NMR SUITE

Pulse program

Reference Manual
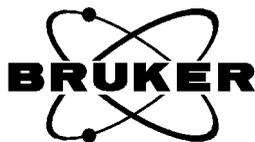
Bruker software support is available via phone, fax, e-mail, Internet, or ISDN. Please contact your local office, or directly:

Contents

Chapter 1

Basic pulse program writing

1.1 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 XWIN-NMR commands **edpul** or **edcpul** (see the Acquisition Reference manual). The XWIN-NMR 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 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.

Spectrometer naming conventions

This manual is written for Avance spectrometers. Nevertheless, a large part of it is also valid for older spectrometers like AMX, ARX and ASX. Since the end of 1999, Bruker delivers a new type of Avance spectrometers that are specified in this manual as Avance-AQS. The conventional Avance spectrometers are spec-

ified as Avance-AQX. You can easily find out which type of spectrometer you have by opening the cabinet door; one of the racks is named either AQS or AQX. Note that there are three types of Avance-AQX spectrometers: DMX, DRX or DPX. These specific names are used in this manual whenever a description only holds for one or two of them.

1.2 Pulse program library

Routine users normally use Bruker pulse programs that delivered with XWIN-NMR. 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 XWIN-NMR. This command copies the pulse programs suitable for your spectrometer into the appropriate directory.

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

1.3 Pulse program display

A graphical representation of a pulse program for Avance type spectrometers can be obtained with the command **pulsdisp**, which is described in the Acquisition Reference manual. After writing your own pulse program, **pulsdisp** will not only check its syntax, but it will also allow you to display any timing detail before you start an experiment.

1.4 Basic syntax rules

Table 1.1 shows *zgcw30* as an example of a simple Bruker pulse program. Here the following pulse programming rues are used:

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

2. 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.

3. #include <filename> or #include "filename"

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

      #include <Avance.incl>

      1 ze
        d11 pl26:f2
        d11 cw:f2
      2 d1
        p1*0.33 ph1
        go=2 ph31
        wr #0
        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 1.1 Pulse program example

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.

4. 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, lo 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.

5. `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 PL[14].

6. `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`. This delay and cw decoupling will begin immediately after the delay specified in item 5 has finished.
Items 5 and 6 illustrate 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.

7. `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.

8. `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 PL[1]. PL[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 PL[7]. It must be put on a separate line, with a delay, before the line with the pulse statement. This gives the transmitter some time to settle before the pulse is executed.

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.

9. `go=2 ph31`
Execute 1 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 signal. During this time the receiver gate is opened. For AQ_mod = DQD and for any value of AQ_mod if you have an RX22 receiver, 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, DE2, 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. AMX/ARX spectrometers require 6 millisec.

10. `wr #0`
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 at any time 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 1000 scans. To accomplish this, you can set NS=1000, and add the line:

```
lo to 1 times 30
```

before the `exit` statement. The pulse program then accumulates a total of 30.000 scans, but stores the result every 1000 scans.

Please note that the loop must include the `ze` 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.

11. `exit`
Specifies the end of the pulse program.

1.5 Pulse generation

Table 1.2 shows the available types of statements for the generation of high frequency pulses.

p0, p1, ... , p31	Generate a pulse whose length is given by the acquisition parameter P[0], ..., P[31].
3.5up, 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.
vp	Generate a pulse whose length is taken from a pulse list.
pulse(*flipangle*, auto, *duration*) pulse(*flipangle*, *power*, auto)	Generate a pulse causing the specified flip angle.

Table 1.2 Pulse generation statements

A high frequency pulse is described by its:

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

The following paragraphs will describe these items.

1.5.1 Pulse duration

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

1.5.1.1 <u>p0</u>-<u>p31</u>

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 width P[1].

1.5.1.2 Fixed length pulses

The statement:

```
10mp
```

executes a pulse of width 10 millisec (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 would omit the terminating "p", a delay would executed instead of a pulse.

1.5.1.3 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.

1.5.1.4 Variable list pulses

The statement:

```
vp
```

executes a pulse whose duration is given by the current value of a *pulse list*. A pulse list is a text file that contains one pulse duration per line. It can be set up with the command **edlist** (described in the Acquisition Reference manual). The statement vp uses the list file given by the acquisition parameter VPLIST. When the pulse program begins, the first duration in the list is used. The statement ivp moves the list pointer to the next duration. If the end of the list is reached, the pointer is set to the first item. The statement ivp must be specified behind a delay, for example:

```
d1 ivp
0.1u ivp
```

The length of the delay is irrelevant; any value is allowed.

You can also set a specific list position with an equation. For example:

```
"vpidx=5"
vp
```

The statement vp will execute the pulse defined at position 5 of the pulse list. To the right of the equal sign, any dimensionless expression is allowed. It may contain any of the parameters listed in Table 1.3.

1.5.1.5 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> Plist = { 10 20 30 }
```

This statement defines the pulse list Plist 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. 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 set 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 msec, change index from 0 to 1
locallist locallist.res ; pulse of 20 msec, set index to 0
locallist[2]            ; pulse of 30 msec (do not interpret the index)
locallist locallist.dec ; pulse of 10 msec, change index from 0 to 3
locallist               ; pulse of 40 msec
"locallist.idx = 3"     ; set index to 3
locallist^              ; pulse of 40 msec, move index to 0
locallist               ; pulse of 10 msec
```

<u>Caution</u>: index operations on pulse lists only take effect in the next line. Further-

more, 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 ms) twice
locallist                 ;the ^ operator takes effect: 20ms
locallist[2] locallist[3] ;executes locallist[3] (40 ms) twice
```

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.

1.5.1.6 Manipulating pulse durations: The operator

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

```
p1*1.5

p30d1H*3.33

p3*oneThird

vp*3
```

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.

1.5.1.7 Manipulating pulse durations: Changing p0-p31 by a constant value

Each pulse statement p0-p31 has been assigned an acquisition parameter INP[0]-INP[31] These parameters take a duration value, in μsec. The pulse program statements ipu0-ipu31 add the value of INP[0]-INP[31] to the current value of p0-p31, respectively. Likewise, dpu0-dpu31 subtract the value of INP[0]-INP[31] from the current value of p0-p31. The statements rpu0-rpu31 reset p0-p31 to their original values, i.e. to the values of the parameters P[0]-P[31]. The statements presented in this paragraph must be specified behind a delay of any length (≥ 0) . Some examples:

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

1.5.1.8 Manipulating pulse durations: Redefining <u>p0</u>-<u>p31</u> via an expression

The duration of the pulses `p0-p31` is normally given by the parameters P[0]-P[31]. 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 Table 1.3.

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 cannot be used in labelled pulse program lines. You can, however, put a small duration behind a label and put the expression in the next line.

Expressions do not cause an extra delay in the pulse program. Pre-evaluation is applied before the pulse program is started, and the result is stored in the available buffer memory to be accessed at run time. At run time, pre-evaluation is performed during the cycle time of the loops in which the statements are embedded. If loops are executed too fast, a run time message is printed.

1.5.1.9 Manipulating the durations of user defined pulses

User defined pulses, as described in section 1.5.1.3, can be manipulated in the same way pulses defined by `p0-p31` are manipulated (see sections above).

```
d0-d31 [sec]
p0-p31 [µsec]
l0-l31 (loop counters)
in0-in31[sec]
inp0-inp31 [µsec]
aq [sec]
dw [µsec]
dwov [µsec]
de1, de2, depa, derx, deadc [µsec]
vd [sec]
vp [µsec]
nbl, ds, ns, nsdone, td, td1, td2
decim
cpdtim1-cpdtim8 [sec]
cnst0-cnst31
```

Table 1.3

1.5.2 Pulse frequency

1.5.2.1 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
vp:f2
```

all execute a pulse on channel f2, with the duration P[1], P[2]*0.33, p30d1H*3.33 and a values 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`, ..., `p31` 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.

