BRUKER

TopSpin

NMR Pulse Programming for AVANCE NEO and Fourier 80

User Manual

Version 015

Innovation with Integrity

NMR

Document Number: 10000059107

P/N: H166633

Contents

1 About This Manual

This manual enables safe and efficient handling of the device.

This manual is an integral part of the device, and must be kept in close proximity to the device where it is permanently accessible to personnel. In addition, instructions concerning labor protection laws, operator regulations tools and supplies must be available and adhered to.

Before starting any work, personnel must read the manual thoroughly and understand its contents. Compliance with all specified safety and operating instructions, as well as local work safety regulations, are vital to ensure safe operation.

The figures shown in this manual are designed to be general and informative and may not represent the specific Bruker model, component or software/firmware version you are working with. Options and accessories may or may not be illustrated in each figure.

1.1 Symbols and Conventions

Safety instructions in this manual and labels of devices are marked with symbols.

The safety instructions are introduced using indicative words which express the extent of the hazard.

In order to avoid accidents, personal injury or damage to property, always observe safety instructions and proceed with care.

⚠ DANGER

DANGER: Indicates a hazardous situation that, if not avoided, will result in death or serious injury. This signal word is limited to the most extreme situations.

This is the consequence of not following the warning.

1. This is the safety condition.

▶ This is the safety instruction.

⚠ WARNING

WARNING: Indicates a hazardous situation that, if not avoided, could result in death or serious injury.

This is the consequence of not following the warning.

1. This is the safety condition.

▶ This is the safety instruction.

⚠ CAUTION

CAUTION: Indicates a hazardous situation that, if not avoided, could result in minor or moderate injury.

This is the consequence of not following the warning.

1. This is the safety condition.

▶ This is the safety instruction.

NOTICE

NOTICE: Indicates information considered important, but not hazard-related (e.g. messages relating to property damage).

This is the consequence of not following the notice.

1. This is a safety condition.

▶ This is a safety instruction.

SAFETY INSTRUCTIONS

SAFETY INSTRUCTIONS are used for control flow and shutdowns in the event of an error or emergency.

This is the consequence of not following the safety instructions.

1. This is a safety condition.

▶ This is a safety instruction.

This symbol highlights useful tips and recommendations as well as information designed to ensure efficient and smooth operation.

2 Basic Pulse Program Writing

2.1 Introduction

A pulse program is an ASCII text consisting of a number of lines. Each line may contain one or more pulse program statements which specify actions to be performed by the acquisition hardware and software. You can set up a pulse program with the TopSpin commands **edpul** or **edcpul** (see the Acquisition Reference Manual). The TopSpin acquisition commands **gs**, **go**, and **zg** execute the pulse program defined by the acquisition parameter PULPROG which can be set with **eda** or **pulprog**. Pulse program execution is a two-step process: After entering **gs**, **go**, or **zg**, the pulse program compiler is invoked which translates the pulse program text into an internal binary form that can be executed. Possible syntax errors are reported. If errors are found, the acquisition will not be started. If, however, the compilation is successful, the compiled pulse program is loaded into the acquisition hardware and the measurement begins.

2.1.1 Spectrometer Naming Conventions

This manual is written for AV spectrometers. Nevertheless, a large part of it is also valid for older spectrometers like AMX, ARX, ASX, and Avance. You can easily find out which type of spectrometer you have at the cabinet door; the name is AV, AV II or AV III (sometimes also AV II+) or AV III HD.

2.1.2 Font and Format Conventions

Type of Information	Font	Examples
Shell Command, **Commands,** “All that you can enter”	Arial bold	Type or enter **fromjdx zg**
Button, Tab, Pane and Menu Names “All that you can click”	Arial bold, initial letters capitalized	Use the **Export To File** button. Click **OK**. Click **Processing**…
Windows, Dialog Windows, Pop-up Windows Names	Arial, initial letters capitalized	The Stacked Plot Edit dialog will be displayed.
Path, File, Dataset and Experiment Names **Data Path Variables** **Table Column Names** **Field Names (within Dialog Windows)**	Arial Italics	*$tshome/exp/stan/nmr/lists* *expno, procno,*
Parameters	Arial in Capital Letters	VCLIST

Program Code
Pulse and AU Program Names
Macros
Functions
Arguments
Variables
	Courier	`go=2`
		`au_zgte`
		`edmac`
		`CalcExpTime()`
		`XAU(prog, arg)`
		`disk2, user2`

Type of Information	Font	Examples
AU Macro	Courier in Capital Letters	REX PNO

Table 2.1: Font and Format Conventions

2.2 Pulse Program Library

Routine users normally use Bruker pulse programs delivered with TopSpin. The **edpul** command displays a list of these pulse programs and allows you to view their contents. Viewing Bruker pulse programs requires that the **expinstall** command was executed once after the installation of TopSpin. This command copies the pulse programs suitable for your spectrometer into the default directories.

If you want to write your own pulse programs, it can be helpful to start with a Bruker pulse program, put it into the user directory, and modify it to your needs.

2.3 Pulse Program Display

A graphical representation of a pulse program for AV spectrometers can be obtained with the command **spdisp**, which is described in the Acquisition Reference manual.

For AV III, the command **hpdisp**, which is also described in the Acquisition Reference manual, displays the pulse program showing the signals as produced by the spectrometer hardware with exact timing of pulses, phases and amplitudes.

2.4 Basic Syntax Rules

```
;zgcw30
;avance-version
;1D sequence with CW decoupling
;using 30 degree flip angle

"d11=30m"
"acqt0=-p1*0.66/PI"
#include <Avance.incl>

1 ze
  d11 pl26:f2
  d11 cw:f2
2 d1
  p1*0.33 ph1
  go=2 ph31
  30m mc #0 to 2 F0(zd)
  d11 do:f2
  exit
```

```
ph1=0 2 2 0 1 3 3 1
ph31=0 2 2 0 1 3 3 1

;pl1 : f1 channel - power level for pulse (default)
;pl26: f2 channel - power level for cw/hd decoupling
;p1 : f1 channel - 90 degree high power pulse
;d1 : relaxation delay; 1-5 * T1
;d11: delay for disk I/O [30 msec]
```

Table 2.2: Pulse program example

The table above shows *zgcw30* as an example of a simple Bruker pulse program. Here the following pulse programming rules are used:

2.4.1 Line Orientation

Pulse programs are line oriented. Each line specifies an *action* to be performed by the acquisition hardware or software.

2.4.2 Comments

A semicolon (;) indicates the beginning of a comment. You can put it anywhere in a line. The rest of the line will then be treated as a comment.

2.4.3 Calculation of Parameters

Parameters can be calculated within the pulse program before and during the execution. Such relations must be enclosed by double quotes. Example: "`d11=30m`", see the table *Table 7.1 [▸ 62]*.

2.4.4 File Import

`#include <filename>` or `#include "filename"`

This statement allows you to use pulse program text that is stored in a different file. It allows you to keep your pulse program reasonably sized, and to use the same code in various pulse programs. If the filename is given in angle brackets (< >), the file is searched for in the directory *$xwinnmrhome/exp/stan/nmr/lists/pp/*. Alternatively, double quotes (" ") can be used to specify the entire path name of the file to be included.

2.4.5 Labels

```
1 ze
```

Any pulse program line can start with a *label* ("1" in the example above). Labels are only required for lines which must be reached by loop or branch statements such as `go=label`, `go to label times n` or `goto label`. You can, however, also use labels for numbering the lines. A label can be a number or an alphanumeric string followed by a comma. An example of the latter is:

```
firstlabel, ze
```

The statement `ze` has the following function:

- Reset the *scan counter* (which is displayed during acquisition) to 0

- Enable the execution of dummy scans. This will cause the pulse program statement `go=label` to perform DS dummy scans before accumulating NS data acquisition scans. If you replace `ze` with `zd`, `go=label` will omit the dummy scans

The statement `zd` automatically resets all phase program pointers to the first element, whereas the statement `ze` sets all phase program pointers such that they are at the first element after DS dummy scans.

2.4.6 Pulse Power

```
d11 pl14:f2
```

Execute a delay whose duration is given by the acquisition parameter D[11]. Behind any delay statement, you can specify further statements to be executed during that delay (note that the delay must be long enough for that statement). In this example, the power level of channel f2 is switched to the value given by the acquisition parameter PLW[14].

2.4.7 Decoupling

```
d11 cw:f2
```

Execute a delay whose duration is given by the acquisition parameter D[11] and, at the same time, turn on continuous wave (cw) decoupling on frequency channel f2. Decoupling will remain active until it is explicitly switched off with the statement `do:f2`.

Delay and cw decoupling will begin immediately with the delay on which it is specified. This illustrates a general feature of pulse programs: the actions specified in two consecutive lines are executed sequentially. Actions specified on the same line are executed simultaneously.

2.4.8 Delay

```
2 d1
```

Execute a delay whose duration is given by the acquisition parameter D[1]. This line starts with the label “2“, the position where the statement `go=2` will loop back to.

2.4.9 Pulse

```
p1*0.33 ph1
```

Execute a pulse on frequency channel f1. The pulse length of this pulse is given by the acquisition parameter P[1] multiplied by 0.33. P[1] is normally used for the pulse width of a 90° flip angle. The statement `p1*0.33` would then execute a 30° pulse. In general, you can specify the operator * behind (not before!) a pulse or delay statement, followed by a floating point number. Note that the channel f1 is not specified; it is the default channel for `p1`, i.e.:

```
p1*0.33
```

is identical to:

```
p1*0.33:f1
```

The pulse is executed with a power (amplitude) defined by the acquisition parameter PLW[1]. PLW[1] is the default power level for channel f1, but you can also use a different parameter. For example, the statement `pl7:f1` sets the channel f1 power to the value of PLW[7]. It must be put on a separate line, with a delay, before the line with the pulse statement.

The phase of this pulse in our example is selected according to `ph1`, the name of a phase program or phase list. It must be specified behind the pulse and defined after the pulse program body. In this example we use the phase program

```
ph1=0 2 2 0 1 3 3 1
```

The phase of the pulse varies according to the current data acquisition scan. For the first scan, `p1` will get the phase *0*90°*, for the second scan *2*90*, for the third scan *2*90*, for the fourth scan *0*90,* etc. After 8 scans, the list is exhausted. The phase program is cycled, so with scan 9 the phase will be set to the first element of the list: *0*90°*. Phase cycling is a method of artefact suppression in the spectrum to be acquired. The receiver phase must be cycled accordingly to ensure that coherent signals of subsequent scans are accumulated, not cancelled. This is achieved by the receiver phase program `ph31` in our example.

2.4.10 Data Acquisition

```
go=2 ph31
```

Execute one data acquisition scan, then loop to the pulse program line with label “2“. Repeat this until NS scans have been accumulated. Note that NS is an acquisition parameter. The data acquisition scans are preceded by DS dummy scans (because the statement `ze` is used at the beginning of the pulse sequence rather than `zd`). A dummy scan does not acquire any data, but requires the same time (given by the acquisition parameter AQ) as a real scan. Dummy scans are used to put the spin system of the sample into a steady state before acquisition starts.

The receiver phase is changed after each scan as described above for the pulse phase. Phase cycling is done according to the phase program `ph31`. Phase cycling is also used during the execution of dummy scans. Both DS and NS must therefore be a multiple of the number of phases in the list.

The `go=label` statement executes a delay, the so-called *pre-scan-delay* to avoid pulse feed through before it starts digitizing the NMR signa. lDuring this time the receiver gate is opened and the frequency is switched from transmit to receive. DE is an acquisition parameter that can be set from **eda** or by entering `de` on the command line. It consists of the sub-delays DE1, DEPA, DERX and DEADC that can be set with the command **edscon** (see the Acquisition Reference manual). Normally, you can accept the default values for DE value and its sub-delays. The total time the `go=label` statement requires to execute a scan is DE+ AQ+3 millisec. The duration of 3 millisec is required for preparation of the next scan. It is valid for all Avance type spectrometers.

2.4.11 The mc Macro

```
30m mc #0 to 2 F0(zd)
```

writes the accumulated data as file *fid* into the EXPNO directory of the current data set. Note that with the *zgcw30* pulse program, data are only stored on disk after all NS scans have been accumulated. You can, however, store the data to disk after any scan during the acquisition by entering the command **tr** on the command line. You can process and plot these data while the acquisition continues. If you want to protect your data against power failures during long term experiments, we recommend that you write the data on disk in regular intervals, for example every 100 scans. To accomplish this, you can set NS=100 and TD0=300 (if the pulse program is written using the `mc`-syntax). The pulse program then accumulates a total of 30.000 scans, but stores the result every 100 scans.

Please note that the `mc` macro must include a `zd` statement. The reason for this is that `wr #0` *adds* the last acquired data to the data already present in the file.

The real time FID display will only show the data currently present in the acquisition processor’s memory.

2.4.12 Decoupler off Command

```
d11 do:f2
```

The decoupling power is switched off.

2.4.13 Pulse Program Exit

```
exit
```

Specifies the end of the pulse program.

2.4.14 Phase Program Definitions

After the `exit` command the phase program definitions follow. The commentary text at the end of the pulse program contains comments for parameters which can be displayed in the editor **ased.**

3 Pulse Generation

The next table shows the available types of statements for the generation of high frequency pulses

`p0, p1, ... , p63`	Generate a pulse whose length is given by the acquisition parameter P[0], ..., P[63].
`p0 :r, ...,p63:r`	Generate a pulse whose length is given by the acquisition parameter P[0], ..., P[63] and which is randomly varied. The maximum variation (in percent) is defined by the acquisition parameter V9.
`3.5 up, 10mp, 0.1sp`	Generate a pulse of fixed length: up = a µsec pulse (mp = millisec, sp = sec).
`P135, p30d1H`	Generate a pulse, whose name is defined by a `define pulse` statement, and whose duration is defined by an expression.

Table 3.1: Pulse generation statements

A high frequency pulse is described by its:

- duration (= pulse width)
- frequency
- phase
- power (=amplitude) and shape

The following paragraphs will describe these items.

3.1 Pulse Duration

The pulse duration is selected according to the name of the pulse statement.

3.1.1 p0-p63

The statement:

```
p0
```

executes a pulse of width P[0]. P[0] is an acquisition parameter that can be set from **eda**, or by typing *p0* on the command line. Likewise, the statement:

```
p1
```

executes a pulse of with P[1].

3.1.2 Fixed Length Pulses

The statement:

```
10mp
```

executes a pulse of width 10 ms (called a *fixed pulse* because its duration cannot be manipulated, see below). The duration must be followed by `up`, `mp` or `sp`. These units indicate microseconds, milliseconds, and seconds, respectively. If you omit the terminating “`p`“, a delay would be executed instead of a pulse.

3.1.3 Random Pulses

The statement:

```
p0:r
```

executes a pulse of length P[0] which is varied randomly. The parameter V9 specifies, in percent, the maximum amount which is added to or subtracted from P[0]. As such, the effective pulse varies between 0 and 2*P0. It can be set from **eda**, or by typing `v9` on the command line.

Please note that the **gs** command ignores the `:r` option.

3.1.4 User Defined Pulses

The statement:

```
p30d1H
```

executes a pulse whose name is defined by the user, and whose duration is determined by an arithmetic expression. For example, the line:

```
define pulse p30d1H
```

defines `p30d1H` to be a pulse statement, and the line:

```
"p30d1H=p1*0.33"
```

defines the expression for its duration. Note that the definition must be within double-quotes (“) .

Both the `define` statement and the defining expression must be placed before the beginning of the actual pulse sequence. It is evaluated at compile time of the pulse program, not at run time. User defined pulses can consist of alphanumeric characters, where the first character must be a alphabetic. The maximum length of the name is 11 characters. Make sure you do not use any of the reserved words like `adc`, `go`, `pulse` etc.

3.1.5 Pulse Lists Defined in the Pulse Program

Instead of setting up a pulse list with **edlist**, a list of pulses can also be specified within the pulse program using a `define` statement, e.g.:

```
define list<pulse> P1list = { 10 20 30 }
```

This statement defines the pulse list `P1list` with values 10 µsec, 20 µsec and 30 µsec. User defined pulse lists must be initialized within the definition. There are two alternatives to assigning values directly in {}-brackets. You can specify the filename of a pulse list or the variable that contains such a filename, both in angle brackets. Examples,

```
define list<pulse> P2list = <mypulselist>
```

```
define list<pulse> P3list = <$VPLIST>
```

In both cases, the file that contains the pulse list can be created with the command **edlist vp.**

According to the `define` statements above:

```
P1list
```

executes a pulse of 10 µsec the first time it is invoked. In order to access different list entries, you can append the `.inc`, `.dec` or `.res` postfix to the pulse statement to increment, decrement or reset the index, respectively. Additionally, two operators are available which can be used to retrieve both the length (`.len`) and the maximum value (`.max`) of a given list. Any index operations are performed cyclically i.e. when the pointer is at last entry of a list, the next increment will move it to the first entry. Furthermore, list entries can be specified directly in squared brackets counting from 0, i.e. the statement:

```
P1list[1]
```

executes a pulse of 20 µsec according to the above definition. Lists can be executed and incremented with one statement, using the caret postfix operator. As such, the statement:

```
P1list^
```

is equivalent to:

```
P1list P1list.inc
```

Finally, you can calculate the index directly in an arithmetic expression within double quote characters, appending `.idx` to the pulse statement. The following example shows the use of a pulse list that is assigned within its definition:

```
define list<pulse> locallist = {10 20 30 40}

locallist locallist.inc  ;  pulse of 10 µsec, change index from 0 to 1
locallist locallist.res  ;  pulse of 20 µsec, set index to 0
locallist[2]             ;  pulse of 30 µsec (leave the index unchanged)
locallist locallist.dec  ;  pulse of 10 µsec, change index from 0 to 3
locallist                ;  pulse of 40 µsec
"locallist.idx = 3"      ;  set index to 3
"l3 = locallist.len"     ;  set loop counter 3 to the length of the list
                         ;  (which is ‘4’ in our current example)
"p3 = locallist.max"     ;  set p3 to the maximum value of the list
                         ;  (which is ‘40’ in our example)
locallist^               ;  pulse of 40 µsec, move index to 0
locallist                ;  pulse of 10 µsec
```

Index operations on pulse lists only take effect in the next line. Furthermore, you cannot access two different entries of the same list on one line. This is illustrated in the following example

```
define list<pulse> locallist = {10 20 30 40}

locallist^ locallist  ;  uses the same list entry (10 µs) twice
locallist             ;  the ^ operator takes effect: 20 µs locallist[2]
locallist[3]          ;  executes locallist[3] (40 µs)
```

Note that names for user defined items may consist of up to 19 characters, but only the first 7 are interpreted: i.e. Pulselist1 and Pulselist2 are allowed names but they would address the same symbol.

3.1.6 Manipulating Pulse Durations: The Multiplication Operator *

A pulse duration can be manipulated with the operator *. Examples of allowed statements:

```
p1*1.5
p30d1H*3.33
p3*oneThird
```

The operator must be placed behind the pulse statement. `oneThird` is the name of a macro which must have been defined at the beginning of the pulse program, e.g.:

```
#define oneThird 0.33
```

Note that fixed pulses cannot be manipulated. So the statement `10mp*0.33` would be incorrect.

3.1.7 Manipulating Pulse Durations: Changing p0-p63 by a Constant Value

Each pulse statement `p0`-`p63` has been assigned an acquisition parameter INP[0]- INP[63] These parameters take a duration value, in µsec. The pulse program statements `ipu0`-`ipu63` add the value of INP[0]-INP[63] to the current value of `p0`-`p63`, respectively. Likewise, `dpu0`-`dpu63` subtract the value of INP[0]-INP[63] from the current value of `p0`-`p63`. The statements `rpu0`-`rpu63` reset `p0`-`p63` to their original values, i.e. to the values of the parameters P[0]-P[63]. The statements presented in this paragraph must be specified behind a delay of any length.

Some examples:

```
d1 ipu3
0.1u dpu0
d1 rpu0
```

3.1.8 Manipulating Pulse Durations: Redefining p0-p63 via an Expression

The duration of the pulses `p0`-`p63` is normally given by the parameters P[0]-P[63]. You can, however, replace these values by specifying an expression in the pulse program. The following examples show how you can do this:

```
"p13=3s + aq - dw*10"
"p13=p13 + (p1*3.5 + d2/5.7)*td"
```

The result of such an expression must have a time dimension. You can therefore include acquisition parameters such as pulses, fixed pulses, delays, fixed delays, the acquisition time AQ and the dwell time DW within the expression. Furthermore, you can include parameters without a dimension such as the time domain size TD. The complete list is shown in the table *Parameters in relations* [▸ 67].

An expression must be specified between double quote characters (" "). It can be placed anywhere in the pulse program, as long as it occurs before the line that contains the corresponding pulse statement (which would be `p13` in our example). Note that the second expression in the example above assigns a new value to `p13` each time the expression is encountered, e.g. if it is contained in a pulse program loop.

Expressions can appear before the start of the pulse program (initialization phase) or during pulse program execution (runtime).

Expressions during runtime do not cause an extra delay in the pulse program. The evaluation is done before the pulse program is loaded, and the result is stored in the available buffer memory to be accessed at run time. If pulse programs execute too fast, a run time message is printed.

3.1.9 Manipulating the Durations of User Defined Pulses

User defined pulses, as described in section *User Defined Pulses* [▸ 18], can be manipulated in the same way pulses defined by `p0`-`p63` are manipulated (see sections above).

3.2 Pulse Frequency

3.2.1 Frequency Channels

The RF frequency of a pulse is selected via the spectrometer channel numbers

f1, ... ,f8 (the actual numbers of the channels depend on your spectrometer type and accessory). A pulse on a particular channel is executed with the frequency defined for that channel. The statements:

```
p1: f2
p2*0.33:f2
p30d1H*3.33:f2
```

all execute a pulse on channel f2, with the duration P[1], P[2]*0.33, p30d1H*3.33 and a value from VPLIST, respectively. The pulse frequency is the value of the acquisition parameter SFO2; the default frequency for channel f2. If the channel is not specified in the pulse statement, `p1`, `p2`, ..., `p63` all use the default channel f1. The default frequencies of the channels f1-f8 are given by the parameters SFO1-SFO8 (see the description of SFO1, NUCLEI, and **edasp** in the Acquisition Reference manual for more information about defining frequencies for a particular channel). These parameters are loaded into the synthesizer(s) before the pulse program starts. This gives the hardware time to stabilize before the experiment begins. f1, f2 ... always denote the logical channel which must not necessarily correspond to the physical channel with SGU1, SGU2 ... The assignment of logical to physical channels is done via routing parameters (see **edasp**).

3.2.2 Setting Frequency from Constants or Numbers

The frequency can also be set to the values of the parameters CNST0-63 or to any number, for example:

```
d1 fq=cnst20 :f1          ; SFO1 [MHz] + CNST[20] [Hz]
d1 fq=3000:f1             ; SFO1 [MHz] + 3000[Hz]
```

set the frequency on channel f1 to the value of CNST20 and to 3000 Hz, respectively. The default settings refer to SFOn as base frequency and the offsets are in Hz. But the offset can be given in PPM as well, and the base channel frequency can be BFn instead of SFOn. This is achieved by specifying options after the frequency setting command.

Example:

```
d1 fq=cnst20 (bf ppm):f1
```

The resulting frequency will be Fn = BFn[MHz](1 + 1.0e-6*CNST[20][PPM]). The following options are possible:

Option	Meaning
`sfo`	Base frequency SFOn
`bf`	Base frequency BFn
`Hz`	Offset in Hz
`ppm`	Offset in ppm

Table 3.2: Frequency modifiers

3.2.3 Frequency Lists Applied to the Reference Frequency

The reference frequency is the intermediate frequency which is used by the receiver to mix the observed signal down to a lower frequency which can be digitized there. Normally this frequency is the same as the frequency of the transmitter pulse. For all types of Avance spectrometers frequency changes will never affect the reference frequency.

If the reference frequency must be changed during pulse program execution, use the (`receive`) option, like for instance:

```
define list <frequency> FQ1=<$FQ1LIST>
1u FQ1(receive):f1
```

3.2.4 Frequency Lists Defined in the Pulse Program

Frequency lists can also be defined in the pulse program using the `define` statement. The name of a list can be freely chosen, for example:

```
define list<frequency> username = { 200 300 400 }
```

The list must be initialized, specifying a list of frequency offsets between braces, separated by white spaces. By default, the entries are taken as frequency offsets (in Hz) to the default frequency (SFOx) of the channel, for which the list is used. However, this behaviour can be changed by specifying a modifier before the first entry of the list.

```
define list <frequency> freqList = {10    ; offset in [Hz]
20 30 40 50}                              relative to SFO

define list <frequency> freqList = {O     ; offset in [Hz]
300 10 20 30 40 50}                       relative to 300 MHz in
                                          this example

define list <frequency> freqList = {p,    ; offset in [ppm]
10, 20, 30, 40, 50}                       relative to SFO

define list <frequency> freqList = {P,    ; offset in [ppm]
10, 20, 30, 40, 50}                       relative to BF

define list <frequency> freqList = {bf    ; offset in [Hz]
hz, 10, 20, 30, 40, 50}                   relative to BF

define list <frequency> freqList = {bf    ; offset in [ppm]
ppm, 10, 20, 30, 40, 50}                  relative to BF

define list <frequency> freqList = {sfo   ; offset in [Hz]
hz, 10, 20, 30, 40, 50}                   relative to SFO

define list <frequency> freqList = {sfo   ; offset in [ppm]
ppm, 10, 20, 30, 40, 50}                  relative to SF
```

Instead of list entries, a list definition can also contain the name of a list file between angle brackets, e.g.:

```
define list<frequency> filefq = <freqlist>
```

The specified file can be created with the command **edlist** *f1*. Alternatively, you can specify $FQxLIST between angle brackets, where x is a digit between 1 and 8. For example:

```
define list<frequency> f1list = <$FQ1LIST>
```

In this case the value of the parameter FQxLIST will be used as filename. The format of frequency lists is the following: the first line contains the modifiers according to the table *Frequency modifiers [▶* 21], the following lines contains the frequencies, one item per line.

