Control-C does not work: typing the command shown cannot be abort it with Control-C.
- Operation within VNMR is possible using the shell command, e.g.,
  \(\text{shell('Vnmr \(-m\)back \(-n2\) wft2da')}\)
- Plotting is possible; e.g.,
  \(Vnmr -m\)back \(-n3\) "pl pscale pap page"
- Printing is possible; e.g.,
  \(Vnmr -m\)back "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:

\(vbg \# \text{command\_string} <\text{prefix}>\)

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 \(vbg\).
4.5 Shell Programming

- “Shell Variables and Control Formats,” page 272
- “Shell Scripts,” page 272

The shell executes commands given either from a terminal or 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 very 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:

<table>
<thead>
<tr>
<th>if command-list (not Boolean)</th>
<th>while command-list</th>
</tr>
</thead>
<tbody>
<tr>
<td>then command-list</td>
<td>do command-list</td>
</tr>
<tr>
<td>else command-list</td>
<td>done</td>
</tr>
<tr>
<td>fi</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>case word in pattern) command-list;</th>
<th>for name (in w1 w2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>do command-list</td>
</tr>
<tr>
<td>esac</td>
<td>done</td>
</tr>
</tbody>
</table>

Shell Scripts

The following shell scripts show two ways a shell script might be written 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:

```bash
: 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:

```bash
: 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, but 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).
Chapter 5. Parameters and Data

Sections in this chapter:

- 5.1 “VnmrJ Data Files,” this page
- 5.2 “FDF (Flexible Data Format) Files,” page 280
- 5.3 “Reformatting Data for Processing,” page 285
- 5.4 “Creating and Modifying Parameters,” page 288
- 5.5 “Modifying Parameter Displays in VNMR,” page 294
- 5.6 “User-Written Weighting Functions,” page 297
- 5.7 “User-Written FID Files,” page 300

5.1 VnmrJ Data Files

- “Binary Data Files,” page 273
- “Data File Structures,” page 275
- “VnmrJ Use of Binary Data Files,” page 278
- “Storing Multiple Traces,” page 279
- “Header and Data Display,” page 280

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 UNIX 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 an Fourier transform, the experiment file expn/datdir/data contains the transformed spectra stored in 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 stores also 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 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 UNIX operating system performs. The command flush 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. Within the 3D data subdirectory /data are the following files and further subdirectories:

- 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 UNIX 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 $f_2$ and $f_1$ transforms, there are two log file 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.
5.1 VnmrJ Data Files

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 f1 and f2 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 is 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:

```c
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.
Chapter 5. Parameters and Data

- 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.

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.

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:

```c
struct datablockhead
{
  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):

- **MORE_BLOCKS**: 0x80 0 = absent, 1 = present
- **NP_CMPLX**: 0x100 0 = real, 1 = complex
- **NF_CMPLX**: 0x200 0 = real, 1 = complex
- **NI_CMPLX**: 0x400 0 = real, 1 = complex
- **NI2_CMPLX**: 0x800 0 = real, 1 = complex

Additional data block header for hypercomplex 2D data:
```c
struct hypercmplxbhead
{
    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:
- Bits 0–3: bit 3 is currently unused
- Bits 4–7: bit 7 is currently unused
- Bits 8–11: bit 11 is currently unused
- Bits 12–15: bit 15 is currently unused

Usage bits for additional block headers (hypercmplxbhead.status)
```c
U_HYPERCOMPLEX 0x2 1 = hypercomplex block structure
```

The actual FID data is typically stored as pairs 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
Chapter 5. Parameters and Data

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

The following example of a simple Fourier transform (performed with the command \texttt{ft}) followed by the display of the spectrum illustrates how VnmrJ uses individual binary data files.

1. Copy processing parameters from \texttt{curpar} into \texttt{procpar}.
2. If a FID is not in the \texttt{fid} file buffer, open the \texttt{fid} file (if not already open) and load it into buffer.
3. Initialize the data file with the proper size (using parameter \texttt{fn}).
4. Store the FID in the data file buffer.
5. Apply dc drift correction and first point correction.
6. Apply weighting function, if requested.
7. Zero fill data, if required.
8. Fourier transform data in data file buffer.

The data file buffer now 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 \texttt{expn/datdir/data} at this time. Joining a different experiment or the command \texttt{flush} would perform such a write operation.

The \texttt{ds} command takes the following steps in displaying the spectrum:

1. If data is not in phasefile buffer or if the phase parameters have changed, \texttt{ds} tries to open the phase file (if not already open) and load data into the buffer (if it is there). If \texttt{ds} is unsuccessful, the data must be phased:
   a. If the data is not in the data file buffer, \texttt{ds} opens the data file (if not already open) and loads it into the buffer.
   b. \texttt{ds} initializes the phasefile buffer with the proper size (using the same parameter \texttt{fn} as used for last FT).
   c. \texttt{ds} calculates the phased (or absolute value) spectrum and stores it in the phasefile buffer.
2. \texttt{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 \texttt{expn/datdir/phasefile} at this time. Joining a different experiment or entering the command \texttt{flush} would perform such a write operation.

Depending on the nature of the data processing, the two files data and phasefile will contain different information, after each of the following processes:

- \textit{ID FT} – data contains a complex spectrum, which can be used for phased or absolute value displays.
- \textit{ID display} – phasefile contains either phased or absolute value data, depending on which type of display had been selected.
• **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 data.

• **The first FT in a 2D experiment** – data contains the once-transformed spectra. This is equivalent to the interferograms, if the data is properly reorganized (see f1 and f2 traces in “Storing Multiple Traces,” page 279). If a display is done now, phasefile contains phased (or absolute value) half-transformed spectra or interferograms.

• **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 just uses a “seek” operation to position the file pointer at the right point in the file and does not try to read traces that have never been calculated.

A 2D experiment resembles a special case of an arrayed experiment that is complicated by the fact that the data often has to be transposed. The directly acquired arrayed FIDs are Fourier transformed creating an array of spectra that are transposed and become the FIDs used for the second dimension Fourier transform. After the second FT, the user might want to work on traces in either the f1 or f2 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 and cannot therefore 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 is 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 f1 and f2 are stored in the same file. The first block(s) contain traces number 1 to fn along the f1 axis; the next block(s) contains traces number 1 to fn1 along the f2 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 f1 or a ‘hole’ followed by the block(s) for the traces along f2. Furthermore, in large 2D experiments, where multiple blocks must be used to store the whole data, 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 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.

5.2 FDF (Flexible Data Format) Files

- “File Structures and Naming Conventions,” page 280
- “File Format,” page 281
- “Header Parameters,” page 282
- “Transformations,” page 284
- “Creating FDF Files,” page 284
- “Splitting FDF Files,” page 285

The FDF file format was developed to support the ImageBrowser, 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 `proccpar`, a standard VnmrJ name.
5.2 FDF (Flexible Data Format) Files

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:

- **Listing 8** 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 `procpar` 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.

**Listing 8. Example of an FDF Header**

```c
#!/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>
```
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. Figure 4 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:
  
  \[
  \begin{align*}
  d_{11} & , d_{12} , d_{13} , d_{21} , d_{22} , d_{23} , d_{31} , d_{32} , d_{33} \\
  x & = d_{11}X + d_{12}Y + d_{13}Z \\
  y & = d_{21}X + d_{22}Y + d_{23}Z \\
  z & = d_{31}X + d_{32}Y + d_{33}Z
  \end{align*}
  \]

  where:

  \[
  \begin{align*}
  X & = d_{11}x + d_{21}y + d_{31}z \\
  Y & = d_{12}x + d_{22}y + d_{32}z \\
  Z & = d_{13}x + d_{23}y + d_{33}z
  \end{align*}
  \]

  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};`).
5.2 FDF (Flexible Data Format) Files

- **location** is the position of the center of the acquired data volume relative to the center of the magnet, in the user’s coordinate 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 coordinates of data points. These axes do not tell how to orient the display of the data, but only what to call the coordinates 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 coordinates 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 coordinate 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"};`).

![Magnet Coordinates as Related to User Coordinates.](image-url)

**Figure 4.** Magnet Coordinates as Related to User Coordinates.

---

01-999379-00 A 0808  VnmrJ 2.2D User Programming  283
Chapter 5. Parameters and Data

- 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 requires interpreting the term “ppm” for the specific nucleus, if ppm to frequency conversions are necessary, and properly labeling axes arising from different nuclei. 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 the 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 its data type. For single precision floating point data, the data is aligned on word boundaries. Currently, the end of 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:

\[ \text{span}'_0 = \text{span}_0 \]
\[ \text{origin}'_0 = \text{origin}_0 - \text{span}'_0 \]

Creating FDF Files

To generate files in the FDF format, the following macros are available to write out single or multislice images:

- For the current imaging software—including sequences sms, 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 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 
  \texttt{svsis(directory<,’f’|’m’>)}, where directory, 'f', and 'm' are 
  defined the same as \texttt{svib}.

Raw data from the FID file of the current experiment can be saved as an FDF file with the 
\texttt{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 \texttt{fid0001.fdf}, \texttt{fid0002.fdf}, etc. The \texttt{procpar} 
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 \texttt{fdfgluer} program. Use the syntax 
\texttt{fdfgluer header_file <data_file <output_file>}} (from UNIX 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, \texttt{fdfgluer} assumes the data is input from the standard input, and 
if the output_file name is not present, \texttt{fdfgluer} puts the FDF file to the standard 
output.

\section*{Splitting FDF Files}

The \texttt{fdfsplit} command takes an FDF file and splits it into its data and header parts. The 
syntax is \texttt{fdfsplit fdf_file data_file header_file} (from UNIX only). If 
the header still has a checksum value, that value should be removed.

\section{5.3 Reformatting Data for Processing}

• “Standard and Compressed Formats,” page 286
• “Compress or Decompress Data,” page 287
• “Move and Reverse Data,” page 287
• “Table Convert Data,” page 287
• “Reformatting Spectra,” page 287

Sometimes, data acquired 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 \texttt{ft2d} and \texttt{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 decompress their data (\texttt{flashc}), move data around 
between experiments and into almost any format (\texttt{mf, mfblk, mfdata, mftrace}), 
reverse data while moving it (\texttt{rfblk, rdfdata, rftrace}), or use a table of values, in this 
case a table stored in \texttt{tablib}, to sort and reformat scans of data (\texttt{tabc, tcapply}).

In this section, standard and compressed data are defined, reformatting options are 
described, and several examples are presented. Table 40 summarizes the reformatting 
commands described in this section. Note that the commands \texttt{rsapply, tcapply, 
tcclose, and tcoopen} are for 2D spectrum data; the remaining commands in the table 
are for FID data.
Chapter 5. Parameters and Data

### Table 40. Commands for Reformatting Data

<table>
<thead>
<tr>
<th>Commands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>flashc*</td>
<td>Convert compressed 2D data to standard 2D format</td>
</tr>
<tr>
<td>mf(from_exp,&gt;to_exp)</td>
<td>Move FIDs between experiments</td>
</tr>
<tr>
<td>mfblk*</td>
<td>Move FID block</td>
</tr>
<tr>
<td>mfclose</td>
<td>Close memory map FID</td>
</tr>
<tr>
<td>mfdata*</td>
<td>Move FID data</td>
</tr>
<tr>
<td>mfnopen&lt;src_expno,&gt;dest_expno&gt;</td>
<td>Memory map open FID file</td>
</tr>
<tr>
<td>mftrace*</td>
<td>Move FID trace</td>
</tr>
<tr>
<td>rfbk*</td>
<td>Reverse FID block</td>
</tr>
<tr>
<td>rfdata*</td>
<td>Reverse FID data</td>
</tr>
<tr>
<td>rftrace*</td>
<td>Reverse FID trace</td>
</tr>
<tr>
<td>rsapply</td>
<td>Reverse data in a spectrum</td>
</tr>
<tr>
<td>tabc&lt;dimension&gt;</td>
<td>Convert data in table order to linear order</td>
</tr>
<tr>
<td>tcapply&lt;file&gt;</td>
<td>Apply table conversion reformatting to data</td>
</tr>
<tr>
<td>tcclose</td>
<td>Close table conversion file</td>
</tr>
<tr>
<td>tcopen&lt;file&gt;</td>
<td>Open table conversion file</td>
</tr>
</tbody>
</table>

* 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,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,num_points>
  * rftrace<src_expno,src_blk_no,src_trace_no,dest_expno,dest_blk_no,dest_trace_no>

### Standard and Compressed Formats

The terms standard and compressed data formats have the following meaning:

- **standard** — the data was acquired using the arrayed parameters \( n_1 \) and \( n_2 \), which specify the number of increments in the second and third dimensions.
- **compressed** — the data was acquired using parameter \( n_f \) to specify the increments in the second dimension.

For multislice imaging, standard means using \( n_1 \) 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** — uses \( n_f \) 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 \( n_v*n_s \) or \( n_v*n_v^2 \), where \( n_v \) is the number of phase-encode increments in the second dimension, \( n_v^2 \) is the number of phase-encode increments in the third dimension, and \( n_s \) is the number of slices.

To give another view of data formats, which will help when using the “move FID” commands, each \( n_1 \) increment or array element is stored as a data block in a FID file and each \( n_f \) 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, Listing 9 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 `rfblk`, `rftrace`, and `rfdata` are similar to their respective `mfblk`, `mftrace`, and `mfdata` commands, except that `rfblk`, `rftrace`, and `rfdata` commands also reverse the order of the data. The `rfblk`, `rftrace`, and `rfdata` commands were implemented to support EPI (Echo Planar Imaging) processing. Listing 10 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 is moved. The `rfblk`, `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 \texttt{ft2d} command to complete the processing.

**Listing 9. Code from a “Move FID” Macro**

```plaintext
if ($seqcon[3] = 'c') and ($seqcon[4] = 'c') then
  "****  Compressed-compressed 3d ****"
  $arraydim = arraydim
  if ($index > $arraydim) then
    write('error','Index greater than arraydim.')
    abort
  endif
  mfblk($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
    mfblk($k,$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)
```

### 5.4 Creating and Modifying Parameters

- “Parameter Types and Trees,” page 289
- “Tools for Working with Parameter Trees,” page 289
- “Format of a Stored Parameter,” page 292

VnmrJ parameters and their 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 detail, refer to the description of the create command in the VnmrJ Command and Parameter Reference):

- 'real' any positive or negative value, and can be positive or negative.
- 'string' composed of characters, and can be limited to selected words by enumerating the possible values with the command setenumeral.
- 'delay' a value between 0 and 8190, in units of seconds.
- 'frequency' positive real number values.
- 'flag' composed of characters, similar to the 'string' type, but can be limited to selected characters by enumerating the possible values with the command setenumeral.
  
  If enumerated values are not set, the 'string' and 'flag' types are identical.
- 'pulse' value between 0 and 8190, in units of microseconds.
- 'integer' 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 vnmr.sys directory of the present UNIX user.
- 'processed' contains the parameters with which the data was obtained. These parameters are from the file procpar 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 41 lists commands for creating, modifying, and deleting parameters.