1.5.2.2 Using frequency lists

You can change the frequency of a channel within a pulse program with the statements `fq1`-`fq8`. They take the current value from a frequency list. A frequency list is a text file whose lines contain frequency values (see the command **edlist** in the Acquisition Reference manual). For example, the statement:

```
d1 fq2:f3
```

which is equivalent to:

```
d1 fq=fq2:f3
```

uses the frequency list whose file name is defined by the acquisition parameter FQ2LIST (`fq1` would use FQ1LIST, etc.). You can set FQ1LIST etc. from the **eda** dialog box, and you can modify a selected list with **edlist**. The example above sets the frequency of channel f3 by taking the *current* value from the list defined by FQ2LIST. When `fq2` is executed the first time, the *current* value is the first value in the list . The next time `fq2` is encountered (e.g. because it occurs several times in the pulse program, or because it is contained in a loop) the current value will be the next value in the list, etc. At the end of the list, the pointer will be set to the first entry of the list. The statements `fq1`-`fq8` not only set a frequency, but also increment the list pointer to the next entry of the list. They must be written behind a delay. The frequency change occurs at the beginning of this delay, which must be at least 2 μsec.

The list can, optionally, contain a frequency offset in MHz. If it does, the frequency list values in Hz are added to this offset. If it doesn't, the list values are added to the channel frequency (SFO1 for f1, SFO2 for f2, etc.).

The frequency can also be set to the values of the parameters CNST0-31 or to any number, for example:

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

```
d1 fq=3000:f1              ; SFOn [MHz] + 3000 Hz
```

set the frequency on channel f1 to the value of CNST20 and to 3000 Hz, respectively.

1.5.2.3 Frequency lists defined in the pulse program

For Avance spectrometers, 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, e.g.:

```
define list<frequency> absfq = { O 300 4000 5000 6000 }
```

The allowed modifiers are shown in table 1.4.

O <basic frequency [MHz]>	offset is in Hz and relative to basic freq. O
p (lower case)	offset is in PPM and relative to SFOx
P (upper case)	offset is in PPM and relative to BFx
no modifier	offset is in Hz and relative to SFOx

Table 1.4

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 must 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.

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 a regular frequency lists (interpreted by the fq*n* statements) and a frequency list defined within the pulse program is that the latter is *not autoincremented*. The list index can, however, be manipulated with postfix operators. The operators .inc, .dec, .res increment, decrement and reset the index, respectively. 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.

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
  d1 fqlist:f1             ; set the frequency SFO1+200
  p1:f1                    ; use frequency SFO1+200
  d1 fqlist.dec            ; decrement pointer
  go=1
exit
```

1.5.3 Pulse phase

1.5.3.1 Phase programs: 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
vp:f4 ph6
```

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
(vp ph6):f4
```

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

1.5.3.2 Phase programs: syntax

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

```
(1) ph1 = 0 0 1 1 2 2 3 3
(2) ph1 = (5) 0 3 2 4 1
(3) ph1 = {0}*4 {2}*4
(4) ph1 = {0 2}^1
(5) ph1 = {0 2}^1^2^3
(6) ph1 = {1 3}^1^2*2
(7) ph1 = {{0 2}*2}^1^2
```

(8) ph1 = {{{0}*2}^2^3^1}^2
(9) ph1 = (5) {1 2}*2^1
(10) ph1 = ph2*2 + ph3

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 degrees, 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 65536 (corresponding to 16 bits). 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 n = 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:

(3') ph1 = 0 0 0 0 2 2 2 2
(4') ph1 = 0 2 1 3
(5') ph1 = 0 2 1 3 2 0 3 1
(6') ph1 = 1 3 2 0 3 1 1 3
(7') ph1 = 0 2 0 2 1 3 1 3 2 0 2 0
(8') ph1 = 0 0 2 2 3 3 1 1 2 2 0 0 1 1 3 3
(9') ph1 = (5) 1 2 1 2 2 3

In (10), the phase program is the sum of two other phase program, 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 ph5 = 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.

1.5.3.3 Phase program position

Phase programs must be specified at the end of the pulse program (see the pulse program example in Table 1.1 at the beginning of this chapter). Any pulse program can contain up to 32 different phase programs (ph0-ph31).

1.5.3.4 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*).

1.5.3.5 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.

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.

Note that there is no statement dpp0 - dpp31.

1.5.3.6 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
```

1.5.3.7 Phase program arithmetic

Each of the phase programs `ph0-ph31` has 3 associated statements:

```
ip0-ip31, dp0-dp31, rp0-rp31.
```

They can also be used with an integer multiplier n:

```
ip0*n - ip31*n, dp0*n - dp31*n.
```

Consider the phase programs `ph3 = 0 2 2 0` and `ph4 = (5) 0 1 2 3`. The pulse program statement:

```
20u 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:

```
20u 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:

```
6u ip3*2
7.5u dp4*2
```

increment `ph3` by 2*90=180° and decrement `ph4` by 2*360/5=144°.

An increment/decrement phase program statement must always be specified behind a delay, which must be long enough for the increment/decrement to be calculated. The required time depends on the number of phases in the phase program and amounts to 1.5 μsec per phase and channel.

1.5.3.8 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 <u>original</u> phase program `ph1` whereas `ip1(*n)` works on the <u>current</u> 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
```

1.5.3.9 Phase presetting

A pulse program statement like:

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

suggests that switching to the current phase of the specified phase program is executed at the time the pulse begins. On Avance-AQX, however, the phase may take a certain amount of time to become stable. For this reason, XWIN-NMR allows you to instruct the pulse program to switch the phase somewhat earlier. Eight parameters PHASPR[1]-PHASPR[8] are available, one for each spectrometer channel. They can be set from the **edscon** parameter editing command or at the beginning of the pulse program by statements like "phaspr1=4u". Their default value is 3 μsec. As such, the phase for any pulse will be set 3 μsec before the pulse begins. If you would change PHASPR[3] to 4 μsec, the phase of all pulses executed on chan-

nel 3 would be set 4 μsec before the pulses begin.

On Avance-AQS, the phase switching delay is negligible and, as such, the parameter PHASPR does not exist.

1.5.3.10 Phase setting without executing a pulse

XWIN-NMR allows you set the phase for a particular spectrometer channel without executing a pulse. In that case, you must specify a phase program behind a delay. Examples:

```
(d1 ph1):f3
(0.1u ph3):f2
```

1.5.3.11 The 4-phase modulator

On Avance-AQX, the phases for the spectrometer channels f1-f8, as presented so far, are realized in the FCU (Frequency Control Unit) of the spectrometer, during the digital frequency generation. For special applications such as certain solid state experiments, an accessory is available called 4-phase modulator. This device continuously provides four phases of the current frequency: 0, 90, 180, and 270°. It therefore allows faster switching between these phases. The following example shows its phase program syntax:

```
ph1 = +x +y -x -y
```

Whenever this syntax is used, the phases will be selected from the 4-phase modulator rather than from the FCU's digital frequency generator.

The above phase program corresponds to the regular phase program:

```
ph1 = 0 90 180 270
```

The 4-phase modulator has a number of internal adjustment parameters which can be set with the command **ed4ph**.

The 4-phase modulator can be used with or without an HPCU. In the former case, it must be connected to the 4-PH connector of the HPCU. In the latter case, it must be connected to the second *rs485* channel of the CCU and must be specified in the hardware list (see also **cf**). Note that spectrometers which are equipped with transistorized amplifiers rather than tube amplifiers are not equipped with (because they do not need) an HPCU.

On Avance-AQS spectrometers, the 4-phase modulator is not needed and, as such, not supported. The reason is that the frequency generation on the SGU doesn't cause any phase switching delays.

1.5.4 Pulse power and shape

1.5.4.1 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 PL[1], PL[2], .., PL[8]. Here, PL is an acquisition parameter that consists of 32 elements PL[0] - PL[31]. It can be set from **eda** or by entering **pl0**, **pl1**, 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 PL[5]. Any pulse executed on this channel will then get the frequency SFO2 and the power PL[5]. The pl0-pl31 statements must be written behind a delay. The power setting occurs within this delay, which must be at least 2 μsec.

1.5.4.2 Power lists

In addition to the PL[0]-PL[31] parameters, 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 = { -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 high power values, in dB, enclosed in braces. Entries must be separated by white spaces.

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 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**. 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 VALIST acquisition parameter.

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 = { 10 30 50 70 }
ze

1 d1 pwl:f1 pwl.inc         ; set power on f1 to 10dB, incr. pointer
  d1 pwl:f2 pwl.dec         ; set power on f2 to 30dB, decr. pointer
  d1 pwl[2]:f3              ; set power on f2 to 50dB
  "pwl.idx = pwl.idx + 3"  ; set the pointer to 0 to 3
  d1 pwl^:f4               ; set power on f4 to 70dB, incr. pointer
  (p1):f1 (p2):f2 (p3):f3 (p4):f4
  go=1
exit
```

1.5.4.3 Shaped pulses

A *shaped* pulse changes its power (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 power 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 power values in the shape file. If this is less than 200 nsec, an error message is displayed which tells you by which amount the pulse duration must be increased.

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 millisec, 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 *shaped pulse parameter table*. This table is displayed when you click the SP07 button within **eda**. The table has 32 entries with the indices 0-31. You may use the statements :sp0 - :sp31 to refer to the entries 0-31, 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.

Note that the statement:

```
(vp:sp4 ph10):f4
```

is <u>incorrect</u> because shaped pulses with vp are not supported.

Each entry of the shaped pulse parameter table has 3 parameters assigned to it: a *power value*, an *offset* and a *file name*.

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 a a power value (in percent) and a phase value (in degree). The power value defines the percentage of the absolute power value (see below).

Offset frequency
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. As such, phase coherency of the frequency is maintained.

Power value [dB]
This is the absolute power value of the pulse shape. 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.

Rather than using power value specified by SP07, 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 PL[9]
p1:sp0(currentpower):f1    ; shaped pulse with abs. power PL[9]
d1                         ; power is reset to PL[1], default for f1
p2:f1                      ; rectangular pulse with power PL[1]
```

If, in this example, the value of PL[9] is very different from the value of SP[1], the pulse shape may be compressed. The reason for this is that the CORTAB correction table for SP[1] is applied rather than for PL[9].

You can access the SP07 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 **sp15** sets the absolute power value of entry 15.

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*3*50 ns = 150 µsec.
- the maximum execution length is a few seconds
- the increment must be a multiple of 50 µsec. Any other increment values might result in spikes after the shape.

When a shape is specified too short or too long, an error message will be printed and the shape will be used with the previous settings!

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

```
ipu1
```

or with a relation like:

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

In both cases, the variation of a shape pulse length takes 4 µsec per channel.

Note that varying the length of a shape with non zero offset frequency will change the offset frequency, as the frequency shift is obtained via phase shifting. This phase shift won't be recalculated during execution, so the offset will be changed inverse proportional to the duration. (Doubling the duration means cutting the offset in half). An warning will be printed, when you change the duration of a shape with offset.

1.5.4.4 Shaped pulse presetting

On Avance-AQX spectrometers, the parameters SHAPPR[1]-SHAPPR[8] are used for shaped pulse presetting. Please refer to the description of the command ***edscon*** in the Acquisition Reference manual for details on SHAPPR. On Avance-AQS spectrometers, phase presetting is not required.

1.5.4.5 Fast Shapes

Regular pulse shapes as described above need a short delay before and after the pulse of ~4 μsec. On Avance-AQS, you can also generate the so-called fast shapes. They do not require this delay so fast shaped pulses can be executed consecutively in a loop or they can be executed right before or right after a rectangular pulse. Fast shape pulses and can be executed with the options `:spf0` - `:spf31`. They differ from normal shapes in the following respects:

- They do not change the power setting but use the current setting.
- The minimum time for each interval is 350 nsec whereas for normal shapes it is between 50 and 100 nsec. If this limit is violated, the pulse programs will stop with the error message: "AQNEXT while FIFO busy".
- The timing of the shape cannot be changed during pulse program execution.
- The total time of the entire shape must be exactly the time for one interval times the number of intervals in the shape pulse, where the timing resolution of the time for the intervals is 50 ns (whereas the time resolution for normal shapes is 12.5 ns).

Fast shapes are typically used for solid states experiments.

1.5.4.6 Amplitude Lists

On Avance-AQS, you can define initialized 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 PL[1] (assuming it was at its

default value PL[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 PL[1]. Furthermore, a statement like:

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

which executes a shaped pulse, resets the power on f1 to PL[1] after is 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 PL[1]
d11 am1:f1          ; set the power on f1 to 70% of PL[1]
p1 ph2              ; rectangular pulse on f1 with 70% of PL[1]
d11
p11:sp1:f1 ph1      ; shape pulse with power SP[1]
d11
p1 ph1              ;rectangular pulse on f1 with power PL[1]
```

1.5.5 Generating pulses with a certain flip angle

The pulse generation statements described so far do not allow you to specify a flip angle. The flip angle is implicitly defined by pulse duration and power. In fact, *flip angle, power,* and *duration* are interdependent: when two of them are known, the third one can be calculated. XWIN-NMR provides the possibility of storing calibrated 90° pulses (using the pulse widths and the corresponding power levels) for different probe heads and solvents (see command ***edprosol***). Based on these values, the statement:

```
pulse(flipangle, power, duration)
```

generates a pulse of the specified flip angle. The value of two of the three arguments must be specified, the third argument can be set to *auto*.

Example 1:

```
pulse(90 deg, 0 dB, auto):f1 ph1
```

Generate a 90° pulse using a power level of 0 dB. The program calculates the required pulse width and executes the pulse on channel f1 using the phase program ph1. Please note that a space character must be specified before the

deg and dB units.

Example 2:

```
pulse(30 deg, pl2, auto):f2 ph1
```

Generate a 30° pulse using the power given by the acquisition parameter PL[2]. The program calculates the required pulse width and executes the pulse on channel f2 using the phase program `ph1`. You can use `pl0`-`pl31` as the second argument, referring to the parameters PL[0]-PL[31].

Example 3:

```
pulse(45 deg, auto, 5u):f2 ph1
```

Generate a 45° pulse using a pulse width of 5 μsec. The program calculates the required power and executes the pulse on channel f2 using the phase program `ph1`. You can append 'u', 'm', or 's' to the number in the third argument to indicate microseconds, milliseconds, or seconds.

Example 4:

```
pulse(45 deg, auto, p1):f2 ph1
```

Generate a 45° pulse using a pulse width defined by the acquisition parameter P[1]. The program calculates the required power and executes the pulse on channel f2 using the phase program `ph1`. Please note that you can specify P[0]-P[31] as the third argument, but not pulses defined by the `define pulse` statement.

Example 5:

```
pulse(90 deg, auto, 10m):sp1:f1 ph1
```

Generate a 90° shaped pulse using a pulse width of 10 msec. The program calculates the required power and executes the pulse on channel f1 using the phase program `ph1`. *Please note:* Power calculation uses the same formula for rectangular and shaped pulses. To account for the deviation from a rectangular pulse, a power correction value must be specified for shaped pulses. The program expects this number (in db) in the corresponding Power entry of the shaped pulse parameter table. In this example (`:sp1`), the correction value is taken from entry 1 (you can set this value from **eda** or by entering **sp1** on the command line).