A maximum of 32 different frequency lists can be defined within a pulse program. The name can be of arbitrary length, but only the first 7 characters are interpreted.

A difference between regular frequency lists (interpreted by the `fqn` statements) and a frequency list defined within the pulse program is that the latter is not auto incremented. The list index can, however, be manipulated with postfix operators. The operators `.inc`, `.dec`, `.res` increment, decrement and reset the index, respectively , whereas `.len` and `.max` can be used in a relation to retrieve the length of the list and the maximum value. Furthermore, you can use a caret operator (^) to execute the list and increment the pointer with one statement. You can also address a list entry by specifying its index in square brackets [ ]. Note that index manipulation statements are executed at the end of the duration. This, for example, means that the statement:

```
d1 fqlist^  :f1 fqlist:f2
```

sets both channels f1 and f2 to the same frequency and afterwards increments the list pointer.

Note that the index runs from 0 and will be treated modulo the length of the list. As such, by incrementing the index, the frequency can be cycled through a list.

You can also set the index with a relation adding the `.idx` postfix to the list name.

Example:

```
define list<frequency> fqlist = { 100 200 300}
ze
1 p1
  d1 fqlist:f1 fqlist.inc;  set freq. to SFO1+100, incr. pointer
  p1:f1                  ;  use frequency SFO1+100Hz
  d1 fqlist^:f1          ;  set frequency and increment pointer
  p1:f1 fqlist.res       ;  use freq. SFO1+200, set pointer to 0
  d1 fqlist:f1           ;  set frequency to SFO1+100
  p1:f1                  ;  use frequency SFO1+100
  d1 fqlist[2]:f1        ;  set frequency to SFO1 +300
  p1:f1                  ;  use frequency SFO1+300
  "fqlist.idx = 1"       ;  set pointer to entry 1
  "l3 = fqlist.len"      ;  set loop counter 3 to length of list
                         ;  (which is ‘3’ in our example)
  "cnst3 = fqlist.max"   ;  set constant to maximum value of list
                         ;  (which is ‘0.0003’ in our example)
  d1 fqlist:f1           ;  set the frequency SFO1+200
  p1:f1                  ;  use frequency SFO1+200
  d1 fqlist.dec          ;  decrement pointer
  go=1
exit
```

3.2.5 Frequency Lists Defined in a File

A frequency list can be defined as a text file whose lines contain frequency values (see the command **edlist** in the Acquisition Reference manual).

For example, the statement

```
define list<frequency> f1list = <$FQ1LIST>
```

uses the frequency list whose file name is defined by the acquisition parameter FQ1LIST.

You can set FQ1LIST etc. from the **eda** dialog box, and you can modify a selected list with **edlis**t.

The first use of `f1list` in the pulse program sets the frequency of the respective channel by taking the current value from the list defined by FQ1LIST. When `f1list` is executed the first time, the current value is the first value in the list. The next time `f1list` is incremented (e.g. by the statement `f1list.inc`) the current value will be the next value in the list, etc. At the end of the list, the pointer will be re-set to the first entry of the list.

The frequency list in the text file has to be formatted in a way that the meta information (see Chapter *Frequency Lists Defined in the Pulse Program [▸ 22]*) is written in the first line of the file followed by one frequency value in each of the following lines.

The following examples illustrate some possible beginnings of a frequency list file:

Example 1:

```
O 300.13
10
20
30
```

Example 2:

```
bf ppm
103
103.5
```

Example 3:

```
bf hz
8000
7000
```

Example 4:

```
sfo ppm
5
3
```

3.2.6 Using Old Style Frequency Lists

You can change the frequency of a channel within a pulse program with the statements `fq1-fq8`.

This syntax is obsolete now and should be replaced by one of the possibilities mentioned above.

3.3 Pulse Phase

3.3.1 Phase Program Definition

Pulse phases are relative phases with respect to the reference phase for signal detection. A phase must be specified behind a pulse statement with the name of a phase program. For example, the statements:

```
10mp:f1 ph3
p2*0.33:f2 ph4
p30d1H*3.33:f3 ph5
```

execute pulses on the channels f1, f2, f3 and f4, respectively. As such, the channel frequencies would be SFO1, SFO2, SFO3, and SFO4. The channel phases are set according to the current value of the phase programs `ph3`, `ph4`, `ph5`, and `ph6`, respectively. If a pulse is specified without a phase program, it will have the last phase that was assigned to the channel on which the pulse is executed. Note that at pulse program start, before any pulse has been executed, the phase on all channels is zero.

The four examples above can also be written in the following form:

```
(10mp ph3):f1
(p2*0.33 ph4):f2
(p30d1H*3.33 ph5):f3
```

This form expresses more clearly that a phase is a property of a spectrometer channel.

3.3.2 Phase Program Syntax

A phase program can be specified as shown in the following examples:

```
ph1 = 0 0 1 1 2 2 3 3           ;(1)
ph1 = (5) 0 3 2 4 1             ;(2)
ph1 = {0}*4 {2}*4               ;(3)
ph1 = {0 2}^1                   ;(4)
ph1 = {0 2}^1^2^3               ;(5)
ph1 = {1 3}^1^2*2               ;(6)
ph1 = {{0 2}*2}^1^2             ;(7)
ph1 = {{{0}*2}^2^3^1}^2         ;(8)
ph1 = (5) {1 2}*2^1             ;(9)
ph1 = ph2*2 + ph3               ;(10)
ph1 = (float, 90.0) 30 60 95.5 ;(11)
```

A phase program can contain an arbitrary number of phases.

Furthermore, the list of phases in a phase program can be spread over several lines, for example:

```
ph1 = 0 2 2 0
     1 3 3 1
```

In (1), the phases are expressed in units of 90°. The actual phase values are 0, 0, 90, 90, 180, 180, 270, 270.

In (2), the phases are expressed in units of 360°/5 , corresponding to the actual phase values 0*72, 3*72, 2*72, 4*72, 1*72 = 0, 216, 144, 288, 72 degrees. The divisor, to be specified in parentheses ( ) and before the actual phase list, can be as large as 2$^{\wedge 16}$=65536. This corresponds to a digital phase resolution of 360°/65536, which is better than 0.006°.

In (3) - (9), the operators “ * “ and “ ^ “ are used, which allow you to write long phase programs in a compact form. For phase programs with less than 16 phases, the explicit forms (1) and (2) are usually easier to read. The operator “*n“ (with n = 2, 3, ...) must be specified behind a list of phases that is enclosed in braces { }. It repeats the contents of the braces (n-1) times. The operator “^m“ (with m = 1, 2, 3, ...) must be specified behind a list of phases that is enclosed in braces { }, or behind a previous “^m“ or behind an “*“ operator. Each “^m“ operator repeats the contents of the braces exactly once, but the repeated phase list will be incremented by m*360/d degrees (modulo d) where d is the divisor of the phase program. If no divisor is specified, the default value of 4 is used. The following lines display the phase programs (3) - (9) in their explicit forms:

```
ph1 = 0 0 0 0 2 2 2 2                        ;(3’)
ph1 = 0 2 1 3                                ;(4’)
ph1 = 0 2 1 3 2 0 3 1                        ;(5’)
ph1 = 1 3 2 0 3 1 1 3                        ;(6’)
ph1 = 0 2 0 2 1 3 1 3 2 0 2 0                ;(7’)
ph1 = 0 0 2 2 3 3 1 1 2 2 0 0 1 1 3 3        ;(8’)
ph1 = (5) 1 2 1 2 2 3                        ;(9’)
```

In (10), the phase program is the sum of two other phase programs, one of which is multiplied with an integer constant. This principle is illustrated by the following example. Assume the following phase programs:

```
ph2 = 0 2 1 3
ph3 = 1 1 1 1 3 3 3 3
```

In order to calculate ph1 = ph2*2 + ph3, we first calculate ph2*2:

```
ph2*2 = 0 0 2 2
```

Then we extend ph2 to the same size as ph3:

```
ph2 = 0 0 2 2 0 0 2 2
ph3 = 1 1 1 1 3 3 3 3
```

Now we calculate the sum of the two:

```
ph1 = 1 1 3 3 3 3 1 1
```

In cases where phase programs are added and the size of one of them is not a multiple of the size of the other, the resulting phase program will have the length of the smallest common multiple of the two phase programs.

In (11), the phases are defined as floating point numbers in degree. In case an ’`ip`’ command is used, the increment is specified as the second argument in parentheses.

3.3.3 Phase Program Position

Phase programs must be specified at the end of the pulse program after the "`exit`" statement (see the pulse program example in table *Pulse program example* [▶ 12], chapter *Basic Pulse Program Writing* [▶ 12]). Any pulse program can contain up to 32 different phase programs (`ph0`- `ph31`).

3.3.4 Phase Cycling

At the start of a pulse program, the first phase of each phase program is valid. The next phase becomes valid with the next scan or dummy scan. When the end of a phase program is reached, it starts from the beginning (phase cycling).

3.3.5 Phase Pointer Increment

The phase pointer in all phase programs is automatically incremented by the `go` statement. However, it is also possible to explicitly switch to the next phase as shown in the following example:

```
p1:f2 ph8^
p2:f2 ph8
```

`p1` is executed with the currently active phase of `ph8`, then `p2` is executed with the next phase in `ph8`. The caret (^) postfix in the first line, increments the phase pointer to the next phase in the list. This phase will become valid with the next pulse program statement that includes this phase program (note that this can be the same statement if it is included in a loop).

The following example is equivalent to the one above:

```
p1:f2 ph8 ipp8
p2:f2 ph8
```

Only in this case the statement `ipp8` is used to increment the pointer in the phase program `ph8`. Please note that `ipp8` is specified on the same line as `p1` and therefore does not cause an extra delay between `p1` and `p2`. The increment statements `ipp0`-`ipp31` are available for the phase programs `ph0`-`ph31`. Increment statements can also be specified with a delay rather than a pulse. For example,

```
d1 ipp7
```

moves the pointer to the next phase in `ph7`.

If explicit phase program manipulation is used in the pulse program, the phase program concerned will no longer be incremented with the `go` command.

The statements `rpp0`-`rpp31` can be used to reset the phase program pointer to the first element. The statement `zd` automatically resets all phase program pointers to the first item, whereas the statement `ze` sets the pointer such that after DS dummy scans the pointer will be at the first element of each phase program. Phase programs that use the autoincrement feature or explicit incrementation with `ipp` are not incremented by the `go` statement at the end of a scan.

One can set the phase program pointer to a value calculated previously:

```
"phval0 = 5;"
rpp7 + phval0
```

sets the pointer in phase program 7 to the 5th item (starting with 0). Note that only `phval0` is allowed as a variable in this context.

`dpp0` - `dpp31` can be used analogously to go back to the previous phase program item.

If all phase programs are concerned one can also use the commands `ippall`, `dppall` and `rppall` to manipulate them.

3.3.6 Adding a Constant to a Phase Program

You can change all phases in a phase program by a constant amount with the `:r` option. Each phase program `ph0`-`ph31` has a constant assigned to it, PHCOR[0]- PHCOR[31]. These can be set from **eda**, or by entering *phcor0* etc. on the command line. For example, with `ph8 = 0 1 2 3` and PHCOR[8]=2°, the phases of the pulse:

```
(p1 ph8:r):f2
```

are 2, 92, 182, 272 degrees. Without the `:r` option, the phase cycle of `p1` would be 0, 90, 180, 270 degrees. The `:r` option can be used together with the caret postfix, e.g.:

```
(p1 ph8^:r):f2
```

3.3.7 Phase Program Arithmetic

Each of the phase programs `ph0-ph31` has three associated statements:
`ip0-ip31`, `dp0-dp31`, `rp0-rp31`.
These statements can also be used with an integer multiplier n, for example

```
ip12*n.
```

Consider the phase programs

```
ph3 = 0 2 2 0
```

and

```
ph4 = (5) 0 1 2 3.
```

The pulse program statement

```
ip3
```

increments all phases of `ph3` by 90°. The next time that `ph3` is encountered, its phase cycle will be `1 3 3 1`.
Likewise, the pulse program statement

```
ip4
```

increments all phases of `ph4` by 360/5 degrees. The next time that ph4 is encountered in the pulse program, its phase cycle will be `(5) 1 2 3 4`.
The statements `dp0-dp31` decrement all phases of the associated phase program. The statements `rp0-rp31` reset all phases of a phase program to their original values, i.e. to the values they had before the first `ip0-ip31` or `dp0-dp31`.
The statements

```
ip3*2
dp4*2
```

increment `ph3` by 2*90=180°and decrement `ph4` by 2*360/5=144°.
It is also not necessary anymore to specify an explicit delay for the `ip/dp/rp` statements as their calculation does not require additional time on modern spectrometers.

3.3.8 Phase Program Modifications at Runtime

The following modifications on a phase program are possible during the execution of an experiment:

```
(1) p1 ph=91.5
(2) (d1 p21:sp2 ph=cnst30):f2
(3) p1 ph1+ph2
(4) p1 ph1+90
(5) p1 ph=cnst30+90
```

In (1), a phase is set to a value given explicitly in degrees.

In (2), the phase is set from the parameter `cnst30`, which can be calculated in a relation at some other place in the pulse program before.

In (3), the phases of 2 phase programs are added together.

In (4), a constant offset is added to a phase program. This is especially useful in subroutines, where a phase program is defined in the main program and phases in the subroutine are set relative to this phase program.

In (5), the parameter `cnst30` is used with an offset.

3.3.9 Calculation and Usage of Phase Programs at Runtime

There are two ways to calculate constants from phase programs and set phases at runtime:

(1)

```
"cnst30=ph1+nsdone*90"
p1 ph=cnst30
```

(2)

```
"ph1=(nsdone%8)*30"
p1 ph1
```

In example (1), `cnst30` is calculated, using a phase program and adding a variable amount to it depending on the current scan counter. `cnst30` the is used some time later in the pulse program to set the phase of pulse `p1`.

In (2), the current value of phase program `ph1` is overwritten with some value calculated from the scan counter. If the scan counter was 3 and the original phase program was

```
ph1(360)=0 90 90 0
```

the modified phase program would be

```
ph1(360)= 0 90 60 0
```

The phase program ph1 will have this value as long as no phase program manipulation command is executed between the two statements. Any such command will replace the current phase value with the value from the original phase program.

3.3.10 Runtime Changes of the Phase Program Increments

The `ip` statement can also be used to add increments other than the amounts defined in the definition of the phase program. This is done using the parameters CNST[0]-CNST[31] (which can have a positive or negative value). For example, the statement:

```
d11 ip1+cnst23
```

adds the value of CNST[23] to each phase of the phase program `ph1`.

A constant can also be defined in the pulse program. As such it is calculated at runtime. For example, the section:

```
"cnst23=d0*360/24;"
d11 ip1+cnst23
```

calculates a phase from the current value of `d0` and then puts it into the parameter `cnst23`. Then it adds this value (in degrees) to each phase of the phase program `ph1`. Note that `ip1+cnst23` works on the **original** phase program `ph1` whereas `ip1(*n)` works on the **current** phase program `ph1`.

As an example, the next pulse program section increments the phase at runtime depending on the number of scans done:

```
"cnst5 = 20"
2 d1
p1 ph1
```

`6u ip1+cnst5` ; set the phase program to the original values + cnst5 "cnst5= nsdone*30"

```
go =2 ph31
ph1 = 0 2 2 0 1 3 3 1
```

3.3.11 Phase Setting without Executing a Pulse

Phases can be set after pulses or delays. TopSpin allows to set the phase for a particular spectrometer channel without executing a pulse. In that case, you must specify a phase program behind a delay. Example:

```
(d1 ph1):f3
(p11:sp1):f3
```

Note that on AV spectrometers there is no frequency while there is no pulse. This kind of phase setting makes sense only if there is no time to set the phase during the pulse (e.g. for ultrafast shape pulses).

3.3.12 Definition of Phase Programs Using List Syntax

Instead of setting up `ph0`-`ph31` at the end of the pulse program, a phase program may also be defined by a list definition, which must then occur before the actual start of the pulse program (beginning with `ze`):

```
define list<phase> PhList1={0.0 180.0 90.0 270.0}
```

This statement defines the phase program *PhList1* with phase values of 0°, 180°, 90° and 270°. Note that in contrast to the previously described definitions of phase programs, all angles are written in degree in this syntax. Instead of initializing the phase program directly with {}-brackets, you may also specify the file name of a phase program or the variable PHLIST which contains such a filename, both in angle brackets. Examples,

```
define list<phase> PhList2=<myphaseprogram>
define list<phase> PhList3=<$PHLIST>
```

In both cases, the file that contains the phase program can be created with the command **edlist phase**.

Phase setting from user-defined phase programs is done in exactly the same way as with the standard phase programs by specifying the phase program after a pulse statement.

After initialization, the current value of a user-defined phase program is its first entry. You can access other entries by using the list operations `.inc`, `.dec`, or `.res` to increment, decrement, or to reset the index. By using the caret postfix operator (^) you can combine phase setting with an increment operation, as with other list types or with the standard phase programs. However, in contrast to other list types, you can neither retrieve a particular entry using the [ ]-bracket notation, nor set the index directly by assigning to `PhList1.idx`. Furthermore, no equivalents to the `ipX`, `dpX`, and `rpX` statements available with the standard phase programs `ph0-ph31` exist for user-defined phase programs. Note that user-defined phase program names may consist of up to 19 characters, but only the first 15 are interpreted. Up to 32 user-defined phase programs may be defined in a single pulse program. It is furthermore possible to define the standard phase programs `ph0` to `ph31` with the above syntax. In this case the phase program base (used for the `ip0-ip31`, and `dp0-dp31` statements) is implicitly 65536 (equivalent to 16 bits). Using an `ipX` command on a standard phase program defined with the above syntax will shift all of its phase entries by 360°/65536=0.006°. After 65536 ipX statements, the phase program entries would have returned to their initial values. To make a 90° phase increments use `ipX*16384`.

3.4 Pulse Power and Shape

3.4.1 Rectangular Pulses

A *rectangular* pulse has a constant power while it is executed. It is set to the current power of the spectrometer channel on which the pulse is executed. The default power for channel f1, f2, ... , f8 is PLW[1], PLW[2], .., PLW[8]. Here, PLW is an acquisition parameter that consists of 64 elements PLW[0] - PLW[63]. It can be set from **eda** or by entering `plw0, plw1`, etc. on the command line. You can set the power for a particular channel with the statements `pl0-pl31`. For example:

```
d1 pl5:f2
```

sets the transmitter power for channel f2 to the value given by PLW[5]. Any pulse executed on this channel will then get the frequency SFO2 and the power PLW[5]. The `pl0-pl63` statements must be written behind a delay. The power setting occurs within this delay, which must be at least 0.2 µsec. For AVIII systems the power can be set together with the pulse and needs no extra delay.

The power can be set not only from these parameters but also from constants, from the parameters SPW0..63 and directly as a numbers:

```
d1 pl=cnst23[Watt]:f1
d1 pl=sp7:f1
d1 pl=3[dB]:f1
```

In case of constants or numbers the unit specifier in square brackets must follow.

3.4.2 Power Lists

In addition to the above possibilities, you can use user defined power lists on Avance spectrometers. A user defined power list is defined and initialized in a single define statement, e.g.:

```
define list<power> pwl = {dB -6.0 -3.0 0}
```

The `define list<power>` key is followed by the symbolic name, under which the list can be accessed in the pulse program. The name is followed by an equal sign and an initialization clause, which is a list of power values, either in `Watt` or in `dB`, enclosed in braces. Entries must be separated by white spaces. The first entry must be the unit specifier.

You can access a power list by specifying its name, e.g.:

```
d1 pwl:f1
```

sets the power of channel f1 to -6.0 dB, when it is used for the first time. You can move the pointer within a power list with the increment, decrement and reset postfix operators `.inc`, `.dec`, `.res`. For example, you can switch to the next entry of the above list with the statement:

```
pwl.inc
```

Alternatively, you can use the caret (^) operator to set the power and increment the list pointer within one statement. For example, the statement:

```
d1 pwl^:f1
```

is equivalent to:

```
d1 pwl:f1 pwl.inc
```

You can also access the list index in a relation, appending `.idx` to the symbolic name, e.g:

```
"pwl.idx = pwl.idx + 1"
```

The above expression is equivalent to:

```
pwl.inc
```

Furthermore it is possible to access the length (using the operator `.len`) or respectively the maximum (using `.max`) value of the list within a relation (see examples below).

It is also possible to access a certain list element by specifying its number in square brackets, for example:

```
pwl[2]
```

Note that list indices start with 0. All index calculations are performed modulo the length of the list. In the above example `pwl[3]` = `pwl[0]` = -6.0.

Note that index manipulations are executed at the end of the duration. This means, for example, that the statement:

```
d1 pwl^:f1 pwl:f2
```

will set both the f1 and f2 channel to the same power level.

As an alternative to initializing a list, you can specify a list file in angle brackets, e.g.:

```
define list<power> fromfile = <pwlist>
```

Such a file can be created or modified with the command **edlist va**.

The format of this power list file has to be as follows:

```
<powerToken>

<value 1>

<value 2>

…

<value n>
```

with <`powerToken`> being either `Watt` or `dB` depending on how you prefer to define the power values.

For example, the following list defines three power values with 200, 300, and 400 Watts, respectively:

```
Watt

200

300

400
```

Instead of a filename you can also specify *$VALIST*, for example:

```
define list<power> fromva = <$VALIST>
```

In this case, the filename is defined by the acquisition parameter VALIST.

Note that the number of user defined lists is limited to 32 for each list type. The length of the name is arbitrary, but only the first 7 characters are interpreted.

The following example shows the use of an initialized power list:

Example:

```
define list<power> pwl = {Watt 10 30 50 70}
```

```
ze
1 d1 pwl:f1 pwl.inc             ; set power on f1 to 10W, incr. pointer
  d1 pwl:f2 pwl.dec             ; set power on f2 to 30W, decr. pointer
  d1 pwl[2]:f3                  ; set power on f2 to 50W
  "pwl.idx = pwl.idx + 3"       ; set the pointer to 0 to 3
  "l3 = pwl.len"                ; set loop counter 3 to length of list
                                ; (in our example to '4')
  "cnst3 = pwl.max"             ; set constant to maximum value of list
                                ; (in our example to '70')
  d1 pwl^:f4                    ; set power on f4 to 70W, incr. pointer
  (p1):f1 (p2):f2 (p3):f3 (p4):f4
  go=1
exit
```

3.4.3 Shaped Pulses

A *shaped* pulse changes its amplitude (and possibly phase) in regular time intervals while it is executing. The pulse shape is a sequence of numbers (stored in a file, see below) describing the amplitude and phase values which are active during each time interval. The interval length is automatically calculated by dividing the pulse duration by the number of amplitude values in the shape file. If this is less than the minimum duration, an error message is displayed which tells you what is the minimum pulse duration for this shaped pulse.

The next 3 examples generate shaped pulses:

```
(10mp:sp2 ph7) :f1
(p1:sp1 ph8):f2
(p30d1H*3.33:sp3 ph9):f3
```

The pulse durations are 10 ms, P[1], and p30d1H*3.33, respectively. The pulses are executed on the frequency channels f1, f2, and f3 (i.e. the pulse frequencies are SFO1, SFO2, and SFO3), respectively. The pulse shape characteristics are described by the entries 2, 1, and 3 (corresponding to `:sp2`, `:sp1`, and `:sp3`) of the table **shaped pulse parameter**. This table is displayed when you click the SHAPE button within **eda**. The table has 64entries with the indices 0-63. You may use the statements `:sp0` - `:sp63` to refer to the entries 0-63, respectively. As you can see from the examples, a phase program can be appended to a shaped pulse in the same way it can be appended to a rectangular pulse. The current phase of the phase program is added to the phase of each component of the shaped pulse.

Each entry of the shaped pulse parameter table has 4 parameters assigned to it: a *power value* SPW, an *offset* SPOFFS, a *file name* SPNAM and a *phase alignment* SPOAL.

File Name

The name of a shape file. A shape file can be generated with the command *st*. or from the Shape Tool interface (command *stdisp*). Shape files are stored on disk in the so called JCAMP format. They reside in the directory:

$xwinnmrhome/exp/stan/nmr/lists/wave/

After its header, a shape file contains a list of entries, one entry for each pulse shape interval. Each entry consists of an amplitude value (in percent) and a phase value (in degree). The amplitude value defines the percentage of the absolute power value (see below).