```
Listing 10. Example of Command Reversing Data Order

"***************************************************************
" epirf(<blkno>) - macro to reverse every other FID
" block & trace indicies start at 1 for rfblk,rftrace,rfdata **
"***************************************************************
mfopen
$i=2
while ($i <= nv) do
  rftrace($1,$i)
  $i = $i + 2
endwhile
mfclose
```
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.

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 Table 41. Commands for Working with Parameter Trees

<table>
<thead>
<tr>
<th>Commands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create (parameter&lt;, type&lt;, tree&gt;&gt;)</td>
<td>Create a new parameter in parameter tree</td>
</tr>
<tr>
<td>destroy (parameter&lt;, tree&gt;)</td>
<td>Destroy a parameter</td>
</tr>
<tr>
<td>destroygroup (group&lt;, tree&gt;)</td>
<td>Destroy parameters of a group in a tree</td>
</tr>
<tr>
<td>display (parameter</td>
<td>'*'</td>
</tr>
<tr>
<td>fread (file&lt;, tree&lt;,'reset'</td>
<td>'value'&gt;&gt;)</td>
</tr>
<tr>
<td>fsave (file&lt;, tree&gt;)</td>
<td>Save parameters from a tree to a file</td>
</tr>
<tr>
<td>getvalue (parameter&lt;, index&lt;, tree&gt;)</td>
<td>Get value of parameter in a tree</td>
</tr>
<tr>
<td>groupcopy (from_tree, to_tree, group)</td>
<td>Copy group parameters from tree to tree</td>
</tr>
<tr>
<td>paramvi (parameter&lt;, tree&gt;)</td>
<td>Edit parameter and its attributes using vi</td>
</tr>
<tr>
<td>prune (file)</td>
<td>Prune extra parameters from current tree</td>
</tr>
<tr>
<td>setdgroup (parameter, dgroup&lt;, tree&gt;)</td>
<td>Set the Dgroup of a parameter in a tree</td>
</tr>
<tr>
<td>setenumeral*</td>
<td>Set values of a string parameter in a tree</td>
</tr>
<tr>
<td>setgroup (parameter, group&lt;, tree&gt;)</td>
<td>Set group of a parameter in a tree</td>
</tr>
<tr>
<td>setlimit*</td>
<td>Set limits of a parameter in a tree</td>
</tr>
<tr>
<td>setprotect*</td>
<td>Set protection mode of a parameter</td>
</tr>
<tr>
<td>settype (parameter, type&lt;, tree&gt;)</td>
<td>Change type of a parameter</td>
</tr>
<tr>
<td>setvalue*</td>
<td>Set value of any parameter in a tree</td>
</tr>
<tr>
<td>* setenumeral (parameter, N, enum1, enum2, ...enumN&lt;, tree&gt;)</td>
<td></td>
</tr>
<tr>
<td>setlimit (parameter, maximum, minimum, step_size&lt;, tree&gt;) or setlimit (parameter, index&lt;, tree&gt;)</td>
<td></td>
</tr>
<tr>
<td>setprotect (parameter, 'set'</td>
<td>'on'</td>
</tr>
<tr>
<td>setvalue (parameter, value&lt;, index&lt;, tree&gt;)</td>
<td></td>
</tr>
</tbody>
</table>

Table 41. Commands for Working with Parameter Trees
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 current protection bits to be replaced with the set specified by `bit_vals` (listed in the 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.
- `setenumeral(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 uses this command currently.

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 in 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<intree>)` command removes a parameter from a parameter tree while the `destroygroup(group<intree>)` 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` instead. 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, requires knowledge of the format of a stored parameter. If an error in the format is made, the parameter may not load. This section describes the format in detail.

The stored format of a parameter is made up of three or more lines:

- Line 1 contains the attributes of the parameter and has the following fields (given in same order as they appear in the file):

  - `name`: the parameter name, which can be any valid string.
  - `subtype`: an integer value for the parameter type: 0 (undefined), 1 (real), 2 (string), 3 (delay), 4 (flag), 5 (frequency), 6 (pulse), 7 (integer).
  - `basictype`: an integer value: 0 (undefined), 1 (real), 2 (string).
  - `maxvalue`: 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`: 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`: a real number for the step size in which parameters can be entered or index to a step size in the parameter `parstep` (found in `/vnmr/conpar`). If stepsize is 0, it is ignored. Applies to real types only.
  - `Ggroup`: an integer value: 0 (ALL), 1 (SAMPLE), 2 (ACQUISITION), 3 (PROCESSING), 4 (DISPLAY), 5 (SPIN).
  - `Dgroup`: an integer value. The specific application determines the usage of this integer.
  - `protection`: a 32-bit word made up of the following bit masks, which are summed to form the full mask:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>Cannot array the parameter</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Cannot change active/not active status</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>Cannot change the parameter value</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>Causes _parameter macro to be executed (e.g., if parameter is named <code>sw</code>, the macro <code>_sw</code> is executed when <code>sw</code> is changed)</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>Avoids automatic redisplay</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>Cannot delete parameter</td>
</tr>
</tbody>
</table>
5.4 Creating and Modifying Parameters

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, list 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 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 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 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.

• 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:

1. 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).

2. Parameter a has 1 value, the real number 24.126400.

3. 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 2 1 8202 1 64
(2) 1 1160
(3) 0
```

The tof file is made up of the following lines:
1. 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 stepsizes 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 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).

2. Parameter `tof` has 1 value, the real number 1160.

3. 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
g  e  g  o  r  e
       r  i  n  g  o
(4) 0
```

The `beatles` file is made up of the following lines:

1. 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.

2. There are four elements to this variable; therefore, it is arrayed. `john` is the first element in the array.

3. `paul`, `george`, and `ringo` are the other three elements in the array.

4. `0` (zero) is the terminating line.

### 5.5 Modifying Parameter Displays in VNMR

- “Display Template,” page 294
- “Conditional and Arrayed Displays,” page 296
- “Output Format,” page 297

The VNMR plotting commands and macros—`ap`, `pap`—are controlled by 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.

Enter `paramvi('ap')` to use the `vi` text editor to modify an existing template parameter, such as `ap`, 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 5 shows an example of a single-string display template (actually the parameter ap) and the resulting plot.

```
ap 2 2 1023 0 0 4 1 6 1 64
1
0
```

**Figure 5.** Single-String Display Template with Output

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:

- Column number (e.g., 2)
- Condition for plot of column (optional, e.g., “4 (ni)”, see “Conditional and Arrayed Displays,” page 296).
- Colon
- Column title (e.g., 2D ACQUISITION)
- Colon
- Parameters to appear in column, separated by commas (for notation, see “Conditional and Arrayed Displays,” page 296)
- 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. Since 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 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 6 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,” page 296.

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').

**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.
- sb(sb):3 means display sb only when sb is active (not equal to 'n').
- 4(ni):2D PROCESSING: means display entire “2D PROCESSING” section only when parameter ni is active and non-zero.

Note that 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 “Programming with MAGICAL,” page 27) 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:

\[ \text{aig*}, \text{dcg*}, \text{dmg*} \]

For tabular output of arrayed parameters, square brackets ([...]) are used. For example:

\[ \text{Sample Table Output: [pw,p1,d1,d2]} \]

Notice that all parameters in the column must be in the brackets; thus, the following is illegal:

\[ \text{Sample Table Output: [pw,p1,d1],d2} \]

Since 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:

\[ \text{Sample Table Output: [pw:6:2,p1:6:2,d1:10:6,d2:10:6]} \]

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.

### 5.6 User-Written Weighting Functions

- “Writing a Weighting Function,” page 298
- “Compiling the Weighting Function,” page 299

The parameter wtfile can be set to the name of the file containing a user-written weighting function. If the parameter wtfile (or wtfile1 or wtfile2) does not exist, it can be created with the commands:

\[ \text{create('wtfile','flag')} \]
\[ \text{setgroup('wtfile','processing')} \]
\[ \text{setlimit('wtfile',15,0,0)}. \]
Chapter 5. Parameters and Data

If wtfile exists but wtfile=' ' (two single quotes), VnmrJ does not look for the file: wtfile is inactive. To enable user-written weighting functions, set wtfile=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 there, 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 in 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 parameter awc 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 that 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:

```c
#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 statement *wtpntr++=x 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 wtfile=filename