Example 6:

```
pulse(auto, pl2, p2):f2 ph4
```

Generate a pulse using a width defined by the acquisition parameter P[2], and a power defined by PL[2]. The flip angle can be viewed with the command **ased**. The above statement is equivalent to:

```
p2:f2 ph4
```

1.6 Delay generation

Table 1.5 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, ... , d31	Generate a delay whose duration is taken from the acquisition parameter D[0], ..., D[31], respectively.
d0:r, ... d31:r	Generate a delay whose duration is taken from the acquisition parameter D[0], ..., D[31] and which is randomly varied. The maximum variation (in percent) is defined by the acquisition parameter V9.
3.5u, 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.
vd	Generate a delay whose duration is taken from in a delay list.
de1, de2, de, depa, derx, deadc	Generate a delay of length DE1, DE2, DE, DEPA, DERX, DEADC, respectively.
dw, dwov	Generate a delay of length DW, DWOV.
aq	Generate a delay of length AQ.

Table 1.5 Delay generation statements

1.6.1 <u>d0</u>-<u>d31</u>

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].

1.6.2 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[1]. As such, the effective delay varies between 0 and 2*D1. It can be set from **eda**, or by typing **v9** on the command line.

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

1.6.3 Fixed length delays

The statement:

```
10m
```

executes a delay of 10 msec (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.

1.6.4 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 obligatory.

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.

1.6.5 Variable list delays

The statement:

```
vd
```

executes a delay whose duration is given by the current value of a variable delay list. A delay list is a text file that contains one delay per line. Delay lists are set up with the command **edlist vd** (described in the Acquisition Reference manual). The statement vd uses the list file defined by the acquisition parameter VDLIST. When the pulse program is started, the first duration in the list is used. The pulse program statement ivd can be used to move the list pointer to the next duration. If the end of the list is encountered, the pointer is reset to the beginning. The statement ivd must be specified behind a delay, for example:

```
d1 ivd
0.1u ivd
```

The length of the delay is irrelevant, any value is allowed.

It is also possible set the list position with an equation. Example:

```
"vdidx=5"
vd
```

Here, vd will execute a delay whose duration is selected from position 5 of the delay list. To the right of the equal sign any dimensionless expression is allowed. This may contain parameters from Table 1.3.

1.6.6 User defined delay lists

As an alternative to using the vd statement, a list of delays can also 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 an 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 0.1seconds 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. Furthermore, a particular list entry can be specified as an argument, in squared 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
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.

1.6.7 Special purpose delays

These are the delay statements de1, de2, de3, dw, and aq as listed in Table 1.5. They are used in pulse programs in which the acquisition is started with the adc statement rather than with go=label.

1.6.8 Manipulating delays: The operator *

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

```
d1*1.5
compensationTime*3.33
d3*oneThird
vd*3
```

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.

1.6.9 Manipulating delays: Changing <u>d0</u>-<u>d31</u> by a constant value

The delays executed by d0-d31 can be incremented or decremented according to the acquisition parameters IN[0]-IN[31]. These parameters contain a duration (in seconds). The pulse program statements id0-id31 add IN[0]-IN[31] to the current value of d0-d31, respectively. Likewise, dd0-dd31 subtract IN[0]-IN[31] from the current value of d0-d31. The statements rd0-rd31 reset d0-d31 to their original value, i.e. to the values of the parameters D[0]-D[31]. 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 D10 are used as incrementable delays for 2D and 3D experiments, IN0 and IN10 are the respective increments which are used to calculate the sweep widths SW(F1) and SW(F2), respectively (see the description of IN0, IN10 in the Acquisition Reference manual).

1.6.10 Manipulating delays: Redefining <u>d0</u>-<u>d31</u>

The duration of the d0-d31 statements is normally given by the parameters D[0]-D[31]. 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 Table 1.3. 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.

1.6.11 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-d31, described in the previous section.

Note: 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, not at run time.

1.7 Simultaneous pulses and delays

1.7.1 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.

3. Pulses and delays which are specified on the same line, and which are enclosed in different sets of parentheses, are executed simultaneously. The first item within a set of parentheses is started at the same time as the first item in any other set of parentheses. You can specify an arbitrary number of sets of parentheses on a line.

1.7.2 Examples

1.7.2.1 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 1.1).
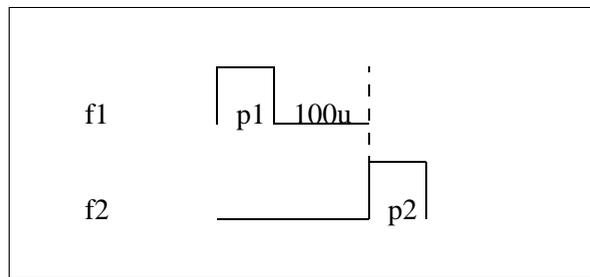
Figure 1.1 Rules 1 and 2: An example

1.7.2.2 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 (Figure 1.1).

1.7.2.3 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 (Figure 1.1)..
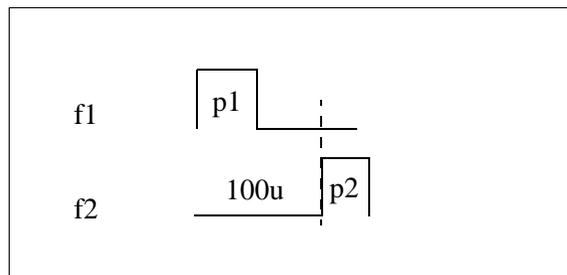
Figure 1.2 Rule 3: example 1

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 (Figure 1.3).
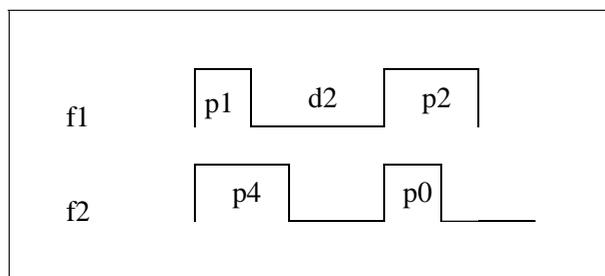
Figure 1.3 Rule 3: DEPT example

The following 2 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 parenthesis, 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.

Chapter 2

Decoupling

2.1 Decoupling

2.1.1 Decoupling statements

Table 1.5 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
`do`	switch decoupling off

Table 2.1 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.

2.1.2 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! 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 1.5.2). 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 msec delay.

2.1.3 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 1.5.3). Examples:

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

Note that phase cycling (see chapter 1.5.3.4) is applied to phase programs speci-fied behind decoupling statements in the same way that phase programs are speci-fied 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.

2.2 Composite pulse decoupling (CPD)

2.2.1 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 differ-ent 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, simulta-neously, CPD sequence 2 on channel f2. CPD sequence 1 is obtained from a text file defined by the acquisition parameter CPDPRG1. Likewise, CPD sequence 2 is obtained from a text file defined by the acquisition parameter CPDPRG2, 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). Table 2.2 shows the statements available to start a CPD sequence; and Table 2.3 shows the statements available to build a CPD sequence.

2.2.2 Syntax of CPD sequences

The syntax of CPD sequences is demonstrated by examples. Table 2.4 shows the

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 2.2 Available cpd statements

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-p31. The Garp sequence shows the usage of the lo to statement.

The Garp sequence, as well as the sequences in Table 2.5, make use of 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 g[2] (PCPD[3]). This allows you 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 2.5 shows two CPD sequences based on shaped pulses. Shapes are specified in the same way they are specified in pulse programs using the :sp0, ... , :sp31 pulse selector options. The examples demonstrate the order in which duration multiplier, shape selector and phase must be specified.