Offset Frequency [Hz] SPOFFS

The shape offset frequency allows you to shift the frequency of the shaped pulse by a certain amount (in Hertz). This shift is realized by applying phase changes during the shaped pulse's time intervals. In this way, phase coherency of the frequency is maintained.

Power Value [W] SPW or [dB] SPdB

This is the absolute power value of the pulse shape when the amplitude of the shape is at 100%. The actual power value of a particular shape interval is the absolute power value multiplied by the relative power value of that interval, as specified in the shape file. The power of the shape is set in the interval before the start of the shape (in this case PLW[1] for channel F1). The power setting which was used before the shape is lost. (For AV I and II instruments it is not possible to execute pulses immediately before and after the shape, because there must be delays in which the power setting is done. These delays must be 3μs before and after the shape and 4μs between two shapes; for AV III instruments no such extra delay is required, but the minimum length of a shape segment which is normally 25ns can be executed only if there is no power setting or phase setting associated with this shape.) After the shape the power setting remains as in the last slice of the shape. For example, in pulse program below the pulse p2 would be executed with zero amplitude if the shape sp0 would be a sine.

```
d20 pl9:f1                ; set power on channel f1 to PLW[9]
p1:sp0:f1                 ; shaped pulse with abs. power SPW[0]
d11
p2:f1                     ; rectangular pulse with power SPW[0]
```

Rather than using power values specified in SHAPE, you can also use the power value that is currently active on the channel that you use. You can do that with the `(currentpower)` modifier of the `sp` statement as shown the following example:

```
d20 pl9:f1                ; set power on channel f1 to PLW[9]
p1:sp0(currentpower):f1   ; shaped pulse with abs. power PLW[9]
p2:f1                     ; rectangular pulse with power PLW[9]
```

The advantage of this method is that you can use pulses immediately before and after the shape.

You can access the SHAPE table entries from **eda**. However, you can also set the entries from the command line. For example, *spnam5* allows you to set the file name of entry 5, *spoffs2* sets the frequency offset of entry 2 and *spw15* sets the absolute power value of entry 15.

Phase Alignment

Shapes with frequency offsets do vary the phase in order to shift the frequency. By this variation the total phase of the shape is affected as well. The parameter SPOAL[0..63] determines whether the phase is aligned relative to the start or the end of the pulse. SPOAL has a range of 0 to 1. If SPOAL = 0, the relative phase shift is 0 at the beginning of the pulse whereas it is determined by the frequency offset SPOFFS and the pulse length at the end of the pulse (this is used for “flip back” type pulses). If SPOAL = 1 the relative phase shift is 0 at the end of the pulse (this is generally used for excitation pulses, and for SPOAL=0.5 the phase shift is zero in the middle of the pulse (used for refocussing pulses).

Using Shapes with Variable Pulse Length

A shaped pulse can be used in connection with a variable duration. For example, the pulse `p1` has a duration P[1] and can be varied with statements like `ipu1` or `"p1=p1+0.5m"`.

For a shape consisting of 1000 points the following restrictions apply:

the minimum execution length is 1000*25 ns = 25 μsec.

When a shape is specified too short or too long, an error message will be printed.

The length of a shaped pulse can be varied with a statement like:

```
ipu1
```

or with a relation like:

```
"p1 = p1 + 0.5m"
```

Note that varying the length of a shape with non-zero offset frequency during the run of a pulse program will rescale the phase shift of the shape correctly because a recalculation is performed during run-time with respect to the actual pulse length. For example, the following construct is possible, yielding the desired offset frequency for each loop cycle:

```
define list<shape> SPL=<wavelist>

define list<pulse> Plist=<$VPLIST>


1 ze

...

(Plist:SPL ph2):f1
```

3.4.4 Shape Lists

Instead of a single shape, one can also use different shapes from a shape list and toggle through the list with the list increment command.

```
define list<shape> shl=<wavelist>
```

This statement defines a new object called `shl` which represents a shape list (with the name “wavelist” in this example) and can be used in the same way as the normal shapes `sp0-sp63`.

With the command

```
shl.inc
```

the next item from the shape list becomes active.

Shape list files are stored in the directory <XWINNMRHOME>/exp/stan/nmr/lists/sp.

A shape list file has a format similar to the shape parameter list in the editor where each entry consists of a line containing 4 items. The next table shows a list containing the same entry twice, the first defining the shape power in W, the second in dB.

Power[dB or W]	Offset [Hz]	Offset alignment	Shape file name
1.0 W	0.0	0.5	Gauss
0.0 dB	0.0	0.5	Gauss

Table 3.3: An example of a shape pulse list

3.4.5 Amplitude Lists

You can define amplitude lists in a pulse program, for example:

```
define list<amplitude> am1={70}
```

The amplitude values represent the percentage of the power of a rectangular pulse. The above list is interpreted by a statement like:

```
d11 am1:f1
```

which reduces the power on the f1 channel to 70% of PLW[1] (assuming it was at its default value PLW[1]). All rectangular pulses on f1 will then be executed with this reduced power. A statement like:

```
d12 pl1:f1
```

will reset the power on channel f1 to 100% of PLW[1]. Furthermore, a shape pulse like

```
p11:sp1:f1 ph1
```

sets the power on f1 to the power SPW[1] and the last amplitude of the shape after it has finished.

An example of a pulse program segment using an amplitude list is:

```
define list<amplitude> am1={70}

p1 ph1                  ; rectangular pulse on f1 with power PLW[1]
d11 am1:f1              ; set the power on f1 to 70% of PLW[1]
p1 ph2                  ; rectangular pulse on f1 with 70% of PLW[1]
...
```

Amplitudes can be set also with the following statements;

```
d11 amp=amp1:f1     ; set the amplitude on f1 to the value AMP[1]
d11 amp=cnst2:f1    ; set the amplitude on f1 to the value CNST[2]
d11 amp=70:f1       ; set the amplitude on f1 to 70%
```

4 Delays

4.1 Delay Generation

The next table shows the available types of statements for the generation of delays. The duration of a delay corresponds to the name of the delay statement.

`d0, d1, ... , d63`	Generate a delay whose duration is taken from the acquisition parameter D[0], ..., D[63], respectively.
`3.5 u, 10m, 0.1s`	Generate a delay of fixed length: u = µsec , m = msec, s = sec.
`compensationTime`	Generate a delay whose name is defined with a `define delay` statement, and whose duration is defined by an expression.
`de1, de, depa, derx, deadc`	Generate a delay of length DE1, DE, DEPA, DERX, DEADC, respectively.
`dw, dwov`	Generate a delay of length DW, DWOV.
`aq`	Generate a delay of length AQ.
`acqt0 (or acqt0_F1)`	Determines 0-point of FID. The actual start of the FID measurement begins after this time.
`acqt0_F2, acqt0_F3, … acqt0_F8`	Same as `acqt0`, but for logical receiver channels 2-8; this may be used for **baseopt** acquisitions in multi-receiver experiments

Table 4.1: Delay generation statements

4.1.1 d0-d63

The statement:

```
d0
```

executes a delay of width D[0], where D[0] is an acquisition parameter. It is set from **eda**, or by typing `d0` on the command line. Likewise, the statement:

```
d1
```

executes a delay of width D[1].

4.1.2 Random Delays

The statement:

```
d0:r
```

executes a delay of width D[0] which is varied randomly. The parameter V9 specifies, in percent, the maximum amount which is added to or subtracted from D[0]. As such, the effective delay varies between 0 and 2*D0. It can be set from **eda**, or by typing *v9* on the command line.

Please note that the *gs* command ignores the `:r` option.

4.1.3 Fixed Length Delays

The statement:

```
10m
```

executes a delay of 10 ms (called a *fixed delay* because its duration cannot be manipulated, see below). The duration must be followed by `u`, `m`, or `s`. These units indicate microseconds, milliseconds, and seconds, respectively.

4.1.4 User Defined Delays

The statement:

```
define delay compTime
```

defines `compTime` to be a delay statement and the statement:

```
"compTime=d1*0.33".
```

is the expression that defines its duration. Note that the double-quote characters ("…") are mandatory.

With the above statements, the statement:

```
compTime
```

executes a delay whose name is defined in the pulse program, and whose duration is determined by an arithmetic expression. The `define` statement must be inserted somewhere at the beginning of the pulse program, before the actual pulse sequence. The defining expression must also occur before the actual pulse sequence. It is evaluated at compile time of the pulse program, not at run time.

Names for user defined delays must consist of alphanumeric characters, and the first character must be an alphabetic character. The maximum length of the name is 11 characters. Caution, do not use any of the reserved words like ‚`adc`', ‚`go`', ‚`pulse`' etc. as a delay name.

4.1.5 User Defined Delay Lists

A list of delays can be specified with a `define` statement in the following way:

```
define list<delay> Dlist = { 0.1 0.2 0.3 }
```

This statement defines the delay list `Dlist` with the values 0.1sec, 0.2sec and 0.3sec. Instead of delay values, you can specify a list filename in the defined statement. There are two way of doing this: you can specify the actual filename or $VDLIST, both in <>. In the latter case, the file defined by the acquisition parameter VDLIST is used. For example:

```
define list<delay> D2list = <mydelaylist>
define list<delay> D3list = <$VDLIST>
```

In both cases, the file can be created or modified with the command **edlist vd**.

In a pulse program that contains the statements above, the statement:

```
D1list
```

executes a delay of 0.1 seconds the first time it is invoked. In order to access different list entries, the list index can be incremented by adding `.inc`, decremented by adding `.dec` or reset by adding `.res`. Index operations are performed modulo the length of the list, i.e. when the pointer reaches the last entry of a list the next increment will move it to the first entry.

The operators `.len` and `.max` can be used in relations to access the length of the list or even its respective maximum value (see examples below).

Furthermore, a particular list entry can be specified as an argument, in square brackets, to the list name. For example, the statement:

```
D1list[1]
```

executes a delay of 0.2 seconds. Note that the index runs from 0 to n-1, where n is the number of list entries.

Lists can also be executed and incremented with one statement, using the caret postfix operator. For example, the statement:

```
D1list^
```

is equivalent to:

```
D1list D1list.inc
```

Finally, you can set the index with an arithmetic expression within double quotes using `.idx` postfix. The following example shows the usage of an initialized delay list:

```
define list<delay> locallist = {0.1 0.2 0.3 0.4}

locallist locallist.inc  ; delay of 0.1s, set index from 0 to 1
locallist locallist.res  ; delay of 0.2s, set index to 0
locallist[2]             ; delay of 0.3s
locallist locallist.dec  ; delay of 0.1s, set index from 0 to 3
locallist                ; delay of 0.4s
"locallist.idx = 3"      ; set index to 3
"l3 = locallist.len"     ; set loop counter 3 to length of list
                         ; (which is ‘4’ in our example)
"d3 = locallist.max"     ; set delay 3 to maximum value of list
                         ; (which is ‘0.4’ in our example)
locallist^               ; delay of 0.4s, set index from 3 to 0
locallist                ; delay of 0.1s
```

Note that there are two restrictions on the multiple use of delay lists within the same line:

- Index operations take effect from the next line on
- Furthermore, you cannot access two different entries of the same list in one pulse program line as illustrated in the following example:

```
locallist^ locallist        ; executes the first list entry (0.1s) twice
locallist                   ; increment takes effect now (0.2s delay)
locallist[2] locallist[3]; executes the third entry (0.4s) twice
```

Note that names for user defined items may consist of up to 19 characters but only the 7 first are interpreted: i.e `Delaylist1` and `Delaylist2` are allowed names but would address the same symbol.

4.1.6 Special Purpose Delays

These are the delay statements `de1, depa, derx, deadc, de` as listed in table *Delay Generation* [▶ 37]. They are used in pulse programs in which the acquisition is started with the `adc` statement rather than with `go=label`. Implicitly they are used in the `go` macro as well and can be set in the edscon table or explicitly within the pulse program. `dw` and `aq` are determined by setting the sweep width and cannot be redefined within the pulse program.

The delay `acqt0` is a delay which is not used by the pulse program compiler but in connection with the *baseopt* option of DIGMOD. It serves to determine the point where t=0 for the FID. This point is for a simple zg program somewhere in the middle of the excitation pulse. One therefore defines this delay as

```
"acqt0= -p1*2/pi"
```

The linear prediction software takes this value to reconstruct the part of the FID which could not be measured.

4.2 Manipulating Delays

4.2.1 The Operator *

A delay can be manipulated by the “*“ operator. Examples of allowed statements are:

```
d1*1.5
compensationTime*3.33
d3*oneThird
```

The * operator must be specified behind the delay statement, not before. `oneThird` is the name of a macro that must be defined at the beginning of the pulse program with a statement like `#define oneThird 0.33`. Note that a statement like `10m*0.33` would be incorrect, since `10m` is a fixed delay.

4.2.2 Changing d0-d63 by a Constant Value

The delays executed by `d0-d63` can be incremented or decremented according to the acquisition parameters IN[0]-IN[63]. These parameters contain a duration (in seconds). The pulse program statements `id0-id63` add IN[0]-IN[63] to the current value of `d0-d63`, respectively. Likewise, `dd0-dd63` subtract IN[0]-IN[63] from the current value of `d0-d63`. The statements `rd0-rd63` reset `d0-d63` to their original value, i.e. to the values of the parameters D[0]-D[63]. The statements presented in this paragraph must be specified behind a delay of any length. Examples:

```
d1 id3
0.1u dd0
d1 rd0
```

In Bruker pulse programs, D[0] and D[10] are used as incremental delays for 2D and 3D experiments, IN[0] and IN[10] are the respective increments which are used. IN[0] can be set in the pulse program by a calculation:

```
"in0=inf1/2"
```

Here the parameter INF1= 1/SW(F1) represents the dwell time for this dimension (see the description of IN0, INF1 in the Acquisition Reference manual). You can also set the sweep width in F1 by calculating INF1 within the pulse program.

Please note that the parameter INF contains a delay in microseconds and is internally rounded to 12.5ns after each calculation. This may also play a role if `inf` is used for calculation in the pulse program.

4.2.3 Redefining d0-d63

The duration of the `d0`-`d63` statements is normally given by the parameters D[0]- D[63]. However, you can overwrite these values in the pulse program using an expression in C language syntax. The following examples show some of the possibilities:

```
"d13=3s + aq - dw*10"
"d13=d13 + (p1*3.5 + d2/5.7)*td"
```

The result of such an expression must have a time dimension. You can therefore include acquisition parameters such as pulses, fixed pulses, delays, fixed delays, acquisition time AQ, dwell time DW etc. within the expression. Furthermore, you can include parameters without a dimension such as the time domain size TD. The complete list is shown in the table *Parameters in relations* [▸ 67]. An expression must be double-quoted (“ “). It can be inserted anywhere in the pulse program, as long as it occurs before the delay statement that uses the expression (`d13` in our example). Please note that the second expression in the example above assigns a new value to `d13` each time the expression is encountered, for example in a loop.

4.2.4 Manipulating the Durations of User Defined Delays

You can define your own delay statements using a `define` statement like:

```
define delay compensationTime
```

at the beginning of the pulse program. This delay is executed by the statement:

```
compensationTime.
```

The delay length must be defined with a statement like:

```
"compensationTime=d1*0.33".
```

For such an expression the same rules apply as for the manipulation of `d0`-`d63`, described in the previous section.

The defining expression of a user defined delay must occur before the start of the actual pulse sequence. It is evaluated at compile time of the pulse program, but you can redefine it at run time.

4.2.5 max, min and random

In order to avoid complicated expressions for the assignment of delays and pulses, the comparison of two delays and/or pulses can be made like this:

```
define delay delta
"delta=max(p4,d2)"
```

In this expression the length of p4 and d2 are compared, and the delay `delta` will become equal to the longer one.

Furthermore, there is also a minimum function where delta will become equal to the shorter value of p4 and d2, respectively:

```
"delta=min(p4,d2)"
```

Possible units are also considered, e.g.

```
“delta = max(4u, 2m)”
```

Will set `delta` to 2m.

In order to randomize a delay or a pulse only at a certain point of the program, one can use the following:

```
"p4=random(p1,20)"
```

which will assign p4 the value of p1*(100+20*R)/100, where R is a random number in the range of -1 to +1. This can be used, where the pulse `p4` should be changed not each time it is used, but for instance only after ns scans. (The usage of `p4:r` within the pulse program randomizes the pulse each time when it occurs).

```
(p1 ph1 100u):f1
(p2 ph2):f2
```

executes a pulse on channel f1, followed by a delay, followed by a pulse on channel f2.

5 Simultaneous Pulses and Delays

5.1 Rules

The following rules apply in pulse programs:

1. Pulses and delays specified on subsequent lines are executed sequentially.
2. Pulses and delays which are specified on the same line, and which are enclosed in the same set of parentheses or without parenthesis are executed sequentially. Such a sequence is called *pulse train* in the following.
3. Pulse trains on the same line are executed simultaneously. The first item within a pulse train is started at the same time as the first item in any other pulse train. You can specify an arbitrary number of sets of parentheses on a line.
4. Pulse trains on different lines which are enclosed by an extra set of parentheses are executed simultaneously.

5.2 Examples

5.2.1 Rule 1

The pulse program section

```
(p1 ph1):f1
100u
(p2 ph2):f2
```

executes a pulse on channel `f1`, followed by a delay, followed by a pulse on channel `f2`.

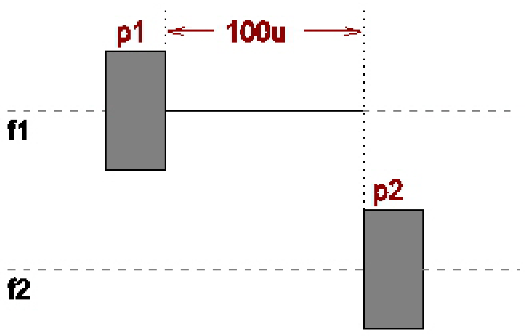

Figure 5.1: An example for rules 1 and 2

5.2.2 Rule 2

The pulse program section

```
(p1 ph1 100u):f1
(p2 ph2):f2
```

executes a pulse on channel `f1`, followed by a delay, followed by a pulse on channel `f2`, see figure *Figure 5.1 [▸ 43]*.

See also

- Rule 1 [▹ 43]

5.2.3 Rule 3

The pulse program section

```
(p1 ph1):f1 (100u)
(p2 ph2):f2
```

executes a pulse on channel `f1`.

At the same time, the 100 µsec delay begins, since it is enclosed in a separate set of parentheses. The pulse on channel `f2` is not executed before either `p1` or `100u` have passed, whichever is longer, see the next figure.

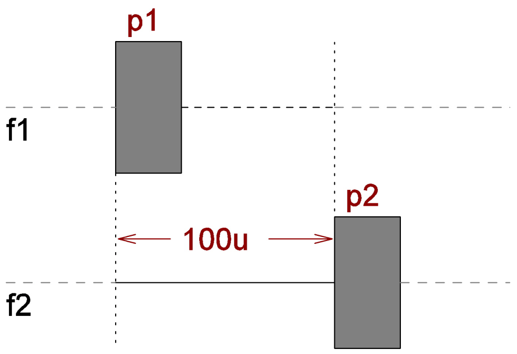

Figure 5.2: An example for rule 3

The following example is a typical section of a DEPT pulse program:

```
(p4 ph2):f2 (p1 ph4 d2):f1
(p0 ph3):f2 (p2 ph5):f1
```

The pulses `p4` and `p1` begin at the same time, `p4` on channel `f2` and `p1` on channel `f1`. The pulses `p0` and `p2` start simultaneously, but not before the sequence with the longest duration of the previous line has finished, see figure *Figure 5.3 [▹ 45]*.

The following two lines have been extracted from the `colocqf` Bruker pulse program.

```
(d6) (d0 p4 ph2):f2 (d0 p2 ph4):f1
(p3 ph3):f2 (p1 ph5):f1
```

We have three sets of parentheses in this case. The first item in each set of parentheses, i.e. `d6` and `d0`, start at the same time. After `d0`, `p4` on channel `f2` and `p2` on channel `f1` start simultaneously. Assuming that `d6` is larger than `d0`+`p4` and `d0`+`p2`, the second line is executed after `d6` has finished.

A final example for rule 3 is a line from the `hncocagp3d` Bruker pulse program:

```
(p13:sp4 ph1):f2 (p21 ph1):f3
```

The shaped pulse `p13` on channel `f2` is started simultaneously with the rectangular pulse `p21` on channel `f3`.

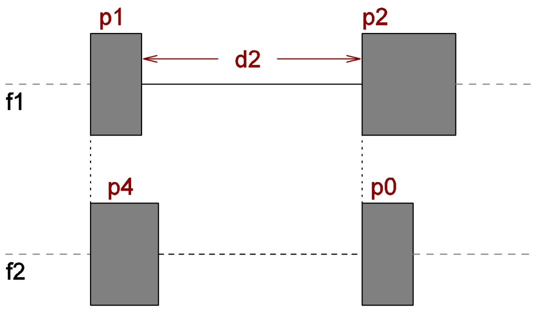

Figure 5.3: An example for rule 3 - DEPT

See also

- Rule 3 [▸ 44]

5.2.4 Rule 4

The example in the previous section can be rewritten according to this rule

```
(
  (d6)
  (d0 p4 ph2):f2
  (d0 p2 ph4):f1
)
(p3 ph3):f2 (p1 ph5):f1
```

It still produces the same pulse sequence.

5.3 Pulse Train Alignment

Pulse trains written in the style of rule 4 can be aligned in different ways.

5.3.1 Global Alignment

There are three global alignment possibilities for the pulse trains: left alignment (`lalign`) is the default, right alignment (`ralign`) will arrange the pulse sequences such that all of them end at the same time, and center alignment (`center`) will start the pulse trains in such a way that they are centered according to the mid-point of the longest of them. The global alignment must be specified after the first opening bracket:

```
(center
  (d6)
  (d0 p4 ph2):f2
  (d0 p2 ph4):f1
)
```

5.3.2 Individual Alignment

Single pulse trains can be aligned individually as well. In this case, the first pulse train in a sequence must be defined as the reference (`refalign`):

```
(
  refalign (d0 p1 ph1 d0):f1
  center (p2 ph2):f2
  ralign (p3 ph4):f3
)
```

Pulse train 3 is now right-aligned relative to pulse train 1, pulse train 2 is centered relative to pulse train 1. From this piece of code several situations may result which are not obvious but best can be shown graphically: only in 2 of the 6 possible cases the sequence begins with the reference pulse train, see the next figures.

If the length of individual pulse trains change during pulse program evolution, the alignment conditions will still be true.

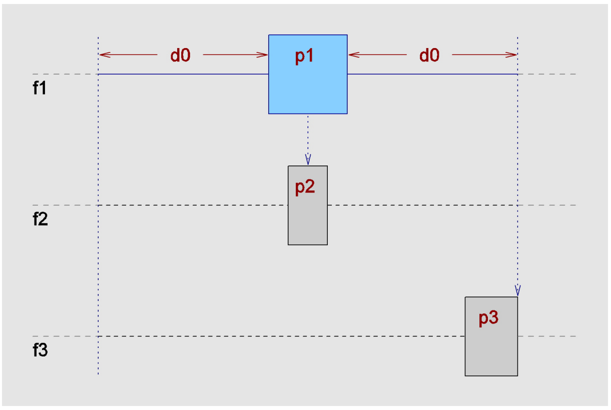

*Figure 5.4: Individual alignment with 2*d0+p1 > p3 > p2*

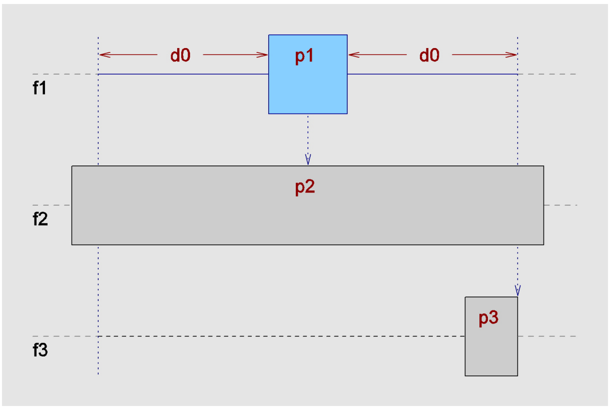

*Figure 5.5: Individual alignment with p2 > 2*d0+p1 > p3*

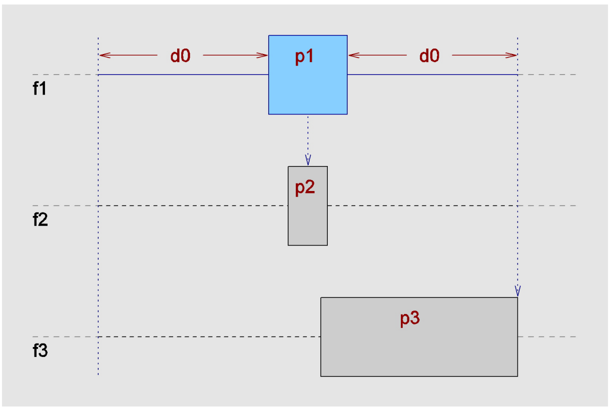

Figure 5.6: Individual alignment with d0+p1/2 > p3 > p2

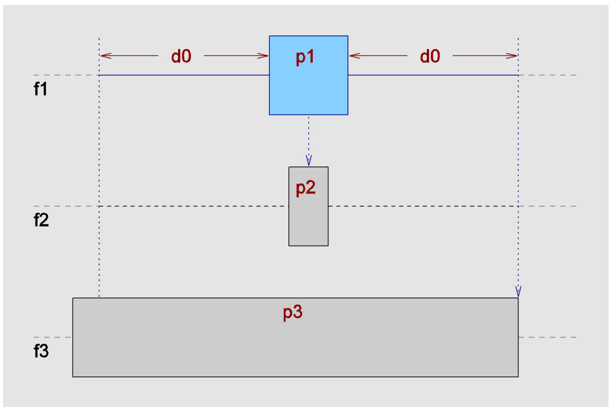

*Figure 5.7: Individual alignment with p3 > 2*d0+p1 > p2*

6 Decoupling

6.1 Decoupling Statements

The next table shows the available types of decoupling statements. Composite pulse decoupling is discussed in more detail in the next section of this chapter.