• Source file filename.c in a user's vnmrsys/wtlib directory:

  ```c
  #include "weight.h"
  wtcalc(wtpntr, npoints, delta_t)
  int npoints;  /* number of complex data points */
  ```
5.6 User-Written Weighting Functions

```c
float *wtpntr, /* pointer to weighting vector */
delta_t;    /* dwell time */

{
    int i;
    for (i = 0; i < npoints; i++)
        *wtpntr++ = (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/shellscript `wtgen` is used to compile `filename` as set by parameter `wtfile` 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 is 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 subsequently can be executed from within VnmrJ. The shellscript `wtgen` can be run from within a shell by typing the name of the shellscript 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`:

1. Checks for the existence of the `bin` subdirectory in the VnmrJ system directory and aborts if the directory is not found.
2. Checks for files `usrwt.o` and `weight.h` in the `bin` subdirectory and aborts if either of these two files cannot be found there.
3. Checks for the existence of the user's directory and creates this directory if it does not already exist.
4. Establishes in the `wtlib` directory soft links to `usrwt.o` and `weight.h` in the directory `/vnmr/bin`.
5. 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.

6. 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.

5.7 User-Written FID Files

User the command makefid(input_file <,element_number,format>) to 
introduce computed data in the experiment. The required input_file argument 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 
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 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 can not 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 since the active value is in 
the processed tree.

The makefid command can modify data returned to an experiment by the rt command. 
To avoid this, 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')

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.
Chapter 6. Panels, Toolbars, and Menus

Sections in this chapter:
• 6.1 “Parameter Panel Features,” page 301
• 6.2 “Using the Panel Editor,” page 301
• 6.3 “Panel Elements,” page 307
• 6.4 “Creating a New Panel,” page 323
• 6.5 “Graphical Toolbar Menus,” page 327

6.1 Parameter Panel Features

The parameter panels in VnmrJ are built using xml files. The 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 experimental, walkup, and LC-NMR interfaces use the name of the pulse sequence, seqfil. Imaging interface panels utilize the parameter layout to select which panels are displayed. The imaging gems protocol sets layout = 'gems'. The parameter can be set to layout = seqfil. Using the layout parameter facilitates sequence development since a panel does not have to be created because a sequence is recompiled under a different name.

6.2 Using the Panel Editor

• “Starting the Panel Editor,” page 301
• “Editing Existing Panel Elements,” page 303
• “Adding and Removing Panel Elements,” page 304
• “Saving Panel Changes,” page 305
• “Exiting the Panel Editor,” page 307

Starting the Panel Editor

1. Click on Edit on the main menu.
2. Select **Parameter Pages**... to display the panel editor window, see Figure 7.

![Figure 7. Panel Editor Window](image)

The current page is displayed in the edit mode with a grid, see Figure 8.

3. Click on **Tools**.

4. Select to **Locator** to display the locator panel, Figure 8, and the basic elements used to build a panel.

![Figure 8. Panel and Locator when Panel Editor is Open](image)
6.2 Using the Panel Editor

Editing Existing Panel Elements
- “Selecting a Panel Element,” page 303
- “Viewing or Changing a Panel Element Attribute,” page 303
- “Changing the Grid Size,” page 304
- “Editing Styles,” page 304
- “Saving Element or Group Changes,” page 305

Selecting a Panel Element
Select a panel element as follows:
1. Double-click on an element (button, toggle, group, etc.) to select it.
   The selected element is highlighted in yellow and is ready for editing.
2. Double-click on an empty area within the group to select a group.
3. Double-click on an empty area within the page to select a page.
4. Double-click on an empty area outside a page to select a folder.
   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 6.3 “Panel Elements,” page 307. 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:
1. Drag the element to the desired location with the mouse.
2. Use the arrow keys to move the element to the new location.
3. 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:
1. Drag the edges of the element with the mouse.
2. Hold the control key down and resize the element using the arrow keys.
3. Enter the size in the **entry boxes** W (Width) and W (Height) in pixels.

**Changing the Grid Size**

The default grid size is 10. The grid size can be changed as follows:

1. Enter a **new value** in the field next to Grid Size, see **Figure 7**.
2. Press **Enter**.

**Editing Styles**

Clicking on 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,” page 304
- “Copying an Element to Another Location on the Same Page,” page 304
- “Copying an Element Between Pages Within a Folder,” page 304
- “Creating a New Page from the Locator,” page 304
- “Removing Panel Items,” page 305

**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:
   - Use the mouse buttons to click on an edge or corner and drag the page to a new size.
   - Use the ctrl-arrow keys to resize the page.
   - 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 desired page in the Locator.
3. Drag the desired page into 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 on it.
2. Click the **Clear** button at the lower right corner of the template editor.
   Removes all items from the page or folder, or deletes the selected item or group (highlighted with yellow border).

**Saving Panel Changes**

- “Saving Element or Group Changes,” page 305
- “Saving Page Changes,” page 306
- “Saving Folder Changes,” page 306

**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.
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.

![Load Button](Load.png)

Loads the element or group using the file name in the element or group entry field.

**Note:** 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

1. Double-click on an empty space within a page, or click on a tab on the left to select a page.

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, Figure 9, for the directory to save the page in.

   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, Figure 10, to set the page type.

   The page type is used for searching for 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

1. Double-click on the area outside a page. Expand the panel display area until larger than the page if no area is available.

   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 from 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 as follows:**
  - Click the **Close** button.

  Closes the panel editor; unsaved changes retained only for the current VnmrJ session. Changes are not saved when VnmrJ is exited.

  Changes are saved and retained only for the current VnmrJ session. Changes are lost when VnmrJ is exited.

- **Exit, 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 on 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. The Save button 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 panel items directory. The page is referenced 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.

  6. Click the **Close** button to exit the panel editor.

### 6.3 Panel Elements

- “Element Style,” page 308
- “Panel Element Attributes,” page 308
Element Style

The font style (plain, bold, italic, bold-italic), size, font, and color that is selected in the Style section at the top of the panel editor window determines 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 42.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label of item</td>
<td>Text label of item.</td>
</tr>
<tr>
<td>Icon of item</td>
<td>Icon of item. This is used only for some elements (button, label).</td>
</tr>
<tr>
<td>Label justification</td>
<td>Justification of label of item. Choices are Left, Right, Center.</td>
</tr>
<tr>
<td>Vnmr variables</td>
<td>VNMR parameters that can change the Value of item, Enable condition, or Show condition of the item.</td>
</tr>
<tr>
<td>Value of item</td>
<td>The value of the item. This string is a MAGICAL expression that sets the value of $VALUE. The value of some items (checkbox, radio, toggle) can be either true (1) or false (0). Other items (comboboxbutton, menu, selmenu) match a value from the Value of choices. For still other items (entry, textmessage) it is a number or string to display.</td>
</tr>
<tr>
<td>Decimal Places</td>
<td>The number of decimal places to truncate to in a real expression in Value of item.</td>
</tr>
<tr>
<td>Vnmr command</td>
<td>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.</td>
</tr>
<tr>
<td>Vnmr command2</td>
<td>The command sent when the item is deselected. This is used only by some items (checkbox, radio, toggle).</td>
</tr>
<tr>
<td>Enable condition</td>
<td>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.</td>
</tr>
<tr>
<td>Label of choices</td>
<td>Text labels used in a menu or comboboxbutton.</td>
</tr>
<tr>
<td>Value of choices</td>
<td>Values in a menu or comboboxbutton used to set the Vnmr command.</td>
</tr>
<tr>
<td>Status parameter</td>
<td>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 Infostat.</td>
</tr>
<tr>
<td>Show condition</td>
<td>The expression which 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).</td>
</tr>
</tbody>
</table>
6.3 Panel Elements

Table 42. Common Attributes of Panel Elements

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vnmr command on show</td>
<td>In a group, the command sent when the group is shown. This string is a MAGICAL expression.</td>
</tr>
<tr>
<td>Vnmr command on hide</td>
<td>The command sent when the group is hidden. This string is a MAGICAL expression.</td>
</tr>
<tr>
<td>Editable</td>
<td>Sets whether or not text may be entered in the item (yes or no).</td>
</tr>
</tbody>
</table>

Panel Elements
• “Basic Panel Elements,” page 309
• “Advanced Panel Elements,” page 319

Basic Panel Elements
• “Button,” page 309
• “Check,” page 310
• “Group,” page 311
• “Menu,” page 313
• “Radio button,” page 315
• “Selmenu,” page 316
• “Spinner,” page 317
• “Toggle button,” page 318
• “Comboboxbutton,” page 310
• “Entry field,” page 311
• “Label,” page 313
• “Parameter,” page 314
• “Scroll,” page 316
• “Slider,” page 316
• “Textmessage,” page 318

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
- Vnmr variables—en-/disable a button based on the parameter value.
- Status parameter—en-/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.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label of item</td>
<td>Acquire Profile</td>
</tr>
<tr>
<td>Icon of item</td>
<td></td>
</tr>
<tr>
<td>Enable condition</td>
<td></td>
</tr>
<tr>
<td>Vnmr command</td>
<td>au</td>
</tr>
<tr>
<td>Vnmr variables</td>
<td></td>
</tr>
<tr>
<td>Status variables</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 6. Panels, Toolbars, and Menus

The check box element selects and de-selects 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 \( \$\text{VALUE} \) evaluates to a positive integer.
- **Enable condition**
- **Vnmr command**
- **Vnmr command2**

Inversion Recovery is a check box example:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label of item</td>
<td>Inversion Recovery</td>
</tr>
<tr>
<td>Vnmr variables:</td>
<td>ir</td>
</tr>
<tr>
<td>Value of element:</td>
<td>( $\text{VALUE} = (\text{ir}='y') )</td>
</tr>
<tr>
<td>Vnmr command:</td>
<td>\text{ir}='y'</td>
</tr>
<tr>
<td>Vnmr command2:</td>
<td>\text{ir}='n'</td>
</tr>
</tbody>
</table>

The commands in the **Vnmr command** and **Vnmr command2** fields are executed when the check box is selected or deselected. The parameter \( \text{ir} \) is set to \( y \) when the box is selected, and when the box is deselected, \( \text{ir} \) is set to \( n \).

The **Value of element** field determines, based on the current value of \( \text{ir} \), whether the check box is shown as selected or deselected. Thus, this element needs to "listen to" the parameter \( \text{ir} \), which requires \( \text{ir} \) 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. Clicking on the button executes the Vnmr command specified in the menu.

The attributes of a comboboxbutton are

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label of item</td>
<td>Acquire Profile</td>
</tr>
<tr>
<td>Vnmr variables</td>
<td>transparent</td>
</tr>
</tbody>
</table>

An example is a **comboboxbutton** that displays the number of complex transform points \( \text{fn}/2 \):

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vnmr variables</td>
<td>( \text{fn} )</td>
</tr>
<tr>
<td>Value of item</td>
<td>( $\text{VALUE} = \text{fn}/2 )</td>
</tr>
<tr>
<td>Enable condition</td>
<td>\text{on('fn'):}$\text{ENABLE}</td>
</tr>
<tr>
<td>Vnmr command</td>
<td>( \text{fn} = $\text{VALUE} * 2 )</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vnmr Variables:</td>
<td>( nt )</td>
</tr>
<tr>
<td>Value of item:</td>
<td>$VALUE = nt</td>
</tr>
<tr>
<td>Enable condition</td>
<td></td>
</tr>
<tr>
<td>Vnmr Command:</td>
<td>( nt = $VALUE )</td>
</tr>
<tr>
<td>Decimal places:</td>
<td></td>
</tr>
<tr>
<td>Disable style:</td>
<td></td>
</tr>
<tr>
<td>Status parameter:</td>
<td></td>
</tr>
</tbody>
</table>

The entry field created in the above example functions as follows after exiting the editor:

- Enter a value into the entry field, e.g., 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:

\[
\text{Vnmr Command: } te = \$VALUE/1000
\]

Entering a list of values for \( te \) in this case, e.g., 10, 20, 30, 40, results in dividing only the last value by 1000 and \( te \) 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.

\[
\text{Vnmr Command: } te = \left[\$VALUE\right]/1000
\]

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, checkboxes, etc.

Groups are used to delineate a collection of basic elements that are connected. There are three types of groups: Major, Minor, and Convenience.
Group attributes are:

- **Label of item**
- **Show condition**
- **Vnmr Command on Hide**
- **Number of Layers**
- **Background Color**
- **Override Panel Enabled**
- **Vnmr variables**
- **Vnmr Command on Show**
- **Type**
- **Edit Layer**
- **Tab to this Group Disabled**

Major groups are outlined with a visible border and can have a label associated at the top of the group. The major group width is restricted to a multiple of 70 pixels, and the group has an automatic margin of 5 pixels inside all edges. Major groups cannot contain major groups.

Minor groups only appear inside major groups, cannot be nested, or contain major groups. The left alignment and width snaps to the grid size. A label given to a minor group is not displayed.