The sequences in Table 2.4 and Table 2.5 do not contain a power setting statement.

p0, ... , p31 10up, 5mp, 2.5sp pcpd1, ... , pcpd8	Generate pulses with durations P[0], ... , P[31]. Generate pulses in micro-, milli-, and seconds 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, ... , d31 10u, 5m, 2.5s	Generate delays with durations D0, ... , D31. Generate delays in micro-, milli-, and seconds.
*3.5 :135.5	Multiplier. Can be appended to p0-p31 or d0-d31. Phase in degrees. Can be appended to pulses.
:sp0, ... , :sp31	Shaped pulse selectors. Can be appended to pulses.

pl=
pl=5
pl=sp13
pl=pl25
	Power specifier (see example in Table 2.5):
	   in dB
	   according to shaped pulse parameters SP[0]-[15]
	   according to PL[0]-[15]

fq= fq=2357 fq=cnst25 fq=fq2	Frequency change in Hz (relative to SFO1 for channel 1, SFO2 for 2...) from the parameters CNST[0]-[31] from the frequency list specified in FQ2LIST
;	Begin of a comment (until end of line)
lo to *label* times *n* jump to *label*	Loop to *label n* times Branch to *label*. Usually the last statement.
#addphase #setphase	Special phase control statements

Table 2.3 statements available to build CPD sequences

Therefore, the current power setting of the main pulse program for the respective channel is valid.

The following section of a pulse program starts a CPD program on channel f2, but keeps the f2 transmitter output disabled (statement cpdngs2) except for the periods given by p2. The p2 pulse actually serves as a gating pulse for CPD decoupling. It should not be specified with a phase program in order to prevent overwriting the CPD program phases.

```
1 90up: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 2.4 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 2.5 MLEVSP and MPF7 CPD sequences

```
1u cpdngs2:f2
(p1 ph1 d1):f1
```

```
1 (p2 d2):f2
  lo to 1 times 10
  lo to 2 times 10
```

2.2.3 Phase setting in CPD programs: <u>#addphase</u>, <u>#setphase</u>

The phase specified within a CPD program can be added to the phase specified in the pulse program (#addphase) or it can overwrite the pulse program phase (#setphase). Note that #addphase is the default mode. It only needs to be specified if #setphase was used and you want to switch back to #addphase.

Example 1:

Pulse program statement to start the CPD sequence:

```
d1 cpds2:f2 ph2
```

CPD program statements:

```
#addphase
pcpd:180
```

Resulting phase of the pcpd pulse: 180 plus the current phase in ph2.

Example 2:

Pulse program statement to start the CPD sequence:

```
d1 cpd2:f2 ph2
```

CPD program statements:

```
#addphase
pcpd:sp15
```

Resulting phase of the pcpd shaped pulse: shaped pulse phase (according to the phases in the shape file) the current phase in plus ph2.

Example 3:

Pulse program statement to start the CPD sequence:

```
d1 cpd2:f2
```

CPD program statements:

```
#setphase
pcpd:sp15:180
```

Resulting phase of the `pcpd` shaped pulse: shaped pulse phase (according to the phases in the shape file) plus 180.

Please note that, on Avance-AQX, a phase program should not be used with a `cpdn` statement since the FCU does not support the real-time addition of more than two phases. On Avance-AQS, however, you can use `cpdn` statements with phase programs.

2.2.4 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.

2.2.4.1 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 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.

2.2.4.2 Frequency setting using the parameters CNST0-31

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.

2.2.4.3 Direct Specification of Frequencies

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

2.2.5 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 loopcounter *n* can be a number or a symbolic loopcounter l0 - l31, where the latter interpret the parameters L[0] - L[31]. It must be equal to of greater than 1.

Loop counters defined in 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.

2.2.5.1 Manipulation of loop counters from the main pulse program

BILEV decoupling

You can manipulate the loopcounter of a CPD program after each scan according to an arithmetic expression in the following way:

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

This means that the loopcounter `l5` will be modified after each scan according to the above equation. The modification will take effect immediately after the scan. The pulse program should be written such that the loopcounter is always greater than zero. The variable loop counter then can be used to modify the CPD program such that the beginning changes with each scan.

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.

If a CPD program contains a `bilev` statement, the `cpd` statement that calls the CPD program works like a `cpds` statement. This means that the CPD sequence is not continued at the point where it was stopped before, but starts from the begin-

ning each time it is called.

Chapter 3

Loops and conditions

3.1 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 first `lo` statement in this example does not cause an extra delay in the pulse program. However, any further `lo` statement will add a delay of 2.5 μsec. XWIN-NMR will display a corresponding message when the pulse program compiler is invoked, i.e. when entering one of the commands **gs**, **zg**, **go**, or **pulsdisp**.

The `lo` statement exists in a number of variations as shown in Table 3.1.

`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 l0`
 ....
 `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 c`	The loop counter is taken from the list defined by the acquisition parameter VCLIST. The list can be created with **edlist vc**. The statement ivc advances the list pointer by 1. The list pointer position can also be set with an equation, e.g. vcidx=5.
`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 3.1 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 μsecond delay. You could replace the loop counter l2 with c in this example, and replace iu2 with ivc 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 millisec, plus 1. The compiler truncates the quotient aq/10m to give an integer. The expression may include any of the parameters shown in Table 1.3.

3.2 Conditional pulse program execution

3.2.1 Conditions evaluated at precompile time

Consider the pulse program at the left part of Table 3.2. 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
```

```
#define PRESAT              #define PRESAT

1 ze                        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 3.2 Using #define, #ifdef, #include statements

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

and correspond to C language pre-processor syntax. 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 3.2, `#define PRESAT` enables the presaturation statement block. Commenting out this line in C-syntax style (`/*#define PRESAT*/`) or 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 display. Just enter the **pulsdisp** command and and click the button *Show program*. Note that in the pre-processed pulse program, all conditional statements beginning with a '#' have been removed.

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 3.2 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.

3.2.1.1 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 without any white spaces in between. 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** .

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

3.2.1.2 Macro Definitions

You can use the statement `#define` not only to define *aFlag*, 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.

3.2.2 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:

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

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

Table 3.3

allowed in pulse programs.

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

Table 3.4 a condition evaluated at compile time

Example: See Table 3.4

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.

3.2.3 Conditions evaluated at run time

The spectrometer TCU has four trigger inputs. Trigger events can be positive or negative edges or levels.

XWIN-NMR supports branching and evaluation of conditions within a pulse program while the pulse program execution is in progress. Table 3.5 lists the available statements. These statements do not cause a delay in the pulse program. 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.

goto *label*	Unconditional jump to *label*
if *expression* goto *label*	Branch to *label* if *expression* evaluates to *true*.

if (*trigger*) goto *label*	Branch to *label* if the *trigger* condition is true.
	Positive *level trigger* specifiers:
	  trigpl1, trigpl2, trigpl3, trigpl4
	Negative *level trigger* specifiers:
	  trignl1, trignl2, trignl3, trignl4

aDelay *trigger*	The same *trigger* specifiers as above are allowed. The next pulse program statement will not be executed until the *trigger* condition becomes *true*. Example:1u trigpl1 Positive *edge trigger* specifiers: trigpe1, trigpe2, trigpe3, trigpe4 Negative *edge trigger* specifiers: trigne1, trigne2, trigne3, trigne4

Table 3.5 Conditional pulse program execution

Example 1:

```
        ze
lab1, d1
        p1
        d0
        if "d0*2 + 7m > 500m" goto lab2
        "d0 = d0 + 10m"
        p2
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 (trigpl2) goto lab3
lab2, d1
```

```
        p1
        aq
        lo to lab2 times ds
        goto lab1
lab3, d1
        p1
        go=lab1
```

The TCU has 4 trigger input channels; signals arriving at the TCU can be checked using the `trig` specifiers. This example performs DS dummy scans to maintain steady state conditions as long as no positive 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.

3.3 Suspend/resume pulse program execution

XWIN-NMR 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 Table 3.6.

suspend	stop execution on the command **suspend**
autosuspend	stop execution
calcsuspend	stop precalculation and stop execution on **suspend**
calcautosuspend	stop precalculation and stop execution

Table 3.6 statements to suspend pulse program execution

After suspension, the program execution can be resumed with the XWIN-NMR command **resume**.

If you use suspend or autosuspend, you should not change any acquisition parameters between suspending and resuming the acquisition. 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 is executed without considering the parameter change.

The statement 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 restart precalculation after **resume**. For example:

```
calcsuspend
2s
```

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

Chapter 4

Data acquisition and storage

4.1 Start data acquisition

XWIN-NMR 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.

4.1.1 The statements <u>go=*label*</u>, <u>gonp=*label*</u>, <u>gosc</u>, <u>goscnp</u>

The left column of Table 4.3 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/DE2/DERX/DEPA/DEADC in the Acquisition Reference Manual). Note that all these delays start simultaneously, at the beginning of DE. The sequence in which the actions are performed, depends upon the length of the individual delays.

1. At the end of DEPA (preamplifier blanking delay), the preamplifier is switched

to observe mode.

2. At the end of DERX (delay for receiver blanking) the receiver gate is opened.

3. At the end of DE1, the intermediate frequency (if used) is added to the frequency of the observe channel. This corresponds to the execution of the statement syrec. The intermediate frequency is only used for AQ_mod = DQD or, if your spectrometers has an RX22 receiver, for any value of AQ_mod.

4. At the end of DE2, the phase of the receiver channel is set to 0. Note that DE2 is used on Avance-AQX but not on Avance-AQS

5. At the end of DEADC (delay for ADC blanking), the digitizer is enabled.

6. 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.

7. At the time the digitizer is started, a delay AQ is executed. This delay lasts until the digitization of the FID is finished.

8. A delay of 3 millisec 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 (if the intermediate frequency is used). This corresponds to the execution of the statement sytra (which is inverse to syrec). The intermediate frequency is only used for AQ_mod = DQD or, if your spectrometers has an RX22 receiver, for any value of AQ_mod.

   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.

Table 4.1 shows that the go statements can be specified in conjunction with other

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.

Table 4.1 Examples of the usage of the go or gonp statement

statements. PH_ref is an acquisition parameter to be defined by the user.

4.1.2 The statements <u>rcyc=*label*</u>, <u>rcycnp=*label*</u>

The statement rcyc executes step 8 of the actions performed by go=*label* and gonp=*label* (see the previous section). The rcycnp statement skips step 8c.

The rcyc statements can be used for acquisition loops based on adc rather than

go=*label* or gonp=*label*. You should *not* specify phase programs behind rcyc and rcycnp. Decoupling statements are allowed although it would not make sense to use them here. Table 4.3 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 3 millisec. Such a delay must be at least 100 μsec.

4.1.3 The statements <u>eosc,</u> <u>eoscnp</u>

The statement eosc executes steps 8a-8c of the actions performed by go=*label* and gonp=*label* (see the previous section). The eoscnp statement only executes steps 8a and 8b.

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 should *not* specify phase programs behind eosc and eoscnp. Decoupling statements are allowed but it would not make much sense to use them here. Table 4.3 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 100 μsec, e.g.:

        100u eosc

In that case, they are then executed during the specified delay rather than during the default 3 millisec.

4.1.4 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 <u>replace</u> 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. Such a delay must be at least 10 μsec and its minimum length depends on the number of phase programs. They are then executed during the delay. If they are not specified with a delay their execution will require 3 millisec.

The statement `zd` is normally executed as a part of the `mc` macro statement. As such it does not appear in most Bruker pulse programs. One example where it is specified explicitly is the pulse program *selno*.

4.1.5 The statement <u>adc</u>

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 Table 4.2.

  *Note that **adc** is implicitly defined with **DE1***

  Here, the statement `de` executes the delay defined by the acquisition parameter DE. The statements `de1`, `de2`, `derx`, `deadc` and `depa` execute a delay

```
define delay rde1
define delay rde2
define delay rdepa
define delay rderx
define delay rdeadc

"rde1=de-de1;"
"rde2=de-de2;"
"rdepa=de-depa;"
"rderx=de-derx;"
"rdeadc=de-deadc;"

#define DE1 (de1 rde1 adc ph31 syrec)
#define DE2 (de2 rde2 ph30:r):f1
#define DE3 (de)
#define DEPA (depa rdepa RGP_PA_ON)
#define DERX (derx rderx RGP_RX_ON)
#define DEADC (deadc rdeadc RGP_ADC_ON)
```

Table 4.2 The contents of the file De.incl

that is defined by the corresponding **edscon** parameters.

- 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. Table 4.3 shows the

```
  ze                  #include De.incl          #include De.incl
2 d1                   ze                          ze
  (p1 ph1):f1         2 d1                       2 d1
;---------             (p1 ph1):f1                 (p1 ph1):f1
  go=2 ph31           ;------------------        ;------------------
;--------              DE1 DE2 DEPA DERX DEADC DE3  DE1 DE2 DEPA DERX DEADC DE3
  wr #0                 aq DWELL_GEN                aq DWELL_GEN
  exit                  rcyc=2                      eosc
                      ;------------------           lo to 2 times ns
                         wr #0                    ;------------------
                         exit                       wr #0
                                                    exit
```

Table 4.3 The same pulse program based on go, adc/rcyc, and adc/eosc

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`. This, however, is only necessary for AQ_mod = DQD or, if your spectrometers has an RX22 receiver, for any value of AQ_mod.

- The dwell time is generated during `aq`. For Avance-AQX, the `adc` statement starts dwell generation on the RCU (here the macro DWELL_GEN has no effect). 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 look at the Bruker pulse program *zgadc* (enter **edpul zgadc**). This program will produce exactly the same result as the program *zg*.

The statement `adc` will send a command to start the digitizer. The digitization doesn't start immediately with this statement but only after a delay DE-DE1. In this way the sampling starts exactly with the beginning of `aq`.

Table 4.4 shows how homodecoupling for Avance-AQX during data acquisition can be realized using `adc`. Homodecoupling requires the receiver to be turned off at regular intervals. This can be achieved with the option `:e`, appended to a pulse or delay statement. It disables the receiver for the duration of the respective pulse or delay. Note that if `:e` does not occur between the statements `adc` (or `DE1`) and `rcyc`/`eosc`, but at a different position, its effect is reversed, i.e. the receiver is enabled rather than disabled.

The receiver phase

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

```
adc ph31
```

This statement sends an interrupt signal to the RCU, telling it 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 receiver phase should be specified before the scan loop using the statement `recph ph31` (see Table 4.5). The statement `ip31` after the recycle loop increments all entries of the phase program `ph31`

```
#include De.incl
define delay dw1
define delay dw2
define pulse pw3
define delay dw4
define delay dw5
define loopcounter tdov
"dw1=0.1u"
"dw2=2u"
"dw4=2.5u"
"pw3=2*dwov/5"; 20% dwell
"dw5=2*dwov-dw1-dw2-pw3-dw4"
"tdov = td *decim / 2"

1 ze
2 (d1 p1 ph31)
   DE1 DE2 DEPA DERX DEADC DE3
10 dw1
   dw2:e
   pw3:f2:e
   dw4:e
   dw5
   lo to 10 times tdov
   50u
   rcyc = 2
   wr #0
   exit

ph30=0
ph31=0 2 2 0 1 3 3 1
```

Table 4.4 Homodecoupling during data acquisition

but does not set the phase. As such, the receiver phase is not changed after each scan but after NS scans.

4.1.6 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 pulses, which activate

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

Table 4.5 Receiver phase setting without and with `recph`

the digitizer in regular time intervals are generated internally (on the RCU) so that the detection of a complete FID is automatically accomplished once initialized via `go` or `adc`. This occurs during the delay `aq` as displayed in the middle and the right columns of Table 4.3.

Certain experiments, however, require the control of the detection of each individual data point of an FID. On Avance-AQX, you can do that with the external dwell that is generated with the **:x** option. The pulse program in Table 4.6 displays an example of how this can be achieved. It works like the middle pulse program in Table 4.3, except that the dwell pulses are generated externally (i.e. by the TCU) by using the **:x** pulse option. A corresponding cable connection between TCU and RCU is required. In this pulse program the waiting time `aq` has been replaced by a loop that generates as many dwell pulses as required to measure TD data points.

Please refer to the Bruker pulse program libraries for high resolution, solids, and imaging experiments for examples using the **:x** option.

On Avance-AQS, the external dwell time is generated on the SGU, using the statement:

```
aq cpdngs29:f1
```

4.2 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

```
define delay dx
define pulse px
"dx=dwov/2"
"px=dwov/2"

  ze
2 d1
  (p1 ph1):f1
;------------------
  DE1 DE2 DE3 DEPA DERX DEADC
3 dx
  px:x              ;external dwell pulse
  lo to 3 times tdov
  rcyc=2
;------------------
  wr #0
  exit
```

Table 4.6 Acquisition on Avance-AQX, based on external dwell pulses

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 scan 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 256, the buffer size will be rounded to the next multiple of 256 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
```

Table 4.6 shows an example, the Bruker pulse program *noedif*. The FID's acquired

```
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 4.7 Usage of st and st0: *noedif* pulse program

with different decoupling frequencies are stored in two memory buffers.

The size of NBL is limited by the constraint that NBL times TD must not exceed the available RCU memory. For example, an RCU equipped with 4 Mb DRAM allows for about 3.8 Mb FID data to be stored (the remainder is needed by the acquisition parameters). If necessary, you can upgrade your spectrometer with more RCU DRAM.

4.3 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. Table 4.8 shows the available pulse program statements

`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.
`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 DSLIST.
`wr ##`	Transfer is performed to the file *fid* or *ser* of the data set which is pointed to by the *data set list pointer*. Its initial position is #0 which always corresponds to the foreground dataset. The list pointer can be incremented by 1, decremented by 1, or reset to the first item of the list using the statements `ifp`, `dfp`, and `rfp`, respectively.
`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).
`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.

Table 4.8 Writing acquisition buffers to disk

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.

It is allowed to specify the statements if, zd, id0-id31, ip0-ip31, and decoupling statements behind the same delay that is used for wr. It is important to use either a zd or ze statement after each wr before the next scan. Otherwise the data will be added to data previously acquired in the same memory region.

The name of the output file is *fid* or *ser*. An *fid* file contains a single FID, whereas a *ser* file contains a series of FID's. 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 filled it 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 (F2-TD)*(F1-TD)*4 corresponds to the size (in bytes) of the ser file. In a 3D experiment, the TD values must be set such that (F3-TD)*(F2-TD)*(F1-TD)*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 an F1-TD values that does not match the size of the ser file, you must set the status F1-TD value before you process the data. You can do that with **1s td**. For a 3D experiments you can adjust the TD values of the indirect dimension with **2s td** and **1s td**.

Although XWIN-NMR does not offer a 4D dataset structure, you can run a 4D experiment by executing a pulse program with a 4D loop structure. If you do that on a 2D dataset, the TD values must be set such that (F2-TD)*(F1-TD)*4 corresponds to the size (in bytes) of the ser file. This means F1-TD must be product of the number of increments in the three indirect dimensions. If you do that on a 3D dataset, the TD values must be set such that (F3-TD)*(F2-TD)*(F1-TD)*4 corresponds to the size (in bytes) of the ser file. Similarly, you can run experiments of more than 4 dimensions.

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 statement:

```
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) is with the command **3s aqseq** .

The wr statements (and all other statements in Table 4.8) can be specified behind a delay (see the example in Table 4.7). The delay must be at least 10 μsec. 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 to acquire data only to a disk that is physically connected to the computer that controls the spectrometer.

Chapter 5

The mc macro statement

5.1 The mc macro statement in 2D

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

```
1 ze    ; initialisation
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
```

In XWIN-NMR 3.0 and newer, the last two lines can be 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 phases 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 Table 5.1.

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

Table 5.1

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 fewer 2D (and 3D) pulse programs are needed. For example, in XWIN-NMR 2.6 and older, the pulse programs *cosytp*, *cosyst* and *cosysh* were used for TPPI, States-TPPI and States, respectively. In XWIN-NMR 3.0 and later, a single pulse program, *cosyph*, can be used for all phase sensitive modes. We will look at the expanded forms of cosyph for different values of F1-FnMODE. 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 expanded pulse programs will have the following header which is the same for the different values of F1-FnMODE:

```
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.

F1-FnMODE = 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
```

F1-FnMODE = 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
```

F1-FnMODE = 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
```

F1-FnMODE = 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. 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 F1-FnMODE = *QSEQ* or *TPPI*, the value for the delay increment is divided by 2 during run time. The parameter ND0, which represents the number of occurrences d0 within the loop, must have same value for all values of FnMODE.

- For F1-FnMODE = *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 F1-FnMODE= *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 F1-TD 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 slice (row) of a 3D dataset by setting F1-TD (F1-TD and F2-TD) to 1.

5.2 The mc macro statement in 3D

The `mc` statement can also be used in 3D pulse programs. In this case, there are two indirect dimensions, F1 and F2. For the F1 dimension, `mc` uses the clauses F1PH, F1EA and F1QF, for the F2 dimension, it uses the clauses F2PH, F2EA and F2QF.

The F2PH clause creates a second loop within which a second delay is varied.

The pulse program *noesyi4pr3d*:

```
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

F2-FnMODE = States-TPPI
F1-FnMODE = 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)
```

5.3 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 loops.

As an example of the F1I clause, we will take the pulse program *noesygpphprxf;* with F1-FnMODE = 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 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.

5.4 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 5.3). 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.

Table 5.2 shows, which expansions will be done for different values of FnMODE.

Note the following things when you view the expanded pulseprogram:

FnMODE	create a double loop	delay-increment div. by 2	phase reset	phase inc inserted	delay inc in inner loop
QF					√
QSEQ	√	√	√	√	√
TPPI		√		√	√
States	√		√	√	
States-TPPI	√			√	
EA	√			√	

Table 5.2 Results of use of different FnMODEs

- 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 # 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.

Chapter 6

Miscellaneous

6.1 Multiple receivers

If your spectrometer is equipped with multiple receivers, you can specify in the pulse program with which receiver you want to acquire the data. The 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`

For example the statement:

`go5=label`

acquires the data with receiver 5

If no number is specified, 1 is assumed, i.e. `go` is equivalent to `go1`):

Parameters for the first RCU are taken from the current dataset. Parameters for the nth RCU are taken from data set n-1 of the data set list DSLIST.

The following parameters are taken from the dataset in the DSLIST dataset:

AQ_mod, DECIM, DIGMOD, DIGTYP, DR, DSPFIRM, DSPFVS, FTLPGN, NBL, OVERFLW, SEOUT, SFO1, SW, SW_h, TD.

All other parameters are taken from the current dataset.

6.2 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**.

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.

6.2.1 Type 1 outputs ("RCP's")

On Avance-AQX, there are 35 outputs called RCP0, ... , RCP34. They can be set with an accuracy of 12.5 nanosec and a minimum of 50 nsec. They can be enabled or disabled from the pulse program in two different ways . This is illustrated by the examples below.

1. The statements:

```
p1:c0
5u:c25
vp:c15
```

generate pulses of duration P[1], 5 μsec, and the current value from the pulse list on the output channels 0, 25, and 15, respectively.

2. The statement:

```
1u setnmr0 | 15
```

activates output channel 15 (using active=low logic). The channel remains active until it is explicitly deactivated, e.g. with the statement:

```
1u setnmr0 ^ 15
```

The characters "|" (vertical bar) and "^" (circumflex) can be used to set and clear a bit in a register consisting of 35 bits. As such, several outputs can be enabled

or disabled simultaneously. For example, the statement:

```
1u setnmr0 | 14 | 13 ^ 15
```

enables the output channels 14 and 13, and disable channel 15.