`cw`	continuous wave decoupling
`hd`	homodecoupling
`cpds1,..,cpds8`	composite pulse decoupling with CPD sequence 1, ..., 8, synchronous mode
`cpd1,..,cpd8`	composite pulse decoupling with CPD sequence 1, ..., 8, asynchronous mode
`cpdhd1,..,cpdhd8`	homodecoupling using CPD seq. 1,...
`do`	switch decoupling off

Table 6.1: Decoupling statements

Each pulse program line can contain one (and only one) decoupling statement. For example, the line:

```
d1 cw:f1
```

turns on cw-decoupling on channel `f1` at the beginning of delay `d1`. The line:

```
go=2 cpds1:f2
```

turns on composite pulse decoupling on channel `f2` at the start of the FID detection.

Decoupling statements are allowed in pulse program lines that contain a delay statement or the `go` statement, but not in lines with pulses, expressions or any lines with any of the statements `adc`, `rcyc`, `lo`, `if` or `goto`.

Once decoupling is turned on, it remains on, until it is explicitly turned off with `do`. For example:

```
0.1u do:f1
```

turns decoupling off on channel `f1`.

cpdhd1..8 is used for homodecoupling during the acquisition. It uses the cpd programs CPDPRG[1..8] but choppsin the HD manner. The decoupling is turned off after the scan automatically and no `do` statement is required:

```
go=2 ph31 cpdhd1:f2
```

6.2 Decoupling Frequency

The decoupling frequency is selected by specifying the spectrometer channel behind the decoupling statement. In contrast to pulse statements, decoupling statements must be specified with a channel. For example:

```
d1 cw:f2
```

turns on cw decoupling on channel `f2`, i.e. with the frequency SFO2. This syntax is the same as used for selecting pulse frequencies (see the chapter *Pulse Frequency* [▸ 21]). The statement:

```
0.1u cpds1:f3
```

turns on the composite decoupling sequence 1 on channel `f3`, i.e. with the frequency SFO3. The statement:

```
3m do:f3
```

terminates decoupling on channel `f3` at the beginning of the 3 ms delay.

6.3 Decoupling Phase

The relative phase of the decoupling frequency can be controlled using a phase program. This is equivalent to controlling the phases of pulses (see chapter *Pulse Phase [▸ 25]*). Examples:

```
(d1 cpds1 ph2):f3
(0.1u cw ph1):f2
```

Note that phase cycling (see chapter *Phase Cycling [▸ 26]*) is applied to phase programs specified behind decoupling statements in the same way that phase programs are specified with pulses.

A simple example demonstrating this feature is the pulse program section:

```
(1m cw ph1):f2
d1 do:f2
```

which is equivalent to:

```
(1mp ph1):f2
d1
```

A 1 millisecond pulse is executed on channel `f2`, followed by a delay `d1`. Its phase is cycled according to phase program `ph1`.

6.4 Composite Pulse Decoupling (CPD)

6.4.1 General

Composite pulse decoupling, as opposed to cw and hd decoupling, offers a large degree of freedom to set up your own decoupling pulse sequences. Up to 8 different CPD sequences can be used in a pulse program. For example, the line:

```
d1 (cpds1 ph2):f3 (cpds2 ph4):f2
```

starts, at the beginning of duration `d1`, CPD sequence 1 on channel `f3` and, simultaneously, CPD sequence 2 on channel `f2`. CPD sequence 1 is obtained from a text file defined by the acquisition parameter CPDPRG[1]. Likewise, CPD sequence 2 is obtained from a text file defined by the acquisition parameter CPDPRG[2], etc. A CPD sequence (= CPD program) can be a Bruker delivered sequence like WALTZ16, GARP or BB or it can be a user defined sequence. CPD sequences can be set up with the command `edcpd` (as described in the Acquisition Reference Manual).

`cpds1`, ... , `cpds8`	Start decoupling using the CPD program CPDPRG1, ... , CPDPRG8. The decoupling sequence will start at line 1.
`cpd1`, ... , `cpd8`	Like `cpds1`-`cpds8`, however, the decoupling sequence will continue from the line where it was stopped using `do`.
`:f1`, ..., `:f8`	Channel selector. To be appended to the `cpd` statements.

`cpdngs1, … ,cpdngs8`
`cpdng1, … ,cpdng8`
	Same as the `cpd(s)` statements above, except that the transmitter gate for the specified channel will not be opened. Gating is controlled by the main pulse program, and can be tailored by the user.

Table 6.2: Available CPD statements

6.4.2 Syntax of CPD Sequences

The syntax of CPD sequences is demonstrated by examples. Table *Table 6.4 [▸ 52]* shows the realization of Broadband and Garp decoupling with CPD sequences. Each sequence is an infinite loop as indicated by the last statement:

```
jump to 1
```

As in pulse programs, the pulse width in CPD programs can be specified as a *fixed pulse*, (e.g. `850up`) or with the statements `p0`-`p63`. The `Garp` sequence shows the usage of the `lo to` statement. The next table shows the statements available to build a CPD sequence.

`p0,.., p63`	Generate pulses with durations `P[0], … , P[63].`
`10up, 5mp, 2.5sp`	Generate pulses in micro-, milli-, and seconds
`pcpd1, … ,pcpd8`	Generate a pulse with duration according to `PCPD[1], … , PCPD[8]`, depending on the channel where the CPD sequence is executed (use **eda** to set PCPD).
`d0, … , d63`	Generate delays with durations `D[0], … , D[63].`
`10u, 5m, 2.5s`	Generate delays in micro-, milli-, and seconds.
`*3.5 :135.5`	Multiplier. Can be appended to `p0-p63` or `d0-d63`. Phase in degrees. Can be appended to pulses.
`:sp0, … , :sp63`	Shaped pulse selectors. Can be appended to pulses.

`pl=5[Watt] pl=sp13`
`pl=pl25`
	Power specifier: in Watt according to shaped pulse parameters `SPW[0]-[63]` according to `PLW[0]-[63]`

`fq=2357 fq=cnst25`
`fq=fq2`
	Frequency change in Hz (relative to SFO1 for channel 1, SFO2 for 2...) from the parameters `CNST[0]-[63]` from the frequency list specified in `FQ2LIST`

`;`	Begin of a comment (until end of line)
`go to label`	Branch to `label` (forward, see the chapter *Manipulation of Loop Counters during Execution (BILEV Decoupling) [▸ 54]*)
`lo to label times n`	Loop to `label n` times.
`jump to label`	Branch to `label` (backward). Usually the last statement.
`#addphase #setphase`	Special phase control statements

Table 6.3: Statements available to build CPD sequences

The Garp sequence, as well as the sequences in table *Table 6.5 [▸ 52]* use the statement `pcpd` to generate pulses. This enables the execution of the same sequence for different nuclei on different channels. For example, when executed on channel `f2` (`f3`), the pulse duration of `pcpd` is given by the parameter PCPD[2] and PCPD[3], respectively. This allows to specify the 90° pulse width for two different nuclei in `PCPD[2]` and `PCPD[3]`, and decouple both nuclei within the same pulse program using the same CPD program.

Table *Table 6.5 [▸ 52]* shows two CPD sequences based on shaped pulses. Shapes are specified in the same way they are specified in pulse programs using the :`sp0`, ... , :`sp63` pulse selector options. The examples demonstrate the order in which duration multiplier, shape selector and phase must be specified.

190up:0	1 pcpd*0.339:0
160up:180	pcpd*0.613:180
240up:0	pcpd*2.864:0
....	pcpd*2.981:180
570up:0	pcpd*0.770:0
680up:180	....
810up:0	pcpd*0.593:0
960up:180	lo to 1 times 2
1140up:0	2 pcpd*0.339:180
1000up:180	pcpd*0.613:0
850up:0	....
710up:180	pcpd*2.843:180
....	pcpd*0.729:0
200up:0	pcpd*0.593:180
110up:180	lo to 2 times 2
jump to 1	jump to 1

Table 6.4: Broadband and GARP CPD sequences

1 pcpd*2:sp15:0	1 pcpd*14.156:sp15:60
pcpd*2:sp15:0	pcpd*14.156:sp15:150
pcpd*2:sp15:180	pcpd*14.156:sp15:0
pcpd*2:sp15:180	pcpd*14.156:sp15:150
pcpd*2:sp15:180	pcpd*14.156:sp15:60
pcpd*2:sp15:0	2 pcpd*14.156:sp15:240
pcpd*2:sp15:0	pcpd*14.156:sp15:330
pcpd*2:sp15:180	pcpd*14.156:sp15:180
pcpd*2:sp15:180	pcpd*14.156:sp15:330
pcpd*2:sp15:180	pcpd*14.156:sp15:240
pcpd*2:sp15:0	lo to 2 times 2
pcpd*2:sp15:0	3 pcpd*14.156:sp15:60
pcpd*2:sp15:0	pcpd*14.156:sp15:150
pcpd*2:sp15:180	pcpd*14.156:sp15:0
pcpd*2:sp15:180	pcpd*14.156:sp15:150
pcpd*2:sp15:0	pcpd*14.156:sp15:60
jump to 1	jump to 1

Table 6.5: MLEVSP and MPF7 CPD sequences

The sequences in the tables above *Table 6.4 [▸ 52]* and *Table 6.5 [▸ 52]* do not contain a power setting statement. Therefore, the current power setting of the main pulse program for the respective channel is valid.

`cpdngs` is used for sequences which do not need the transmitter. For instance the dwell time can be generated in this way (see pulse program zgadc in the Bruker library).

6.4.3 Phase Programs in CPD Programs

In AV III systems phase programs can be used and defined in the same way as in the main pulse program. Phase program phases are added to the phases defined with the CPD program and to the phases defined in the shapes. Phase programs defined in CPD programs and phase programs defined in the main pulse program can be used in both programs. In CPD programs the incrementation of phase program pointers must be done with the autoincrement command. Example:

```
1 p15:sp15 ph21^
  jump to 1
ph21=(360) 0 15 165 180
```

The phase program pointer of ph21 can be manipulated in the main program as well if no autoincrement is used or explicitly with the command `ipp21`. With this syntax the performance of the CPD program is best if the total length of the CPD sequence is adjusted such that shapes are not cut into pieces by the main program. This can be achieved as illustrated in the following example:

```
define loopcounter adia
"adia=aq/p15"
define delay aqround
"aqround=p15*adia+1"

1 ze
d11 pl12:f2
2 30m
d1
p1 ph1
ACQ_START(ph30, ph31)
aqround DWELL_GEN:f1 cpd2:f2
100u do:f2
rcyc=2
```

6.4.4 Frequency Setting in CPD Programs

There are three ways to change the frequency of the channel where the CPD sequence is applied. Frequency setting in CPD programs is the same as in pulse programs except that the channel specification after the statement is not necessary.

6.4.5 Frequency Setting from Lists

The first method to set the frequency is using a frequency list. The statements `fq1`- `fq8` interpret the parameters FQ1LIST-FQ8LIST, set the frequency from the current list entry and move the list pointer to the next entry. In contrast to the lists used in pulse programs, lists used in CPD programs are expanded at compile time, not at run time. In the following example, the first `fq1` statement uses the first entry of the frequency list, the next statement the second entry. If the frequency list contains more than 2 entries, only the first two will be used.

```
1 pcpd:0 fq=fq1
pcpd:180 fq=fq1
jump to 1
```

Like in pulse programs, the frequency offset can be specified in two ways: either the offset is at the top of the list in MHz, or no offset is specified in the list. In the latter case, the measure frequency of the appropriate channel (SFO1 for `f1`, SFO2 for `f2`, etc.) is used as list offset.

6.4.6 Frequency Setting Using the Parameters CNST0-63

The statement `fq=cnst25` will set the frequency SFO1 + CNST25 [Hz]. The parameter CNST25 can also be modified from the `gs` window. If used on channel `f2`, the basic frequency SFO2 instead of SFO1 will be used etc.

6.4.7 Offset Frequencies

You can also use the offset frequencies O1 to O8 (in [Hz]) as an rvalue directly in a relation in the pulse program.

For example,

```
"spoffs25=bf1*(cnst54/1000000)-o1"
```

uses O1 to calculate the offset frequency of a shaped pulse.

On the other hand it may also be necessary to calculate and store O1 for indirect dimensions directly from within the pulse program in order allow a correct automatic processing later on.

In this case, the parameters `o1_F1` to `o1_F8` [Hz] can be used with F denoting the indirect dimensions as defined in eda, for example

```
“o1_F1=cnst25”
```

In case of a 2D experiment, only `o1_F1` can be used whereas in case of a 3D experiment only `o1_F1` and `o1_F2` can be used etc.

Assignments for indirect dimensions that actually do not exist for the current experiment are silently ignored.

6.4.8 Direct Specification of Frequencies

The statement `fq=3000` will set the frequency SFO1(2,3...) + 3000 Hz.

6.4.9 Loop Statements in CPD Programs

The general form of a loop statement is:

```
lo to label times n
```

where `label` can be any number. The loop counter `n` can be a number or a symbolic loop counter `l0 - l31`, where the latter interprets the parameters L[0] - L[31]. It must be equal to or greater than 1.

User defined loop counters from the pulse program can also be used in the CPD program.

For infinite CPD programs (which are terminated from the pulse program by the statement `do:fn`) there is a special `jump to label` statement which executes an unconditional jump to the specified label. For calculated jumps forward (see next section), there exists the command `go to ln` where `ln` is a loop counter `L0 .. L31`.

6.4.10 Manipulation of Loop Counters during Execution (BILEV Decoupling)

You can manipulate the loop counter of a CPD program after each scan according to an arithmetic expression in the following way (this statement must be on the first line of the CPD program):

```
bilev: "l5=nsdone%4+1"
```

This means that the loop counter `l5` will be modified after each scan according to the above equation. The modification will take effect immediately after the scan. The expression should be written such that the loop counter is always greater than zero. The variable loop counter then can be used within the CPD program such that the first section changes with each scan. This can be done in two ways:

the CPD program contains a loop using this loop counter.

the CPD program uses a go to instruction to go to a calculated label:

```
bilev: "l31=nsdone%4+1"
go to l31
1 pcpd*3:180
  pcpd*4:0
...
2 pcpd*2:0
...
3 pcpd*3:180
...
```

The `go to l31` instruction has the effect that the CPD program continues at the label which corresponds to the current value of the loop counter calculated above. Of course, each possible loop counter value must have a corresponding label.

A `bilev` statement in a CPD program automatically changes the CPD statement in the pulse program into the corresponding `cpds`. This means that the CPD sequence is not continued at the point where it was stopped before, but starts from the beginning each time it is called.

6.4.11 Relations in CPD Programs

It is – to a limited extend – possible to use relations in the CPD program in order to modify pulses, delays, or loop counters.

For example, the following CPD program fragment calculates a pseudo-random value for `l31` and varies the length of the following decoupling loop.

```
1 1up

"l31=random(4,100)+0.5"

  1up

2 p60 :0 pl=pl31

  p60*2:90

  p60 :0

  lo to 2 times l31
```

7 Gradients

The term gradient refers to a magnetic field gradient that is added to the homogeneous field of the spectrometer magnet. A gradient is supplied by a gradient coil and can be applied in the x, y and/or z spatial dimension. If a gradient is applied in the x-dimension, the magnetic field will be constant within a y-z plane. In the y-z plane through the center of the receiver coil, the x-gradient field is zero. In a y-z plane at one edge of the receiver coil the x-gradient field is +M, whereas in a y-z plane at the opposite edge it is -M. Here, M is the maximum gradient strength which depends on the gradient amplifier. For y and z-gradients, the same principle holds concerning the x-z plane and x-y plane, respectively.

A rectangular gradient has a constant strength during the time it is applied, whereas a shaped gradient has a variable strength.

7.1 Rectangular Gradients

A rectangular gradient has a strength that is constant during its execution. It can be created with one of the statements `gron0`, `gron1`, ..., `gron31`. The statement `gron0` creates a gradient whose strength is determined by the parameters GPX0, GPY0 and GPZ0. Similarly, `gron1` creates a gradient whose strength is determined by the parameters GPX1, GPY1 and GPZ1 etc. The `groff` statement switches off all gradients that were switched on by a `gron*` statement.

For example, the pulse program section:

```
300u gron2
1m
100u groff
```

switches on a gradient defined by GPX2, GPY2 and GPZ2, at the beginning of a 300 µs delay. This gradient remains on during a period of 1.3 ms.

The parameters GPX0, GPY0 and GPZ0 can be set by entering **gpx0**, **gpy0**, **gpz0**, respectively, on the command line. As the gradient strength is expressed as a percentage of the maximum strength, it takes a values between 0 and 100. The parameter, GPX1, GPY1, GPX2 etc. can be set from the command line in a similar way. Alternatively, you can set all gradient parameters from the **eda** window by clicking the **GP031 edit** button.

7.1.1 Gradient Ramping

Of course in the real world it is not possible to switch *rectangular* gradients. Therefore it is possible to define a ramping time within

$XWINNMRHOME/exp/stan/nmr/parx/preemp/$PROBEHEADID/default

($PROBEHEADID is retrieved from *$XWINNMRHOME/conf/instr/probehead*.)

by editing the parameters RAMPTM.., e.g.

##$RAMPTM=100

RAMPTM > 0 means that a linear ramp is applied (in µs), whereas RAMPTP = 0 switches the gradients without ramping.

Please note that **gron/groff** commands always switch the gradients without ramping (whereas in ParaVision these commands actually do apply a linear ramp).

7.2 Shaped Gradients

A shaped gradient has a strength that varies during its execution. The gradient strength as a function of time is called the gradient shape. It is defined by a list of values between -1.0 and 1.0. The number of values in the list defines the number of time intervals. Each element in the list defines the relative gradient strength during a particular time interval. The interval length is defined by the length of the entire gradient shape divided by the number of intervals. The length of the shape (duration) must be specified in the pulse program The gradients are reset to zero at the end of the shape, if no gradient statement is immediately following.

The following 3 examples generate shaped gradients:

```
10mp:gp2
```

```
p1:gp1
```

```
gradPulse*3.33:gp3
```

They are applied for 10 ms, P[1], and gradPulse*3.33 and are described by the gradient parameter table entries 2, 1, and 3, respectively. This table can be opened by clicking on the **GP031** button in **eda**. It has 32 entries with indices 0- 31. The statements `:gp0` interprets entry 0, `:gp1` interprets entry 1, etc.

Each entry of the gradient parameter table has assigned 4 parameters: GPX, GPY, GPZ (the gradient strength multipliers for the 3 spatial dimensions), and a file name (of the file that contains the gradient strength values).

GPX, GPY, GPZ

These are multipliers with values between 0 to 100. They are applied to the gradient strength values (which range from -1.0 to 1.0) in the shape file to obtain the total gradient field strength.

File Name

File name is the name of a gradient file. A gradient file can be generated from Shape Tool window (command **stdisp**) or from the command line with the command **st** (for more information click *Help Online Manual* from the Shape Tool window).

Gradient shape files are stored in JCAMP-DX format in the directory:

$xwinnmrhome/exp/stan/nmr/lists/gp/

Note that if you specify an internal gradient shape, you don't need a shape file, however you should define the length of the shape as described below.

All gradient parameters can be set from the **eda** window by clicking the **GP031 edit** button. Alternatively, they can be set by entering **gpx0, gpy0, gpz0, gpnam0, gpx1** etc. on the command line. As the gradient strength is expressed as a percentage of the maximum strength, it takes on values between 0 and 100.

As described in the next section, you can also define gradient shape functions in the pulse program rather than using shaped gradient files.

7.3 Gradient Lists

You can multiply rectangular, ramped or shaped gradients with list values. At each moment, the gradient strength is calculated according to the current list value.

The list pointer can be manipulated in the following way:

`sin.res;`	use 1st list value
`sin.inc;`	use next list value

`sin.dec;`	use previous list value
`sin.store;`	save list pointer (stack with depth=1)
`sin.restore;`	restore the last saved list pointer

The gradient list is defined directly within the pulse program, read from a gradient file (like gradient shapes) or an internal function that is calculated during pulse program compilation.

In any case a corresponding `define list` command should be given at the beginning of the pulse program. For example:

`define list<grad_scalar> gl1 = { -1.0 0 0.5 0.75 .9 }`

`define list<grad_scalar> gl2 = <sin.100> ;` read from file

`define list<grad_scalar, 100> sin;` sine function with 100 values

Internal Gradient Functions:

The following internal functions are available:

1. `plusminus`

   can take the value 1 or -1.
2. `r1d, r2d and r3d`

   linear ramps from -1 to 1, where the final value is never reached.
3. `step`

   linear ramp from 0 to 1 and the final value will always be reached.
4. `sin`

   sine function from 0 to π (excluding π). The angle increment depends on the length of the function (see above).
5. `cos`

   cosine function from 0 to π (excluding π).
6. `sinp`

   sine function from 0 to π (including π).
7. `gauss <truncval>`

   which is a gaussian function with truncation level (e.g. `gauss2.5` for 2.5% truncation level)

7.4 Manipulation of Rectangular or Shaped Gradients

7.4.1 Manipulation with Functions

Both rectangular and shaped gradients can be manipulated with a constant and/or a gradient function. Here, manipulation can be addition or multiplication.

Example:

```
1 300m gron2 * - 0.5 * plusminus
p1:gp1 * sin(100) * cnst0
igrad plusminus
igrad sin
lo to 1 times 100
```

If a rectangular gradient is manipulated with a gradient function, the latter must be specified without parameters. For example:

```
300m gron2 *sin
```

If, however, a shaped pulse is manipulated with a function, the latter can be specified with or without parameters.

Example:

```
p1:gp1 * sin
p2:gp2 * sin(100)
```

7.4.2 Manipulation by Recalculation

The gradient parameters GPX0..31, GPY0..31, and GPZ0..31 can be recalculated in relations during compile time and during runtime.

Example:

```
“gpz27=cnst23*l2”
```

7.5 General Gradient Statements

Since the TopSpin gradient software is also used by **ParaVision**, it has features which were actually designed for medical imaging. With gradient statements of the form:

```
delay grad_ramp <options>{<1st dim>, <2nd dim>, <r3d dim>}
delay grad_shape<options>{<1st dim> , <2nd dim>, <r3d dim>}
delay grad_off <options>
```

you can use these features even without ParaVision, but in a restricted manner:

- You can specify *Object Oriented Gradients* that are converted into *Physical Gradients.* This allows for:
  - Acquisition of images with different *slice orientation* while using the same pulse program. The gradients may be specified in spatial coordinates other than x, y and z. The pulse program compiler multiplies the gradients with a rotation matrix (see below) to get x, y and z.
  - Acquisition of images with different slice thickness and field of view, every spatial dimension may be multiplied by a scaling factor.
- The gradients are defined as a percentage of maximum_gradient strength, as scalar values or functions, which may be combined by addition and multiplication.
- Gradient lists and shapes are either internal functions, which are handled accordingly by the compiler, or can be defined directly within the pulse program or read from *gradient files* containing the list or shape values (see above).
- Scaling and rotation can be suppressed with the following options:
  - *rps_coord*: Gradient is scaled and rotated. This is the default.
  - *magnet_coord*: Gradient is not scaled and not rotated at all.
  - *object_coord*: To support animal or human patient investigation it is possible to rotate the gradients according to the position within the magnet. Only this rotation is done.

Examples:

; Apply phase gradient (2nd dimension) = -100, -99, .., 99

; The gradients are switched via a linear ramp within 100 µsec.

```
define list<grad_scalar,200> r2d
1 ..
1m grad_ramp<100u> {0, r2d*100, 0}
1m grad_off<100u>
..
```

```
r2d.inc
lo to 1 times 200
```

; Apply a shaped gradient within 500us.

; The shape (indicated by brackets) is read from gradient_file SIINE.32.

; The gradient is rotated from the 1st to the 2nd dimension in 20 steps.

```
define list<grad_scalar, 20> sin
define list<grad_scalar, 20> cos
define list<grad_shape> gsh1 = <SINE.32>
2 ..
5m grad_shape<500u> {gsh1()*cos*50, gsh1()*sin*50, 0}
1m grad_off
..
sin.inc
cos.inc
lo to 2 times 20
```

; Apply three saturation slices within the object coordinate system,

; where each slice is defined by a gradient vector, containing the three ;dimensions.

; A vector is indicated by square brackets.

; Lists may contain values, but of course the finally

; value should not exceed 100 %.