Convenience groups are simply used to make editing pages easier and have no visible effect on the page. They can be used anywhere and have no restrictions on alignment. They are independent of the hierarchical restrictions placed on other types of groups.

Major and minor groups may also be mutable. Their contents can change depending on other parameters. Multiple layers are available. Set the number of layers greater than one to enable this property. Use the editor to select the current active for editing. Mutable groups have a distinctive look.

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 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. Convenience groups can be nested and "hidden" within each other, if desired.
Groups can have multiple layers, hidden 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.

**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.

Example: **Transform size** in front of an entry field for entering the number of transformed points.

The attributes of a label are:

- **Label of item**
- **Icon of item**
- **Vnmr variables**
- **Enable condition**
- **Label justification**

An example is:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label of item</td>
<td>Transform size</td>
</tr>
<tr>
<td>Icon of item</td>
<td></td>
</tr>
<tr>
<td>Vnmr variables</td>
<td></td>
</tr>
<tr>
<td>Enable condition</td>
<td></td>
</tr>
<tr>
<td>Label justification</td>
<td>Left</td>
</tr>
</tbody>
</table>

**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**
- **Editable**

The menu for np where the value displayed is the number of complex pairs, i.e., np/2 is an example:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vnmr variables</td>
<td>np</td>
</tr>
<tr>
<td>Value of element</td>
<td>$\text{VALUE} = \frac{\text{np}}{2}$</td>
</tr>
<tr>
<td>Vnmr command</td>
<td>np = $\text{VALUE} \times 2$</td>
</tr>
<tr>
<td>Label of choices</td>
<td>&quot;32&quot; &quot;64&quot; &quot;128&quot; &quot;256&quot; &quot;512&quot; &quot;1024&quot;</td>
</tr>
<tr>
<td>Value of choices</td>
<td>&quot;32&quot; &quot;64&quot; &quot;128&quot; &quot;256&quot; &quot;512&quot; &quot;1024&quot;</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value of element</td>
<td>$VALUE = np</td>
</tr>
<tr>
<td>Vnmr command</td>
<td>np = $VALUE</td>
</tr>
<tr>
<td>Label of choices</td>
<td>&quot;32&quot; &quot;64&quot; &quot;128&quot; &quot;256&quot; &quot;512&quot; &quot;1024&quot;</td>
</tr>
<tr>
<td>Value of choices</td>
<td>&quot;64&quot; &quot;128&quot; &quot;256&quot; &quot;512&quot; &quot;1024&quot; &quot;2048&quot;</td>
</tr>
</tbody>
</table>

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 checkbox, 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:

<table>
<thead>
<tr>
<th>Parameter element function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>Style permits changing font and units.</td>
</tr>
<tr>
<td>Check box</td>
<td>Enables or disables selected conditions.</td>
</tr>
<tr>
<td>Entry field</td>
<td>Enter a value with optional decimal places</td>
</tr>
<tr>
<td>Units</td>
<td>Label for parameter units. Selected from a menu is optional.</td>
</tr>
</tbody>
</table>

Entry Size and Unit Size establish the size of the label box, and Units Label adds the desired units description at the end of the box.

The label and menu have the same font.

The following example uses fixed units. Type Label.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter Name</td>
<td>temp</td>
</tr>
<tr>
<td>Label of item</td>
<td>Temperature</td>
</tr>
<tr>
<td>Enable Condition</td>
<td>vnmrinfo('get','tempExpControl'):$tc=0 then $ENABLE=-1 else on ('temp'):$ENABLE endif</td>
</tr>
<tr>
<td>Vnmr variables</td>
<td>temp tin</td>
</tr>
<tr>
<td>Checkbox Enable</td>
<td>vnmrinfo('get','tempExpControl'):$tc $ENABLE=$tc*2-1</td>
</tr>
<tr>
<td>Checkbox Value:</td>
<td>on ('temp'):$VALUE</td>
</tr>
<tr>
<td>Checkbox Vnmr Command:</td>
<td>on ('temp') tin=Y</td>
</tr>
<tr>
<td>Checkbox Vnmr Command2:</td>
<td>off('temp') tin='n'</td>
</tr>
<tr>
<td>Entry value:</td>
<td>$VALUE=temp</td>
</tr>
<tr>
<td>Entry size:</td>
<td>80</td>
</tr>
<tr>
<td>Entry Vnmr Command:</td>
<td>temp=$VALUE tin=Y</td>
</tr>
<tr>
<td>Entry Decimal Places:</td>
<td>1</td>
</tr>
<tr>
<td>Entry Disable Styles:</td>
<td>Grayed out</td>
</tr>
</tbody>
</table>
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**
- **Value of element**
- **Vnmr variables**
- **Enable condition**
- **Vnmr command**
- **Vnmr command2**
Chapter 6. Panels, Toolbars, and Menus

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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Chess</th>
<th>WET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vnmr variables</td>
<td>wss</td>
<td>wss</td>
</tr>
<tr>
<td>Value of element</td>
<td>$VALUE=(wss='chess')</td>
<td>$VALUE=(wss='wet')</td>
</tr>
<tr>
<td>Enable condition</td>
<td>$VALUE=(ws='y')</td>
<td>$VALUE=(ws='y')</td>
</tr>
<tr>
<td>Vnmr command</td>
<td>wss='chess'</td>
<td>wss='wet'</td>
</tr>
</tbody>
</table>

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. The “Spinner,” page 317, 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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vnmr variables</td>
<td>dmm</td>
</tr>
<tr>
<td>Value of element</td>
<td>$VALUE = dmm</td>
</tr>
<tr>
<td>Enable condition</td>
<td>dmm = '$VALUE'</td>
</tr>
<tr>
<td>Vnmr command</td>
<td>&quot;ccc&quot; &quot;ccw&quot; &quot;ccg&quot;</td>
</tr>
</tbody>
</table>

**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.

**Slider**

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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vnmr variables</td>
<td>Value of element</td>
</tr>
<tr>
<td>Enable condition</td>
<td>Vnmr command</td>
</tr>
<tr>
<td>Vnmr command</td>
<td><code>setshim ('Z0', $VALUE)</code></td>
</tr>
<tr>
<td>Status parameter</td>
<td>20</td>
</tr>
<tr>
<td>Limits parameter</td>
<td>20</td>
</tr>
<tr>
<td>Min displayed value</td>
<td></td>
</tr>
<tr>
<td>Max displayed value</td>
<td></td>
</tr>
<tr>
<td>Coarse adjustment value</td>
<td>10</td>
</tr>
<tr>
<td>Fine adjustment value</td>
<td>1</td>
</tr>
<tr>
<td>Number of digits to display</td>
<td>6</td>
</tr>
<tr>
<td>Ms between updates while dragging</td>
<td>0</td>
</tr>
</tbody>
</table>

**Spinner**

The spinner element applies a defined step size 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. The “Scroll,” page 316, is similar but does not require a defined step size.

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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vnmr variables</td>
<td>vtairflow</td>
</tr>
<tr>
<td>Value of item</td>
<td>$VALUE = vtairflow</td>
</tr>
<tr>
<td>Enable condition</td>
<td>$SHOW = (vtairflow&gt;6)</td>
</tr>
<tr>
<td>Vnmr command</td>
<td></td>
</tr>
<tr>
<td>Status parameter</td>
<td>Vnmr Variables</td>
</tr>
<tr>
<td>Limits parameter</td>
<td></td>
</tr>
<tr>
<td>Min displayed value</td>
<td>7.0</td>
</tr>
<tr>
<td>Max displayed value</td>
<td>25.0</td>
</tr>
<tr>
<td>Mouse click adjustment</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Textmessage

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status parameter to display</td>
<td></td>
</tr>
<tr>
<td>Vnmr Variables</td>
<td>np</td>
</tr>
<tr>
<td>Enable condition</td>
<td></td>
</tr>
<tr>
<td>Vnmr expression to display</td>
<td>$VALUE = np/2</td>
</tr>
<tr>
<td>Number of digits</td>
<td>0</td>
</tr>
</tbody>
</table>

Toggle button

A toggle button is used to execute one action when the button is selected and another action when the button is de-selected. 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. To 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)
6.3 Panel Elements

Example:

<table>
<thead>
<tr>
<th>Label of item</th>
<th>Vnmr variables</th>
<th>Value of item</th>
<th>Enable condition</th>
<th>Vnmr command</th>
<th>Vnmr command2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status variables</td>
<td>Selecting status values</td>
<td>Enabling status values</td>
<td>Button 1</td>
<td>Button 2</td>
<td></td>
</tr>
</tbody>
</table>

Example:

<table>
<thead>
<tr>
<th>Label of item</th>
<th>Vnmr variables</th>
<th>Value of item</th>
<th>Enable condition</th>
<th>Vnmr command</th>
<th>Vnmr command2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status variables</td>
<td>Selecting status values</td>
<td>Enabling status values</td>
<td>Button 1</td>
<td>Button 2</td>
<td></td>
</tr>
</tbody>
</table>

Advanced Panel Elements

- “Dial,” page 319
- “Filemenu,” page 320
- “Page,” page 320
- “Shimbutton,” page 321
- “Selfilemenu,” page 321
- “Shimset,” page 322
- “Statusbutton,” page 322
- “Textfile,” page 323

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 can not be set using the dial.

The attributes of a dial are:

- Vnmr variables
- Value of item
- Status variable
- Max value
- Max value elastic
- Number of hands
- Digital readout
- Show max value
- Max marker color
- Show pic slice
- Show color bars

Example:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vnmr variables</td>
<td>fidarea</td>
</tr>
<tr>
<td>Value of item</td>
<td>$VALUE = fidarea</td>
</tr>
<tr>
<td>Enable condition</td>
<td>Status variable</td>
</tr>
<tr>
<td>Min value</td>
<td>0.0</td>
</tr>
<tr>
<td>Max value</td>
<td>1000.0</td>
</tr>
</tbody>
</table>
Chapter 6. Panels, Toolbars, and Menus

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, where 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:

<table>
<thead>
<tr>
<th>Label of item</th>
<th>Selection variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content variables</td>
<td>Value of item</td>
</tr>
<tr>
<td>Enable condition</td>
<td>Vnmr command</td>
</tr>
<tr>
<td>Menu source</td>
<td>Menu type</td>
</tr>
<tr>
<td>Show dot files</td>
<td></td>
</tr>
</tbody>
</table>

An example of a filemenu is the orientation menu in Plan page in the Imaging interface, where the orientation of previously acquired data is dynamically added to the orientation menu during a study:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label of item</td>
<td>orient planValue</td>
</tr>
<tr>
<td>Selection variables</td>
<td>sqdir studyid</td>
</tr>
<tr>
<td>Content variables</td>
<td>$VALUE = planValue</td>
</tr>
<tr>
<td>Enable condition</td>
<td></td>
</tr>
<tr>
<td>Vnmr command</td>
<td>iplanType = 0 planValue='$VALUE' setgplan('$VALUE')</td>
</tr>
<tr>
<td>Menu source</td>
<td>$VALUE=sqdir+ '/plans'</td>
</tr>
<tr>
<td>Menu type</td>
<td>file</td>
</tr>
<tr>
<td>Show dot files</td>
<td>yes</td>
</tr>
</tbody>
</table>

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       |
6.3 Panel Elements

Selfilemenu

The selfilemenu 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 selfilemenu is that the selfilemenu always displays the same text (the Label), regardless of what was last selected. The exception to this is if the selfilemenu is "Editable", in which case it displays the last selected option.

The attributes of a selfilemenu are the same as for a filemenu.

Shimbutton

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 step size. 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 on the value with the left mouse button. The step size can be changed by clicking the middle mouse button (goes through 3 values), or a new step size 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:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vnmr variables</td>
<td>Z0</td>
</tr>
<tr>
<td>Value of item</td>
<td>Vnmr command</td>
</tr>
<tr>
<td>Label</td>
<td>setshim ('Z0', $VALUE)</td>
</tr>
<tr>
<td>Vnmr command</td>
<td>Z0</td>
</tr>
<tr>
<td>Status parameter</td>
<td>Z0</td>
</tr>
<tr>
<td>Limits parameter</td>
<td>Z0</td>
</tr>
<tr>
<td>Min allowed value</td>
<td>False</td>
</tr>
<tr>
<td>Max allowed value</td>
<td>True</td>
</tr>
<tr>
<td>Pointy style</td>
<td>False</td>
</tr>
<tr>
<td>Rocker style</td>
<td>True</td>
</tr>
<tr>
<td>Arrow feedback</td>
<td>True</td>
</tr>
<tr>
<td>Arrow color</td>
<td>GraphForeground</td>
</tr>
<tr>
<td>Values wrap around</td>
<td>False</td>
</tr>
</tbody>
</table>
Shimset

The shimset element brings up an entire set of shimbuttons, corresponding to the shim hardware.

The attributes of a shimset are:

- Border type
- Vnmr shim set parameter
- Shim setting command
- Vnmr variables for shim
- Freeze layout
- Vnmr shim set value
- Status parameter for shim
- Vnmr expression for shim

Statusbutton

A status button brings up 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.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Title</td>
<td>FID Shim</td>
</tr>
<tr>
<td>Status Color</td>
<td>$fg$</td>
</tr>
<tr>
<td>Status Variable</td>
<td>$fidarea$</td>
</tr>
<tr>
<td>Vnmr variables</td>
<td>$VALUE=fidarea$</td>
</tr>
<tr>
<td>Value of item</td>
<td>$VALUE=fidarea$</td>
</tr>
<tr>
<td>Vnmr command</td>
<td>$fid_scan$</td>
</tr>
<tr>
<td>Vnmr command2</td>
<td>aa('exit FID shim')</td>
</tr>
</tbody>
</table>
6.4 Creating a New Panel

- “Writing Commands,” page 323
- “Creating a New Panel Layout,” page 324
- “Creating a New Page,” page 324
- “Defining and Populating a Page,” page 325
- “Saving and Retrieving a Panel Element,” page 325
- “Files Associated with Panels,” page 326

Writing Commands

The panel editor uses the MAGICAL command syntax and a special variable, $VALUE, that 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 42, page 308.

<table>
<thead>
<tr>
<th>Index</th>
<th>freq(gHz)</th>
<th>intensity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>153.843</td>
<td>16.134</td>
</tr>
<tr>
<td>2</td>
<td>134.517</td>
<td>40.552</td>
</tr>
<tr>
<td>3</td>
<td>127.254</td>
<td>38.1514</td>
</tr>
<tr>
<td>4</td>
<td>120.546</td>
<td>29.1125</td>
</tr>
<tr>
<td>5</td>
<td>113.534</td>
<td>36.3057</td>
</tr>
<tr>
<td>6</td>
<td>103.459</td>
<td>55.9474</td>
</tr>
<tr>
<td>7</td>
<td>77.0098</td>
<td>99.4765</td>
</tr>
<tr>
<td>8</td>
<td>76.5847</td>
<td>50.5</td>
</tr>
<tr>
<td>9</td>
<td>68.7526</td>
<td>41.5768</td>
</tr>
<tr>
<td>10</td>
<td>61.3561</td>
<td>91.537</td>
</tr>
<tr>
<td>11</td>
<td>54.4663</td>
<td>30.4191</td>
</tr>
<tr>
<td>12</td>
<td>11.6124</td>
<td>33.1797</td>
</tr>
</tbody>
</table>

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 Layout

A new layout is created using either a blank panel or an existing layout that is similar to the desired 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

1. Load the protocol, pulse sequence, and or parameter set that will use the new layout.
2. Set seqfil or layout='mygems'.
   The current panels are edited if seqfil or layout is not set to the new name.
3. Open the panel editor:
   a. Click on Edit on the main menu.
   b. Select Parameter Pages.
4. Modify any page.
5. Double click within a page or select the tab to the left.
   The selected page border is highlighted in yellow.
6. Click the Save button and save the page.
   Save the entire folder if new pages were created:
   a. 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 button Save is followed by the word Folder and an entry field.
