SpinsolveExpert – Pulse Programming Guide
Version 2.00 (DSP) January 2024

Table of Contents

1 Introduction 2
2 File organisation 5
3 The Proton experiment examined 6
3.1 The Pulse Program Macro 8
3.2 The interface description 9
3.3 The relationships list 11
3.4 The group list 12
3.5 The variable list 13
3.6 The pulse program 13
3.7 The phase cycle matrix 14
3.8 The returned variables 14
3.9 Getting factory parameters 14
4 Writing a new pulse program 16
5 The Experiment Control Macro 17
5.1 The execpp (execute pulse program) procedure arguments 18
5.2 The execpp procedure in detail 20
5.3 Modifying the execpp procedure to vary the pulse duration 23
Appendix A Pulse program commands 29
A.1 Acquire NMR data 29
A.2 Clear the DSP data memory 30
A.3 Delay for a specified time 31
A.4 Frequency modifications 31
A.5 Looping 32
A.6 RF pulse production 33
A.7 Shim and gradient commands 36
A.8 Table commands 37
Appendix B Available control groups 37
B.1 Acquisition 37
B.2 AcquisitionBW 38
B.3 AcquisitionBW2D 38
B.4 AcquisitionFIR 39
B.5 Display_Std 39
B.6 Display_2D 39
B.7 Display_Hz 40
B.8 File_Settings 40
B.9 Jres_processing 40
B.10 Processing_std 40
B.11 Processing_autophase 41
B.12 Processing_filter_autophase 41
B.13 ProcessingBW 41

Appendix C Procedures to change pulse program values 41
Appendix D Acquisition Explained 42
Appendix E Using Tables 43
Appendix F Programming multi-X spectrometers 45
Appendix G An introduction to Prospa programming 46
G.1 Data types 47
G.2 Variables....47
G.3 Control statements 48
G.4 Procedures 48
G.5 Commands 49
G.6 Debugging 49
G.7 Expressions....50
G.8 Matrices and vectors 50
G.9 User interface objects 51
G.10 Using the Prospa text editor 52
G.11 Additional help....52
Appendix H System specifications 52

1 Introduction

This manual will help you to understand and write pulse programs for the SpinsolveExpert application. This version targets version 2.0 which can drive either legacy DSP based Spinsolves or those with the newer FX3 processors. The software interface is the same for both systems however there are some subtle differences which may be important. This documentation is for DSP based systems.

The NMR pulse program controls a series of electronic modules in the spectrometer in such a way as to generate the desired NMR signal. At a minimum, this means producing radio frequency (RF) pulses of a specific amplitude, phase, frequency and duration at precise times, and then after a certain delay the NMR signal is captured and returned to the computer for processing and display. In addition, there may be control of digital (TTL) levels and magnetic field gradients or shims. If good quality data is to be collected, all of these functions must be applied at very specific times, with high precision and stability.

The Spinsolve spectrometer before 2023 use a DSP (digital signal processor) to control the timing and destination of the various signals. A short program runs on the processor to perform this. This program must be first written in a high-level language and then translated (automatically) into the required machine code. Because the DSP can only perform a single function at a time it is necessary to write the code in this way:

```
Perform an action
Wait a certain time
Perform next action
Wait a certain time
…
```

Much of the complexity of coding the sequence comes from determining how long to wait between actions.

The code for translating the high-level language used for pulse programming is defined in two DLLs (dynamic linked libraries) called UCSPPRun.dll and UCSLockPPRun.dll which are stored in the Expert/Prospa DLLs folder. (UCS stands for Ultra Compact Spectrometer – an early name for the Spinsolve). Each DLL includes translation information for some 50 commands used for controlling the spectrometer. However, of these, only a handful are commonly used. These are:

```
acquire ..  collect NMR data
delay ....  wait a certain number of microseconds.
wait .....  wait a certain number of microseconds (long version)
pulse ....  generate an RF pulse (or pulses) of specific amplitude, phase,
            duration and frequency.
loop .....  repeat a section of code a number of times.
endloop ..  define the end of the looped region
shim16 ...  set the current in one of the spectrometer shims.
txon .....  switch on the RF
txoff ....  switch off the RF
```

A full list may be found in Appendix A. All pulse programs are surrounded by two other commands initpp and endpp

```
initpp

…

endpp
```

You can see more complete information about each command by typing:

> help command_name

at the common line prompt or by pressing F1 when the insertion point is on the command name (this is valid for all Prospa commands).

When a pulse program is generated on the PC, each of these commands will produce a block of DSP assembly language code which will be stitched together by the endpp command and saved to a file for later compilation. The initpp command has an argument which specifies the filename for the pulse program output.

The parameters for each command can be of two kinds:

- Fixed – which means the command will always use this parameter value e.g. pulse(1, 16383, 0, 10). This will produce a full power pulse on channel 1 with phase 0 for 10 µs using the currently defined transmitter frequency. In this case you may need to know the conversion factor for the argument value from user, to digital units.

- Variable – in this case the parameters are variable names which will later be given the desired values e.g. pulse(1, a1, p1, d1). The names of these variables are important – the first letter corresponds to the type of the parameter according to this table

  ```
  aXXX ... An RF pulse amplitude in dB.
  pNNN ... A phase in degrees/90
  dXXX ... A delay in microseconds
  fXXX ... A frequency in MHz.
  nXXX ... An integer number
  tXXX ... A table of numbers
  ```

  Here XXX can be any valid Prospa variable name or an integer and NNN is an integer starting from 1. The phase numbers should be sequential (i.e. p1, p2, p3)

Before a pulse program is run, the compiled code is sent to the spectrometer and stored in a specific location (normally p:0x2000, where 0x means a hex number and p refers to program memory). If a command expects a variable, it will have a reference to another location in memory where this is stored. This is called the pulse program parameter block. This starts at location x:0x0000. Note that the DSP has 3 memories x, y and p. i.e. two data and one program memory. So this syntax means; x memory, address 0. This parameter block is updated just before the program starts and may be updated between scans if you need to change pulse parameter values. The order of the parameters in the parameter block will match the order found in the sequence.

So, for example, with the following pulse and collect sequence

```
pulse(1, a1, p1, d1, f1)
delay(d2)
acquire("overwrite",n1)
```