; Since no ramp delay is given, the standard ramp is applied

;(see Gradient ramping).

```
define list <grad_vector> fovSat = { 10.0, 20.0, 30.0,
0.0, 0.0, 50.0,
50.0, 20.0, 0.32 }
3 ..
1m grad_ramp <object_coord> { fovSat[] }
fovSat.inc
lo to 3 times 3
```

7.6 Rotation and Scaling

Rotation and scaling of the gradients can be done with the elements of the acquisition parameter CAGPARS. The three first elements define the scaling in read, phase and slice dimension. The remaining 9 elements define a rotation matrix.

For example table *Table 7.1 [▸ 62]*

CAGPARS[0] = 0.5 means 1st (or read) dimension is scaled by 0.5

CAGPARS[1] = 0.5 means 2nd (or phase) dimension is scaled by 0.5

CAGPARS[2] = 0.8 means 3rd (or slice) dimension is scaled by 0.8

CAGPARS[3] = 0.707

CAGPARS[4] = 0.707

CAGPARS[5] = 0.0

means 1st (or read) dimension is rotated 45 degree from x to z.

CAGPARS[6] = -0.707

CAGPARS[7] = 0.707

CAGPARS[8] = 0.0

means 2nd (or phase) dimension is rotated 45 degree from y to -x.

CAGPARS[9] = 0.0

CAGPARS[10] = 0.0

CAGPARS[11] = 1.0

means 3rd (or phase) dimension is the z direction.

Multiple Rotation Matrices

It is even possible to define and use multiple rotation matrices. But in this case a list named *grad_matrix* has to be defined within the pulseprogram, containing multiple 3x3 matrices. Then instead CAGPARS[3] .. CAGPARS[11] the rotation matrices within this list are used.

Example to scan 3 orthogonal slices:

```
define list <gradient> grad_matrix =
{ 1, 0, 0, 0, 1, 0, 0, 0, 1,
0, 0, 1, 1, 0, 0, 0, 1, 0,
0, 0, 1 0, 1, 0, 1, 0, 0 }

slice, ..
grad_matrix.inc         ; Next slice orientation
lo to slice times 3
```

Defining Other Object Coordinate Systems

To support animal or human patient investigation it is possible to rotate the gradients according to the position within the magnet.

Within the pulseprogram 8 different object coordinate system can be defined.

`patient_pos = 0`; head first, supine

`patient_pos = 1`; head first, prone

`patient_pos = 2`; head first, right

`patient_pos = 3`; head first, left

`patient_pos = 4`; feet first, supine (this is the default and neutral setting)

`patient_pos = 5`; feet first, prone

`patient_pos = 6`; feet first, right

`patient_pos = 7` ; feet first, left

Then the pulseprogram compiler changes the sign of the gradients accordingly and exchanges x and y if the patient lies on his left or right side.

0.5	0.5	0.8	;Scaling of 1st (read or x) dimension
0.707	0.707	0.0	;rotation matrix

-.707	0.707	0.0	;gradients are rotated by
0.0	0.0	1.0	;45 degrees

Table 7.1: Example of a CAGPARS setting

8 Relations

8.1 The Relation Interpreter

There is a relation interpreter which can be called from the pulse program to calculate parameters. The interpreter language is "C" with a reduced functionality. Pulse program lines which are enclosed by double quotes are submitted to this interpreter. The interpreter can be used either to calculate or recalculate parameters which are used by the pulse program:

```
"d11=30m"
```

or to evaluate conditions (boolean expressions):

```
if "l0%2" goto 5
```

The second form is used after an 'if'-statement in the pulse program whereas the calculations should be in separate pulse program lines.

In the relation above, the `goto` statement will be executed if the content of the condition evaluates to a value different from zero which is here the case for all od numbers.

8.1.1 Syntax Elements

All calculations are executed using the C-type "double". The usual logical expressions like 'if', 'else', 'while' are available, but the 'do', the 'switch' statement and the statement 'else if' cannot be used. Multiple statements can be put into one line separated by semicolons.

8.1.2 Constants

There are some constants which are understood by the relation interpreter:

DEG	180/PI
E	2.718…
LN10	2.302…
PI	3.141…
RAD	PI/180

Table 8.1: Constants in the relation interpreter

8.1.3 Functions

The following mathematical functions can be used:

`acos`	arc cosine
`atan`	arc tangent
`asin`	arc sine
`abs`	absolute value
`cos`	cosine
`exp(b)`	e^b
`max(a,b)`	a if a>b, otherwise b

`min(a,b)`	a if a<b, otherwise b
`trunc(a,b)`	truncate a to next multiple of b
`tdmax(a,b,c)`	limit a to b/c
`trunc(a)`	truncate a to next multiple of 1
`log`	natural logarithm
`log10`	decadic logarithm
`pow(a,b)`	a^b
`random(a)`	random var. of a
`sin`	sine
`sqrt`	square root
`tan`	Tangent
`kronecker_delta(a, b)`	1 if a==b, otherwise 0 (a,b are integer values)

Table 8.2: Functions in the relation interpreter

The `random` function takes 2 arguments: the first (a) is the delay which is varied, the second (b) is the percentage by which the delay is varied.

The resulting value will be in the range from `a -b*a/100` to `a + b*a/100`.

The function `tdmax(a,b,c)` can be used to calculate the maximum possible td in a constant time experiment where `a = td` is the original `td, b` is the constant time delay and `c` is the time increment. The result will be `a` if `b/c` is greater than or equal to `a`, otherwise the result will be `b/c`.

The `trunc` function can be used with 1 or 2 arguments. The second argument can specify a truncation base value, e.g. `trunc(114, 10)` = 110. If the second argument is missing the base is 1.

Please note that arguments of trigonometric functions have to be specified in radians.

8.1.4 Variables

It is not possible to define new variables, instead there are 10 double variables called $d0 .. $d9 and 10 boolean variables $b0 .. $b9. They can be used to store intermediated results. Their values are not kept between subsequent calls of the relation interpreter.

8.2 Compile Time and Run Time

The relation interpreter can be called before the start of the pulse program (usually a pulse program starts with `ze`) or during pulse program execution. The rules and the effects are different.

8.2.1 Compile Time

Relations which stand before the first pulse program statement `ze` are interpreted at compile time. In these statements more parameters can be used and changed. The changed parameters are visible in **ased** and their values are stored after pulse program execution in the acquisition status parameter set. All parameters which are changed there, will appear in

ased as constant, i.e. their values cannot be changed within the editor since the value set by the user is overwritten by the calculation within the pulse program. Parameters defined by the pulse program with a `define` statement must be assigned a value by such a relation.

8.2.2 Run Time

At runtime only a few parameters can be recalculated. The changed parameters are not stored in the status parameters. Other parameters can be recalculated but the recalculation has no effect upon the pulse program.

8.2.3 Parameters

For the list of parameters which are available within relations see the next table.

- Some of them cannot be set but only used on the right-hand side of an equation. They are marked with ^{1}.
- Some of them can only be used at compile time, not at run time; they are marked with ^{2}.
- Phases in phase programs (ph0-31) can only be calculated and used at runtime. They are marked with ^{3}.

See also section *Phase Program Modifications at Runtime [▶ 28]*.

d0-d63	p0-p63	l0-l31	cnst0-63
de1 ^{2}	pcpd1-8^2	nbl	sfo1-8 ^{2}
deadc ^{2}	gpx0-31	ns	bf1-8 ^{2}
depa ^{2}	gpy0-31	ds	inf1-8 ^{2}
dw	gpz0-31	td ^{1}	spoffs0-63 ^{2}
dwov	grpdly^2	spincnt ^{1}	spw0-63 ^{2}
acqt0 ^{2}	acqt0_F1-8	td0 ^{2}	plw0-63 ^{2}
in0-in63	inp0-63 ^{1}	td1-8 ^{2}	ph0-31 ^{3}
bwfac0-63 ^{1,2}	integfac0-63 ^{1,2}	totrot0-63 ^{1,2}	gradcc^{1,2}

Table 8.3: Parameters in relations

9 Loops and Conditions

9.1 The Unconditional goto Statement

This statement has the form

```
goto label
```

The pulse program is continued at `label` which may be either before or after the statement.

9.2 Loop Statements

The general form of a loop statement is:

```
lo to label times n
```

Example 1:

```
label1,d1

p1:f2

lo to label1 times 10

p2:f2
```

Note that a label can be an arbitrary string, such as `label1`, followed by a comma, or a number, such as `2`, without a comma. The `lo` statement in this example, although specified on a separate line, does not cause an extra delay between the `p1` and `p2` pulse statements.

Example 2:

```
label1, p1:f1
label2, d1
        p1:f2
        lo to label2 times 10
        lo to label1 times 5
        p2:f2
```

The `lo` statement exists in a number of variations as shown in the next table.

`lo to label times 5`	The loop counter is a constant.
`lo to label times td`	The loop counter is TD, the time domain size in the acquisition dimension (to be set with the command **td,** or in the left column in **eda**).
`lo to label times td1`	Only used in 2D or 3D pulse programs. The loop counter is F1-TD (to be set with command **1 td** for 2D data sets or in the right column in **eda**).
`lo to label times nbl`	The loop counter is the parameter NBL (see the statements `wr`, `st`, `st0`)
`lo to label times 10` .... `lo to label times l31`	The loop counter is `L[0] - L[31]` (to be set with the commands **l0, ..., l31**, or the L array in **eda**). The pulse program statements `iu0`- `iu31` increment the counters `l0-l31` by 1, `du0-du31` decrement them by 1, and `ru0-ru31` reset them to `L[0] - L[31]`.
`lo to label times lclist`	The loop counter is taken from a loop counter list which, in this example, is named `lcList` (see Chapter '*Loop Counter Lists'* [▸ 70]).

`lo to label times myCounter`	The loop counter must be defined at the beginning of the pulse program by means of a `define` statement and an expression, e.g. `define loopcounter myCounter "myCounter=aq/10m +1"` The result of the expression must be dimensionless.

Table 9.1: The lo statements

Example 3:

```
         ze
label1, (d1 p1):f1
        lo to label1 times l2
        1u iu2
        p2:f2
        go=label1
```

Assume the parameter L[2] is set to 1 using the command **l2 1**, or by setting L[2]=1 in **eda**. Then, `(d1 p1):f1` would be executed once before scan 1, twice before scan 2 etc. The `lo` statement does not cause an extra delay in the sequence. The increment statement `iu2` is executed during the specified 1 µs delay. You could alternatively replace the loop counter `l2` with a loop counter list (see Chapter '*Loop Counter Lists'* [▸ 70]) and replace `iu2` with the respective increment operator of the list to use the number of loops specified in a list file.

Example 4:

```
define loopcounter myCounter
"myCounter=aq/10m +1"
        ze
label1, (d1 p1):f1
        lo to label1 times myCounter
        go=label1
```

Here the variable myCounter represents a loop counter. An arithmetic expression assigns a value to it: the parameter AQ, divided by 10 ms, plus 1. Please note that the compiler rounds the floating point quotient aq/10m automatically to the numerically adjacent integer value. This is different to older versions of TopSpin (< 3.0) where a floating point value was simply truncated instead of rounded. The expression may include any of the parameters shown in table Parameters in relations.

9.3 Loop Counter Lists

A list of loop counters can be specified with a `define` statement in the following way:

```
define list<loopcounter> lcList = {1 2 3}
```

This statement defines the loop counter list `lcList` with the values 1,2 and 3. Instead of explicit loop counter values, you can specify a list filename in the defined statement.

There are two way of doing this:

You can either specify the actual filename or the token `$VCLIST`, both in <>. In the latter case, the file defined by the acquisition parameter VCLIST is used.

For example:

```
define list<loopcounter> ll1 = <myLoopCounterList>
define list<loopcounter> ll2 = <$VCLIST>
```

In both cases, the file can be created or modified with the command **edlist vc**. In a pulse program that contains the statements above, the statement

```
ll1
```

executes a loop counter of 1 the first time it is invoked.

In order to access different list entries, the list index can be incremented by adding `.inc`, decremented by adding `.dec` or reset by adding `.res`. Index operations are performed modulo the length of the list, i.e. when the pointer reaches the last entry of a list the next increment will move it to the first entry.

The operators `.len` and `.max` can be used in relations to access the length of the list or even its respective maximum value (see examples below).

Furthermore, a particular list entry can be specified as an argument, in square brackets, to the list name.

For example, the statement:

```
ll1[1]
```

executes a loop counter of 2 in our example above. Note that the ndex runs from 0 to n-1, where n is the number of list entries.

Finally, you can set the index with an arithmetic expression within double quotes using the `.idx` postfix. The following example shows the usage of an initialized loop counter list:

```
define list<loopcounter> lcList = { 2 3 4 6 }

 lo to 1 times lcList               ; loop to label '1' for 2 times
 lcList.inc                         ; loop counter of 3, set index to 1
 lo to 1 times lcList[2]            ; use loop counter of 4
 "lcList.idx = 3"                   ; set list index to 3 (i.e. a loop counter of 6)
 "l3 = locallist.len"               ; set loop counter 3 to length of list (which
                                    is '4' in our example)
 "l3 = locallist.max"               ; set loop counter 3 to maximum value of
                                    list (which is '6' in our example)
```

9.4 Conditional Pulse Program Execution

9.4.1 Conditions Evaluated at Precompile Time

Consider the pulse program at the left part of the next table:

#define PRESAT 1 ze	#define PRESAT 1 ze
2 d11	2 d11
3 0.1u	3 0.1u
#ifdef PRESAT	#include <Presat.incl>
d12 pl9:f1	p1 ph1
d1 cw:f1	d0
d13 do:f1	p0 ph2
d12 pl1:f1	go=2 ph31
#endif	d11 wr #0 if #0 id0 zd
p1 ph1	lo to 3 times td1
d0	exit
p0 ph2	
go=2 ph31	

`d11 wr #0 if #0 id0 zd`
`lo to 3 times td1`
`exit`	

Table 9.2: Using `#define, #ifdef, #include` statements

It combines two experiments in one pulse program, a simple Cosy and a Cosy with presaturation during relaxation. The required pulse program statements to select or deselect presaturation are:

```
#define aFlag
#ifdef aFlag
#ifndef aFlag
#endif
```

and correspond to C language pre-processor syntax where it is mandatory that the "#" is the very first character on the line. Note that `aFlag` is just a place holder, it can be any name. If the pulse program contains the statement:

```
#define aFlag
```

the identifier *aFlag* is considered to be *defined*, otherwise it is considered to be *undefined*. If *aFlag* is *undefined,* the statement:

```
#ifdef aFlag
```

causes the pulse program to ignore all subsequent statements until the statement:

```
#endif
```

If *aFlag* is *defined,* these statements will be executed. The statement:

```
#ifndef aFlag
```

has the opposite effect.

In table *Using #define, #ifdef, #include statements* [▸ 71], `#define PRESAT` enables the presaturation statement block. Commenting out this line in C-syntax style (*/*#define PRESAT*/*), (not in pulse program style ;#define PRESAT), would make the PRESAT flag undefined, and the presaturation block would not be executed.

The `#ifdef` and `#ifndef` statements are evaluated by a pre-processor. The pulse program compiler will use the pre-processed pulse program. For this reason, these statements do not cause any timing changes. You can view a pre-processed pulse program from the pulse program view. Just click the “**S**” in the tab *PulseProg*. Note that in the pre-processed pulse program, all conditional statements beginning with a ’#’ have been replaced.

The example could be extended to include double quantum filtering. For this purpose, an additional flag (e.g. `#define DQF`) could be defined.

The right part of table *Using #define, #ifdef, #include statements* [▸ 71] shows the same pulse program in a more condensed form. The presaturation block is now contained in a separate file, *Presat.incl*, which is included with the `#include` statement.

9.4.2 Setting of Precompiler Conditions

Conditions can be set or unset not only within the pulse program but also on the command line with the **zg** command using the option *-D*. For example, the command **zg** *-DDQF* has the same effect as the line:

```
#define DQF
```

at the beginning of the pulse program. The argument must follow the *-D* option with or without white space in between. The **-D** option can be given more than once. As an alternative to command line options to **zg**, you can also set the acquisition parameter ZGOPTNS. Once this parameter is set, the corresponding option is used by **zg** and **go**. Thus, setting ZGOPTNS to "-DDQF -DPRESAT" and typing **zg** has the same effect as the command **zg -DDQF -DPRESAT**.

All statements beginning with a '#' character must start at the beginning of a line. Spaces or tabs before '#' are not allowed.

9.4.3 Macro Definitions

You can use the statement `#define` not only to define *a flag*, but also, as in C language, to define a macro.

Example 1:

```
#define macro1 (p1 d1) (p2):f2
macro1
```

This pulse program section is equivalent to:

```
(p1 d1) (p2):f2
```

Example 2:

```
#define macro2 (p1 d1) \n\
            (p2):f2
macro2
```

This pulse program section is equivalent to:

```
(p1 d1)
(p2):f2
```

The definition of macro2 extends over 2 lines using the \n\ character sequence. In example 1, `p1` and `p2` start at the same time, while in this example `p2` starts after `(p1 d1)` has finished.

Example 3:

```
#define macro3 (p1 d1) \n (p2):f2
macro3
```

This pulse program is equivalent to:

```
(p1 d1)
(p2):f2
```

The definition of macro3 requires only one line. However, the \n character sequence enforces a new line when the macro is evaluated. As such, the pulse programs of the examples 2 and 3 are identical.

Attention: Macro symbols are replaced by the body of the macro everywhere in the text, even in comments and strings. If you use a macro in a comment, where you don't want it to be replaced, you should use a C-style comment instead.

Example 4:

```
; the macro3 produces two pulses on two channels
```

This will be expanded by the precompiler to

```
; the (p1 d1)
(p2):f2 produces two pulses on two channels
```

which will produce a syntax error. The following statement is correct:

```
/* the macro3 produces two pulses on two channels */
```

because the C-precompiler removes the line entirely.

9.4.4 Conditions Evaluated at Compile Time

Whereas conditions controlled by `#if` statements are evaluated at precompile time, conditions controlled by `if` statements are evaluated at compile time.

The `if` statement can be used in connection with the parameters L[0] - L[31] as shown in the following example:

`if (l7==0)`	if l7 is zero
`if (l8!=0)`	if l8 is not zero
`if (l9 op`(*arithmetic expression*))	`op` can be ==, !=, >, <, >=, or <=

Table 9.3: Different compile time conditions

The condition must be followed by an if-block and, optionally, can be followed by an else-block. The statement '`else if`', as it is used in C language, is not allowed in pulse programs.

```
if (l5 > 2)
{
  p1 ph1
}
else
{
  p1 ph2
}
```

Table 9.4: A condition evaluated at compile time

The if-block is executed if the condition is true at compile time; in the above example if `l5` is greater than 2, `p1` is executed with phase program `ph1`, if not, it is with phase program `ph2`. If `l5` changes during the experiment, and the condition becomes false, the execution mode doesn't change.

9.4.5 Conditions Evaluated at Run Time

In this case the condition can be either a trigger input or a parameter relation. The spectrometer has four trigger inputs. Trigger events can be positive or negative edges or levels.

TopSpin supports branching and evaluation of conditions within a pulse program while the pulse program execution is in progress.

`goto label`	Unconditional jump to `label`
`if expression goto label`	Branch to `label` if `expression` is true.

`if (trigpe2) goto label`
`if (trigne3) goto label`
	Branch to `label` if the `trigger` condition is true. Please note that only `trignl` (1-4) can be used in this kind of `if`-statement

`if(trigpl3)`
`{`
`10u`
`}`
`else`
`{`
`20u`
	Executed first or second block depending on whether the trigger condition is true.
	Please note that only `trigpl` (1-4) can be used in this kind of `if`-statement

}	

`aDelay trigpe1`
`aDelay trignl2`
`aDelay trigpl3`
`aDelay trigpn4`
	In this construct all trigger specifiers (`trigpl`, `trignl`, `trigpe`, `trigne`) are allowed. The next pulse program statement will not be executed until the trigger condition becomes true.

Table 9.5: Conditional pulse program execution

The table above lists some possible statements. Triggers can be either a positive (p) or a negative (n) level (l) or a positive or negative edge (e) in the incoming signal of one of the 4 inputs 1-4. These statements do not cause a delay in the pulse program. Triggering on a level only makes sense where the program waits for a certain condition. At run time, pre-evaluation is performed during the cycle time of the loops in which the statements are embedded. If, in a particular pulse program, loops are executed too fast, a run time message is printed.

Example 1:

```
      ze
lab1, d1
      p1
      d0
      if "d0*2 + 7m > 500m" goto lab2
       p2
      "d0 = d0 + 10m"
lab2, go=lab1
```

Assume that we will start with d0=10m. The pulse `p2` will no longer be executed when the expression "`d0*2 + 7m > 500m`" becomes true.

Example 2:

```
      ze
lab1, if (trignl2) goto lab3
lab2, d1
      p1
      aq
      lo to lab2 times ds
      goto lab1
lab3, d1
      p1
      go=lab1
```

The spectrometer has 4 trigger input channels numbered 0-3 and corresponding to the trigger commands 1-4; signals arriving there can be checked using the `trig` specifiers. This example performs DS dummy scans to maintain steady state conditions as long as no negative level is detected on input channel 2. If such a level is detected, NS data acquisition scans are executed, then the pulse program again checks the external trigger signal.

Example 3:

```
      ze
lab1, d1 trigpl2
      p1
      go=lab1
```

This example starts executing the pulse sequence as soon as a positive level is detected on input channel 2. After each scan, the pulse program will wait until the next trigger signal is detected.

Example 4:

```
      ze

lab1, d1
```

```
      p1
      lo to lab1 times l2
      0.1u iu1                ; count number of scans
      0.1u iu2                ;  increment l2
      if "l1 <= 3" goto lab2  ;  if scancounter < 4
      0.1u ru2                ;  reset l2 to L2
lab2, go=lab1
```

This example repeats the sequence `(d1 p1)` L[2] times before scan 1, L2+1 times before scan 2, and L2+2 times before scan 3. Then, `l2` is reset to its initial value L[2]. Before all remaining scans the sequence `(d1 p1)` is generated L[2] times. L[1] must be set to 1 before starting the sequence. Note that the branching of pulse programs cannot be repeated without a larger delay between the 2 branches.

9.5 Suspend/Resume Pulse Program Execution and Precalculation

TopSpin allows you to stop (suspend) the pulse program execution at specified positions in the pulse program. Pulse program suspension can be done conditionally or unconditionally using the statements shown in the next table.

`suspend`	stop execution on the command *suspend*
`autosuspend`	stop execution
`calcsuspend`	stop precalculation and stop execution on *suspend*
`calcautosuspend`	stop precalculation and stop execution
`stop_calc`	stop precalculation
`resume_calc`	resume precalculation

Table 9.6: Statements to suspend pulse program execution

After suspension, the program execution can be resumed with the TopSpin command *resume*.

If you use `suspend` or `autosuspend`, you should not try to change any acquisition parameters between suspending and resuming the acquisition, because this will not have the wanted effect. The reason is that the acquisition uses the principle of precalculation which means a part of the pulse program is interpreted (precalculated) before it is actually executed. After resume, the precalculated part which was calculated before the parameter change is executed without considering the parameter change.

The statements `calcsuspend` or `calcautosuspend`, however, stop precalculation. Here you can change parameters between suspending and resuming the acquisition. Note that you must specify a delay which is long enough to start and do a reasonable amount of precalculation after **resume**.

Example:

```
calcsuspend
2s
```

If, after resuming the acquisition, you would get the error message "timing too short", you must increase this delay.

9.5.1 Changing Parameters Interactively during Pulse Program Execution

For pulse programs where parameters should be changed interactively during pulse program execution, there exist two commands to control the precalculation of the pulse program. Normally, the program precalculates as much of the pulse program as possible. If any parameter is to be changed, the effect of this change would not become effective until all code which has been precalculated up to this time has been executed. To restrict the amount of precalculation, the command `stop_calc` can be used. Of course, before program execution reaches this point, the precalculation must be resumed. For this purpose the command `resume_calc` must precede the `stop_calc`. The time between these two commands must be long enough to allow for a sufficient amount of precalculation. Parameters which are changed after `stop_calc` and before `resume_calc` become efficient immediately in the code after `stop_calc`. To enable the changing of parameters the pulse program must be started not with **zg** but with **zg interactive** instead.

Parameters which can be modified with the current version are SPOFFS0-63, SPOAL0-63, SPW0-63, and the shape itself (but not the shape file name).

Additionally, the frequency SFO1 as well as pulse lengths (parameter P), delays (parameter D) and loopcounters (parameter L) can be modified provided that they are used in the pulse program and are not explicitly defined by a relation like “`d1=200u`”.

A parameter change can also be triggered by entering the following command into the topspin command line:

```
sendirptcmd gsload <parameter>#<new value>
```

For example, the following command would set `d1` to 0.1sec:

```
sendirptcmd gsload D1#0.1
```

In order to trigger a parameter change in an AU-program the command `sendgui` has to be prepended:

```
CPR_exec(“sendgui sendirptcmd gsload D1#0.1”, WAIT_TERM);
```

9.6 Pulse Program Execution on Independent Channels

For current spectrometers each pulse program channel is executed on its own processor. Normally all these processors execute the same pulse program but only that part of the pulse program which is relevant for the specific channel.

In certain situations it is desirable that each channel executes its own code, and only after some time all channels continue with the common part of the pulse program.