   b. Click on 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 6.2 “Using the Panel Editor,” page 301 for details.

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.
   - Use the ctrl-arrow keys to resize the page.
   - Type in values for size W(idth) and H(eight).

### 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.

   ![Type dropdown menu](image)

   Undo any changes made since the most recent save by clicking on 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 on 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 on 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 desired 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 43 for panels and locations.

<table>
<thead>
<tr>
<th>Panel Owner</th>
<th>Panel</th>
<th>Location</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>VnmrJ</td>
<td>Experimental</td>
<td>/vnmr/templates/layout</td>
<td>Named according to the pulse sequence name and used by all interfaces.</td>
</tr>
<tr>
<td>Walkup</td>
<td>/vnmr/walkup/templates/layout</td>
<td>Walkup panels not shared with the experimental interface.</td>
<td></td>
</tr>
<tr>
<td>User</td>
<td>user defined</td>
<td>~/.user/vnmrsys/templates/layout</td>
<td>User panels named according to the pulse sequence name or layout.</td>
</tr>
</tbody>
</table>

The panel search path is defined in Applications dialog 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 example:

COSY panels in the walkup interface, with a DEFAULT file of set default default2d.

1. ~/.vnmr/templates/layout/COSY
2. /vnmr/walkup/templates/layout/COSY
3. /vnmr/templates/layout/COSY
4. ~/.vnmr/templates/layout/default2d
5. /vnmr/walkup/templates/layout/default2d
6. /vnmr/templates/layout/default2d
7. ~/.vnmr/templates/layout/default
8. /vnmr/walkup/templates/layout/default
9. /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. Login as the system administrator, typically vnmr1.
2. Open a terminal window.
3. Enter cd /vnmr/bin
4. Enter cp vnmrj vnmrj.backup
5. 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.
6. Save the change and exit the editor.
7. Restart VnmrJ to make VnmrJ read the new value of the parameter.

6.5 Graphical Toolbar Menus

- “Editing the Toolbar Menu,” page 327
- “Graphics Toolbar Parameters,” page 328
- “Icons,” page 328
- “Menu File Description Example, deoni,” page 328

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/vnmsys
- any appdir accessible path.

Menus and toolbars are a special form of 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:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>micon</td>
<td>Saves the name of the GIF file associated with the button in micon[i]. This parameter is typically arrayed with one icon for each button and is set when a menu is called. A noicon.gif is used if an icon does not exist.</td>
</tr>
<tr>
<td>mlabel</td>
<td>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. mlabel[i] contains the tooltip for the i-th button.</td>
</tr>
<tr>
<td>mstring</td>
<td>Stores command or macro strings to be executed when a VnmrJ menu button is clicked. Usually the mstring parameter is arrayed, with one string for each button in the menu. The string can be any string of commands that can otherwise 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.</td>
</tr>
</tbody>
</table>

Icons

VnmrJ icons available to all users are .gif files located in: /vnmr/iconlib
Size all button icons to 24x24 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 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 (ermode='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[$vjm] = '2D1cur.gif') when the cursor operation is in the box mode (ermode='b').
Clicking on the button changes the button to Box (mlabel[$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 optional displayed depending upon 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.
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
6.5 Graphical Toolbar Menus

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 mfShowAxis=1 endif repaint'
57  micon[$vjm] = '1Dscale.gif'
58
59  $vjm=$vjm+1
60  mlabel[$vjm] = 'Projections'
61  mstring[$vjm] = 'newmenu('dconi_proj') dconi('restart')'
62  micon[$vjm] = '2Dvhproj.gif'
63
64  $vjm=$vjm+1
65  mlabel[$vjm] = 'Redraw'
66  mstring[$vjm] = 'dconi('again')'
67  micon[$vjm] = 'recycle.gif'
68
69  $vjm=$vjm+1
70  mlabel[$vjm] = 'Rotate'
71  mstring[$vjm] = 'if trace='f2' then trace='f1' else trace='f2' endif dconi('again')'
72  micon[$vjm] = '2Drotate.gif'
73
74  if appmode='imaging' then
75
76  $vjm=$vjm+1
77  mlabel[$vjm] = 'Scale +7%'
78  mstring[$vjm] = 'vs2d=vs2d*1.07 dconi('redisplay')'
79  micon[$vjm] = '2Dvs+20.gif'
80
81  $vjm=$vjm+1
82  mlabel[$vjm] = 'Scale -7%'
83  mstring[$vjm] = 'vs2d=vs2d/1.07 dconi('redisplay')'
micon[$vjm] = '2Dvs-20.gif'

else

$vjm=$vjm+1

mlabel[$vjm] = 'Scale +20%'
mstring[$vjm] = 'vs2d=vs2d*1.2 dconi('again')'

micon[$vjm] = '2Dvs+20.gif'

$vjm=$vjm+1

mlabel[$vjm] = 'Scale -20%'
mstring[$vjm] = 'vs2d=vs2d/1.2 dconi('again')'
micon[$vjm] = '2Dvs-20.gif'

endif

$vjm=$vjm+1

mlabel[$vjm] = 'Phase2D'
mstring[$vjm] = 'newmenu('dconi_phase') dconi('trace')'
micon[$vjm] = '1Dphase.gif'

if appmode<>'imaging' then

$vjm=$vjm+1

mlabel[$vjm] = 'Peak Picking'
mstring[$vjm] = 'newmenu('ll2d') dconi('restart')'
micon[$vjm] = '2Dpeakmainmenu.gif'
endif

$vjm=$vjm+1

mlabel[$vjm] = 'Return'
micon[$vjm] = 'return.gif'

if (lastmenu<>'') then

mstring[$vjm] = 'menu(lastmenu) lastmenu='''
else

if appmode='imaging' then

mstring[$vjm] = 'menu('main')'
else

mstring[$vjm] = 'menu('display_2D')'
endif
endif
Appendix A. Status Codes

Codes marked with an asterisk (*) are not used on MERCURYplus/-Vx systems. Codes marked with a double asterisk (**) apply only to Whole Body Imaging systems.

Table 44. Acquisition Status Codes

<table>
<thead>
<tr>
<th>Done codes:</th>
<th>11. FID complete</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>12. Block size complete (error code indicates bs number completed)</td>
</tr>
<tr>
<td></td>
<td>13. Soft error</td>
</tr>
<tr>
<td></td>
<td>14. Warning</td>
</tr>
<tr>
<td></td>
<td>15. Hard error</td>
</tr>
<tr>
<td></td>
<td>16. Experiment aborted</td>
</tr>
<tr>
<td></td>
<td>17. Setup completed (error code indicates type of setup completed)</td>
</tr>
<tr>
<td>101. Experiment complete</td>
<td></td>
</tr>
<tr>
<td>102. Experiment started</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Error codes:</th>
<th>Warnings</th>
</tr>
</thead>
<tbody>
<tr>
<td>101. Low-noise signal</td>
<td></td>
</tr>
<tr>
<td>102. High-noise signal</td>
<td></td>
</tr>
<tr>
<td>103. ADC overflow occurred</td>
<td></td>
</tr>
<tr>
<td>104. Receiver overflow occurred*</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Soft errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>200. Maximum transient completed for single precision data</td>
</tr>
<tr>
<td>201. Lost lock during experiment (LOCKLOST)</td>
</tr>
<tr>
<td>300. Spinner errors:</td>
</tr>
<tr>
<td>301. Sample fails to spin after 3 attempts to reposition (BUMPFAIL)</td>
</tr>
<tr>
<td>302. Spinner did not regulate in the allowed time period (RSPINFAIL)*</td>
</tr>
<tr>
<td>303. Spinner went out of regulation during experiment (SPINOUT)*</td>
</tr>
<tr>
<td>395. Unknown spinner device specified (SPINUNKNOWN)*</td>
</tr>
<tr>
<td>396. Spinner device is not powered up (SPINNOPOWER)*</td>
</tr>
<tr>
<td>397. RS-232 cable not connected from console to spinner (SPINRS232)*</td>
</tr>
<tr>
<td>398. Spinner does not acknowledge commands (SPINTIMEOUT)*</td>
</tr>
<tr>
<td>400. VT (variable temperature) errors:</td>
</tr>
<tr>
<td>400. VT did not regulate in the given time VTtime after being set</td>
</tr>
<tr>
<td>401. VT went out of regulation during the experiment (VTOUT)</td>
</tr>
<tr>
<td>402. VT in manual mode after auto command (see Oxford manual)*</td>
</tr>
<tr>
<td>403. VT safety sensor has reached limit (see Oxford manual)*</td>
</tr>
<tr>
<td>404. VT cannot turn on cooling gas (see Oxford manual)*</td>
</tr>
<tr>
<td>405. VT main sensor on bottom limit (see Oxford manual)*</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>406. VT main sensor on top limit (see Oxford manual)*</td>
</tr>
<tr>
<td>408. VT oc/ss error (see Oxford manual)*</td>
</tr>
<tr>
<td>496. VT device not powered up (VTONPOWER)*</td>
</tr>
<tr>
<td>498. VT does not acknowledge commands (VTTIMEOUT)</td>
</tr>
<tr>
<td>501. Sample changer has no sample to retrieve</td>
</tr>
<tr>
<td>503. Sample changer arm unable to move down during retrieve</td>
</tr>
<tr>
<td>505. Invalid sample number during retrieve</td>
</tr>
<tr>
<td>507. Gripper abort during retrieve</td>
</tr>
<tr>
<td>509. Illegal command character during retrieve*</td>
</tr>
<tr>
<td>511. Sample tray size is not consistent*</td>
</tr>
<tr>
<td>513. Illegal sample changer command during retrieve*</td>
</tr>
<tr>
<td>515. Air supply to sample changer failed during retrieve*</td>
</tr>
<tr>
<td>526. Invalid temperature during sample changer insert*</td>
</tr>
<tr>
<td>528. Sample out of range during automatic insert</td>
</tr>
<tr>
<td>530. Robot arm failed to find home position during insert*</td>
</tr>
<tr>
<td>532. Sample changer power failure during insert*</td>
</tr>
<tr>
<td>534. Gripper failed to open during insert*</td>
</tr>
<tr>
<td>593. Failed to remove sample from magnet*</td>
</tr>
<tr>
<td>595. Sample failed to insert properly</td>
</tr>
<tr>
<td>597. Sample changer not connected to RS-232 interface</td>
</tr>
<tr>
<td><strong>600. Shimming errors:</strong></td>
</tr>
<tr>
<td>601. Shimming user aborted*</td>
</tr>
<tr>
<td>604. Lock saturation while shimming*</td>
</tr>
</tbody>
</table>
Table 44. Acquisition Status Codes (continued)

608. A shim coil DAC limit hit while shimming*

700. Autolock errors:
701. User aborted (ALKABORT)*
702. Autolock failure in finding resonance of sample (ALKRESFAIL)
703. Autolock failure in lock power adjustment (ALKPOWERFAIL)*
704. Autolock failure in lock phase adjustment (ALKPHASFAIL)*
705. Autolock failure, lost in final gain adjustment (ALKGAINFAIL)*

800. Autogain errors.
801. Autogain failure, gain driven to 0, reduce p\text{w} (AGAINFAIL)

Hard errors
901. Incorrect PSG version for acquisition
902. Sum-to-memory error, number of points acquired not equal to \( n_p \)
903. FIFO underflow error (a delay too small?)*
904. Requested number of data points (\( n_p \)) too large for acquisition*
905. Acquisition bus trap (experiment may be lost)*

1000. SCSI errors:
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*

1100. Host disk errors:
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*

1400–1500. RF Monitor errors:
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 **
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 **
Table 44. Acquisition Status Codes (continued)

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 **
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 **
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 **
Table 44. Acquisition Status Codes (continued)

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 **
1559. Channel 2 RF Power Monitoring BYPASSED **
1560. Channel 2 RF Amp errors BYPASSED **
## Index

### Symbols

- "..." (double quotes) notation, 24, 30
- # notation (pulse shaping file), 109
- $ (dollar sign) notation, 28, 32
- $# special input argument, 35
- $0 special input argument, 35
- $1, $2,... input arguments, 35
- $VALUE, 323
- & (ampersand) notation (UNIX), 269
- '...' (single quotes) notation, 25, 28
- (...) (parentheses) notation, 34
- (...)# notation (AP table file), 84
- * (asterisk) notation (display template), 297
- += notation (AP table file), 84
- . (single period) notation (UNIX), 268
- .. (double period) notation (UNIX), 268
- .c file extension, 55
- .fd file extension, 281
- / notation (UNIX), 268
- . (colon) notation, 26
- .(semicolon) notation (UNIX), 268
- ; (semicolon) notation, 59
- ; (semicolon) notation (UNIX), 268
- < notation (UNIX), 269
- <...> (angled brackets) notation, 25
- > notation (UNIX), 269
- >> notation (UNIX), 269
- ? (question mark) notation (UNIX), 269
- [...] notation (display template file), 297
- [...] notation (square brackets), 32
- [...]# notation (AP table file), 84
- \ (backslash) notation, 28
- \ (S) notation (UNIX), 268
- \ (backslash single quote) notation, 328
- x macro name, 25
- ‘...’ (reverse single quotes) notation, 328
- {...} (curly braces) notation, 36, 59
- {...}# notation (AP table file), 84
- | (vertical bar) notation (UNIX), 269
- ~ (tilde) notation (UNIX), 268

### Numerics

- 1D data file, 275
- 1D display, 278
- 1D Fourier transform, 278
- 2D data file, 279
- 2D FID display, 279
- 2D FID storage, 279
- 2D hypercomplex data, 275
- 2D phased data storage, 279
- 2D plane of a 3D data set, 40
- 2D plane selection without display, 40
- 2D pulse sequence in standard form, creating, 123
- 2D, 3D, and 4D data sets, 122
- 3D coefficient text file, 274
- 3D parameter set, 274
- 3D pulse sequence in standard form, creating, 123
- 3D spectral data default directory, 274
- 4D pulse sequence in standard form, creating, 123
- 63-dB attenuator, 71, 116
- 79-dB attenuator, 71, 116

### A

- abort command, 38
- abort current process (UNIX), 269
- abortoff command, 38
- aborton command, 38
- abs command, 43
- abs macro, 38
- A-codes, 80
- acos command, 43
- acq_errors file, 60
- acqi command, 46, 99, 102
- Acqstat command, 46
- acqstatus parameter, 60
- acquire data explicitly, 140
- acquire data points, 105
- acquire statement, 105, 106, 119
- acquisition bus trap, 335
- Acquisition codes, 80
- Acquisition Controller boards, 141
- acquisition CPU, 121
- acquisition phase (AP) tables. See AP table
- acquisition processor memory, 146
- acquisition statements, 60
- acquisition status codes, 60
- acquisition time, 89
- Acquisition window, 99, 102
- active parameter test, 48
- ADC overflow warning, 333
- add AP table to second AP table, 253
- add integer to AP table, 252
- add integer values, 141
- add statement, 78, 141
- alfa parameter, 60
- alias (UNIX), 268
- amplifiers
  - blanking channels, 145
  - duty cycle, 63
  - gating, 62
  - turn off, 145
  - turn on, 145
- ampmode parameter, 63
- analyze command, 42
- analyze.inp file, 42
- and operator, 30
- angled brackets (< or >) notation, 21, 25
- AP bus commands, 72
- AP bus delay, 121, 141
- AP bus delay constants, 117
- AP bus instruction, 119
- AP bus pulse shaping, 142, 143
- AP bus registers, 76, 226, 234, 263
- ap command, 294
- ap parameter, 294, 297
- AP table, 83
  - add integer to elements, 252
  - add to another table, 253
  - autoincrement attribute, 86, 232
  - divide by second AP table, 254
  - divide integer into elements, 252
  - divn-factor, 86
  - file location, 83
Index

load from file, 85
loading statements, 83
multiply by a second AP table, 254
multiply integer with elements, 252
receiver phase cycle, 232
receiver variable, 86
retrieve element, 86
scalar operations, 86
set divn-return and divn-factor, 232