the parameters will be stored in this order, from address x:0000

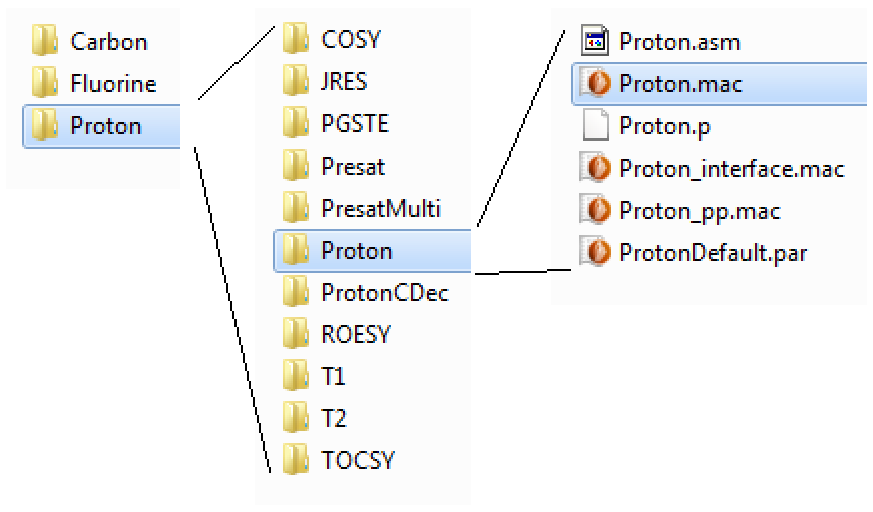
```
x:0000
...
system parameters
...
a1 ... pulse amplitude (a 14 bit number for amplitudes from -85 to 0 dB)
p1 ... pulse phase (an 18 bit number for phases from 0 to 359.999 degrees)
d1 ... pulse duration (a 24 bit number as a multiple of 20 ns)
f1 ... pulse frequency (two 16-bit words)
d2 ... pulse-acquisition delay (a 24 bit number as a multiple of 20 ns)
n1 ... number of points to acquire (a positive 24 bit integer)
```

The `system parameters` are present for all sequences and includes things like the default transmit and receive frequencies, the receiver amplifier gain and the digital receiver parameters.

The use of special first characters means the system automatically knows how to convert from user units (e.g. dB for RF amplitude) to system units (in this case a 14 bit integer).

2 File organisation

Experiments are typically grouped together, first based on the nucleus and then on the experiment name. Several files are required to run an experiment on the Spectrometer.



All of these files are stored in an experiment folder – 'Proton' in this case.

Each of these files has a particular purpose (for other experiments the name 'Proton' will be replaced by the experiment name):

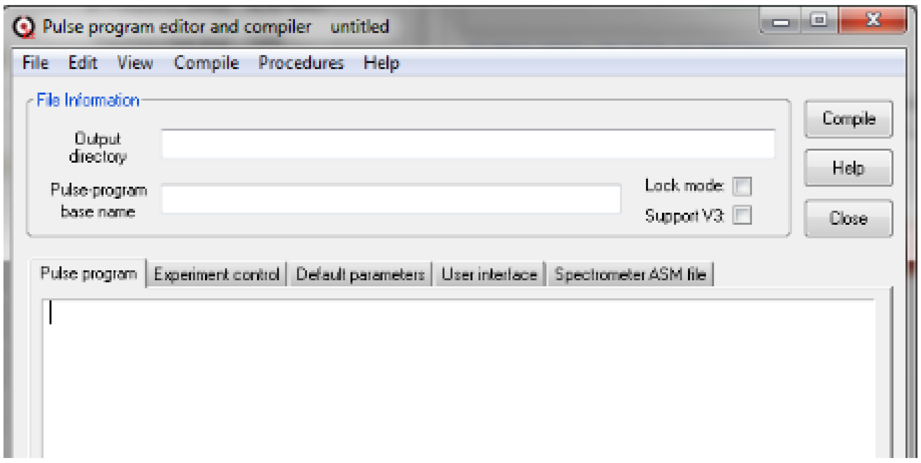
Proton.mac	the experiment control macro. This contains the code to start the experiment, collect and display the data and it also includes loops to accumulate scans or change parameters if it is a multidimensional experiment.
Proton_interface.mac ..	this defines the experiment user-interface as it will appear in the SpinsolveExpert application. It defines which parameters are visible, how the plot layout is organised and what post processing options are available. Part of this is automatically generated.
Proton_pp.mac	this defines the pulse program and parameters which are visible to the end user. It includes a table which defines the relationship between the user parameters and pulse program parameters. It also defines the phase cycling matrix for the experiment. This macro is only executed during the compilation phase.
Proton.asm	this is generated by the pulse program compiler and is an assembly language representation of the pulse program found in Proton_pp.mac. This is not used after the pulse program is compiled and is kept for reference only.

Proton.p	this is the machine level file generated from Proton.asm. This file is used to program the DSP and is reused once, each time the experiment is run.
ProtonDefault.par	this contains most of the parameters used in the experiment, with sensible values which should allow the experiment to run 'out of the box'. Notice that some the parameters here such as B1 frequency, pulse lengths and powers, so called common or factory parameters, may be overwritten after the file is loaded into Expert since they depend on the particular Spectrometer you are using. This latter information is read from a FLASH memory block stored on the Spectrometer DSP.

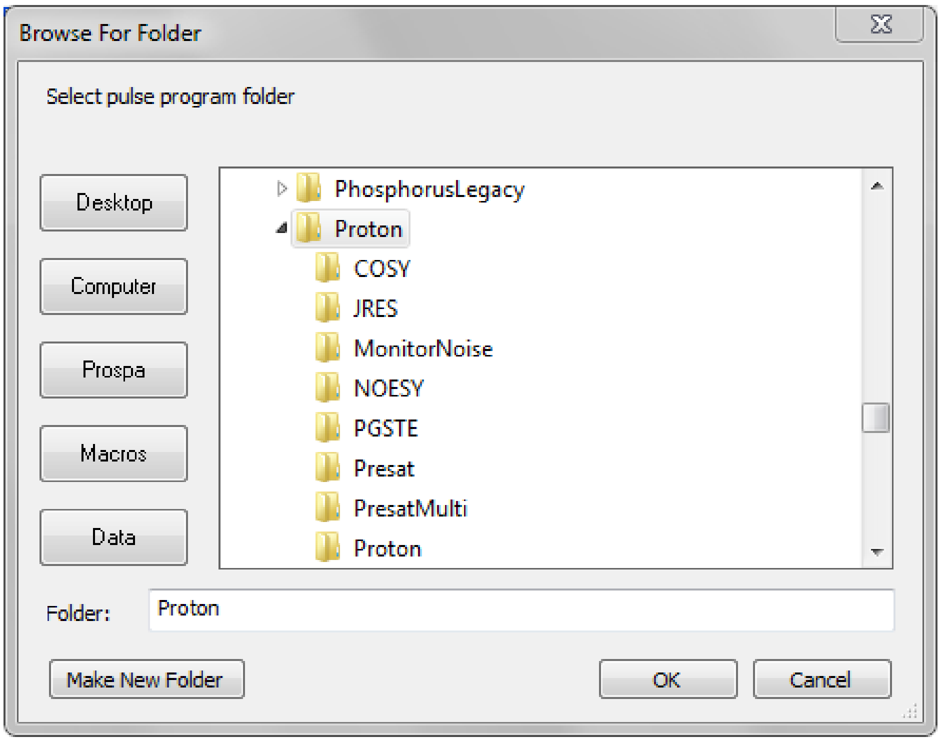
3 The Proton experiment examined

To make this clearer we will now look at the most basic experiment, 'Proton', in more detail.

To view the various experiment files, select the experiment from the appropriate menu (e.g. 1H) holding down the shift key at the same time. Alternatively, if the experiment is visible in the history or batch lists press the enter key or select the option 'Edit current pulse program' from the history list contextual menu. Finally, you can also access the pulse program editor from the options in the 'Experiment' menu. Either with the option 'Edit current protocol (Ctrl-Shift-P), or if you choose the option 'Open pulse program editor' then it will open with a blank interface.