`exec_on_chan:t1:f1:f2:g1`	execute only on specified channels
`exec_on_other`	executed on remaining channels
`exec_wait`	wait for other channels to finish before continuing
`exec_enable`	(obsolete)

Table 9.7: Commands for independent channels

The following commands have to be used to control the program flow:

A channel is specified first with the command `exec_on_chan` followed by the channel specification. This is as usual `:f1-:f8` for frequency channels, but you may also specify the timing controller as `:t1` and the gradient controller as `:g1`. The sequence must end with the command `exec_wait` which waits until all other channels have also been reached their `exec_wait` statement before continuing.

The code for the remaining channels is specified in one or more additional `exec_on_chan/exec_wait` blocks. Please note that - in contrast to AVIII spectrometers – it does not have to be considered anymore which channel is more time-consuming, i.e. it is not necessary to use a separate the `exec_enable` statement for the channel whose execution time is the longest.

The example below shows a program where two loops with different timings are executed independently on two channels.

```
10u

exec_on_chan:t1:f1 ; on channel TCTRL and F1

4 p1:f1 ph1
d1
lo to 4 times l1

exec_wait          ; end of TCTRL - F1

exec_on_other      ; on all other channels (F2, ... GCTRL)

5 p2:f2 ph2
p9:gp2
d2
lo to 5 times l2
10u

exec_wait          ; end of channel specific code

10u
```

This structure can be used for more than two channels as well. As `exec_wait` uses firmware triggers on the spectrometer this may lead to a seemingly wrong timing of the execution blocks relative to each other in the simulation when running **hpdisp**. However, this does not reflect the real workflow of the pulse program on the spectrometer but is a shortcoming of the simulation which cannot properly include the delays of the firmware triggers.

9.7 Rotor Triggering in Pulse Programs

The following commands can be used to synchronize the pulse program execution with an external periodic signal - the principal application is the rotor synchronization in MAS experiments:

`spincnt_use1..4`	use trigger input 1..4 for commands	header
`store_spinrate_d0..63`	Measure rotor period and store in d0..63	pulse pr.
`spincnt_measure`	measure time to next rising edge of trigger input	pulse pr.
`spincnt_start`	start timer	pulse pr.
`spincnt_wait`	start timer and wait for trigger	pulse pr.

Table 9.8: Commands for rotor controlled experiments

`spincnt_use1` means that the all spincnt commands will use trigger input 1. This command must appear in the header of the pulse program to enable the spincnt logic. If the `spincnt_use` is encountered in the pulse program, the hardware will measure the time between 2 trigger signals on trigger input. This can be used to measure the rotor period, i.e while such a programm is running the rotor period can be examined with the TopSpin command spincnt.

`store_spinrate_d31` is used to measure the time of one rotor period and store it in the parameter D31.

`spincnt_measure` measures the time from the point in the pulse program to the next rising edge of the trigger signal (the start of the next rotor period). This time is stored in a register and delays the trigger signal after the commands `spincnt_wait` or `spincnt_start`.

The parameter SPINCNT determines how many rotor periods the trigger signal is delayed after the commands `spincnt_wait` or `spincnt_start`.

`spincnt_wait` waits for the trigger, then counts down the rotor periods according to SPINCNT and the sync delay if there was a command `spincnt_measure` before. after this the pulse program is continued.

`spincnt_start` does the same, but the trigger command `trigpe` at which the pulse program continues has to be added somewhere after the `spincnt_start`. Between these two commands there may be other delays and pulses whose length must not exceed the delay determined by SPINCNT.

The following pulse program examples show how these commands can be used to synchronize the pulse program execution with an external periodic signal.

9.7.1 Wait for Defined Number of Rotations

The following pulse program waits SPINCNT rotations before applying the next pulse:

```
spincnt_use4     ; use trigger input 4
...
2 d1
...
1u spincnt_wait
p1
go = 2
```

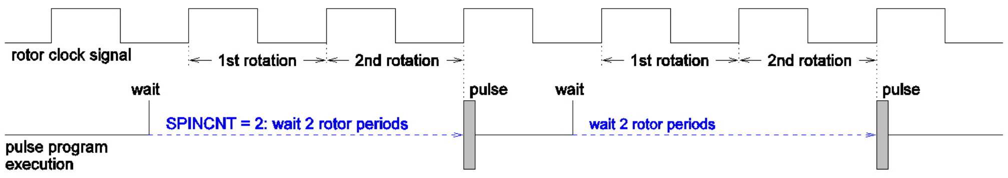

Figure 9.1: Pulse starts immediately after 2 rotation periods

The figure above shows the execution flow for SPINCNT = 2.

9.7.2 Wait for Defined Number of Rotations and a Sync Delay

The following pulse program uses an extra delay between the rotor clock and the pulse:

```
spincnt_use3           ; use trigger input 3


10u
10u spincnt_measure  ; measure sync delay
p1
...
2 d1
...
1u spincnt_wait        ; wait for SPINCNT rotations
                     ; + sync delay
```

```
p1
go = 2
```

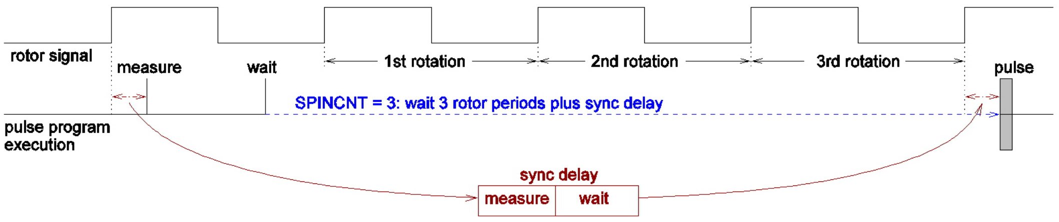

Figure 9.2: Pulse starts after sync delay

Figure *Figure 9.2 [▸ 80]* shows the execution flow for SPINCNT = 0 and figure *Figure 9.3 [▸ 80]* shows the execution flow for SPINCNT = 3.

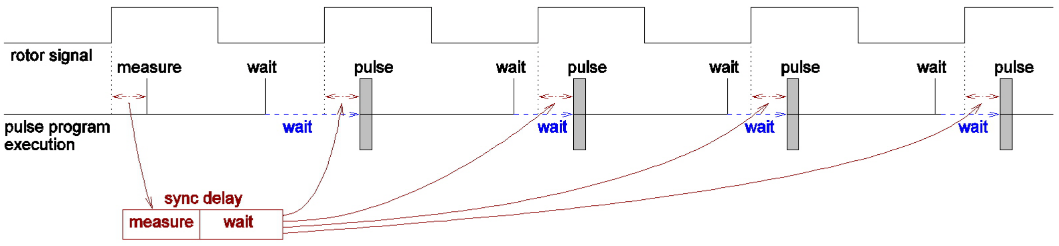

Figure 9.3: Pulse starts after spincnt rotor periods and sync delay

9.8 Synchronization of a Pulse to a Constant Rotation Angle

Using the features described in the two experiments above one can ensure that a sequence is always started at the same position of the rotor (constant angle) independent of the spin rate:

1. Measure the period of the rotation
2. Calculate the delay used between the begin of one rotation period and the start of the sequence.

The following pulse program demonstrates a pulse which is positioned in the middle of the rotation period:

```
spincnt_use3           ; use trigger input 3 for
                       ; the rotor clock signal


1 1m                   ; wait several periods
                       ; of the rotor clock
10u store_spinrate_d31 ; read rotation period
                       ; into delay d31
d5                     ; if d5 is too short
                       ; expect 'empty FIFO' error

1u
"d30 = (d31-p1) / 2"   ; calculate d30 ~ 50% of
                       ; the rotation period
2 d31                  ; wait 2 periods
d31
```

```
1 spincnt_wait          ; wait for SPINCNT rotations
d30                     ; move p1 below into the center of the rotation period
p1
go = 2
```

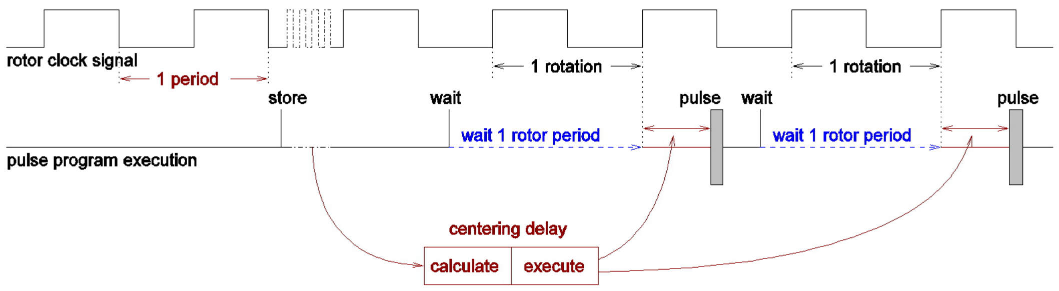

Figure 9.4: Pulse starts after 1 rotation period in the middle of the next

The figure above shows the execution flow for SPINCNT = 1 and the next figure for SPINCNT = 0.

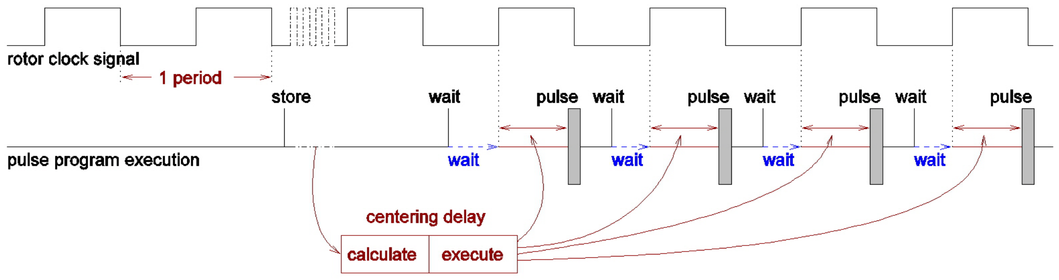

Figure 9.5: Pulse starts in the middle of the next rotation period

`store_spinrate_dn` reads the rotation period at run time (during execution of the pulse program by the sequencer) while the relation which determines the angle delay (D30 in the example above) is calculated during load time (executed by the DSP). Therefore it is necessary to stop loading before the relation, and to continue loading after the period has been read. This is done automatically by the DSP when the command `store_spinrate_dn` is reached. However, since the loading is stopped while the sequencer is running the DSP may need some time to calculate the next lines of the pulseprogram after the loading is continued. This time should be spent together with the `spinrate_measure` command (see `d5` in the pulse program shown above).

You must wait at least one complete period between the start of the pulse program and the command `store_spinrate_dn`. Otherwise the delay where the period is stored will be zero.

Due to the execution delay of the SGUs there is always a lag of 10 µs after trigger actions (e.g. between the pulse command from the F-controller and the appearance of pulse at the output of the SGU). This should not lead to problems though, as the experiments do not rely on the absolute position of the rotor.

10 Data Acquisition and Storage

10.1 Start Data Acquisition

TopSpin provides 5 basic pulse program statements to start data acquisition:

`go=label`, `gonp=label`, `gosc`, `goscnp` and `adc`.

The most commonly used statement is `go=label`. Actually, `go` is a macro statement, i.e. it includes a number of different actions required for data acquisition. The statement `adc` can be used to control fine details of the acquisition process. All five acquisition statements place the digitized signal into a memory buffer. The `wr` statement, described in a later section, writes the buffer contents to disk.

10.1.1 The Statements go=label, gonp=label, gosc, goscnp

The left column of the table *Table 10.1 [▸ 84]* shows a simple example of how to use `go=label` in a pulse program. All `go` type statements perform the 8 actions described below. A parallel sequence of 5 pre-scan subdelays is executed (see the description of DE1/ DERX/ DEPA/DEADC in the Acquisition Reference Manual). Note that all these delays end simultaneously, at the end of DE. The sequence in which the actions are performed, depends upon the length of the individual delays. The sequence must be

1. At the end of DE-DEPA (preamplifier blanking delay), the preamplifier is switched to observe mode.
2. At the end of DE-DERX (delay for receiver blanking) the receiver gate is opened.
3. At the end of DE-DE1, the intermediate frequency (if used) is added to the frequency of the observe channel and the observe SGU switches from transmit to observe mode. This corresponds to the execution of the statement syrec.
4. At the end of DE- DEADC (delay for ADC blanking), the digitizer is enabled.
5. After a total delay of DE the digitizer is started. Please refer to the description of the parameters DW/DWOV/DIGMOD on how the sampling rate is selected. The result will be a digitized FID signal of TD data points, where the time domain size TD is defined by the user (from **eda**, or by typing **td**). The FID will be put into the current memory buffer. The contents of memory buffers can be transferred to disk with the wr pulse program statement or with the `tr` command. The section Acquisition memory buffers discusses the usage of memory buffers and the size restrictions of TD.
6. At the time the digitizer is started, a delay AQ is executed. This delay lasts until the digitization of the FID is finished.
7. A delay of 1.4 µsec is executed. During this time the following tasks are performed:
   - a) The scan counter, visible during real time FID display, is incremented to inform the user about the number of scans performed since the last executed `ze` or `zd` statement.
   - b) The frequency of the observe channel is switched back to the frequency of the observe nucleus. This corresponds to the execution of the statement `sytra` (which is inverse to `syrec`).
   - c) The pointers of all phase programs are incremented to the next phase, corresponding to the execution of the statements `ipp0`, ... , `ipp31`. This step is skipped by `gonp=label` and `goscnp`.
   - d) The statements `go=label` and `gonp=label` perform a loop to `label`, whereas `gosc` and `goscnp` do not loop. The pulse program statements between `label` and `go` or `gonp` are executed DS+NS times. During the first DS loops (dummy scans to

achieve steady state conditions), the digitizer is not activated. In all other respects, the dummy scans are identical to the NS data acquisition scans. If no dummy scans are desired, DS must be set to 0.

– **Please note:** Even if DS > 0, no dummy scans will be executed if the pulse program statement `zd` (rather than `ze`) was executed before a `go` loop is entered (see the description of `ze` and `zd`). This feature is, for example, used in 2D experiments where dummy scans are only required before the first FID is measured.

– e) Commands after an additional `finally` statement are executed during this delay. In this way decoupling can be switched on during the acquisition time only and need not be terminated outside the go command.

1	`go=2 ph31`	Receiver phase = ph31, realized via add/subtract and channel A/B switching. Allowed phase values: 0, 90 180, 270 degrees.
2	`go=2 ph30 :r`	Receiver phase = ph30 + PH_ref, realized via the phase of the reference frequency of the observe channel. Allowed phase values: any.
3	`go=2 ph31 ph30:r`	Combination of (1) and (2). The receiver phase is the sum: ph31 + ph30 + PH_ref
4	`go=2 ph31 ph30:r cpd1:f2`	Decoupling starts at the same time the receiver is opened, and automatically stops when the loop is executed.
5	`go=2 ph31 ph30:r cpd1:f2 ph29`	As example 4, with a phase program for the CPD sequence.
6	`go=2 cpd1:f2 finally do:f2`	As example 4, but decoupling ends after the acquisition time

Table 10.1: Examples of the usage of the go or gonp statement

The table above shows that the `go` statements can be specified in conjunction with other statements. PH_ref is an acquisition parameter to be set by the user.

10.1.2 The Statements rcyc=label, rcycnp=label

The statement `rcyc` executes step 7 of the actions performed by `go=label` and `gonp=label` (see the previous section). The `rcycnp` statement skips step 7c.

The `rcyc` statements can be used for acquisition loops based on `adc` rather than `go=label` or `gonp=label`. You must *not* specify phase programs behind `rcyc` and `rcycnp`. Decoupling statements are allowed although it would not make sense to use them here. The table *Table 10.4 [▸ 87]* shows an example of an acquisition loop with `rcyc`. Note that the `adc` statement is part of the DE1 macro statement.

The `rcyc` statements can also be specified behind a delay, e.g. `100u rcyc=2`. They are then executed during that delay instead of the default 1.4µsec. The specified delay must be at least 1.4µsec of course.

10.1.3 The Statements eosc, eoscnp

The statement `eosc` executes steps 7a - 7c of the actions performed by `go=label` and `gonp=label` (see the previous section). The `eoscnp` statement only executes steps 7a and 7b.

The `eosc` statements can be used in pulse programs with data acquisition based on `adc`. In contrast to `rcyc`, you must add the appropriate loop statements.

You must *not* specify phase programs behind `eosc` and `eoscnp`. Decoupling statements are allowed but it would not make much sense to use them here. The table *Table 10.4 [▸ 87]* shows an example of an acquisition loop based on `eosc`. Note that the `adc` statement is part of the DE1 macro statement.

The statement `eosc` or `eoscnp` can also be specified behind a delay of at least 1,4 µsec, e.g.:

```
100u eosc
```

In that case, they are then executed during the specified delay rather than during the default 1,4 µsec.

See also

- The Statement adc [▸ 85]

10.1.4 The Statements ze and zd

The statements `ze` and `zd` perform the following actions:

1. They set the scan counter, which is visible during real time FID display, to 0 or to -DS. A negative value indicates that dummy scans are in progress.
2. They set a flag which triggers the next `go`, `gonp`, `gosc`, `goscnp`, or `adc` statement to replace any existing data in the acquisition memory rather than add to them. This counts for all NBL memory buffers. If `ze` or `zd` are placed outside an acquisition loop, the *replace* mode will only be valid for the first scan performed by the loop. The FID's of all the scans that follow will be added to the data present in the memory buffer.
3. The statement `zd` automatically resets all phase program pointers to the first element, whereas the statement `ze` sets all phase program pointers such that they are at the first element after DS dummy scans.
4. The difference between `ze` and `zd` is that `zd` prevents the execution of dummy scans by `go`, `gonp`, `gosc`, `goscnp`, and by `adc` (combined with `rcyc` or `eosc`), even if DS > 0.

The statements `ze` and `zd` can be written behind a delay statement. They are then executed during the delay. If they are not specified with a delay their execution will require 3 ms.

The statement `zd` is normally part of one of the specifications of the `mc` macro statement. One example where it is specified explicitly is the pulse program *selno*.

10.1.5 The Statement adc

The statement `adc` starts the digitizer and, at the same time, opens the receiver. Please refer to the description of the parameters DW/DWOV/DIGMOD in the Acquisition Reference Manual for information on how the sampling rate is calculated. The result of `adc` will be a digitized FID signal of TD data points. TD is an acquisition parameter that must be set by the user. The FID will be placed in the current memory buffer (see the section *Acquisition memory buffers*).

When you use the `adc` statement rather than `go`, you must consider the following:

- Whereas the `go` statement automatically executes the required switching delays, these must be specified explicitly when you use `adc`. For this purpose, the macros DE1, DE2, DE3, DEPA, DERX and DEAC are available. They are defined in the file *De.incl* that can be included in the pulse program with the statement:
- `#include < De.incl>`
- The contents of this file is shown in the next table.

```
define delay rde1
define delay rdepa
define delay rderx
define delay rdeadc

"rde1=de-de1;"
"rdepa=de-depa;"
"rderx=de-derx;"
"rdeadc=de-deadc;"

#define DE1(phrec) (rde1 de1 adc phrec syrec)
#define DE2(phref) (1u 1u phref:r):f1
#define DE3 (de)
#define DEPA (rdepa depa RGP_PA_ON)
#define DERX (rderx derx RGP_RX_ON)
#define DEADC (rdeadc deadc RGP_ADC_ON)

#define ACQ_START(phref,phrec) DE1(phrec) DE2(phref) DERX DEADC
DEPA DE3
```

Table 10.2: The contents of the file De.incl for AV and AV-II spectrometers

Note that `adc` is implicitly defined with DE1

Here, the statement `de` executes the delay defined by the acquisition parameter DE. The statements `de1`, `derx`, `deadc` and `depa` execute a delay that is defined by the corresponding *edscon* parameters.

```
define delay rde1


"rde1=de-de1"


#define DE1(phrec) (rde1 sync de1 adc phrec syrec)

#define DE2(phref) (1u 1u phref:r):f1

#define DE3 (de)


#define ACQ_START(phref,phrec) DE1(phrec) DE2(phref) DE3
```

Table 10.3: The contents of De.incl for AV-III spectrometers

In the case of an AV-III spectrometer, the three delays DEPA, DERX and DEADC are generated automatically by the hardware and can be replaced by one single DE2. The DE1 and DE2 macros have the arguments `phrec` and `phref` which are used to define the phase program used by the receiver and the reference frequency, respectively. The latter phase program can be specified after the `go` macro using the syntax `phnn:r`.

ze 2 d1 (p1 ph1):f1 ;---------- go=2 ph31 ;-------- wr #0 exit	#include De.incl ze 2 d1 (p1 ph1):f1 ;------------------- DE1 DEPA DERX DEADC DE3 aq DWELL_GEN rcyc=2 ;------------------- wr #0 exit	#include De.incl ze 2 d1 (p1 ph1):f1 ;------------------- DE1 DEPA DERX DEADC DE3 aq DWELL_GEN eosc lo to 2 times ns ;------------------- wr #0 exit

Table 10.4: The same pulse program based on go, adc/rcyc, and adc/eosc

- For end-of-scan handling, you must specify one of the statements `eosc`, `eoscnp`, `rcyc`, or `rcycnp`. Multiple `adc` statements can be used in conjunction with, for example, a single `eosc` statement. The table above shows the same pulse program realized via `go=label`, `adc` in conjunction with `rcyc`, and `adc` in conjunction with `eosc`.
- You must enable the intermediate frequency using the statement `syrec`.
- The dwell time is generated during `aq`. For Avance-AQS, the dwell time is generated on the SGU with the macro `DWELL_GEN`.

For an example of how to use the `adc` statement rather than `go`, please have a look at the Bruker pulse program `zgadc` (enter **edpul zgadc**). This program will produce exactly the same result as the program `zg`.

10.1.6 The Receiver Phase

In pulse programs using the `adc` statement, the receiver phase must be specified behind `adc`, e.g.:

```
adc ph31
```

This statement tells the receiver which phase is to be used for the next scan to account for the receiver phase setting. Note that there must be sufficient time between the end-of-scan interrupt signal of one scan and the receiver phase interrupt signal of the next scan. Normally, the recycle delay is long enough for this purpose. However, for some applications (like imaging experiments) the recycle delay can be too short for correct interrupt handling. In that case, the

1 ...
  ...
2 d1
  10u adc ph31
  aq DWELL_GEN:f1
  rcyc=2
  10u ip31
  lo to 1 times l1
  ...
	1 ...
	  ...
	  2u recph ph31
	2 d1
	  10u adc
	  aq DWELL_GEN:f1
	  d2 rcyc=2
	  10u ip31
	  lo to 1 times l1

	...

Table 10.5: Receiver phase setting without and with recph

receiver phase should be specified before the scan loop using the statement `recph ph31` (see the table above). The statement `ip31` after the recycle loop increments all entries of the phase program `ph31` but does not set the phase. As such, the receiver phase is not changed after each scan but after NS scans.

10.1.7 External Dwell Pulses

The `go` and `adc` statements instruct the digitizer to acquire the desired number of data points with a rate given by the *dwell time*. The dwell pulse (only a short dwell pulse is necessary to acquire a complete scan) is generated by the TRX which is dedicated to the observe channel. This is the meaning of the macro `DWELL_GEN` which evaluates to the statement:

```
aq cpdngs17:f1
```

Certain experiments, however, require the control of the detection of each individual data point of an FID. In this pulse program the waiting time `aq` has been replaced by a loop that collects as many dwell points as required to measure TD data points.

On AV III systems, the pulse program instructions `DWL_CLK_ON/DWL_CLK_OFF` have been used to gate an acquisition, but those commands will not compile anymore on systems newer than AV III, and for technical reasons had to be replaced by three new pulse program instructions. Furthermore, the pulse program statement ‘dwellmode explicit’ is now obsolete and ignored by the compiler.

The new pulse program instructions are:

```
START_NEXT_SCAN
DWELL_HOLD
DWELL_RELEASE
```

The usage is in a way similar to `DWL_CLK_ON/DWL_CLK_OFF`, but some details are different:

1. `START_NEXT_SCAN`:
   This command is mandatory. It instructs the digital part of the receiver to start to acquire TD data points (this behavior is similar to `DWL_CLK_ON`). Acquisition will start immediately after this command (if no `DWELL_HOLD` was set before). It will automatically finish if TD points have been acquired. If, however, TD points have not yet been collected and another `START_NEXT_SCAN` command is executed the acquisition will abort with an error message.
2. `DWELL_HOLD`: This command is optional. It instructs the digital part of the receiver to stop acquiring data points until the next `DWELL_RELEASE` command.
3. `DWELL_RELEASE`:
   This command is optional. It instructs the digital part of the receiver to continue acquiring data points until TD points have been acquired. Requires a preliminary `START_NEXT_SCAN` to take effect.

Please also refer to the Bruker pulse program libraries for high resolution, solids, and imaging experiments for examples using these commands.

The digitizer itself has a constant rate of 240 MHz and cannot be stopped and started, instead it can be gated and the resulting data stream is filtered and decimated according to the chosen sweep width and decimation.

By using the `DWELL_ENABLE/DWELL_RELEASE` commands it is possible to *pause* the acquisition (on the digital side of the ADC) and for example apply another pulse sequence before continuing to acquire the FID.