store integer array, 85, 233
subtract from second AP table, 254
subtract integer from elements, 253
table handling statements, 85
vector operations, 86
apa command, 40
apdelay.h file, 119, 121
apovrride statement, 72, 120, 141
applicability of statements, 55
apshaped_dec2pulse statement, 143
apshaped_decpulse statement, 142
apshaped_pulse statement, 144
arc cosine of a number, 43
arc sine of a number, 43
arc tangent of a number, 43
arc tangent of two numbers, 43
argument number, 35
arguments passed to commands and macros, 25
array defined, 31
arraydim parameter, 123, 279
arrayed experiment, 279
arrayed parameter values, 180
arrayed shaped gradient generation, 237
arrayed string variables, 33
arrayed variables, 30, 32
arraying acquisition parameters, 122
ASCII format, 273
asin command, 43
assign integer values, 144
assign statement, 78, 144
asterisk (*) character, 269, 297
asynchronous decoupling, 233
at parameter, 89
atan command, 43
atan2 command, 43
attenuators-based shaped pulses, 116
attributes of parameter, 292
attributes of variables, 31
auto file, 274
Autogain, see automatic gain
autoincrement attribute, 84, 86, 232
Autolock, see automatic lock
automatic execution of macros, 292
automatic gain
  errors, 335
automatic lock
  errors, 335
automatic macro execution, 26
automatic variables, 31
automation file, 274
autoscale command, 42
autoscaling, 42
average command, 43
average value of input, 43
awe parameter, 298
awk command (UNIX), 269
axis command, 46
axis labels, 46
axis parameter, 46

B
back slash single quote (\'...) notation, 328
background process (UNIX), 269
background processing, 271
backslash (\) notation, 28
backward single-quote (\"...\"), 28
bandinfo macro, 115
banner command, 40
beeper sound, 46
beepoff command, 46
beepon command, 46
binary files, 273
binary information file, 274
blanking amplifiers, 75, 145, 153, 205
blankingoff statement, 145
blankingon statement, 145
blankoff statement, 75, 145
blankon statement, 75, 146
block size complete, 333
block size counter, 77
block size variable, 80
Boolean expressions, 35
Boolean operations, 30
bootup macro, 25, 46, 48
box mode, 40
Breakout panel, 75, 76, 226
bs parameter, 77, 80, 81, 333
bsctr real-time variable, 77, 81
bsval real-time variable, 77, 81
buffering in memory, 274
button labels, 328

C
C loop, 118
C programming language, 55
C programming language framework, 58
cat command (UNIX), 269
cd command (UNIX), 269
cf parameter, 107
change bar, 21
change current directory, 269
channel control, 122
channel identifiers, 122
channel selection, 63
char-type variables, 59
checkpw macro, 37
checksum of FDF file data, 284
chemical shift, 49
chmod command (UNIX), 269
clear command, 41
clearapdatatable statement, 106, 146
clearing a window, 41
cmp command (UNIX), 269
coarse attenuators, 71
code file, 274
coef file, 274
Index

coherence transfer selective phase cycling, 69
 colon (:) notation, 26
 command entry, 268
 command interpreter, 24
 command output to variables, 26
 command tracing, 38
 comments, 30
 in macros, 24
 comparing two files (UNIX), 269
 compilation error messages, 57
 compiling source code, 56
 completed transient counter, 77
 complex pair of FID data, 277
 compressed acquisitions, 130
 compressed data format, 286
 compressed files, 281
 compressed loop, 201, 215
 Compressed-compressed data format, 286
 concatenate and display files (UNIX), 269
 concatenate strings, 29
 conditional execution, 178, 187
 conditional statements, 24, 36
 config command, 289
 conpar file, 289, 292
 constant delay time for changing the status, 82
 constant phases, 77
 constant strings, 25
 constants, 28
 continuous decoupling caution, 72
 continuous wave (CW) modulation, 65, 233
 conventions used in manual, 21
 conversion units, 49
 copying files (UNIX), 268
 copying macros, 44
 cos command, 43
 cosine value of an angle, 43
 COSY-NOESY sequence, 107
 course power control, 71
 cp command (UNIX), 268
 cp parameter, 77
 cr parameter, 39
 crcom command, 44
 create command, 290, 292
 create_delay_list statement, 131, 146
 create_freq_list statement, 131, 148
 create_offset_list statement, 131, 148
 createatable macro, 128
 creating
 directories (UNIX), 269
 FDF files, 284
 new parameter, 290
 slider in Acquisition window, 102
 user macros, 44
 variable without value, 31
 ct variable, 77, 84
 curly braces {...} notation, 36, 59
 curpar file, 274, 278, 289
 current experiment files, 274
 current parameter tree, 289
 current parameters text file, 274
 current-type parameter tree, 289
 cursor mode, 40
 cursor position, 39
 curve fitting, 42

D

d0 parameter, 81
d2 parameter, 77, 123
d3 parameter, 77, 123
 d4 parameter, 77, 123
 DANTE sequence, 116, 118
 Data Acquisition Controller boards, 59, 141
 data acquisition statements, 60
 data block, 275
data block header, 275
data buffers, 274
data directory, 274
data file, 274, 278, 279
data file header, 275
data file in current experiment, 280
data point acquisition, 105
data portion of FDF file, 281
data transposition, 279
data.h file, 275
datablockhead structure, 276
datadir3d directory, 274
datafilehead structure, 275
date command (UNIX), 269
dbl statement, 78, 151
dc drift correction, 278
dcphase statement, 119
dcpltr2phase statement, 68, 106, 119, 152
dcpltr3phase statement, 68, 106, 119, 153
dcpltrphase statement, 68, 106, 119, 151
ddf command, 280
ddff command, 280
ddfp command, 280
debug command, 38
dec file suffix, 109
dec2blank statement, 75, 153
dec2off statement, 75, 155
dec2offset statement, 70, 155
dec2on statement, 75, 156
dec2phase statement, 106, 157
dec2power statement, 71, 106, 119, 158
dec2progoff statement, 114, 120, 160
dec2prgon statement, 75, 114, 120, 161
dec2prgoff statement, 72, 106, 119, 163
dec2prgoff statement, 65, 106, 165
dec2shaped_pulse statement, 112, 118, 120, 168
dec2spinlock statement, 115, 120, 170
dec2stepsizex statement, 69, 172
dec2unblank statement, 75, 173
dec3blank statement, 75, 154
dec3off statement, 75, 155
dec3offset statement, 70, 155
dec3on statement, 75, 157
dec3phase statement, 67, 106, 158
dec3power power, 119
dec3power statement, 71, 106, 159
dec3progoff statement, 114, 120, 160
dec3prgon program, 114
dec3prgoff statement, 75, 120, 161
dec3shaped pulse statement, 112, 120, 169
dec3spinlock statement, 115, 120, 166
dec3stepsizex statement, 69, 172
dec3unblank statement, 75, 173
dec4offset statement, 156
dec4phase statement, 158
dec4power statement, 159
dec4rgpulse statement, 166
decblank statement, 75, 153
DECCh, DEC2Ch, DEC3Ch devices, 148, 149
declvloff statement, 72, 106, 154
declvlon statement, 72, 106, 154
decoff statement, 74, 154
decoffset statement, 70, 155
decon statement, 74, 156
decoupler
  blank associated amplifier, 75, 153
  fine power, 163, 224, 229
  fine power adjustment, 72
  fine power with IPA, 194
  full power, 154
  gate channel, 241
  gating, 73, 74, 249
  high-power level, 162
  linear modulator power, 224, 229
  linear modulator power with IPA, 194
  modes, 74
  modulation mode, 74
  normal power, 154
  offset frequency, 69, 70, 155, 208
  pattern type, 109
  phase, 67
  phase control, 68
  power adjustment, 71
  power level, 72, 158, 222, 229
  power level switching, 71
  programmable decoupling, 159, 160
  pulse shaping via AP bus, 142
  pulse with IPA, 186
  pulse with receiver gating, 162, 164
  quadrature phase, 157
  set status, 233
  shaped pulse, 167
  simultaneous pulses, 65
  small-angle phase, 151
  small-angle phase step size, 250
  spin lock waveform control, 170
  status, 248
  step size, 171
  turn off, 154
  turn on, 156
  two-pulse shaped pulse, 113
  unblank amplifier, 172
  WALTZ decoupling, 67
  waveforms, 111
decoupler mode, 233
decoupling, switching, 173
decphase statement, 67, 106, 157
decpower statement, 71, 106, 119, 158
decprgoff statement, 111, 114, 120, 160
decprgong statement, 74, 111, 114, 120, 160
decpulse statement, 64, 106, 162
decpwr statement, 162
decpwrfl statement, 72, 106, 119, 163
decr statement, 78, 164
decrement integer value, 164
decrgpulse statement, 64, 106, 164
decshaped_pulse statement, 112, 118, 120, 167
decspinlock statement, 115, 120, 170
decstepsize statement, 69, 171
decunblank statement, 75, 172
delay
  create delays table, 146
  for synchronizing sample rotor, 231
  initialize, 191
  interincrement, 81
  intertransient, 81
  parameter type, 289
  real-time incremental, 188
  routine, 183
  specified time, 173
  specified time with IPA, 187
  timebase fixed and real-time count, 259
  with possible homospoil pulse, 185
delay statement, 61, 102, 106, 173
delay-related statements, 61
delays
  initializing next for hardware shimming, 184
delcom command, 44
deleting files (UNIX), 268
deleting user macros, 44
destroy command, 292
destroygroup command, 292
device gating, 180
dg2 parameter, 295
Dgroup field, 292
Dgroup of a parameter, 291
dhp parameter, 72, 154
diff command (UNIX), 269
differentially compare files (UNIX), 269
diffusion analysis, 42
digital resolution measurement, 39
dimensioning statement, 33
directory information, 47
disk blocks, 275
disk cache buffering, 274
disk file errors, 335
display command, 291
displaying
  controlling pulse sequence graphical display, 82
  date and time (UNIX), 269
  FID file, 280
  file headers, 280
  macros, 44
  memory usage, 280
  part of file (UNIX), 269
  pulse sequences, 82
dividing an AP table into a second AP table, 254
dividing an integer into AP table elements, 252
dividing integer values, 174
divn factor, 84, 86, 232
divn statement, 174
divn-return attribute, 84, 86, 232
dll command, 26
dm parameter, 73
dm2 parameter, 74
dm3 parameter, 74
dmm parameter, 64, 73, 120, 121, 233
dmm2 parameter, 74, 120
dm33 parameter, 74, 120
DODEV, DO2DEV, DO3DEV constants, 63
### Index

<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>dof parameter, 69</td>
<td>69</td>
</tr>
<tr>
<td>dof2 parameter, 69</td>
<td>69</td>
</tr>
<tr>
<td>dof3 parameter, 69</td>
<td>69</td>
</tr>
<tr>
<td>dollar-sign (?) notation, 28, 32</td>
<td></td>
</tr>
<tr>
<td>done codes, 60, 333</td>
<td></td>
</tr>
<tr>
<td>double integer value, 151</td>
<td></td>
</tr>
<tr>
<td>double quotation marks (&quot;...&quot;) notation, 30</td>
<td></td>
</tr>
<tr>
<td>double-precision, 31</td>
<td></td>
</tr>
<tr>
<td>double-type variables, 59</td>
<td></td>
</tr>
<tr>
<td>dp parameter, 274</td>
<td></td>
</tr>
<tr>
<td>dps command, 56, 82, 174</td>
<td></td>
</tr>
<tr>
<td>dps_off statement, 82, 174</td>
<td></td>
</tr>
<tr>
<td>dps_on statement, 82, 174</td>
<td></td>
</tr>
<tr>
<td>dps_ps_gen command, 56</td>
<td></td>
</tr>
<tr>
<td>dps_show statement, 174</td>
<td></td>
</tr>
<tr>
<td>dps_skip statement, 177</td>
<td></td>
</tr>
<tr>
<td>dpwr parameter, 72, 118, 154</td>
<td></td>
</tr>
<tr>
<td>dpwr2 parameter, 72</td>
<td></td>
</tr>
<tr>
<td>dpwr3 parameter, 72</td>
<td></td>
</tr>
<tr>
<td>draw pulses for graphical display, 174</td>
<td></td>
</tr>
<tr>
<td>dres command, 39</td>
<td></td>
</tr>
<tr>
<td>ds command, 278</td>
<td></td>
</tr>
<tr>
<td>ds command, 39</td>
<td></td>
</tr>
<tr>
<td>dsn command, 39</td>
<td></td>
</tr>
<tr>
<td>dsnmax command, 39</td>
<td></td>
</tr>
<tr>
<td>du command (UNIX), 269</td>
<td></td>
</tr>
<tr>
<td>duty cycle, 63</td>
<td></td>
</tr>
<tr>
<td>dynamic range of shaped pulse, 116</td>
<td></td>
</tr>
<tr>
<td>dynamic variable gradient pulse generation, 188, 238</td>
<td></td>
</tr>
<tr>
<td>dynamic variable shaped gradient pulse generation, 239</td>
<td></td>
</tr>
</tbody>
</table>

### F

<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>f3 file, 274</td>
<td></td>
</tr>
<tr>
<td>FALSE Boolean value, 35</td>
<td></td>
</tr>
<tr>
<td>FDF files</td>
<td></td>
</tr>
<tr>
<td>attach header to data file, 285</td>
<td></td>
</tr>
<tr>
<td>creating, 284</td>
<td></td>
</tr>
<tr>
<td>directory naming convention, 280</td>
<td></td>
</tr>
<tr>
<td>format, 281</td>
<td></td>
</tr>
<tr>
<td>header format, 281</td>
<td></td>
</tr>
<tr>
<td>magic number, 282</td>
<td></td>
</tr>
<tr>
<td>splitting data and header parts, 285</td>
<td></td>
</tr>
<tr>
<td>transformations of data, 284</td>
<td></td>
</tr>
<tr>
<td>why developed, 280</td>
<td></td>
</tr>
<tr>
<td>fdf files, 281</td>
<td></td>
</tr>
<tr>
<td>fdfgluer command, 285</td>
<td></td>
</tr>
<tr>
<td>fdfsplt command, 285</td>
<td></td>
</tr>
<tr>
<td>FID complete, 333</td>
<td></td>
</tr>
<tr>
<td>FID data, 277</td>
<td></td>
</tr>
<tr>
<td>fid file, 274, 278</td>
<td></td>
</tr>
<tr>
<td>fid file extension, 273</td>
<td></td>
</tr>
<tr>
<td>FID files, 273, 280, 300</td>
<td></td>
</tr>
<tr>
<td>FIFO underflow error, 335</td>
<td></td>
</tr>
<tr>
<td>file</td>
<td></td>
</tr>
<tr>
<td>binary format, 273</td>
<td></td>
</tr>
<tr>
<td>existence test, 46</td>
<td></td>
</tr>
<tr>
<td>header of binary file, 273</td>
<td></td>
</tr>
<tr>
<td>information, 47</td>
<td></td>
</tr>
<tr>
<td>protection mode (UNIX), 269</td>
<td></td>
</tr>
<tr>
<td>text format, 273</td>
<td></td>
</tr>
<tr>
<td>fine attenuators, 72</td>
<td></td>
</tr>
<tr>
<td>fine power, 194, 224, 229</td>
<td></td>
</tr>
<tr>
<td>control, 71</td>
<td></td>
</tr>
<tr>
<td>decoupler, 163</td>
<td></td>
</tr>
<tr>
<td>transmitter, 207</td>
<td></td>
</tr>
<tr>
<td>fine power routine, 103, 183</td>
<td></td>
</tr>
<tr>
<td>fine-grained pulse shaping, 118</td>
<td></td>
</tr>
<tr>
<td>first point correction, 278</td>
<td></td>
</tr>
<tr>
<td>fixpar macro, 26</td>
<td></td>
</tr>
<tr>
<td>flag of a parameter test, 48</td>
<td></td>
</tr>
</tbody>
</table>
Index

flag-type parameter, 289
FLASH pulse sequence, 75
flash command, 289
flexible data format files. See FDF files
floating constant, 28
floating point, 31
float-type variables, 59
flush command, 274, 278
fm-fm modulation, 233
fn parameter, 278
focus command, 47
format command, 41
format of weighting function, 298
formatting for output, 41
forward slash notation (UNIX), 268
Fourier transform process, 278
fourth decoupler
  offset frequency, 156
  power level, 159
  pulse with receiver gating, 166
  quadrature phase, 158
fractions in integer mathematics, 78
framework for pulse sequences, 58
fread command, 291
frequency
  control, 69
  create frequencies table, 148
  offsets table, 148
  set based on position, 220
  set from position list, 220, 221
  set on position, 220
  table indexing, 261
frequency and intensity from line list, 39
frequency limits of region, 39
frequency lists, 148
frequency offset lists, 263
frequency offset routine, 101, 183
frequency-type parameter, 289
fsave command, 274, 291
ft command, 278
ft3d command, 274

G
G_Delay general routine, 98, 102, 183
G_Offset general routine, 98, 101, 183
G_Power general routine, 98, 103, 183
G_Pulse general routine, 98, 99, 100, 102, 183
gap command, 47
GARP modulation, 233
gate pulse sequence from an external event, 264
gate statement, 180
gating control statements, 73
Gaussian pulse, 116
gcoil parameter, 128
gedit, 270
generic delay routine, 102, 183
generic pulse routine, 100, 183
getarray statement, 131, 180
getelem statement, 86, 180
getfile command, 47
getl command, 39
getorientation statement, 181