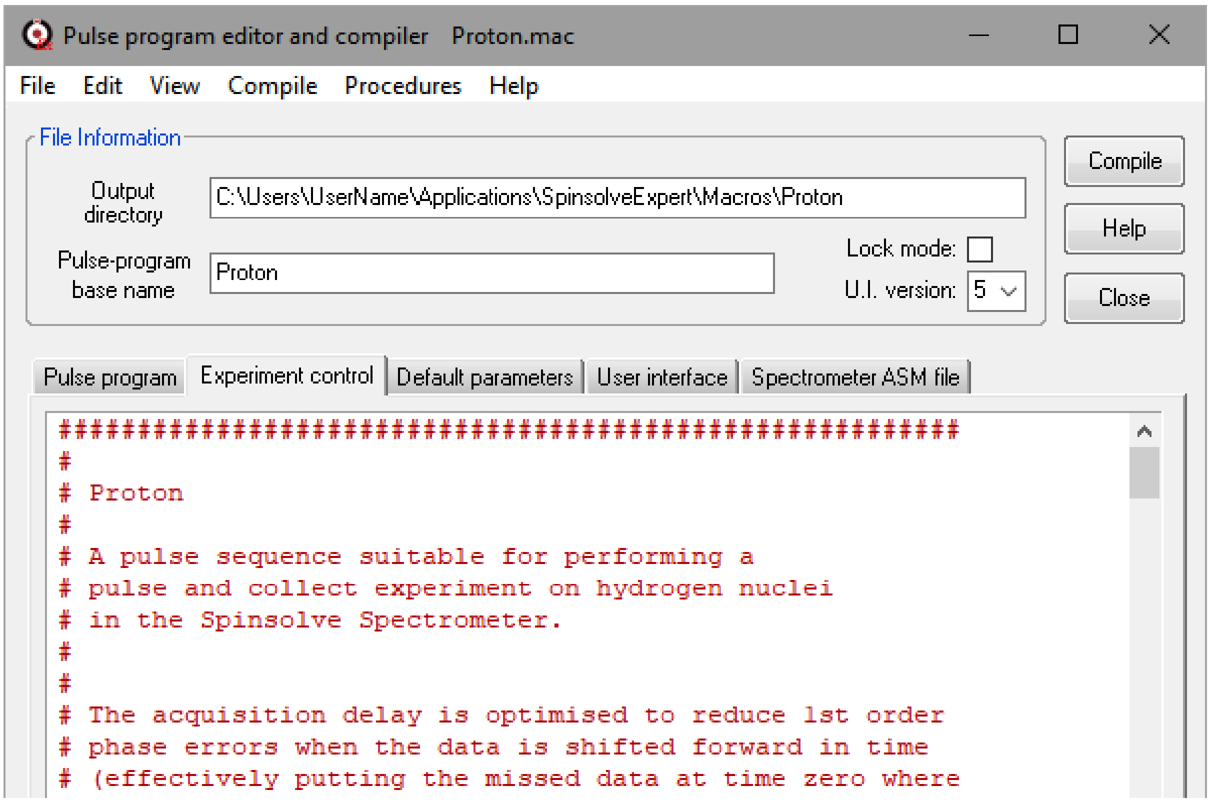


Then you can select the pulse program to load from the File->Open Pulse Program menu option (Ctrl+O).



Folder: Proton

With the Proton folder selected, press the OK button or press Enter. By default, this will open the Experiment control macro 'Proton.mac'.



Output directory: C:\Users\UserName\Applications\SpinsolveExpert\Macros\Proton

Pulse-program base name: Proton

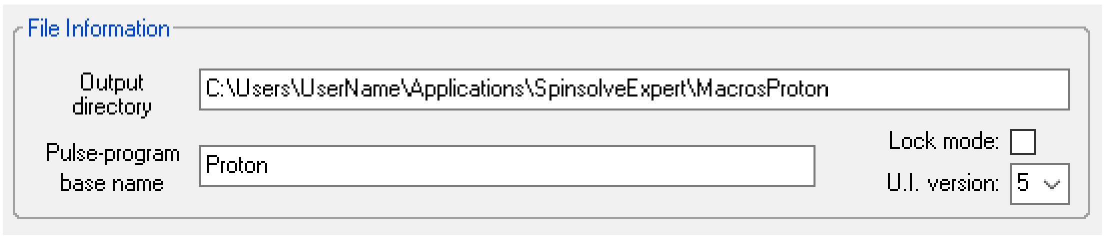
Lock mode: ☐

U.I. version: 5

```
##########################################################
#
# Proton
#
# A pulse sequence suitable for performing a
# pulse and collect experiment on hydrogen nuclei
# in the Spinsolve Spectrometer.
#
#
# The acquisition delay is optimised to reduce 1st order
# phase errors when the data is shifted forward in time
# (effectively putting the missed data at time zero where
```

The other important macros associated with the experiment are accessible from the different tabs above the editor. The name of the currently displayed file is visible in the title bar.

Two important file locations are given; the folder the pulse program files are in (Pulse program base name) and the parent directory (Output directory).