In this way data points are acquired explicitly but both decimation and digital filters are nevertheless used for the acquisition.

In case less than TD points have been acquired when setting up the next scan, an error is reported.

The following is a draft of an example pulse program with an explicit dwell mode generation.

Please note that, in case `Avancesolids.incl` is used instead of `Avance.incl`, the macros `RG_ON/RG_OFF` have to be used instead of `REC_UNBLANK/REC_BLANK`.

```
#include <Avance.incl>
#include <De.incl>
define loopcounter tdh
define loopcounter firstFidSegment
define loopcounter secondFidSegment
; loop over TD/2 because with each dwell clock two points are acquired
; d12 is the dwell time for one complex data point (i.e. two data points of TD)
"tdh=td/2"
"firstFidSegment = tdh/2"          ; just for clarification
"secondFidSegment = tdh/2"         ; just for clarification
ze
; now excite spin system, this section is omitted here
ACQ_START1(ph30, ph31)             ; now start acquisition
(1u REC_UNBLK):f1                   ; unblank (analog part of) receiver
1u DWELL_HOLD ; prevent data acquisition
1u START_NEXT_SCAN ; tell (digital part of ) receiver to acquire a total of TD points
; turn dwell clock on to acquire two data points.
; Note that d12 must be long enough

lo to 4 times firstFidSegment      ; now repeat to acquire the next
                                     two data points
1u REC_BLK                         ; blank receiver path
4 d12 DWELL_RELEASE ; now start acquiring points
0.1u DWELL_HOLD ; now stop acquiring points
; now the first half of the FID has been acquired
; do something else here, e.g. apply some pulses
; now the second half of the FID is acquired:

(1u REC_UNBLK):f1                  ; unblank (analog part of) receiver
5 d12 DWELL_RELEASE  ; now start acquiring points
0.1u DWELL_HOLD  ; now stop acquiring points
lo to 5 times secondFidSegment
1u DWELL_RELEASE  ; see explanation below for this command
1u REC_BLK                         ; blank receiver path
100u eoscnp                        ; end of scan
                                   ; OK, FID has been acquired
```

```
10m wr #0                              ; write fid down to disk
exit
ph1=0
ph30=0
ph31=0
```

Please note the somewhat surprising `DWELL_RELEASE` command at the end of the acquisition sequence. This is necessary because although up to this point in the pulse program all of the TD points have been acquired there may still be some few points in the filter pipeline of the TRX (especially if the acquisition time is exactly calculated, e.g. as d12 in the example above). Those points also have to be flushed out, which is done by the final `DWELL_RELEASE` command.

There is one potential drawback when using explicit dwell mode generation: The digital filter is not cleared between the FID segments.

The consequence in the example above is that – when starting to acquire the second half of the FID - there are still the last GRPDLY (group delay) points of the first FID segment in the filter pipeline which will now affect the first points of the second half of the FID. This does not matter if e.g. the subsequent chunks of data seamlessly fit together concerning the development of the chemical shift, but it has to be kept in mind that there is a potential source of error depending on the current experiment.

The group delay is available in the pulseprogram as compile-time variable **grpdly**.

It is defined as number of complex points, i.e. a group delay of 78 means that the first 156 points of the FID are affected by the length of the filter pipeline.

10.2 Acquisition Memory Buffers

The acquisition statements `go=label`, `gonp=label`, and `adc` put the acquired data points into a *memory buffer* where they reside until new data points are added, or until they are replaced by new data (*replace* mode is turned on by the statements `ze` and `zd`). A memory buffer provides space for TD data points, where TD must be set by the user.

In most 1D experiments, one FID is measured and stored in one memory buffer. After NS scans have been accumulated, the contents of that memory is written to disk (with the `wr` statement). Multi-dimensional experiments, imaging experiments, experiments varying parameters such as the decoupling frequency or recovery time generate several FID's. In that case you can use one or several memory buffers. If a single buffer is used, the buffer contents must be transferred to disk before the next FID can be measured. If several buffers are used, several FID's can be measured before a disk transfer is required. The latter method is appropriate if the FID's of the experiment succeed one another so quickly that no disk transfer is possible in between them.

The acquisition parameter NBL determines the number of memory buffers used (default: NBL=1). Each buffer has a size TD. If TD is not a multiple of 128, the buffer size will be rounded to the next multiple of 128 data points. The acquisition commands will put the FID into the *current* buffer. The default *current* buffer is buffer 1. The pulse program statement `st` makes the *next* buffer the current buffer whereas the statement `st0` makes the *first* buffer the current buffer. When the number of buffers is exhausted, i.e. when `st` is executed for the NBL'th time, the *first* buffer becomes the current buffer.

The statements `st` and `st0` must be specified behind a delay which must be at least 10 µsec, e.g.:

```
  10u st
1 ze
  d11 pl14:f2
  d11 fq2:f2 st0
```

```
2 d1
3 d20 cw:f2
  d13 do:f2
  p1 ph1
  go=2 ph31
  d1 fq2:f2 st
  lo to 3 times l4
  d11 wr #0 if #0
exit
```

Table 10.6: Usage of st and st0: noediff pulse program

The table above shows an example, the Bruker pulse program *noediff*. The FID's acquired with different decoupling frequencies are stored in two memory buffers.

The commands `ze` and `zd` reset the scan counters for all NBL buffers. Of course, the pulse program must do NS (NS+DS) scans in each buffer before advancing to the next buffer with the command `st`. If dummy scans should be done only before acquisition starts in the first block, it is better to write a separate loop before the actual acquisition loop in the pulse program.

10.3 Writing Data to Disk

Data acquisition statements `go=label`, `gonp=label`, `gosc`, `goscnp`, and `adc` put the digitized data into a memory buffer, but do not store them to disk. Therefore, every pulse program must contain at least one disk write statement to transfer the acquired data to disk.

`mc #0`	Macro statement that executes the statements `wr #0`, `if` and `zd`. Normally `mc` is specified with one or more clauses which expand to loop structures. See chapter *The mc Macro Statement [▸ 95]*.
`wr #0`	Transfer the acquisition buffer to the file *fid,* or transfer NBL acquisition buffers to the file *ser* of the current data set. For *ser* files: `wr` starts writing into the file at the current position of the disk file pointer, which initially is at the beginning of the file.
`wr #1`, `wr #2`, `wr #3`, ...	Transfer is performed to the file *fid* or *ser* of the data set with the number 1, 2, 3, ... contained in the data set list defined by the acquisition parameter MULEXPNO.
`wr ##`	Transfer is performed to the file *fid* or *ser* of the data set which is pointed to by the dataset list pointer. Its initial position is #0 which always corresponds to the foreground dataset. The dataset list pointer itself can be manipulated with the commands `ifp`, `rfp` and `dfp`.
`if #0`, `if #1`, `if #2`, ...	Advance the disk file pointer for *ser* files by TD*NBL (note that TD is rounded to the next multiple of 256 data points if it is not a multiple of 256).
`if ##`	Advance file pointer of current file (see `wr ##`)
`df #0`, `df #1`, `df #2`, ...	Decrement the file pointer (inverse of `if`).
`rf #0`, `rf #1`, `rf #2`, ...	Reset the file pointer to the beginning of the *ser* file.
`rf #0 m`, `rf #1 m`, `rf #2 m`, ...	Set the file pointer to position `m`*TD*NBL of the *ser* file, where `m` is an integer number.

`ifp`	The dataset list pointer `##` is incremented by 1. It initially points to the foreground dataset (`#0`), after the first increment to the first item of the dataset list (`#1`) and so on (NB: There is an automatic wrap-around, i.e. the list pointer jumps back to the foreground dataset when it points to the last data set list item and is further incremented).
`dfp`	The dataset list pointer `##` is decremented by 1. (NB: There is an automatic wrap-around, i.e. the list pointer jumps to the last item of the dataset list when it points to the foreground dataset and is further decremented).
`rfp`	The dataset list pointer `##` is reset to the first item (the foreground data set `#0`).

Table 10.7: Writing acquisition buffers to disk

The table above shows the available pulse program statements to access disk files.

Transferring data to disk means *adding* the data to the data contained in an existing *fid* or *ser* file, or *replacing* these data. If no such file exists, it will be created. *Replacement* will take place if started with **zg**, *addition* will take place if the pulse program is started with the command **go**. However, data replacement only occurs the first time a memory buffer is transferred to disk. Any further execution of the `mc` or `wr` statement will cause the buffered data to be added to the data in the file.

Nevertheless, for certain experiments it may be desirable to replace (instead of accumulating) the stored FID any time a `wr` command is executed. This can be achieved by writing the statement

```
wrmode replace
```

at the beginning of the pulse program, i.e. above the first `ze` statement usually marking the beginning of the acquisition loop.

It is allowed to specify the statements `if`, `zd`, `id0-id63`, `ip0-ip31`, and decoupling statements behind the same delay which is used for `wr`.

For example, one could write

```
30m wr #0 if #0 zd
```

in one line of the pulse program(`df` or `rf` instead of `if` are also possible).

Please note that it is important to use either a `zd` or `ze` statement after each `wr` before the next scan. Otherwise the receiver will not clear its local data buffer, and as a consequence the same data will be accumulated again with the next `wr` command.

In rare cases or very complicated experiments, it may also be necessary to increase the file pointer more than once after a `wr` statement. This is also possible by writing for example

```
30m wr #0
1u if #0
1u if #0
```

The name of the output file is *fid* or *ser*. A *fid* file contains a single FID, whereas a *ser* file contains a series of FIDs. The appropriate name is automatically chosen by the pulse program compiler: if a pulse program contains one of the increment, decrement, or reset file pointer statements, or `st`/`st0`, a *ser* file will be created.

If the pulse program uses a *ser* file, the acquisition command checks if a *ser* file already exists and if it has the correct size. If this is the case, the first occurrence of a `wr` statement will overwrite the *ser* file section defined by the current file pointer, TD, and NBL. If a *ser* file does not exist or has the wrong size, a new *ser* file will be created and filled with zeroes before acquisition starts. As such, the *ser* file is not required to grow during the experiment. This method avoids the risk of running out of disk space while acquisition is in progress.

In a 2D experiment, the TD value must be set such that TD_{F1}*TD_{F2}*4 corresponds to the size (in bytes) of the ser file. In a 3D experiment, the TD values must be set such that TD_{F1}*TD_{F2}*TD_{F3}*4 corresponds to the size (in bytes) of the ser file. If they are not, a warning is displayed even though the experiment can still be executed. If, for some reason, you have performed a 2D experiment with TD values that do not match the size of the ser file, you must set the status TD_{F1} value before you process the data. You can do that with *1s td*. For 3D experiments you can adjust the TD values of the indirect dimension with *2s td* and *1s td*.

These rules apply to acquisition dimensions higher than 3 in a similar way. TopSpin allows acquisition dimensions up to 8D.

In 3D pulse programs, the acquisition status parameter AQSEQ describes the order (*321* or *312*) in which the 1D FID's of a 3D acquisition are written into the *ser* file (3 = the acquisition dimension, 1 and 2 = the orthogonal dimensions). AQSEQ is automatically set and stored in the parameter file *acqus* according to the pulse program loop structure. A 3D pulse program usually contains a double nested loop with loop counters `td1` and `td2`. If `td1` is used in the inner loop and `td2` in the outer loop, AQSEQ is set to 312. Otherwise it is set to 321. Note that, in most 3D pulse programs, the `td1` and `td2` loop is implicitly defined by an `mc` statement. If a 3D pulse program contains a different loop structure (not defined by `td1`, `t2d`, or `mc`) AQSEQ should be explicitly set with one of the statements:

```
aqseq 321
```

```
aqseq 312
```

before the actual pulse sequence. Without this statement, the status parameter AQSEQ would be set to an arbitrary value. In that case you can still set it after the acquisition has finished (before processing) with the command **3s aqseq**. In dimensions higher than 3D only the natural order (e.g. 4321) is supported and there is no corresponding aqsec command.

The `wr` statements and all other statements in the table *Writing acquisition buffers to disk [▸ 91]* can be specified behind a delay. See also the example in table *Usage of st and st0: noediff pulse program [▸ 90]*. The only timing requirement for `wr` is that the disk transfer is finished before `wr` is called again. If it is not, a run-time error message is printed. The actual execution time of a disk write depends on the computer hardware, the operating system, and the system load according to currently active processes and users. Bruker recommends acquiring data only to a disk that is physically connected to the computer that controls the spectrometer.

However, there is another limit if `wr` is used in conjunction with other commands for data handling. The commands `wr`, `if` (or `df` or `rf`, respectively) and `zd` may be combined on a single line. Other commands, like `ze` must be on a separate line. Any sequence containing one of these commands more than once must have a delay of 10 ms between two of them.

11 The mc Macro Statement

11.1 The mc Macro Statement in 2D

A 1D experiment can be based on the following pulse program sequence:

```
1 ze       ; initialization
2 d1       ; starting delay
  p1       ; pulsing
  d0       ; waiting
  go=1     ; acquiring FID and loop for adding
  d1 wr #0 ; write to buffer
```

You can turn this sequence into a 2D sequence by taking the following steps:

- Increment the file pointer after each disk write
- Initialize the buffer after each disk write
- Increment a delay, by convention `d0`, in each loop
- Add a loop outside of the `wr #0` statement to a second label - the size of which is usually `td1`
- For phase sensitive acquisition: add a phase increment

When the indirect dimension is acquired phase insensitive, the 2D pulse program would have the following form:

```
1 ze
2 d1
3 p1
  d0
  go=2
  d1 wr #0 if #0 zd id0
  lo to 3 times td1
```

The last two lines can be replaced by the `mc` statement. In the above sequence, this would take the form:

```
d1 mc #0 to 2 F1QF(id0)
```

The statement `mc` is a macro that includes a disk write (`wr`), a file increment (`if`) and memory initialization (`zd`). It can be used with one or more clauses, e.g. `F1QF`, which expands to a loop structure. Each clause can take one or more pulse program statements, e.g. `id0`, as arguments. These statements are executed within the loop created by the clause. Different `mc` clauses are used for phase sensitive, phase insensitive and echo-antiecho experiments. However, the same `mc` clause, i.e. the same pulse program, can be used for different types of phase sensitive experiments like QSEQ, States, TPPI and States-TPPI. The experiment type is determined by the F1 acquisition parameter FnMODE. The allowed combinations of FnMODE and `mc` clauses are listed in the next table.

`mc` clause	Mode	Possible values of FnMODE[F1]
`F1QF`	phase insensitive	QF
`F1PH`	phase sensitive	QSEQ, States, TPPI, States-TPPI
`F1EA`	Echo-Antiecho	Echo-Antiecho

Table 11.1: Allowed combinations of FnMODE and mc clauses

If an incorrect combination of FnMODE and `mc` clause is used, such as `F1PH` - `QF`, the **zg** command will show an error message and quit.

2D and 3D processing commands interpret the **acquisition status** parameter FnMODE and set the **processing status** parameter MC2 accordingly. However, if FnMODE = undefined, they interpret the **processing** parameter MC2 and set the **processing status** parameter MC2 accordingly.

By using `mc` instead of the `wr` and `lo to label` statements the same 2D (and 3D) pulse programs can be used for TPPI, States-TPPI and States, respectively. We will look at the expanded forms of `cosyph` for different values of FnMODE[F1]. The unexpanded pulse program as it appears with **edpul cosyph**:

```
"d0=3u"

1 ze
2 d1
3 p1 ph1
  d0
  p0 ph2
  go=2 ph31
  d1 mc #0 to 2 F1PH(ip1, id0)
  exit

ph1 =0 2 2 0 1 3 3 1
ph2 =0 2 0 2 1 3 1 3
ph31=0 2 2 0 1 3 3 1
```

The `mc` command will put into the expanded pulse programs the following header which is the same for the different values of FnMODE[F1]:

```
define delay MCWRK
define delay MCREST
"MCREST = d1 - d1"
```

As such, it is not specified in the expanded pulse programs below.

Note that MCWRK, MCREST are general delays that are defined during the expansion of the `mc` statement.

MCREST is zero for all expansions of `cosyph` but can be non- zero for other pulse programs.

MCWRK, however, is different for different expansions.

Note that the phase programs are, for each value of FnMODE, the same as in the unexpanded pulse program.

FnMODE[F1] = QSEQ:

```
define loopcounter ST1CNT
"ST1CNT = td1 / ( 2 ) "
"MCWRK = 0.500000 * d1"

1           ze
            "in0 = in0 / 2"
2 MCWRK
  LBLSTS1,  MCWRK
  LBLF1,    MCREST
3           p1 ph1
            d0
            p0 ph2
            go=2 ph31
            MCWRK wr #0 if #0 zd ip1 id0
            lo to LBLSTS1 times 2
            MCWRK rp1
            lo to LBLF1 times ST1CNT
exit
```

FnMODE[F1] = States

```
define loopcounter ST1CNT
"ST1CNT = td1 / ( 2 ) "
"MCWRK = 0.500000 * d1"
1          ze
2 MCWRK
  LBLSTS1, MCWRK
  LBLF1,   MCREST
3          p1 ph1
           d0
           p0 ph2
           go=2 ph31
           MCWRK wr #0 if #0 zd ip1
           lo to LBLSTS1 times 2
           MCWRK rp1 id0
           lo to LBLF1 times ST1CNT
exit
```

FnMODE[F1] = TPPI

```
"MCWRK = d1"
1        ze
         "in0 = in0 / 2"
2 MCWRK
  LBLF1, MCREST
3 p1 ph1
         d0
         p0 ph2
         go=2 ph31
         MCWRK wr #0 if #0 zd ip1 id0
         lo to LBLF1 times td1
exit
```

FnMODE[F1] = States-TPPI

```
define loopcounter ST1CNT
"ST1CNT = td1 / ( 2 ) "
"MCWRK = 0.500000 * d1"
1          ze
2 MCWRK
  LBLSTS1, MCWRK
  LBLF1,   MCREST
3 p1 ph1
           d0
           p0 ph2
           go=2 ph31
           MCWRK wr #0 if #0 zd ip1
           lo to LBLSTS1 times 2
           MCWRK id0
           lo to LBLF1 times ST1CNT
exit
```

The expanded version of the pulse program can be found in the *expno* directory of the dataset in the file *pulseprogram*. Note that the `mc` statement performs the following actions:

- In QSEQ, States, States-TPPI and Echo-Antiecho mode, `mc` creates two loops and sets the corresponding labels and delays. The delay at the line to which `mc` loops back to is split into two equal parts: one for the inner loop label and one for the outer loop label.

- For FnMODE[F1] = QSEQ or TPPI, the value for the delay increment is divided by 2 during run time. In older program versions, the parameter ND0 represented the number of occurrences of `d0` within the loop. In TopSpin 2.1 and later versions, this parameter is no longer used. It is replaced by an equation which determines the dwell time for this dimension. Both of them are independent from FnMODE.
- For FnMODE[F1] = QSEQ or States, an `rp1` statement is included within the outer loop. This causes the phases of `ph1` to be reset to their original values.
- For FnMODE[F1] = States, States-TPPI and Echo-Antiecho, the statements specified in the first argument of the `mc` clause are executed in the inner loop and the statements specified in the second argument are executed in the outer loop.
- For FnMODE = QSEQ, the statements specified in the first and second argument of the `mc` clause are executed in the inner loop.
- For FnMODE = TPPI, only one loop is created so the statements specified in the first and second argument of the `mc` clause are executed in that loop.
- For FnMODE = QF, the `mc` clause contains only one argument whose statements are executed in the only loop that is created.

For large 2D data sets, it is often useful to test the experiment with the first increment. This can be done by setting the parameter TD[F1] to 1. The dimension of the generated dataset will be 1D and can be processed as such. Note that you do not have to change the value of the parameter PARMODE; it is still set to 2D. In the same way, you can acquire a plane of a 3D dataset by setting TD[F1] or TD[F2] to 1, and a single row by setting TD[F1] and TD[F2] to 1.

11.2 The mc Macro Statement in n-Dimensional Experiments

The `mc` statement can also be used in nD pulse programs. In this case, there are n-1 indirect dimensions, F1 to Fn-1. For the F1 dimension, `mc` uses the clauses `F1PH`, `F1EA` and `F1QF`, for the F2 dimension, it uses the clauses `F2PH`, `F2EA` and `F2QF` and so on.

The `F2PH` clause creates a second loop within which a second delay is varied.

The pulse program `noesyhmqcpr3d`:

```
aqseq 312

1 d11 ze
2 d11 do:f2
3 d12 pl9:f1 pl2:f2
...
  go=2 ph31 cpd2:f2
  d11 do:f2 mc #0 to 2
  F1PH(ip1 & ip29, id0)
  F2PH(rd0 & ip5, id10)
  exit
```

with

FnMODE[F2] = States-TPPI

FnMODE[F1] = States-TPPI:

expands to :

```
aqseq 312

define delay MCWRK
define delay MCREST
define loopcounter ST2CNT
"ST2CNT = td2 / ( 2 ) "
define loopcounter ST1CNT
```

```
"ST1CNT = td1 / ( 2 ) "
"MCWRK = 0.166667 * d11"
"MCREST = d11 - d11"
1 d11 ze
2 MCWRK*2 do:f2
  LBLSTS2, MCWRK
  LBLF2, MCWRK*2
  LBLSTS1, MCWRK
  LBLF1, MCREST
3 d12 pl9:f1 pl2:f2
  ...
  go=2 ph31 cpd2:f2
  MCWRK do:f2 wr #0 if #0 zd ip1
  MCWRK ip29
  lo to LBLSTS2 times 2
  MCWRK id0
  lo to LBLF2 times ST1CNT
  MCWRK rd0
  MCWRK ip5
  lo to LBLSTS1 times 2
  MCWRK id10
  lo to LBLF1 times ST2CNT
  exit
```

If you reverse the acquisition order of this pulse program, i.e. if you specify:

```
aqseq 321
```

you have to change the `mc` clauses to:

```
F1PH(rd10& ip1 & ip29, id0)
F2PH(ip5, id10)
```

For dimensions higher than 3 only the natural order is supported which is for example in 4D 4-3-2-1.

11.3 Additional mc Clauses

Apart from the `mc` clauses specified above two further clauses are available:

- `F1I`
- this clause is typically used for interleaved experiments where parameters have to be varied independently from the `ip`/`id` scheme required for the actual 2D.
- `F0`
- this clause is used when a parameter needs to be varied without incrementing the data file pointer.

Both `F1I` and `F0` expand to an additional inner loop.

As an example of the `F1I` clause, we will take the pulse program `noesygpphprxf`; with FnMODE[F1] = States-TPPI:

```
1 ze
  d11 pl12:f2
2 d11 do:f2
3 d12 pl9:f1
  ...
  go=2 ph31 cpd2:f2
  d11 do:f2 mc #0 to 2
  F1I(ip3*2, 2, ip13*2, 2)
  F1PH(ip4 & ip5 & ip29, id0)
  exit
```

will expand to

```
"ST1CNT = td1 / ( 2 * 2 * 2 ) "
"MCWRK = 0.166667 * d11"
"MCREST = d11 - d11"
1 ze
  d11 pl12:f2
2 MCWRK do:f2
  LBLF1I1, MCWRK
  LBLF1I2, MCWRK*3
  LBLSTS1, MCWRK
  LBLF1, MCREST
3 d12 pl9:f1
...
  go=2 ph31 cpd2:f2
  MCWRK do:f2 wr #0 if #0 zd ip3*2
  lo to LBLF1I1 times 2
  MCWRK ip13*2
  lo to LBLF1I2 times 2
  MCWRK ip4 MCWRK ip5 MCWRK ip29
  lo to LBLSTS1 times 2
  MCWRK id0
  lo to LBLF1 times ST1CNT
  exit
```

The pulse program line below shows how the F0 clause can be used:

```
d1 mc #0 to 1 F0(id9) F1QF(id0)
```

will be expanded to:

```
...
d1*0.5 id9
lo to 2 times td0
d1*0.5 wr #0 if #0 zd id0
...
```

As a loop counter, the parameter TD0 is evaluated.

In order to be able to switch dimensions, timing of statements within the loops must be controlled by the `mc` statement. So delays or pulses should not be used as argument to the `F0`, `F1PH` ... clauses of the `mc` statement. But in some cases statements must be separated by an delay. Precautions have been taken for this case: the & symbol used within an argument of F0, ... will be substituted by an equal fraction of the delay with which the `mc` statement was specified, e.g.

```
d1 mc #0 to 1 F0(ip1 & ip3)
```

will expand to

```
MCWRK ip1 MCWRK ip3
lo to 3 times td0
```

For 3D pulse programs, the clauses `F1I` and `F2I` are available for the two indirect dimensions.

11.4 General Syntax of mc

The syntax for the `mc` statement is

```
label <delay1> <options>
...
...
<delay2> <options> mc #<buffer> to <label>
F0(<statements>)
```

```
F1I(<sts>,<no. of loops>,<sts>,<no. of loops>, ...)
F1PH(<statements>,<statements>)
F1QF(<statements>)
F1EA(<statements>,<statements>)
F2I(<sts>,<no. of loops>,<sts>,<no. of loops>, ...)
F2PH(<statements>,<statements>)
F2QF(<statements>)
F2EA(<statements>,<statements>)
```

The following rules hold:

- `<label>` must be followed by one delay and can be followed by options.
- `<delay1>` must be greater than or equal to `<delay2>`.
- multiple clauses like `F0()`, `F1PH()`, … can be specified on the same line or on consecutive lines. Do not specify any other statements between the clauses.
- The order in which `F0()`, `F1PH()`, ... clauses occur is irrelevant.
- In 3D, the statement `aqseq 312` determines the order of the F1 and F2 loop.
- The pulse program must contain a `ze` statement after the parameter definitions.
- The symbol `&` is required between multiple statements of the same type (e.g. multiple phase increments) that are specified within one argument.