The statement `setnmr0` must be specified behind a delay (in the above examples it is 1 μsec). The minimum delay is 200 nanosec.

The following pulse program section sets and clears a 4 microsecond pulse on RCP7:

```
4u setnmr0 | 7
1u setnmr0 ^ 7
```

6.2.2 Type 2 outputs ("NMR control words")

These are 128 outputs which can be set with an accuracy of 25 nanosec and a minimum of 50 nsec. They are organized in 8 registers of 16 bit size (called NMR control words, outputs or registers).

The statements `setnmr1`, ... , `setnmr8` can be used to enable or disable the channels 0-15 of each register. The syntax is identical to `setnmr0` as described in the previous section. For example, the statement:

```
1u setnmr3 | 0 | 13 ^ 15
```

enables the output channels 0 and 13 and disables output channel 15 of register 3.

6.3 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 spacial 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.

6.3.1 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 GPZ1etc. 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 μsec delay. This gradient remains on during a period of 1.3 msec.

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.

6.3.2 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
```

```
vp:gp4 ; Incorrect! Shaped gradients with vp are not supported.
```

They are applied for 10 millisec, 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 4 assigned 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 from the **eda** window by clicking the *GP031 edit* button. Alternatively, they can 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 a 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.

6.3.3 Gradient Functions

You can use gradient shapes as gradient functions. Then the current function value is used to calculate the gradient.

Shaped gradients can be defined in the pulse program as a gradient function. At

each moment, the gradient strength is set to the current function value.

The function index can be manipulated with the following statements:

```
zgrad sin; zero index -> use 1st function value
igrad sin; increment index
dgrad sin; decrement index
sgrad sin; save index (stack with depth = 1)
rgrad sin; restore index

sgrad sin; save the current index
rgrad sin; restore the last saved index
```

The length of an internal gradient function (or shape) must be specified at the beginning of the pulse program, e.g.:

```
lgrad sin = 100; sine function with 100 values
```

Internal Gradient Functions:

A gradient function is either a gradient shape that is defined in a gradient file, or an internal function that is calculated during pulse program compilation. The following internal functions are available:

- *plusminus*
  can take the value 1 or -1.

- *r1d*, *r2d* and *r3d*
  linear ramps from -1 to 1, where the final value is never reached.

- *step*
  linear ramp from 0 to 1 and the final value will always be reached.

- *sin*
  sine function from 0 to π (excluding π). The angle increment depends on the length of the function (see above).

- *cos*
  cosine function from 0 to π (excluding π).

- *sinp*
  sine function from 0 to π (including π).

- *gauss <truncval>*
  which is a gaussian function with truncation level (e.g. gauss2.5 for 2.5% truncation level)

- *rnd*
  random function.

6.3.4 Manipulation of rectangular or shaped gradients

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. For example:

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

6.3.5 General Gradient Statements

Since the XWIN-NMR gradient software is also used by ParaVision, it has features, that actually designed for medical imaging. With gradient statements of the form:

```
delay grad{<1st dim> | <2nd dim> | <r3d dim>}
```

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 pulseprogram. 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*.

- The functions are either *Internal functions*, which are handled accordingly by the compiler, or *gradient files* containing the function values (see above).

- Scaling and rotation can be suppressed with the following options:

  no_scale: Gradient is not scaled

  direct_scale or *shim_scale*: Gradient is not scaled and not rotated

- Hardware dependencies can be accounted for by specifying different values for xyz.

Examples:

```
10u grad{(0)|r2d(100)|(0)}; Ramp in the 2nd (or phase) dimension.
1m grad{sin(50,200)*r3d(89|90|91)+cos(50,200) |
(20)|(2|1|3,direct_scale)}
```

The 1st (or read) dimension contains *sin(50,200),* that means: a *sine* function with 50 % amplitude. The 2nd parameter indicates a gradient shape, consisting of 200 values, every value applied 1/200 ms = 5 us.

Every *sine* value is multiplied with the current value of *r3d(89|90|91)*. The amplitude of *r3d* is different for xyz to account for hardware dependencies.

The 1st dimension also contains a 2nd gradient shape *cos(50,200).* You can combine several gradient shapes in one statement, but the same length should be used.

The 2nd (or phase) dimension contains *(20)*, indicating a scalar gradient with 20 per cent amplitude.

The 3rd (or slice) dimension contains *(2|1|3, direct_scale)***,** indicating a scalar gradient with 2 per cent amplitude in x direction, 1 per cent in y and 3 per cent in z, independent of rotation and scaling.

6.3.6 Rotation and Scaling

If the EXPNO directory of the current data set contains a text file cag_par, the

rotation and scaling is done, as specified in this file.

Else if $XWINNMRHOME/exp/stan/nmr/lists/gp contains a text file cag_par, the rotation and scaling is done, as specified in this file:

Example:

0.5	; Scaling of 1st (or read) dimension
0.5	; Scaling of 2nd (or phase) dimension
0.8	; Scaling of 3rd (or slice) dimension

In this case you can acquire 2 slices with different orientation. Like function indices, you can manipulate slice indices with the statements zslice, islice, dslice, sslice, rslice.

1.0	0.0	0.0	;	Scaling of 1st (read or x) dimension
0.0	1.0	0.0	;	Scaling of 2nd (or y) dimension
0.0	0.0	1.0	;	Scaling of 3rd (or z) dimension
1.0	0.0	0.0	;	1st rotation matrix
0.0	1.0	0.0		
0.0	0.0	1.0		
0.707	0.707	0.0	;	2nd rotation matrix
-.707	0.707	0.0	;	the 1st and 2nd dimensions are rotated by
0.0	0.0	1.0	;	45 degrees

Table 6.1 example of a cag_par file

6.4 Miscellaneous statements

6.4.1 Switching on/off Presetting of Blanking Pulses: <u>preset</u>

The preset off statement switches off the presetting of blanking pulses. The

program will then behave as if all *preset* parameters (command **edscon**) are set to 0. Switching of the presetting must occur at the beginning of the pulse program and can not be undone.

6.4.2 Assignment of Preamplifier Blanking Pulses: <u>blkpa</u>

The preamplifier blanking pulses are normally set by the **edscon** preset parameters (BLKPA[1...8]). They can, however, also be declared at the beginning of the pulse program using the syntax:

```
blkpa<channel number> = <duration>
```

Example:

```
"blkpa1=3u"
```

6.4.3 Assignment of Transmitter Blanking Pulses: <u>blktr</u>

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 the pulse program using the syntax:

```
blktr<channel number> = <duration>
```

Example:

```
"blktr1=3u"
```

6.4.4 Generation of Blanking Pulses: <u>gatepulse</u>

Blanking pulses are automatically generated according to the **edscon** preset parameters. If, however, the pulse program contains the statement preset off the generation of blanking pulses is disabled. In that case, you can selectively enable the generation of blanking pulses on a particular channel. This can be done with the gatepulse statement. The syntax is:

```
delay gatepulse 1 [ | 2...]
```

Example:

```
3u gatepulse 1 ;generate blanking pulse for f1
p1:f1
d1
2u gatepulse 1|2 ;generate blanking pulses for f1 and f2
(p1):f1 (p2):f2
```

Note that `gatepulse` statement will only enable generate the transmitter blanking pulses, the preamplifier blanking pulses and the ASU blanking pulses.

6.4.5 Printing messages

The statement

```
print "Hello World"
```

prints the message *Hello World* during runtime of an experiment. The timing of the printout is not necessarily correlated to the execution of the pulse program because the TCU interprets the pulse program in advance of its execution. However, for debugging complex pulse programs it could be helpful.

Index