Output directory: C:\Users\UserName\Applications\SpinsolveExpert\MacrosProton

Pulse-program base name: Proton

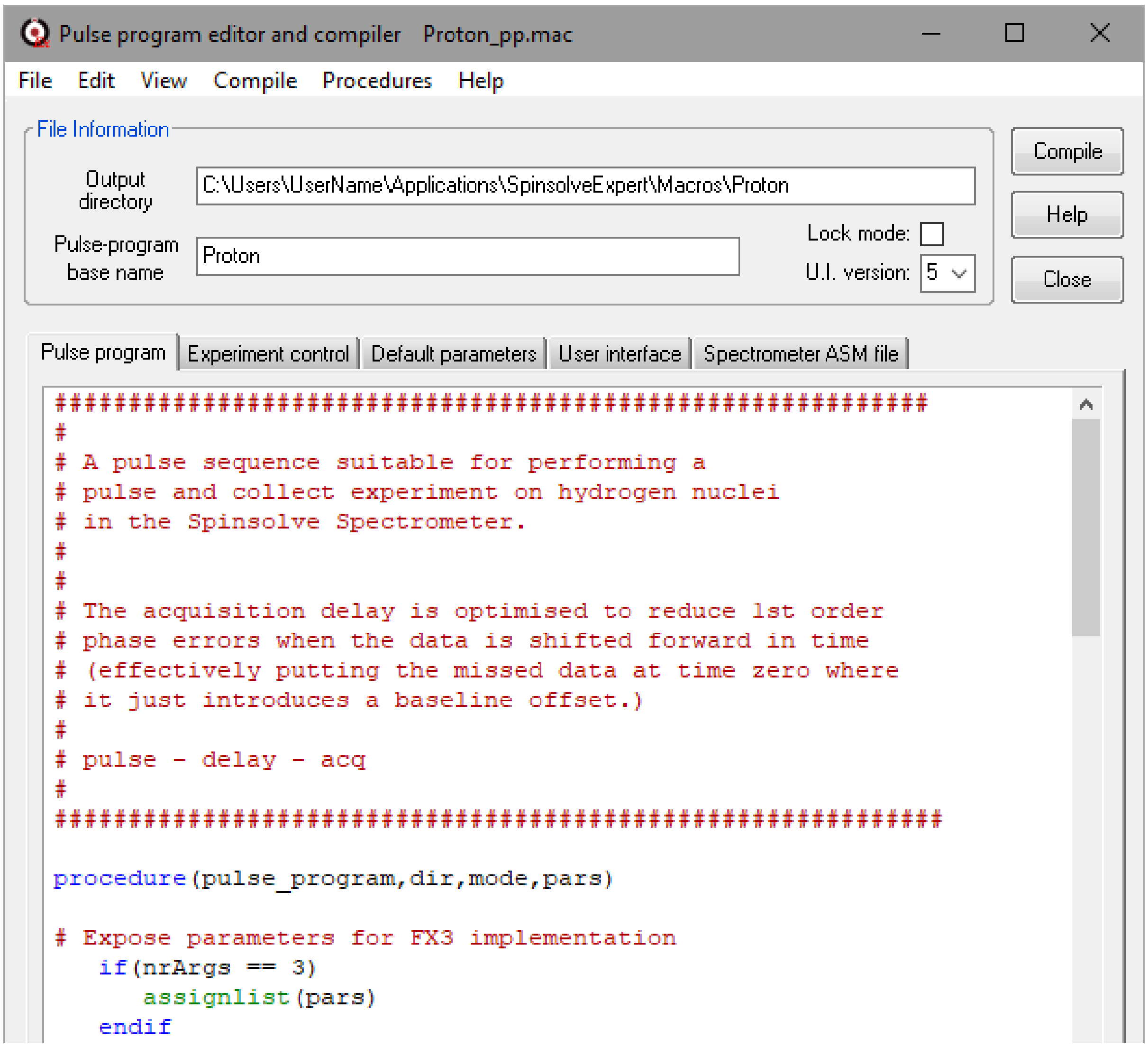
Lock mode: ☐

U.I. version: 5

The Lock mode check box is used when compiling lock associated pulse programs (which for most users will be never), while the user interface (U.I.) version should be set to 5.

3.1 The Pulse Program Macro

We will start by looking at the pulse-program macro. Click on the pulse program tab:



Output directory: C:\Users\UserName\Applications\SpinsolveExpert\Macros\Proton

Pulse-program base name: Proton

Lock mode: ☐

U.I. version: 5

```
############################################################
#
# A pulse sequence suitable for performing a
# pulse and collect experiment on hydrogen nuclei
# in the Spinsolve Spectrometer.
#
#
# The acquisition delay is optimised to reduce 1st order
# phase errors when the data is shifted forward in time
# (effectively putting the missed data at time zero where
# it just introduces a baseline offset.)
#
# pulse - delay - acq
#
############################################################

procedure(pulse_program,dir,mode,pars)

# Expose parameters for FX3 implementation
   if(nrArgs == 3)
      assignlist(pars)
   endif
```

This macro contains the procedure pulse_program. It has three argument, ‘dir’ which is the location where the pulse program will be processed and ‘mode’ which is the particular pulsing mode. The third parameter ‘pars’ is not used in the DSP version. The ‘mode’ argument will always be ‘1’, standing for channel 1 (the proton channel). ‘mode’ is a carry-over from the Kea spectrometer which could have internal or external RF amplifiers. For Spinsolve applications you won’t need to use ‘mode’ and you should explicitly specify which channel you wish to access. However, for backward compatibility this line should not be modified.

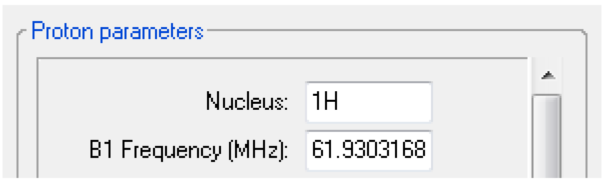
The procedure is broken into several parts; the interface description, the relationships list and optional group interface, the variables list, the pulse program and the phase cycling matrix. We will look at each of these in turn.

3.2 The interface description

The interface is described by a Prospa 2D list[1], each row of which describes one user interface element and label. Note that the last symbol on each row (except in the very last row) must be a semicolon. As an example, the first two rows in Proton are shown below:

```
  interface = ["nucleus",     "Nucleus",                "tb",     "readonly_string";
               "b1Freq1H",    "B1 Frequency (MHz)",     "tb",     "freq";
```

These define the nucleus and the B1 NMR frequency for protons. These generate the first two entries in the Expert Proton parameters user interface.



Nucleus: 1H
B1 Frequency (MHz): 61.9303168

For each parameter, the interface list entries are; variable name, interface label, type of user interface element (or control) and some special information about the entered data (data type). The abbreviations in column 3 are defined below.

```
tb ......  a text entry (box) field
tm ......  a text menu (a text entry field with a pull down menu)
cb ......  a check box
rb ......  radio buttons
bt ......  a standard button
dv ......  a line with a label to divide controls into groups
```

Note that 3 letter symbols are sometimes used, but they are only relevant in V3 interface (e.g. ‘w’ makes the control wider, however all Expert text entry fields are already wide).

The data type field can take a number of different parameters and formats:

```
double  .........  a general double precision floating point number.
```

[1] A 1D list can be used, but it requires two addition parameters per row which are only used in the V3 legacy interface. Some experiments still use the 1D list.

float	a general single precision floating point number.
freq	we expect the entry to be a frequency in MHz. This makes it double precision and it must fall within a certain range of values as define in the Spinsolve preferences file (5-110 MHz).
integer	a general positive integer number [1 ... 1e8].
ldelay	a long delay. For use with the wait command. This can range from 2 µs to 167 seconds. Units default to microseconds.
ldelayms	a long delay[2]. For use with the wait command. This can range from 2 µs to 167 seconds. Units default to milliseconds.
other	the same as float.
pulseamp	a pulse amplitude in dB. This number can range from -85 (no power) to 0 (maximum power). Note that this is a logarithmic scale with a step size of 6 dB; very closely corresponding to twice the amplitude or quadruple the power.
pulselength	a pulse length in µs. This can range from 0.5 to 1000 µs.
readonly_string .	a string which cannot be changed by the end user.
reptime	a repetition time in milliseconds. This cannot be less than 1 ms.
sdelay	a short delay[2] – for use with the delay command. This can range from 2 µs to 327.67 ms. Units default to microseconds.
string	a text string.