getreg command, 39
getstr statement, 58, 96, 182
getval statement, 58, 96, 182
getvalue command, 290
Ggroup, 291, 292
global file, 289
global list, 148, 149
  statements, 131
global PSG parameters, 89
global variables, 31
global-type parameter tree, 289
go command, 80
gradaaxis parameter, 128
gradient
  control, 125
  set to specified level, 228
  simultaneous shaped, 198
  variable angle, 256
  variable angle gradient pulse, 256
  variable angle shaped gradient, 257
  variable angle shaped gradient pulse, 258
  waveforms, 109, 111
  zero all gradients, 265
gradient function, 188
gradient level set by real-time math, 261
gradient pattern file, 191
gradient pulse, 127
  generation, 238
  on z channel, 266
  simultaneous shaped, 199
gradient statement, 131
gradables directory, 128
gradtype parameter, 120, 125
graphical display of a sequence, 56
graphical display of pulse sequences, 82
graphical display of statements, 174
graphics display status, 47
graphis command, 47
GRD file suffix, 109
grep command (UNIX), 269
gripper abort, 334
group of parameters, 291
groupcopy command, 291

H
half value of integer, 184
half-transformed spectra, 279
hardloop nesting, 106
hardware loop, 104, 178
  end of loop, 178
  start of loop, 248
hardware phase control, 68
hardware shimming
  initializing next delay, 184
  hardware WALTZ decoupling, 67
hardwired 90° phase, 68
head command (UNIX), 269
header of FDF file, 281
HET2DJ pulse sequence, 123
hidden delay, 119
hidecommand command, 44
high-band nuclei, 64
high-noise signal, 333
high-speed device control, 75
Index

high-speed line propagation delay, 121
hlv statement, 78, 80, 184
HMQC experiment, 63
hom2dj.c sequence listing, 56
HOM2DJT pulse sequence, 87
home directory for user (UNIX), 268
homo parameter, 64, 65
homo2 parameter, 65
homo3 parameter, 65
homodecoupler gating, 65
homonuclear J-resolved pulse sequence, 87
homonuclear-2D-J pulse sequence, 55
homospoil gating, 73, 74, 248
homospoil pulse, 61, 185
host disk errors, 335
hs parameter, 61, 73
hsdelay statement, 61, 74, 106, 185
hst parameter, 61, 74
hwlooping.c module, 67
hypercmplxblockhead structure, 277
hypercomplex 2D, 123
i2pul.c pulse sequence, 99
id2 pointer, 58, 77, 124
id3 pointer, 58, 77
id4 pointer, 58, 77
idecpulse statement, 65, 186
idecrg pulse statement, 65, 186
idelay statement, 61, 187
identifier, 28, 36
if, then, else, endif conditional form, 36
ifzero statement, 79, 83, 187
image file names, 281
image plane orientation, 181
imaginary component of FID data, 277
imaging module, 125
imaging-related statements, 129
implicit acquisition, 60
implicit expressions, 35
implicitly arrayed delay, 123
inactive parameter, 48
incdelay statement, 61, 188
increment an integer value, 189
increment counts, 58
increment index, 124
incremental delay, 61, 188, 191
incrementing a loop, 37
index out of bounds, 34
indices of an array, 32
indirect detection, 189
indirect detection experiments, 122
indirect statement, 189
info directory, 274
init_gradpattern statement, 132, 191
init_rfpattern statement, 132, 190
inidelay statement, 61, 191
initialize incremental delay, 191
initialize parameters for imaging sequences, 192
initialize real-time variable, 192
initialize string variable, 32
initparms_sis statement, 75
initival statement, 79, 192
input arguments, 35
input command, 41
input tools, 40
integ command, 39
integer array stored in AP table, 233
integer mathematical statements, 78
integer values
  add, 141
  assign, 144
  decrement, 164
  divide, 174
  double, 151
  half value, 184
  increment, 189
  modulo 2, 200
  modulo 4, 200
  modulo n, 200
  multiply, 201
  subtract, 251
integer-type parameter, 289
intensity of spectrum at a point, 40
interactive parameter adjustment (IPA), 98
  change fine power, 194
  change linear modulator power, 194
  change offset frequency, 193
  delay specified time, 61, 187
  fine power control, 72
  pulse decoupler, 65, 186
  pulse transmitter, 63, 192, 193, 194
interferograms, 279
interincrement delays, 81
internal hardware delays, 119
internal variables, 76
intertransient delays, 81
int-type variables, 59
iobspulse statement, 63, 192
ioffset statement, 70, 193
ipa, See interactive parameter adjustment (IPA)
ipulse statement, 63, 193
iopwrf statement, 72, 194
ipwrm statement, 72, 194
irgpulse statement, 63, 194
ix variable, 57
j
jexp command, 31
k
keyboard entries recording, 46
keyboard focus to VNMR input window, 47
keyboard input, 41
kill command (UNIX), 269
kinetic analyses, 42
l
largest integral in region, 39
last used parameters text file, 274
latching, on PTS synthesizers, 117
Index

length command, 47
length of macros, 38
lib directory, 132
libparam.a object library, 56
libpsglib.a directory, 56, 132
library directory, 132
line frequencies and intensities, 40
line list, 32, 39
linear amplifier systems
  power control, 70
  power level, 221, 228
  stabilization, 64
linear attenuator used for pulse shaping, 112
linear modulator power, 229
linear modulators, 72
lines in a file, 41
linewidth measurement, 39
link loading, 56
lint command (UNIX), 56
Linux
  shell, 270
  tools, 267
list files in a directory (UNIX), 268
listenoff command, 47
listenon command, 47
listing names of macros, 45
lists
  frequency, 148
  global, 148, 149
  offset, 149
lk_hold statement, 106, 127, 195
lk_sample statement, 106, 127, 195, 197
llamp parameter, 32
llfrq parameter, 32
ln command, 44, 268
loading AP table elements from file, 85, 196
loading AP table statements, 83
loading macros into memory, 26, 45
loadable statement, 83, 85, 196
local variables, 31, 32, 34
lock correction circuitry, 127
  set to hold, 195
  set to sample, 195
lock feedback loop, 127
lock level, 48
log directory, 274
log files, 271, 274
logarithm of a number, 44
logical frame, 131
login command, 48
login command (UNIX), 269
login macro, 25, 26, 46
login procedure, 267
logout (UNIX), 269
long-type variables, 59
lookup command, 41
loop
  end, 178
  multislice end, 178
  multislice start, 200
  phase-encode end, 179
  phase-encode start, 215
  start, 196
  statements, 131
  types, 37
loop statement, 79, 104, 118, 196
low-band nuclei, 65
low-core acquisition variables, 80
lower shell script, 272
low-noise signal, 333
lp command (UNIX), 269
ls command (UNIX), 268

M
macilib directory, 25
macilibpath parameter, 25
macro
  automatic execution, 26, 292
  calling a macro in a loop, 27
  clear system macro, 27
  concept, 23
  defined, 23
  directory, 25
  editing, 26
  execution, 25
  existence test, 46
  faster execution, 26
  files, 25
  loading into memory, 26
  output to variables, 26
  parsing, 26
  passing information, 32
  remove from memory, 27
  VNMR activation, 48
macro name list, 45
macro parameter, 26
macro tracing, 38
macrocat command, 44, 45
macrop command, 44
macrod command, 45
macroedit macro, 26, 45
macrol command, 26, 27, 45
macrorun command, 45
macros.h file, 100
macrosystem cat command, 45
macrosyscp command, 45
macrosysdir command, 45
macrosysrm command, 45
macrovi command, 26, 45
magic number, 282
MAGICAL language defined, 23
MAGICAL language features, 27
magradient statement, 197
magradpulse statement, 128, 129, 198
mail command (UNIX), 269
makefid command, 300
man command (UNIX), 269
manual directory, 60
manual entry (UNIX), 269
mark command, 40
MAS rotor, 230
mashapedgradient statement, 129, 198
mashapedgradpulse statement, 199
mathematical expression, 35
mathematical functions, 43
matrix arithmetic, 30
matrix transposition, 279

346  VnmrJ 2.2D User Programming  01-999379-00 A 0808
Index

maximum value of parameter, 292
maxpk macro, 37
MAXSTR dimension, 59
mean of data in regression.inp, 42
memory usage by VNMR commands, 280
memory usage statistics, 45
MEMS pulse sequence, 75
meme size parameter (UNIX), 274
menu files, 327
message display with large characters, 40
mf command, 287
mfblk command, 287
mfdata command, 287
mftrace command, 287
micon, 328
microimaging pulse sequences, 127
minimum value of parameter, 292
mixing shapes, 209
mkdir command (UNIX), 268
mlabel, 328
MLEV-16 modulation, 233
mod2 statement, 78, 200
mod4 statement, 78, 200
modn statement, 78, 200
modulation frequency, 233
modulation frequency change delay, 121
modulation mode change delays, 120
modulo 2 integer value, 200
modulo 4 integer value, 200
modulo n integer value, 200
modulo number, 78
move data in FID file, 287
move FID commands, 287
moving files into a directory, 268
MREV-type sequences, 105
msloop statement, 131, 200
mstat command, 45, 280
mstring, 328
mult statement, 78, 201
multidimensional NMR, 122
multiple command separator (UNIX), 268
multiple FID acquisition, 107
multiple trace or arrayed experiments, 279
multiply AP table by second AP table, 254
multiply integer values, 201
multiply integer with AP table elements, 252
multislice loops, 131, 200
multiuser protection, 271
mv command (UNIX), 268

N
n1-n3 parameters, 31
name replacement, 36
name.errors text file, 57
natural logarithm of a number, 44
nested macros, 38
nested multiple hardloops, 106
nf parameter, 107
ni parameter, 77
ni2 parameter, 77
ni3 parameter, 77
nll command, 40
NMR algorithms, 23
NMR language, 23
non-observe pulse, 64
notational conventions, 21
np parameter, 335
records command, 41
nth2D variable, 215
null string, 31, 32
number of arguments, 35
numeric parameter value lookup, 96, 182
numreg command, 40

O
object code, 56
object file, 132
object libraries, 56
obl_gradients statement, 202
obl_shapedgradient statement, 203
oblique gradient, 202
oblique gradient statements, 131
oblique gradient with phase encode in 1 axis, 212,
216
oblique gradient with phase encode in 2 axes, 212
oblique gradient with phase encode in 3 axes, 213,
217
oblique shaped gradient with phase encode in 1 axis,
136, 213, 217
oblique shaped gradient with phase encode in 2 axes,
214
oblique shaped gradient with phase encode in 3 axes,
215, 218
oblique gradient statement, 131, 202
oblique_shapedgradient statement, 203
obs_mf parameter, 74
obsblank statement, 205
OBSch device, 148, 149
observe channel gating, 241
observe transmitter modulation, 233
observe transmitter power, 205
observe transmitter pulse, 62
obsoffset statement, 70, 205
obspower statement, 71, 106, 205
obsprgoff statement, 120, 206
obsprgon statement, 74, 114, 120, 206
obsrun statement, 63, 100, 106, 206
obsrswf statement, 72, 106, 119, 207
obsstepsize statement, 68, 69, 207
obsunblank statement, 207
off command, 48
offset frequency, 155, 193, 205
offset lists, 149
offset macro, 35
offset statement, 69, 101, 106, 119, 208
offset_table, 263
on command, 48
one pointer, 77
operating system, 267
operators, 29
oph variable, 77, 107
order of precedence, 29
orientation of image plane, 181
Output boards, 59, 141
output from commands and macros, 26
output to various devices, 42
Index

output tools, 40
overhead delays, 131
overhead operations, 82
override internal software AP bus delay, 141

P
package files, 268
page
creating new, 324
defining, 325
panel
creating a new layout, 324
editor, 323
saving and retrieving elements, 325
search path, 326
size, 327
pap command, 297
par2d macro, 123
par3d macro, 123
par4d macro, 123
paramedit command, 290, 294
parameter
attributes, 292
create new parameter, 290
enumerable values, 293
maximum value, 292
minimum value, 292
table, 58
template, 294
trees, 288
typical parameter file, 293
values, 293
parameters
accessing the value, 290
arrayed parameter values, 180
as global variables, 31
as variables, 24
categories, 89
change type, 291
conditional display, 296
display field width, 297
display formats, 297
display values in text window, 41
editing attributes, 290
existence test, 46
get value, 290
global PSG parameters, 89
look up value, 96
plotting automatically, 40
protection bit, 25
protection bits, 291
set up for pulse sequence, 41
spectroscopy imaging sequences, 192
step size, 292
types, 289
user created, 89
parameters retrieved from a parameter file, 48
paramvi command, 290, 292, 294
parent directory (UNIX), 268
parentheses (…) notation, 34
parlib directory, 41
parmax parameter, 292
parmin parameter, 292

parsimmming macros, 26
parstep parameter, 292
pattern scanning and processing (UNIX), 269
Pbox, 109
mixing shapes, 209
Pbox experiments, 209
pe_gradient statement, 131, 212
pe_shapedgradient statement, 213
pe2_gradient statement, 212
pe2_shapedgradient statement, 214
pe3_gradient statement, 213
pe3_shapedgradient statement, 215
peak command, 24, 26, 40
peak width of solvent resonances, 48
pepl loop statement, 131, 215
Performa XYZ PFG module, 128
pexpl command, 42
PFG (pulsed field gradient), 128
phase angle, 110
phase calculation, 77
phase carryover, 69
phase control, 77
phase cycle storage, 83
phase cycling, 87
phase encode loops, 131
phase file in the current experiment, 280
phase parameter, 123
phase step size, 250
phase_encode_gradient statement, 131, 216
phase_encode_shapedgradient statement, 217
phase_encode3_gradient statement, 217
phase_encode3_shapedgradient statement, 218
phase1 integer, 123
phase1 variable, 58
phase2 parameter, 123
phase3 parameter, 123
phased 2D data storage, 279
phased spectral information, 274
phased spectrum, 278
phase-encode loop, 179, 215
phasefile file, 274, 278, 279
phase-pulse technique, 219
phase-related statements, 67
phase-sensitive 2D NMR, 123, 277
phaselshift statement, 219
phi angle, 129
phi parameter, 131
pipe, 269
plotif macro, 38
plotting curves, 42
pmode parameter, 274
poffset statement, 131, 220
poffset_list statement, 131, 220
pointer to memory, 76
pointers to constants, 77
poly0 command, 42
polynomial curves, 42
position list, 220, 221
position statements, 131
position_offset statement, 131, 220
position_offset_list statement, 131, 221
position-based frequency, 220
power control statements, 70
power level of shaped pulse, 116
power statement, 71, 72, 106, 117, 118, 119, 221
power statements, back-to-back, 71
ppm of solvent resonances, 48
precedence of operators, 29
presaturation, 71
print files (UNIX), 269
probe damage caution, 72
procad file, 274
process status (UNIX), 269
processed-type parameter tree, 289
procpar file, 274, 278, 280, 281, 289
procpar3d file, 274
program execution, 24
programmable control of transmitter, 206
programmable control statements, 114
programmable decoupling
  ending, 159
  starting, 160
programmable phase and amplitude control, 114
programmable pulse modulation, 233
programming
  imaging pulse sequences, 127
  Performa XYZ PFG module, 127
prompt for user input, 41
propagation delay, 121
protection bits, 25, 291, 292
prune command, 292
ps command (UNIX), 269
ps delay, 71
ps directory, 132
ps macro, 80
psgen shell script, 132
pslib directory, 55
psset command, 41
psi parameter, 131
PTS synthesizers with latching, 117
pulse
  non-observe, 64
  pulse channels simultaneously, 241, 242
  pulse control, 109
  pulse decoupler, 162
  pulse decoupler with IPA, 186
  pulse decoupler with receiver gating, 164
  pulse four channels simultaneously, 242
  pulse interval time, 114
  pulse observe transmitter, 62
  pulse program buffer, 104
  pulse routine, 183
  pulse sequence control statements, 79