After expansion, each statement will appear with a separate delay (see the example in section *Additional mc Clauses [▸ 99]*). Multiple statements of a different type (e.g. a phase increment and a delay increment) can be specified with a & symbol or with a white space in between. In the latter case, after expansion, they will appear together behind one delay.

The next table shows, which expansions will be done (X) for different values of

FnMODE.

FnMODE	**create a double loop**	**delay-increment div. by 2**	**phase reset**	**phase inc inserted**	**delay inc in inner loop**
QF					X
QSEQ	X	X	X	X	X
TPPI		X		X	X
States	X		X	X	
States-TPPI	X			X	
EA	X			X	

Table 11.2: Results of use of different FnMODEs

Note the following when you look at the expanded pulse program:

- MCWRK is a fraction of `<delay2>` and is calculated according to the number of arguments of the `mc` clauses.
- MCREST is the difference between `<delay1>` and `<delay2>`.
- The generated labels have names like LBLF*. Please do not use labels with these names in your own user-defined pulse programs.
- A line starting with a `#` is a comment to the statement(s) that follow it.

The comment contains the respective line number in the original pulse program, and, if applicable, the expansions that were made.

11.5 Advanced nD Sampling Techniques

11.5.1 The Parameter FnTYPE

While in traditional nD acquisition each plane is measured completely before the next plane begins, there are new techniques where planes are not measured completely and in various orders. The new techniques are described by the parameter FnTYPE where FnTYPE="*traditional*" stands for the conventional technique.

FnTYPE="*full(points)*" stands for a technique where all 2^{n-1} FIDs belonging to a complex point of an nD acquisition are measured together.

FnTYPE="*non uniform sampling*" is a technique where the complex points which are measured are described by a list of coordinates. Each complex point (FID) of an nD experiment is described by n-1 coordinates. These coordinates have to be put in a variable loopcounter list in the order Fn-1, Fn-2, ..., F1. The number of FIDs measured is determined by the length of the list divided by (dimension - 1). The name of the list must be set with the parameter NUSLIST. The non-uniform sampling technique reduces the number of points measured and in this way the total experiment time.

FnTYPE="*projection spectroscopy*" will eventually result in 2D projections of a nD experiment onto the Fn-1 dimension. For this the increments are scaled according to the projection angles specified in **eda** (ProjAngle) via sin/cos functions. For each point in time a hypercomplex datapoint is acquired and all delays are incremented at the same time. The data need to be split into different 2D projections (for + and - alpha (3D), +alpha/+beta, +alpha/-beta, -alpha/+beta, -alpha/-beta (4D) etc.) prior to Fourier Transformation.

Each FnTYPE requires a different kind of processing. The new `mc` syntax is compatible with all FnTYPEs and no modification in the pulse program is necessary if different FnTYPEs are used. The processing software can detect the order in which data are stored on disk from the status parameter FnTYPE.

11.5.2 Parameter Calculation from Complex Point Indices

For all these kinds of acquisition the conventional form of the `mc` command is no longer sufficient. Instead, the increments of the incremental delays and the phase programs must be calculated from the indices of the complex point in each dimension. For this purpose 6 new commands are available which can be used to calculate these increments:

`calph(phX, increment)`	calculates the phase program increment and increments the phase program.
`caldel(dX, increment)`	calculates the delay which is necessary for a set of indices.
`calgrad(list, increment)`	calculates and sets the list position of a gradient list.
`calclc(lX, increment)`	calculates the loop counter `lX` for a set of indices.
`calclist(listX, increment)`	calculates and sets a list pointer for the listX.
`calphptr(phX, increment)`	calculates and sets the pointer in phase program `phX`.

Table 11.3: Parameter calculation from complex point indices

An example from a 3D pulse program illustrates this:

Conventional pulse program:

```
F1PH(ip5, id0)
```

```
F2PH(igrad EA & rd0 & rp5 & ip7 & ip9, dd10 & id20 & id30)
```

Using the new syntax the same is expressed as

```
F1PH(calph(ph5), caldel(d0))
F2PH(calgrad(EA) & calph(ph7) & calph(ph9) & caldel(d10, -in10) &
caldel(d20) & caldel(d30))
```

One can see that the new form of the command needs less arguments than the conventional one since it is no longer necessary to reset delays and phase programs.

The second argument of the `cal...` commands describes the increment, for `calph` it is the increment of the phase program in degrees, for `caldel` it is the increment of the delay in sec. The argument can be omitted, in this case it defaults to 90º for phases and for delays to the IN parameter of the delay specified in the first argument (IN0 for d0 etc.). For the `calgrad` command the first argument is the name of the list, second (optional) argument is a multiplier for the increment (+1 per default, -1 corresponds to the `dgrad` command). For `calclc` the first argument is the name of the loopcounter and the second the increment (+1 per default, -1 corresponds to the `du` command). `calclist` calculates a list pointer. If the calculation results in a pointer outside the list, a modulo calculation is done to keep the result within the list. The same is true for `calgrad` and `calphptr`.

If a `cal...` statement appears before the comma, the variable which underlies the calculation is the variable of the inner loop, i. e. the phase loop. Statements after the comma refer to the outer loop, i. e. td loop. If there is no comma, because the statement has no inner loop, the result will refer to the outer loop. If the same statements appear before and after the comma, the results of both calculations are added, for example

```
F1PH(calph(ph5), calph(ph5))
```

will result in a phase `ph5` which varies in the inner and in the outer loop. If however the same statement appears in different dimensions, the second calculation overwrites the result of the first calculation.

```
F1PH(calph(ph5), ...) ; useless, result will be overwritten
                        by the next line
F2PH(calph(ph5), ...)
```

In such a case different phase programs have to be used for different dimensions in conjunction with pulses where the phase programs are added:

```
p5:f1 ph5+ph6
...
F2PH(calph(ph5), ... )
F1PH(calph(ph6), ... )
```

11.5.3 Old and New Syntax

If the new syntax is used, all other commands within the `mc` clauses are forbidden. There are two exceptions:

- The inner loops `FnI` must contain only conventional commands.
- If a conventional command must be used in conjunction with the new syntax, one can put it into an `exec` statement in order to switch off the syntax check, for example:

  ```
  F1PH(...., exec(fxlist.inc))
  ```

12 Subroutines

12.1 Definition

Like in most programming languages, also in pulse programs certain parts of a program can be implemented within more or less generally applicable subroutines - for example if a certain set of instructions has to be executed several times. A subroutine definition can be in the header of a pulse program or in an include file. It consists of the keyword subroutine and the subroutine name followed by the argument list in parentheses and by the subroutine body in curly braces:

```
subroutine SR1(<arguments>)
{
  <subroutine body>
}
```

It is important to emphasize at this point that, in contrast to other programming languages, a call of a subroutine leads to a place holder which later gets replaced by the actual instructions defined in the subroutine. This has the important consequence that it is not possible to declare local variables within a subroutine. Furthermore, there exists no concept like a return value for a subroutine. A class name may optionally be appended to the subroutine name. This serves to identify the subroutine and to prevent it from illegal usage:

```
subroutine SR1:CLASS1(...)
```

These classes are dedicated to improving the code structure in large pulse programs with many subroutines. The class concept is only maintained within a certain pulse program (and its include files), i.e. it does not provide limitations towards the usage of a subroutine to a special subset of pulse programs.

The allowed argument types in the definition of a subroutine are restricted to the following set:

```
pulse, delay, loopcounter, phase, gradient, any.
```

Hereby `any` allows to use arguments of a type not explicitly contained in the above set (e.g. a channel) but then, in contrast to the explicit types, no type check is performed.

A simple example for the definition of a subroutine is given below:

```
subroutine SR1:CLASS1(delay dla, pulse pls, any chn)

{

   dla pls:chn ;delay and a pulse on a channel in a subroutine

}
```

Subroutine calls within subroutines are possible. In contrast to the introductory remarks about the absence of local variables, labels in subroutines are local to this subroutine and may be used by loops within the same subroutine.

12.2 Subroutine Execution

A subroutine is called with the keyword `subr`. For the small example subroutine from above this is shown below:

```
subr SR1:CLASS1(d1, p1, f1)
```

This call defines a place holder which during pulse program compilation will be replaced by the following code:

```
d1 p1:f1
```

For this example the call `subr SR1:CLASS1(ph1, p1, f1)` would lead to a compilation error, because the type `phase` of the first argument in the call is not identical to the type `delay` of the first argument in the definition of the subroutine. On the other hand, the call `subr SR1:CLASS1(d1, p1, sp1:f1)` would not lead to an error (if the shape `sp1` is known) because the type of the last argument in the subroutine definition is `any` and the instruction `p1:sp1:f1` is valid syntax - even though it might not do what it originally was intended to.

If subroutines are defined with a class specification, it is mandatory that the same specification is added to the call of the subroutine. If labels are specified, the pulse program compiler changes them into unique names to avoid multiple definitions.

12.2.1 Subroutines with Variable Names

Different subroutines can be called from a pulse program without changing the pulse program itself by using the parameters SUBNAM1-16:

```
subr <$SUBNAM1>(d1, p1, f1)
```

The compiler will insert the subroutine whose name is in SUBNAM1 at this place.

The precompiled pulse program `pulseprogram` in the raw data directory always shows the expanded subroutine together with lines which mark its begin and end.

12.2.2 Subroutines with Variable Names and Classes

It is allowed that subroutine definitions can have the same name within different classes. For instance, the class `tocsydef` defines subroutines which are necessary in the definition phase, and the class `tocsy` subroutines in the execution phase of the program. Both subroutines can then be called with the same name SUBNAM1. In this case it is necessary that for each member of class `tocsy` there exists a corresponding member of class `tocsydef` and vice versa.

13 Miscellaneous

13.1 Multiple Receivers

For multi-receiver experiments, you can specify in the pulse program which receivers you want to use to acquire the data. The logical receiver number (1-8) can be appended to the following statements:

`go, gonp, gosc, goscnp, adc, rcyc, rcycnp, eosc, eoscnp, ze, zd, st, st0, aq, dw, dwov, recph, wr, if, acqt0_F`

For example the statement

```
go5=label
```

acquires the data with the receiver which is connected to the logical channel F5.

If no number is specified, 1 is assumed, i.e. `go` is equivalent to `go1`.

Parameters for the first transceiver are taken from the current dataset. Parameters for the nth transceiver are taken from data set n. The EXPNO of this dataset is defined in the acquisition parameter MULEXPNO[n].

The following parameters are taken from the dataset defined by MULEXPNO:

AQ_mod, DECIM, DIGMOD, DIGTYP, DQDMODE, DR, DSPFIRM, DSPFVS, FTLPGN, NBL, OVERFLW, SEOUT, SFO1, SW, TD.

All other parameters are taken from the current dataset.

The command `wr <m> #<n>` then writes out the data of this m-th receiver channel to the dataset n (defined via MULEXPNO[n]). To avoid conflicts one should use for m and n numbers with m=n+1 in the same command.

The routing of the receivers can be inspected and changed within **edasp**. If you click **show receive routing** then you can see from right to left the logical receive channels with their respective nuclei and transceivers. The acquisition parameters RECCHAN, RECSEL and RECPRE describe these connections.

13.2 Real Time Outputs

The spectrometer TCU provides a number of real time outputs which are used to control various spectrometer components, such as gating and blanking the transmitters. Please refer to your hardware documentation to find out which output is connected to a particular device. The pulse program compiler will select the correct output automatically, e.g. for a statement like `p1:f2`.

The file *$xwinnmrhome/exp/stan/nmr/lists/pp/Avance.incl* contains a number of macro definitions based on the outputs, which can be used in pulse programs. This file can be viewed with the command **edpul Avance.incl**. Depending upon the spectrometer type this file can have different contents after **expinstall**.

The hardware documentation will also inform you which of the outputs are free for special purposes, e.g. for controlling a laser from a pulse program.

13.2.1 Type 1 Outputs (“NMR control words“)

On AV systems, there are 3 outputs called RCP32, RCP33, and RCP34 in NMR0 and 32 outputs in NMR3 and 4, respectively. They can be enabled or disabled from the pulse program in a way illustrated by the examples below.

```
1u setnmr0 | 32 ; set gradient blanking for z gradient
```

activates output channel 15 (using active=low logic). The channel remains active until it is explicitly deactivated, e.g. with the statement:

```
1u setnmr0 ^ 32
```

The characters “|“ (vertical bar) and “^“ (circumflex) can be used to set and clear a bit in a register consisting of 3 bits. As such, several outputs can be enabled or disabled simultaneously. For example, the statement:

```
1u setnmr3 ^8^9 ; switch QNP to X
```

disables the output channels 8 and 9 of NMR3.

The statement `setnmr` must be specified behind a delay (in the above examples it is 1 µsec). The minimum delay is 50 ns.

An alternative format for the `setnmr` command allows the input of whole words in hexadecimal:

```
setnmr3 0xffff0000 ; set the high 16 bits of nmr word 3.
```

13.2.2 NCO Control

The control of the NCOs has been simplified and put together into the

`reset` command (see the next table). Reset of each NCO in each channel can be done individually (please note that a delay has to be specified at the beginning of the line). Different reset commands for different channels can be combined:

```
1ureset:f1:f2         ; reset all NCOs on channel f1 and f2
1ureset1:f2 reset4:f3 ; reset NCO1 on channel f2 and NCO3
                     ; from NCO2 on f3
```

Command	Action
`reset`	set accumulator to 0, reset NCO1, 2 and 3 from accum.
`reset1`	reset NCO1 to 0
`reset2`	reset NCO2 to 0
`reset3`	reset NCO3 to 0
`reset4`	reset NCO3 with phase from NCO2
`reset5`	reset NCO3 with phase from NCO1
`reset6`	reset NCO2 with phase from NCO1
`reset7`	reset NCO3 from NCO2, reset NCO2 from NCO1
`reset8`	reset NCO3 and NCO2 from NCO1
`reset9`	reset NCO1 from accumulator
`reset10`	reset NCO2 and NCO1 from accumulator
`reset11`	reset NCO3 from NCO2, NCO2 and NCO1 from accum.
`reset12`	= reset

Table 13.1: Reset commands

13.2.3 Real Time Pulses RTP

The real time outputs are used for the control of the digitizer. They are set implicitly with the `go` command in pulse programs. If an explicit control is needed, the command `setrtp<logical channel> | <bit> ^ <bit>` can be used. For channel F1, for example, the command `setrtp1` would be used.

The usage of `setrtp` is analogous to the `setnmr` commands. Table *Table 13.2 [▸ 109]* describes the assignment of these bits.

Bit#	Symbolic Name in Pulse Program	Action
0	`START_NEXT_SCAN`	Tell receiver to acquire TD points (is switched off automatically)
1	`DWELL_HOLD/ DWELL_RELEASE`	Mute/Continue acquisition of TD points
26	`REC_BLK/REC_UNBLK`	Start/stop acquisition on receiver (generates RGP_HPPR, RGP_ADC, RGP_RX automatically)

Table 13.2: Real Time Pulses on the TRX

The receiver gating pulses RGP_ADC, RGP_RX and RGP_HPPR are generated automatically. An individual setting of these pulses is not possible.

Additionally, there are some real-time parameters that are set via GTU.

The syntax is similar to the real-time pulses above, but index zero is used instead of the logical channel in the `setrtp` command:

```
setrtp0 | <bit> ^ <bit>
```

Bit#	Symbolic Name in Pulse Program	Action
0	`LOCKH_ON/LOCKH_OFF`	Lock Hold
1	`SHIM_XY_RAMP_ON/ SHIM_XY_RAMP_OFF`	
2	`HOMOSPOIL_ON/ HOMOSPOIL_OFF`	Homospoil gradient on/off
6	`TTL1_LOW/TTL1_HIGH`	
7	`TTL2_LOW/TTL2_HIGH`	
8	`TTL3_LOW/TTL3_HIGH`	
9	`TTL4_LOW/TTL4_HIGH`	
47	`RCP_HPPR_MODE (former INTERLEAVE_INCR)`	Switch to next mode in Hppr module

Table 13.3: Real Time Pulses on the GTU

13.3 Miscellaneous Statements

13.3.1 Receiver Gain Lists

It is possible to vary the receiver gain of the used receiver channels during pulse program execution.

This can be achieved by using receiver gain lists.

Similar to the other lists occurring at several places in this manual such a receiver gain list is defined and initialized as follows:

```
define list<receiverGain> rgList = {1 2 4 8 45.2}
```

Please note that the assigned values can be floating point numbers. The user not necessarily has to take care of the exact values he provides (due to the fact that the hardware only allows for certain discrete receiver gain values) because the software will automatically select the actually allowed values closest to the list values.

The receiver gain where the list pointer currently points to can be assigned to a logical channel by the following syntax:

```
d1 rgList:f1
```

where the delay `d1` and the channel `f1` were just exemplarily chosen. When this statement is executed after the list definition without further modifications, in the above example then the value 1 will be loaded to the receiver routed to the logical channel `f1`.

To modify the list pointer, the usual modification methods were provided:

```
rgList.inc          ;increment the list pointer
d1 rgList:f1        ;now the value 2 will be assigned to channel
                    f1
rgList.dec          ;decrement the list pointer
d1 rgList:f1        ;now the value 1 will be assigned to channel
                    f1
"rglist.idx = 3"    ;set the list pointer to position 3
                    ;(counted from 0)
d1 rgList:f1        ;now the value 8 will be assigned to channel
                    f1
rgList.res          ;reset the list pointer to position 0
d1 rgList:f1        ;again the value 1 will be assigned to channel
                    ;f1
```

To directly assign a specific value to a channel without globally changing the list pointer one can also use the [ ]-operator:

```
d1 rgList[2]:f1     ;assign the third value, here 4, to channel f1
d1 rgList:f1        ;again the value 1 will be assigned to channel
                    ;f1
```

In order to fit loop counters in a pulse program to a receiver gain list, the length of the list can be accessed via:

```
"l1 = rgList.len"
```

13.3.2 Assignment of Transmitter Blanking Pulses: blktr

The transmitter blanking pulses are normally set by the *edscon* preset parameters (BLKTR[1...8]). They can, however, also be declared at the beginning of the pulse program using the syntax:

```
blktr<channel number> = <duration>
```

Example:

```
"blktr1=3u"
```

NB: for AV I and II the assignment of the BLKTR parameters to channels is different to those of AV III. In the former the assignment was a logical assignment, i.e. if F1 controlled SGU2 and from there amplifier 3, BLKTR1 was applied to channel F1 and in this way to amplifier 3, whereas in AV III BLKTR is always applied to the amplifier with the same number (which is determined by the blanking signal of this amplifier).

13.3.3 Printing Messages

It seems that almost all programming languages have been invented to offer some way to write "*hello world*": here is the way how to achieve this with the pulse programming language:

The statement `print` prints messages during the execution of the pulse program. Such messages should begin either with "error" or with "warning". They are useful in cases where the pulse program itself detects an error condition or follows a different path (see `goto` in the section *The Unconditional goto Statement [*▸ 69]). An Example is

```
print "warning: Hello World"
```

prints the message *Hello World* during runtime of an experiment. The timing of the printout follows the interpretation of the pulse program on the TCU and therefore may be ahead in time of the execution of the pulse program. However, for printing an error message in a case where the program must be aborted and for debugging complex pulse programs it could be helpful.

You can also write to a file with this command: if the statement in quotes begins with the word `file:` the next word is interpreted as a filename and the rest of the string as message which is written into that file. **Example:**

```
print "file: tcutext hello from tcu"
```

will generate the file *tcutext* in the raw data directory and its contents will be "*hello from tcu*".

13.3.4 Specify a Minimum Required Version

If you write and distribute your own pulse programs you may want to specify a minimum version of TopSpin that is at least required for your pulse program to work properly. This can be achieved by the optional pulse program statement `require`.

The general syntax is:

```
require "<program name> <major version>.<minor version>.
<patchlevel>"
```

Currently, only “TopSpin” is supported as program name (white spaces inside the relation will be ignored).

For example

```
require "TopSpin 4.1.0 "
```

will abort with an explanatory error message if your pulse program is executed in any version of TopSpin older than 4.1.0.

Please note that this functionality is available starting with TopSpin 4.0.8.

Older versions of TopSpin will nevertheless also abort with an error message, even though the error message will then be a compilation error due to an unknown syntax element and therefore less clear to understand.

13.3.5 Force FID-Scaling

For certain experiments that vary the number of scans per FID (NS) during the runtime of the pulse program, it might be useful to scale each FID individually by the respective number of accumulated scans (NS).

This can be achieved by using the statement `scaleByNs` at the beginning of the pulse progam.

For example, in the following hypothetical segment of a 2D pulse program, NS is set to ‘4’ for the first FID, and then is always increased by a factor of 2 for each of the next FIDs.

The `scaleByNs` command ensures that all FIDs are divided by their respective NS and therefore become comparable at the end of the experiment.

```
scaleByNs



"l13=4"


1 ze
2 30m
3 d1
"ns = l13"
p1 ph1
go=3 ph31
30m wr #0 if #0 ze
"l13 = l13 * 2"
lo to 2 times td1
exit
```

13.3.6 Generating a Manifest File

General Idea:

A manifest block is a special block of text in the pulse program which will not directly be interpreted as an actual pulse program statement but written to a text file “manifest.txt” in the current dataset directory.

The main purpose of this feature is that such a manifest block can be used to define statements and processing recipes for the current experiment in a hierarchical, machine readable data format (such as XML or JSON) which can be further processed by other third party software (such as XML-parsers or interfaces to data bases).

In order to allow as much flexibility as possible in terms of the target format or language, the manifest block itself will not be checked for valid syntax by TopSpin; this responsibility is left to the user and his dedicated tool chain or application for post-processing of the manifest files.

The only point where the pulse program syntax comes into play is that there exists the possibility to evaluate pulse program variables within a manifest block – then the actual current value of the variable will be written into the manifest file.

Syntax

A manifest block is enclosed between the two tokens `manifest_append` and `manifest_end` and can be defined (in an arbitrary number) and used only at compile time, i.e. before the first `ze` command in the pulse program.

The first manifest block of the pulse program will override an existing manifest file in the current data set, but all succeeding blocks will be appended.

In case no manifest block is defined in the pulse program a potentially existing manifest file will be deleted in the current dataset.

Between the manifest tokens, any kind of text is allowed, i.e. alphanumeric characters, blanks, tabs, newlines, as well as special characters (i.e. "!,:;+\-*/=(){}<>^[]|#&_%?@~°§ etc.)

A manifest block can be written either in one single line or may also contain several lines in the pulse program. If it contains several lines, each line in the pulse program will also result in a single line in the file “manifest.txt”. Thereby also spaces will be kept in order to maintain the manifest block’s indentation in the file “manifest.txt”.

A backslash at the end of a line will result in an ignored line break and no backslash in the file “manifest.txt”. If a backslash at the end of a line should explicitly be printed into the file “manifest.txt”, this backslash should be followed by a space.

The character $ is also allowed but has a special meaning as it can be used to retrieve the value of a variable known to the pulse program parser. Therefore, all alphanumeric characters following a $ without blanks will be interpreted as variable names.

And as a result, the current value of these variables will be printed into the manifest file.

For example,

```
define delay timespan
"timespan=42.0"
"p1=5"

manifest_append<first>
< timespan = "$timespan" unit="s" />
< p1 = "$p1" unit="us" />
</first>manifest_end

"p1=p1/2"
"timespan=17"

manifest_append
<second>
                < timespan = "$timespan" unit="s" />
                <inner>
                               < p1 = "$p1" unit="us" />
                               < timespan+p1 = "$timespan + $p1" />
                </inner>
</second>
manifest_end
```

will generate the following entries in the manifest file:

```
<first>
< timespan = "42.000000" unit="s" />
< p1 = "5.000000" unit="us" />
</first>

<second>
                < timespan = "17.000000" unit="s" />
                <inner>
                          < p1 = "2.500000" unit="us" />
                          < timespan+p1 = "17.000000 + 2.500000" />
                </inner>
</second>
```

Please note that, in case a variable is modified by multiple relations (like `timespan` and `p1` in the example above), always the most recent value will be printed into the manifest file.

In this way, the change of a variable can be tracked by positioning several manifest blocks between the modifications.

14 Contact

Manufacturer

Bruker BioSpin GmbH
Rudolf-Plank-Str. 23
D-76275 Ettlingen
Germany

E-Mail: *nmr-support@bruker.com*
http://www.bruker.com
WEEE DE43181702

Bruker BioSpin Hotlines

Contact our Bruker BioSpin service centers.

Bruker BioSpin provides dedicated hotlines and service centers, so that our specialists can respond as quickly as possible to all your service requests, applications questions, software or technical needs.

Please select the service center or hotline you wish to contact from our list available at:

https://www.bruker.com/service/information-communication/helpdesk.html

List of Figures

List of Tables

Glossary

Index

Bruker Corporation

info@bruker.com
www.bruker.com

Part No: H166633