If used in text-boxes, the generic numeric types `integer`, `float` and `double` can also have an allowable range value by using a two element array e.g. `float, [1,100]` which means only numbers between 1 and 100 inclusive are acceptable. Alternatively, if the control is a text menu, then the array is interpreted as the various menu options e.g. `float, [1,2,3,4,5,6,7,8,9]`

Some control types require additional information in the data type field:

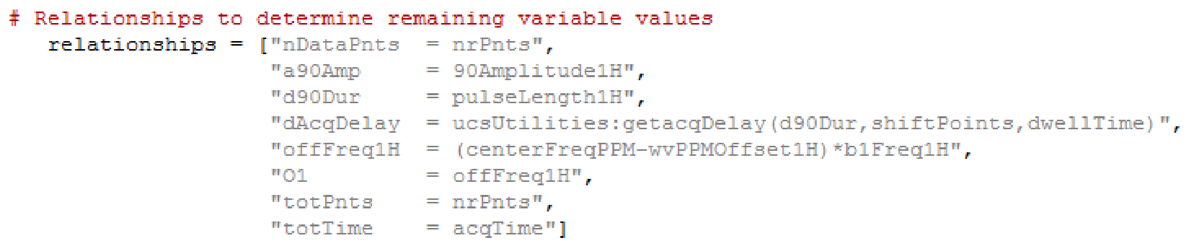
checkboxes	the data type should always be a string and the two options (unchecked and checked) are separated by a comma e.g. "no,yes"
radiobuttons ...	the data type should always be a string and the various options are separated by a comma e.g. "18,opt1,opt2,opt3,Option1,Option2, Option3". This will generate 3 radio buttons with variables names optX and labels OptionX. The first number is the vertical spacing in pixels between the buttons.
textmenus	if the allowable menu options are strings then you should supply a Prospa list e.g. [\"xy\",\"yx\",\"xz\",\"zx\",\"yz\",\"zy\"]. Note that because the data type is a string, the menu sub-strings must have their quotes escaped with a backslash. As mentioned above numerical arrays can be supplied if the options are all numbers.

[2] See the pulse program commands section.

```
buttons ....... the type in this case is a string which contains the
                callback procedure name for that button. e.g. "zeroShims".
                The callback procedure must in the Experiment-Control
                macro.
```

3.3 The relationships list

As noted previously, the variables used in the pulse program description must conform to strict naming conventions. Also, they are typically not the same as the variables we wish to expose to the end user. For this reason, we define a relationships list which defines each pulse program parameter in terms of the user interface parameters. In the case of the Proton experiment these relationships are quite simple



```
# Relationships to determine remaining variable values
   relationships = ["nDataPnts   = nrPnts",
                    "a90Amp      = 90Amplitude1H",
                    "d90Dur      = pulseLength1H",
                    "dAcqDelay   = ucsUtilities:getacqDelay(d90Dur,shiftPoints,dwellTime)",
                    "offFreq1H   = (centerFreqPPM-wvPPMOffset1H)*b1Freq1H",
                    "O1          = offFreq1H",
                    "totPnts     = nrPnts",
                    "totTime     = acqTime"]
```

Note that the relationship list is expressed as a list of strings, since these must be stored for later evaluation.

Each entry in the list defines a pulse program variable. Note that it can also define other variables which are used throughout the software. In this case O1 is the offset frequency, totPnts the total number of points collected (which for CPMG experiments may be more than the nrPnts, the number of points collected per echo), and similarly the totTime is the total acquisition time which might be longer than the individual acquisition time; acqTime (also the case for the CPMG).

This relationships list also shows an example of a function call to update a parameter – in this case the acquisition delay, which depends on several other parameters if we wish to avoid first order phase corrections. Note that we need to include the file name in addition to the procedure name when including procedures in the relationships list. i.e:

```
file_name:procedure_name(arguments)
```

If filenames include a hyphen, then you should quote them to prevent this being interpreted as a numerical operation e.g.

```
"acqShift = \"PGSTE-Li_pp:getAcqShift\"(dwellTime)",
```

Note that some parameters used here and in other relationships list are defined elsewhere in the Expert code. These include:

acqTime	the ideal duration of the data acquisition (dwellTime * nrPnts) in milliseconds.
bandwidth	the bandwidth of the data acquisition (in kHz) Equal to 1000/dwellTime.
bandwidth2 ..	the bandwidth of the second dimension in a 2D experiment (in kHz).

dwellTime	the sampling interval in microseconds.
nrPnts	the number of points to collect during the acquire command.
pgo	the pulse gate overhead delay. This is defined in the Spinsolve preferences dialog and defaults to 5 µs. It is the time between the start of an RF pulse command and when the pulse actually appears.
rxLat	receiver latency. A time shift required to align the collected data with the ideal start of the sampling interval. (In microseconds).
x/y/zshim	the shim values found for the linear gradients. These are updated after a shim experiment is performed. These are required when crusher gradients are applied.

3.4 The group list

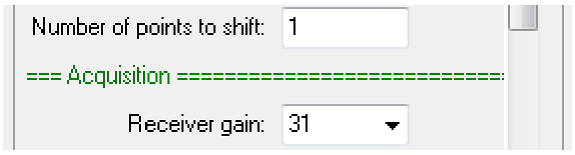
It is possible to define all the graphical user elements in the interface list and that is the case with the Proton sequence. However, if you wish to minimise the number of elements in the list you can define a 'groups' list. These are predefined groups of controls which will be included with a single name. e.g. the Proton interface is included with the following list (currently commented out):

```
   groups = ["Pulse_sequence","Progress","Acquisition",
             "Processing_Std","Display_Std","File_Settings"]
```

Here the first entry includes those items defined in the interface list. The name for the group list in older versions of Expert was 'tabs' and refers to the legacy interface (V3) which grouped controls into tabs. In the case of the newer Expert interface the controls groups are separated by dividers (dv) e.g

```
                  "",    "Acquisition", "dv", "";
```

Note that the divider control has no associated variable or data type. It produces the following result in the user interface:


Number of points to shift: 1
=== Acquisition ===========================
Receiver gain: 31

Control groups are defined in the following folder:

$appdir$\\Macros\\UCS-PP\\Tabs\\AlternateInterface

For example, the Display_Std interface macro looks like this:

```
procedure(interface_description)

  interface = [
      "usePPMScale","Use ppm scale?","cb", "string", "no,yes",
      "dispRangeMinPPM","Minimum ppm value","tb", "float", "[-2000,2000]",
      "dispRangeMaxPPM","Maximum ppm value","tb", "float", "[-2000,2000]",
      "dispRange","Display range (Hz)","tb", "float", "[0,2e6]"]

endproc(interface,"Display")
```

Note that currently, control group interfaces use 1D lists.

For a description of the supplied group lists please see Appendix B. Of course, you can also add your own groups using the supplied ones as examples.

If you are not going to use groups, then you don't need to include this list.

3.5 The variable list

This is a list of the pulse program variables that you wish to modify during the experiment. For example, if you are performing an experiment where the RF pulse amplitude is varied you would include the name of the amplitude variable, e.g. in the Proton experiment this might be `a90Amp`. In this case you would write:

```
   variables = ["a90Amp"]
```

If there are no variables to modify then make a single empty element i.e. `[""]`.