  Pulse Sequence Controller board, 141
  pulse sequence gated from external event, 264
  pulse sequence generation (PSG), 57
    directory, 55
    statement categories, 61
pulse sequences
  compiling, 56
  execution control, 77
  files, 55
  general form, 59
  graphical display, 56, 82
  imaging, 127
  internal hardware delays, 119
  object code, 58
object file, 132
parameter set up, 41
programming, 55
synchronization, 108
pulse shape definitions, 109
pulse shaping programming, 109
pulse shaping through AP bus, 112
pulse shaping via AP bus, 118, 142, 143
pulse statement, 63, 100, 106, 222
pulse transmitter with IPA, 192, 193, 194
pulse transmitter with receiver gating, 206, 222, 227
pulse width array, 32
Pulsed Field Gradient module, 125
pulsed field gradient module, 128
pulseinfo macro, 115
pulsesfunction function, 58, 80
pulsesfunction.o file, 132
pulse-type parameter, 289
pulsing channels simultaneously, 65
pulsing the decoupler transmitter, 64
purge command, 27, 46
pw parameter, 63, 335
pwd command, 269
pwr statement, 72, 106, 117, 119, 224
pwm statement, 72, 117, 224
pwsadj macro, 114
Q
quadrature phase, 68
quadrature phase of decoupler, 157, 158
quadrature phase of transmitter, 255
quadrature phase shift, 67
question mark (?) character, 269
quotation mark ("."") notation, 24
R
r1, r2, ..., r7 parameters, 31, 32
rcvroff statement, 75, 225
rcvon statement, 75, 225
read parameters from a file, 291
readlk command, 48
readuserap statement, 76
real command, 31
real component of FID data, 277
real number formatting for output, 41
real parameters, 31
real-number arguments, 59
real-time gradient statements, 131
real-time incremental delay, 61, 188
real-time statements, 80
real-time variables, 59, 76, 77, 79, 192
real-type parameter, 289, 291
real-type variables, 31
receiver
  default state, 192
  gating, 62, 75, 206, 222, 227
  phase control, 77
  phase cycle, 232
  turn off, 227
  turn on, 225
receiver gate, 225, 227
receiver overflow warning, 333
Index

recoff statement, 227
recon statement, 227
record macro, 46
records in file, 41
rectangular pulse, 116
recursive calls, 25
redefinition warning, 58
reference to statements, 133
reformatting data for processing, 285
reformatting spectra, 287
regions in spectrum, 40
regression analysis, 42, 43
regression.inp file, 42
removing an empty directory (UNIX), 268
removing macros, 45
removing macros from memory, 46
renaming a directory (UNIX), 268
renaming a file (UNIX), 268
repeat, until loop, 37
reserved words, 28
resto parameter, 220
retrieve element from AP table, 86, 180
retrieving individual parameters, 48
return command, 38
Return key, 21
returning a value, 38
reverse a spectrum, 287
reverse FID commands, 287
reverse order of data, 287
reverse single quotes (‘...’) notation, 328
rf channels control, 122
RF file suffix, 109
RF monitor errors, 335
rf pattern file, 189
rf pulses waveforms, 109, 110
rf shape file, 110
rfblk command, 287
rfchnchannel parameter, 63, 122
rfdata command, 287
rftrace command, 287
RG1 and RG2 delays, 62, 64
rgpulse statement, 62, 83, 105, 106, 227
rgradient statement, 120, 126, 127, 128, 228
rinput command, 42
rlpower statement, 228
rlpwrm statement, 72, 117, 230
rm command (UNIX), 268
rmr command (UNIX), 268
rof1 parameter, 63
rof2 parameter, 63
root directory (UNIX), 268
rot angle, 230
rotor control statements, 108
rotor period, 108, 230
rotor position, 230
rotorperiod statement, 108, 230
rotosysync statement, 108, 230
RS-232 cable, 333
rsapply command, 287
rt command, 26, 31, 300
rtcommand, 26, 31
rtv command, 26, 48
run program in background, 269
run-time statements, 80
S
sample changer
errors, 334
saved display file, 274
scalelimits macro, 42, 43
scalesw parameter, 46
scaling factors for axis, 46
SCSI errors, 335
searching a text file, 41
searching files for a pattern (UNIX), 269
second decoupler
blank associated amplifier, 153
fine power, 163
fine power adjustment, 72
gating, 75
homodecoupler gating, 65
offset frequency, 69, 70, 155
phase control, 68
power adjustment, 71
power level, 158
programmable decoupling, 160, 161
pulse shaping via AP bus, 142
pulse with receiver gating, 165
quadrature phase, 157
shaped pulse, 168
simultaneous pulses, 66
small-angle phase, 152
spin lock waveform control, 170
step size, 172
turn off, 155
turn on, 156
unblank decoupler, 173
selecet command, 40
semicolon (;) notation, 59
semicolon (;) notation (UNIX), 268
separators, 30
seqcon parameter, 131, 201
seqgen command, 56, 57, 80
seqgen command (UNIX), 56
seqlib directory, 57, 80
set2d macro, 123
set3dproc command, 274
setautoincrement statement, 86, 232
setdgroup command, 291
setdivfactor statement, 86, 232
setenumeral command, 289, 291
setgroup command, 291
setlimit command, 31, 291
setproct command, 291
setreceiver statement, 77, 86, 107, 232
setstatus statement, 74, 120, 121, 233
settable statement, 83, 85, 233
settype command, 291
setuserap statement, 76
setuserpsg shell script, 132
setvalue command, 291, 300
sh2pul macro, 109
shaped gradient, 258
Index

- pulse generation, 235, 237, 239
  - variable angle, 257
- shaped oblique gradient, 203
- shaped pulse
decoupler, 167
delays, 121
information, 115
on transmitter, 234
  - simultaneous three-pulse, 244
  - simultaneous two-pulse, 243
time truncation error, 115
using attenuators, 116
waveform generator control, 112
  - shaped two-pulse experiment, 109
- shaped pulse statement, 112, 118, 120, 235
- shaped2Dgradient statement, 237
- shapedgradient statement, 127, 131, 235
- shapedLineGradient statement, 131, 238
- shapedyGradient statement, 131, 239
- shapelib directory, 109, 142, 235
shell command, 48, 270, 271
shell programming, Linux and Unix, 272
shell scripts, 272
shimming
errors, 334
short-type variables, 59
signal-to-noise measurement, 39
sim3pulse statement, 66, 106, 242
sim3shaped_pulse statement, 113, 120, 244
sim4pulse statement, 66, 242
simpulse statement, 65, 106, 241
simshaped_pulse statement, 120, 243
  - simultaneous gradient, 197
  - simultaneous pulses, 65, 66
  - simultaneous shaped gradient, 198
  - simultaneous shaped gradient pulse, 199
sin command, 44
sine value of angle, 44
single period notation (UNIX), 268
single quotes (‘...’) notation, 25, 28
size operator, 29, 32
SLI board, 245
SLI lines
  - setting lines, 245
  - sli statement, 130, 245
slider action, 103
SLIDER_LABEL attribute, 99, 102
small-angle phase increment, 68
small-angle phase of decoupler, 151, 152, 153
small-angle phase of transmitter, 265
small-angle phase shifts, 67
small-angle phase step size, 250
sn file, 274
soft loop, 104, 118
solppm command, 48
solvent resonances, 48
sort command (UNIX), 268
sort files (UNIX), 268
source code, 55, 132
sp#off statement, 75, 246
sp#on statement, 75, 247
SPARE 1 connector, 75
spare line gating, 246, 247
spare lines, 75
spectral analysis tools, 39
spectrometer control statements, 61
spectrometer differences, 61
spectroscopy imaging sequences, 192
spectrum gap, 47
spectrum intensity at a point, 40
spectrum selection without display, 40
spell command (UNIX), 269
spelling errors check (UNIX), 269
spin lock control on transmitter, 247
spin lock control statements, 115
spin lock waveform control on decoupler, 170
spinlock statement, 115, 120, 247
spinner errors, 333
tsqt operator, 29
square brackets (…) notation, 32
square brackets notation, 297
square root, 29
square wave modulation, 233
squish, 327
ss parameter, 77, 80
sctor real-time variable, 77, 80
svsal real-time variable, 77, 80
standard data format, 286
standard deviation of input, 43
standard PSG variables, 58
standard.h file, 58, 100
start loop, 196
starthardloop statement, 104, 248
status of transmitter or decoupler, 233
status statement, 73, 82, 106, 120, 121, 248
statusdelay statement, 74, 82
steady-state phase cycling, 81
steady-state pulses, 80
step size
decoupler, 171
  - parameters, 292
  - transmitter, 207
steps in shaped pulse, 116
stepsize statement, 151, 250
store array in AP table, 85
stored format of a parameter, 292
storing multiple traces, 279
string command, 31
string constant, 28
string formatting for output, 41
string length, 47
string parameter value lookup, 96, 182
string parameters, 31
string template, 294
string variables, 31
strings displayed in text window, 41
string-type parameter, 289, 291
sub statement, 78, 251
substr command, 49
substring from a string, 49
subtract AP table from second AP table, 254
subtract integer from AP table elements, 253
subtract integer values, 251
sum of integer values, 141
sum-to-memory error, 335
svdf macro, 285
svib macro, 284
svsis macro, 285
swapping rf channels, 63
swept-square wave modulation, 233
synchronization of a pulse sequence, 108
synchronous decoupling, 233
Synchronous Line Interface (SLI) board, 245
sysgcoil parameter, 128
sysmaclibpath parameter, 25
system identification, 269
system macro, 45
system macro library, 25
systemglobal-type parameter tree, 289

T

T1 analyses, 42
t1–t60 table names, 84
T2 analyses, 42
T2PUL pulse sequence, 87
tabc command, 287
table names, 84
table of delays, 146
table of frequencies, 148
table of frequency offsets, 148
tablib directory, 83
tail command (UNIX), 269
tallest peak in region, 40
tan command, 44
tangent value of angle, 44
tape backup (UNIX), 268
tar command (UNIX), 268
tcapply command, 287
template parameters, 294
temporary variables, 24, 28, 31, 32
terminating a calling macro, 38
terminating zero, 98
test4acq procedure, 67
text display status, 49
text file, 274
text file lookup, 41
text format files, 273
textedit command (UNIX), 269, 270
textis command, 49
thermal shutdown, 63
theta angle, 129
theta parameter, 131
third decoupler
blank associated amplifier, 154
fine power, 163
fine power adjustment, 72
gating, 75
homodecoupler gating, 65
offset frequency, 69, 70, 155
phase control, 68
power adjustment, 71
power level, 159
programmable decoupling, 160, 161
pulse with receiver gating, 166
quadrature phase, 157
shaped pulse, 169
simultaneous pulses, 66
small-angle phase, 153
spin lock waveform control, 171
step size, 172
turn off, 155

turn on, 157
unblank amplifier, 173
three pointer, 77
three-pulse pulse, 66
three-pulse shaped pulse, 113, 244
tilde character notation (display templates), 297
tilde character notation (UNIX), 268
time increments, 59
time-sharing pulse shaping, 116
timing in a pulse sequence, 82
tip angle, 111
TODEV constant, 63
tof parameter, 69
token defined, 28
toolbar, editing, 327
total weighting vector, 298
tenpi experiments, 124
TPPI phase increments, 58
tpwr parameter, 72, 118
transformations of FDF data files, 284
transformed complex spectrum storage file, 274
transformed phased spectrum storage file, 274
transformed spectra storage files, 273
transient blocks, 77
transmitter
blanking, 205
fine power, 207, 224, 229
fine power adjustment, 72
fine power with IPA, 194
gating, 74, 111, 264
hardware control of phase, 68
linear modulator power, 224, 229
linear modulator power with IPA, 194
offset frequency, 69, 205, 208
phase control, 67, 68
power adjustment, 71
power level, 205, 222, 229
programmable control, 114, 206
pulse shaping via AP bus, 143
pulse with IPA, 192, 193, 194
pulse with receiver gating, 206, 222, 227
pulse-related statements, 62
quadrature phase, 255
set status, 233
shaped pulse, 112, 234
simultaneous pulses, 65
small-angle phase, 265
small-angle phase step size, 250
spin lock control, 115, 247
step size, 207
unblank, 207
troubleshooting
acquisition status codes, 60
troubleshooting a new sequence, 57
TRUE Boolean value, 35
trunc operator, 29
truncate real number, 29
tssadd statement, 86, 252
tstdiv statement, 86, 252
tsmult statement, 86, 252
tsub statement, 86, 253
ttadd statement, 86, 87, 253
tttdiv statement, 86, 254
ttmtmult statement, 86, 254
Index

ttsub statement, 86, 254
two attenuators system, 118
two periods notation (UNIX), 268
two pointer, 77
two-pulse pulse, 66
two-pulse sequence T2PUL, 87
two-pulse shaped pulse, 113, 243, 244
txphase statement, 67, 69, 106, 255, 257
type of parameter, 291
typeof operator, 29, 36
types of parameters, 289, 292

U
U+ H1 Only label, 122
uname command (UNIX), 269
unblank amplifier, 75, 172
underline prefix, 25
uniform excitation, 71
uninitialized variable, 58
unit command, 49
units command (UNIX), 269
UNIX
commands, 267
file names, 268
shell, 270
shell startup, 48
text commands, 269
updtgcoil macro, 128
user AP lines, 76
user AP register, 226, 234, 263
user library, 56, 118
user macro, 44
user macro directory, 25
user-created parameters, 89
user-customized pulse sequence generation, 132
user-written weighting function, 297

V
v1, v2, ... v14 real-time variables, 59, 77
vagradient statement, 128, 129, 256
vagradpulse statement, 128, 256
values of a parameter, 293
variable angle gradient, 256
variable angle gradient pulse, 256
variable angle shaped gradient, 257
variable angle shaped gradient pulse, 258
variable declaration, 32, 59
variable gradient pulse generation, 238
variable shaped gradient pulse generation, 239
variable types, 31
variables using parameters, 24
vashapedgradient statement, 131, 260
vashapedgradpulse statement, 129, 258
vbg shell script (UNIX), 271
vdelay statement, 61, 259
vdelay_list statement, 131, 260
vertical bar notation (UNIX), 269
vfreq statement, 131, 261
vgradient statement, 120, 126, 131, 261
vi command (Linux and UNIX), 270
vi command (UNIX), 269
vi command (VNMR), 270

vi text editor, 290
VNMR
macros executed at startup, 46, 48
source code license, 275
Vnmr command (UNIX), 270
VNMR Command and Parameter Reference manual, 24
vnmreditor variable (UNIX), 27
VnmrJ
background processing, 271
vnmrsys directory, 25, 57
voffset statement, 131, 263
vsadj macro, 24
vsetuserap statement, 76
vsli statement, 130
vsmult macro, 35
VT errors, 333
vttime parameter, 333

W
w command, 270
w command (UNIX), 269
WALTZ decoupling, 67
WALTZ-16 modulation, 233
warning error codes, 333
warning messages, 57
waveform generation, 233
waveform generator control, 112, 115
waveform generator delays, 121
waveform generator gate, 111
waveform initialization statements, 132
waveform resolution, VnmrS systems, 109, 131
wbs command, 60
weighting function, 278, 297, 298
werr command, 60
wex command, 60
WFG OFFSET_DELAY macro, 121
WFG2 OFFSET_DELAY macro, 121
WFG3 OFFSET_DELAY macro, 121
which macro, 25
while, do, endwhile loop, 37
who is on the system (UNIX), 269
wildcard character (UNIX), 269
wnt command, 60
working directory (UNIX), 268
write command, 42
writing parameter buffers into disk files, 274
wtcalc function, 298
wtf file extension, 298
wtfile parameter, 297, 299
wtfile1 parameter, 297
wtfile2 parameter, 297
wtgen shell script, 298, 299
wti command, 298
wtlib directory, 298, 299
wtp file extension, 298

X
xgate statement, 108, 264
xmtroff statement, 74, 264
xmtron statement, 74, 264
xmtrphase statement, 68, 69, 106, 119, 265
Index

XY32 modulation, 233

Z
z channel gradient pulse, 266
zero acquired data table, 106
zero all gradients, 265
zero data in acquisition processor memory, 146
zero fill data, 278
zero pointer, 77
zero_all_gradients statement, 265
zgradpulse statement, 120, 127, 128, 266
zip, 268