3.6 The pulse program

This is the core of this file. These commands describe the pulse program which will be run on the spectrometer. They must be surrounded by the initpp and endpp commands.

```
# Pulse sequence
   initpp(dir)                     # Define compile directory and clear parameter list

   delay(5000)                     # Wait 5 ms, allowing time to finish lock scan
   pulse(1,a90Amp,p1,d90Dur)       # RF pulse on channel 1 with phase p1
   delay(dAcqDelay)                # Pulse - acquire delay
   acquire("overwrite",nDataPnts)  # Acquire FID

   parList = endpp()               # Combine commands and return ordered parameter list
```

These commands are rather special – unlike other Prospa commands they don't execute the action immediately but build up the pulse program in an intermediate assembly language file which will be compiled and executed later on the spectrometer processor. The intermediate language file is stored in the directory 'dir' passed to the initpp command. As mentioned previously, either constants or special variables should be used as arguments. The endpp completes the file assembly process and returns a list of variable names in the order they appear in the pulse sequence. In this case

```
parList = ["a90Amp", "p1", "d90Dur", "dAcqDelay", "nDataPnts"]
```

Note that inside the pulse program macro, unquoted strings are treated as quoted strings (as they are in the command line interface). This keeps the pulse sequence description cleaner and easier to read. To prevent an external global variable from accidentally replacing an unquoted pulse sequence argument, global and window variables are not accessible from within this macro.

3.7 The phase cycle matrix

Phase cycling is performed by replacing the parameters p1, p2 ... with the phases stored in the phaseList 2D matrix before each scan. Each row of the matrix corresponds to a different phase variable while the last row is the acquisition phase. Each column is applied in a different scan. Column 1 for the first scan, column 2 for the second and then wrapping around to the start when we run out of columns. The numbers in the matrix represent multiples of 90 degrees. Fractional numbers can be used for smaller increments. (e.g. 0.5 == 45 degrees)

So, in the Proton example,

```
   phaseList  = [0,1,2,3;    # p1 : Pulse phase
                  0,1,2,3]    # pA : Acquire phase
```

the phase of the RF pulse steps from 0 to 90 to 180 to 270 as the scans are incremented. The acquisition phase is stepped in the same way.

If phase cycling is disabled in the user interface using the ‘Phase cycle’ check box then the phases in the first column will be used in all scans.

3.8 The returned variables

The various lists generated in the pulse program need to be returned to the compiler. This is done with a parameter list in the endproc command.

```
endproc(parList,group,interface,relationships,variables,dim,phaseList)
```

In addition to the lists we have just discussed, there is also a **dim** variable which is used in the legacy interface (V3). This should be replaced by zero if it is not used, however it must still be included for backward compatibility.

If the group list is not used then you can replace this variable with a null list i.e. list(0). So for example, in the Proton pulse sequence file, if the group list is not used, the return parameters would look like this

```
endproc(parList,list(0),interface,relationships,variables,0,phaseList)
```

3.9 Getting factory parameters

To simplify the selection of NMR parameters it is recommended that you include the procedure *getFactoryBasedParameters* in the pulse program macro. If this procedure exists it is called when the experiment is selected from the menu.

It is worth considering how the user interface parameters are chosen when selecting an experiment from the main menu.

Once the parameter interface list has been created it is populated with the default parameter list – the file for this is displayed in the Default parameter list.

Next the program will look for the getFactoryBasedParameters procedure in the Pulse program or the Experiment control file. If it exists, then this procedure will be executed. Typically, this procedure will contain a command to read the factory parameter settings from the spectrometer (these are manually accessible using the 'Open Spinsolve parameter dialog' option in the main File menu.) This command looks like this in the proton experiment:

```
   specPar = SpinsolveParameterUpdater:readDSPParameters("1H")
```

SpinsolveParameterUpdater.mac is the file and readDSPParameters is the procedure. The argument should be the nucleus of interest. This is important since some spectrometers support multiple nuclei and the parameters returned may depend on that selection.

specPar is then a structure returned with these parameters. You can add a print statement to see the contents of this structure. Once loaded this structure is made into local variables with the assignlist command and then a new parameter list is formed using these values e.g.

```
   par = ["rxGain        = $modelPar->rxGain$",
          "pulseLength1H = $PulseLength_1H$",
          "90Amplitude1H = $PowerLevel_1H$",
          "b1Freq1H      = $Frequency_1H$"]
```

PulseLength_1H, PowerLevel_1H and Frequency_1H are all spectrometer parameters – values which have been calibrated for your spectrometer in the factory. The rxGain parameter will depend on the spectrometer frequency but this is more generic and is stored in the modelPar structure which is returned from ucsUtilities:getModelBasedParameters procedure called before the above lines.

This parameter list is then returned to the calling program and is used to update those parameters listed on the left of this list.

The final step is to replace the frequencies with the values found during the last lock and calibrate step. These are called common parameters.

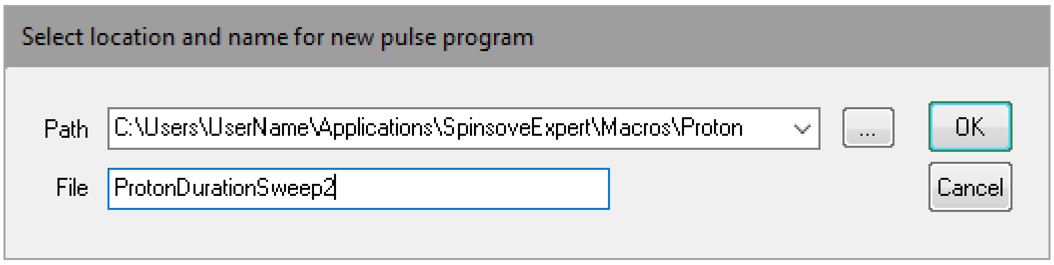
If getFactoryBasedParameter is not present, then those values stored in the default parameter file will be used instead (and these will usually be inappropriate for your spectrometer.)

Note that you can always manually change these parameters before the experiment starts (except for the common parameters which are read only).

4 Writing a new pulse program

With this brief overview of the Proton sequence and how parameters are loaded, we can move on to looking how a new pulse program might be written. Because of the number and complexity of the files involved it doesn't make sense to write an experiment from scratch. Rather you should take an existing experiment and modify it.

Start by opening an experiment which is similar to the one you want to write. For example, let's write a pulse duration-sweep experiment to determine the correct 90-degree pulse length. Since the core experiment here is a simple Pulse and Collect, first open Proton in the pulse sequence editor and compiler window. Then in the file menu, select the option "New pulse program from existing file". Select Proton from the dialog window which appears and enter the name for the new experiment – in this case ProtonDurationSweep2:


Select location and name for new pulse program

Path C:\Users\UserName\Applications\SpinsoveExpert\Macros\Proton

File ProtonDurationSweep2

Note that experiment names should be unique as Expert will simply run the first one it finds with this name. Since ProtonDurationSweep already exists in the Proton menu we have added a '2' at the end.

Press OK. This will make a copy of all the Proton files, but with the new name substituted. If you compile this experiment (press the compile button) it will produce identical results to the Proton experiment. The new experiment will now also be visible in the Proton (1H) menu. It is recommended that you choose a different path for your own experiments to prevent accidental deletion when updating the software. In this case you will need to add that path to the main menu to access the experiment. This can be done by dragging and dropping the folder corresponding to this path onto the Expert main menu or adding it using the 'Experiments Menus' option in the SpinsolveExpert preferences dialog found in the Files menu. Note that you must drag and drop the *parent* directory of ProtonDurationSweep2 onto the menu bar not the ProtonDurationSweep2 directory itself.

Now we can modify the experiment. Since we want to modify the pulse duration, we need to expose the pulse length in the pulse program. To do this, modify the variable list to include the pulse length:

```
            variables = ["d90Dur"]
```

Save the pulse program (Ctrl+S) and recompile (Press the Compile button or Ctrl+Shift+C)

Before we modify the Experiment Control Macro to sweep the pulse duration let's have a look at the structure of this file.

5 The Experiment Control Macro

This macro contains several procedures:

Main procedure (name will be the same as the macro) ... this provides the option to add the experiment to the user interface or if the shift key is pressed, to edit the experiment.

backdoor ... this allows the experiment to be run without using the legacy V3 user interface. Instead, a list of parameters is passed. This is used by all Expert experiments.

getseqpar ... get the sequence parameters. This returns the important parameters which were generated when the pulse program was compiled. This includes the relationships list, the variables list, the list of pulse program parameters, the pulse program name, and the phase cycling list.

execpp ... executes the pulse program. This contains the commands to send the pulse sequence file (*.p) and the list of parameter values to the spectrometer. It runs the sequence and then waits for it to finish. It then reads the collected data, processes and displays it. It then repeats this process for the desired number of scans, phase cycling the RF pulses and accumulating the collected data as specified in the phase list. When the scans have completed the data are saved to the folder specified by the Experiment base-path, date and time hierarchy and file comment.

getPlotInfo ... extracts the file names associated with the various plots in the user interface. This is used for saving the data in each plot and also for reloading the last data set when the experiment is reselected.

expectedDuration ... this returns the expected duration of the experiment in seconds.

saveProcPar ... saves the processing parameters when the experiment is complete. This is useful if one wants to reprocess the raw FID data in the same way at a later date.

Once an experiment has been selected from the experiment menus and the Run button is pressed, the following actions will occur before the execpp procedure is called.

1. A check is made to see if the system is locked and if the temperature is within acceptable limits. This does not prevent the experiment from running, but updates the Status indicators.
2. The user interface is modified for Run mode. This means the required plots are displayed and various controls are disabled to prevent unwanted user interactions while the experiment is running.
3. A separate thread is run to control the progress bar based on the value returned by the expectedDuration procedure.
4. Another thread is started to execute the experiment. Runing the progress bar update and experiment in separate thread allows the user interface view to be modified and windows resized during the experiment.

5. In the experiment thread the experimental parameters are read from the user interface and merged with spectrometer, and common parameters. All parameters are checked for invalid values.
6. The backdoor procedure is then called. This calls an initialisation procedure (`initAndRunPP`). Here the experiment data folder is created and an acquisition parameter file acqu.par is written based on the chosen experimental parameters.
7. The user defined experimental and pulse sequence parameters are then combined with fixed spectrometer parameters and system preferences to produce the required low level list of pulse sequence parameters which will be passed to the spectrometer processor. For example, this involves converting RF amplitudes from dB to the 14 bit digital value required by the spectrometer's digital synthesiser.
8. The pulse program in binary form is sent to the spectrometer via USB.
9. The execpp procedure in the Experiment Control macro is the run with the pulse sequence parameter list as an argument (ppList) along with other lists.

The execpp procedure then performs the following functions:

1. It sends the experiment parameters to the spectrometer (which may change from scan to scan).
2. It executes the pulse program, collecting and scaling the data.
3. The data is then processed, displayed and saved.

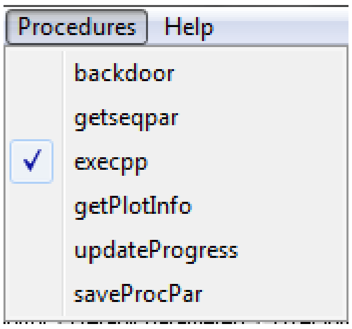
Following the execution of execpp the interface is enabled and restored to normal operating mode (which allows menu activation and all plot interactions).

We will now look at the function of execpp in more detail.

5.1 The execpp (execute pulse program) procedure arguments

If the pulse sequence code in the Pulse Sequence macro defines the experiment, then the execpp procedure in the Experiment Control macro controls it. Let's looks at the Proton experiment macro.

First note that you can jump to different procedures in a macro by using the Procedures menu in the Pulse program editor and compiler window (or any Prospa editor).



The first line of the procedure lists the passed arguments:

```
procedure(execpp,guipar,ppList,pcList,pcIndex,varIndex)
```

execpp is the procedure name, guipar is the list consisting of all the parameters defined in the Proton parameter list plus all the parameters in the relationships list defined in the pulse program macro. Here are some of them:

```
guipar = {
       90Amplitude1H = 0
       a90Amp        = 90Amplitude1H
       accumulate    = "yes"
       acqTime       = 3276.8
       b1Freq1H      = 61.9303168d
       bandwidth     = 5
       d90Dur        = pulseLength1H
       ...
       yshim         = -721.995
       zf            = 1
       zshim         = -1753.47
     }
```

It also includes a number of global variables such as receiver calibrations and gyromagnetic ratios for the various nuclei, as well as other information which may be needed in the macro.

You can list these (or other parameters) yourself by adding the line

```
pr sortlist(guipar)
```

to the execpp code. To see all locally define variables add the line

```
pr local
```

The ppList contains all the generic parameters values required by the spectrometer along with the (highlighted) pulse program parameter values in the same order as they were introduced in the pulse sequence.

```
 0 000F00                                18 00000F
 1 000500                                19 0F4240
 2 004E20                                20 0000E8
 3 000023                                21 000001
 4 000000                                22 000001
 5 00002E                                23 000006
 6 000001                                24 0000E8
 7 000002                                25 0000C8
 8 000001                                26 000000
 9 000A78                                27 000000
 10 008620                               28 003FFF
 11 000200                               29 000000
 12 000004                               30 0002CF
 13 0068B5                               31 0006D8
 14 003D40                               32 004000
 15 000010
 16 00001F
 17 00C4AC
```

To print it in this format, add the following lines after the start of the execpp procedure

```
    for(k = 0 to size(ppList)-1)
       pr "$k$ $hex(ppList [k],24)$\n"
    next(k)
```

If you open the Spectrometer ASM file, then at the beginning of the file you will see the corresponding parameter for each of these values. Here we just show the pulse program parameters 28-32

```
TXA90Amp            ds      1       ; 28 - Tx amplitude 90Amp word 0
TXP1                ds      1       ; 29 - Tx phase 1
DELAY90Dur          ds      1       ; 30 - Delay 90Dur
DELAYAcqDelay       ds      1       ; 31 - Delay AcqDelay
NRDataPnts          ds      1       ; 32 - Number DataPnts
```

pcList is the phase cycle list, but now scaled to be in hardware relevant units (90 degrees == 16384)

```
   pcList =

       0   16384   32768   49152
       0   16384   32768   49152
```

while pcIndex is their zero based position in the ppList.

```
   pcIndex =

   29
```

varIndex is the position of the variables to modified in ppList. (Not used in Proton)

5.2 The execpp procedure in detail

Now we can break down the various parts of the execpp procedure. The first step is to extract all the guipar list parameters and make them local variables. This is done with the assignlist command:

```
# Make all gui parameters available
   assignlist(guipar)
```

Next, we allocate space for the collected data. totPnts is the number of complex points in the FID which is also the same as nrPnts in this case,

```
# Allocate space for output data
   sumData = cmatrix(totPnts)
```

We then determine the axes for the time (FID) and frequency (Spectral) plots

```
# Calculate suitable time and frequency axes
   tAxis = ([0:1:totPnts-1]/totPnts)*totTime*1000 # ms
   fAxis = [-totPnts*zf/2:totPnts*zf/2-1]/(totTime*zf)*1000 # Hz
```

In addition to `totPnts` this also uses `totTime` (== `acqTime`) and the zero-fill parameter zf. This last parameter was part of the guipar list.

If the user wants to apply a time domain apodization to improve SNR we extract this from the filters macro

```
# Time domain filter
   if(filter == "yes")
      flt = filters:get_filter(filterType,"FTFid",tAxis/1e6)
   else
      flt = matrix(totPnts)+1
   endif
```

If not, then the filter is just a constant vector equal to 1.

Next, we indicate that we want 2 plot regions, one for the FID and one for the Spectrum. This function returns plot objects we can use for performing the plot.

```
# Get plot regions
   (prt,prf) = ucsPlot:getPlotReferences()
```

To simplify the generation of axes labels and scales which depend on the ppm/Hz setting and frequency offsets, we call a procedure in the ucsPlot macro to calculate this for us.

```
# Work out frequency axis scale, label and range
   (fAxisDisp,fAxisLabel,fRange) = ucsPlot:generate1DFrequencyAxis(prf,
fAxis, guipar)
```

The core elements of a simple pulse and collect experiment are shown below. First we have the Scan loop:

```
# Accumulate scans
for(scan = 0 to nrScans-1)
```

The next command combines several steps – it checks the timing, updates the parameters on the spectrometer, runs the sequence and returns the data

```
    # Check timing, update the parameters, run the sequence and return the
data
      (data,pAcq,status) =
ucsRun:runSequence(guipar,ppList,pcList,pcIndex,scan)
```

The collected data is accumulated, with the acquisition phase cycle being applied.

```
    # Accumlate the data
      sumData = ucsRun:accumulate(accumulate,pAcq,sumData,data)
```

The accumulated data is zero filled and then Fourier transformed:

```
    # Process data
```

```
      (phasedTimeData,spectrum,ph0) =
      ucsRun:transformData(zerofill(datacorr.*flt,zf*totPnts,"end"),fAxis,g
      uipar,"fid")
```

And finally plotted

```
    # Plot the data
      ucsPlot:graphTimeAndFreq(prt,prf,tAxis,datacorr,fAxisDisp,spectrum,sc
      an,guipar, "Time data","Spectral data", "Time (ms)","Amplitude
      (\G(m)V)", fAxisLabel,"Amplitude")
```

And then we check to see if the Complete button has been pressed during the experiment

```
    # Check if complete button pressed
      if(status == "finish")
         scan = scan+1
         exitfor()
      endif

   next(scan)
```

The Proton experiment also includes some commands to minimise first order phase shifts by setting the acquisition delay carefully and modifying the first few data points to account for the digital filters. These are not essential for operation and so for simplicity will not be discussed here.

The next step is to save the followed data

```
# Save the data
   ucsFiles:savePlot(prt,:getPlotInfo("pt1"),guipar,"noReport")
   ucsFiles:savePlot(prf,:getPlotInfo("pt2"),guipar,"simpleReport")
   ucsFiles:saveMNovaData(prt,"",guipar,"simpleReport")
```

This saves the FID and spectral plots in a proprietary Prospa format so we can reload the data later, and also in a more basic format ‘data.1d’ for use with the 3rd party application Mnova. Note the references to the procedure getPlotInfo which returns the filenames for the FID and spectral files.

Finally, the processing parameters used are saved for future reference in a proc.par file

```
# Save the processing parameters
   :saveProcPar(guipar,ph0,fRange)
```

and then the data is packed up into a structure and returned to the calling procedure:

```
# Pack the data into a structure
   result = struct()
   result->tx = tAxis
   result->ty = sumData/scan
   result->fx = fAxisDisp
   result->fy = spectrum/scan
   result->par = struct(guipar)

# Return result
```

```
   return(result)
```

5.3 Modifying the execpp procedure to vary the pulse duration

Now we have covered the basic pulse and collect experiment we will consider how to incorporate changing the pulse length parameter as discussed earlier in *Writing a new pulse program*. There we included a non-empty variable parameter list in the pulse sequence file:

```
variables = ["d90Dur"]
```

In the execpp macro this will appear in the varIndex argument as the value 30

varIndex = [30]

This refers to element 30 in the parameter list ppList:

```
TXA90Amp             ds       1         ; 28 - Tx amplitude 90Amp word 0
TXP1                 ds       1         ; 29 - Tx phase 1
DELAY90Dur           ds       1         ; 30 - Delay 90Dur
DELAYAcqDelay        ds       1         ; 31 - Delay AcqDelay
NRDataPnts           ds       1         ; 32 - Number DataPnts
```

To use this parameter we add an extra for-loop outside the scan loop. We also add extra storage for each spectrum recorded:

```
# Calculate the pulse length matrix
   minLength = pulseLength1H
   maxLength = minLength+nrLengthSteps*stepNr
   pulseLengths = linspace(minLength, maxLength, nrLengthSteps)

# Storage for the fids and spectra
   fid2d = cmatrix(totPnts, nrLengthSteps)
   spectra = cmatrix(totPnts*zf, nrLengthSteps)

# Vary the pulse duration
   for(stepNr = 0 to nrLengthSteps-1)

   # Allocate space for output data
      sumData = cmatrix(totPnts)

   # Accumulate scans
      for(scan = 0 to nrScans-1)
```

Note that since we have introduced three new user variables; minLength, lengthStep and nrLengthSteps we need them to the pulse sequence user interface as well:

```
"minPulseLength",  "Min. pulse length (µs)",    "tb",  "pulselength";
"lengthStep",      "Pulse step size (µs)",      "tb",  "pulselength";
"nrLengthSteps",   "Nr. of length steps",       "tb",  "integer,[1,100]";
```

Don't forget to recompile after making changes to the *_pp.mac file.

The next step is to save each collected spectrum into the 2D spectra matrix. This is done between the end of the scan loop and the end of the pulse length loop.

```
      next(scan)

     # Save the FID and Spectrum
      fid2d[~,stepNr] = sumData
      spectra[~,stepNr] = spectrum

   next(stepNr)
```

The tilda symbol (~) means replace a row. A colon (as in Matlab) can also be used. Note that the 2D matrix index order in Prospa is column, row (x,y) not row, column as it is in other languages. (We considered the x,y,z order for 3D matrices more logical than y,x,z).

We also save the fids to allow export to MNova.

To give some feedback to the user we can plot the accumulated spectra into an additional plot. To do this we need to add the extra plot pt3 in the User Interface macro:

```
procedure(plot_run_layout)

   layout = ["pt1", "pt2"; "pt3"]

endproc(layout)


procedure(plot_load_layout)

   layout = ["pt3"]

endproc(layout)
```

These define the layout of plots when running and reloading experiments.

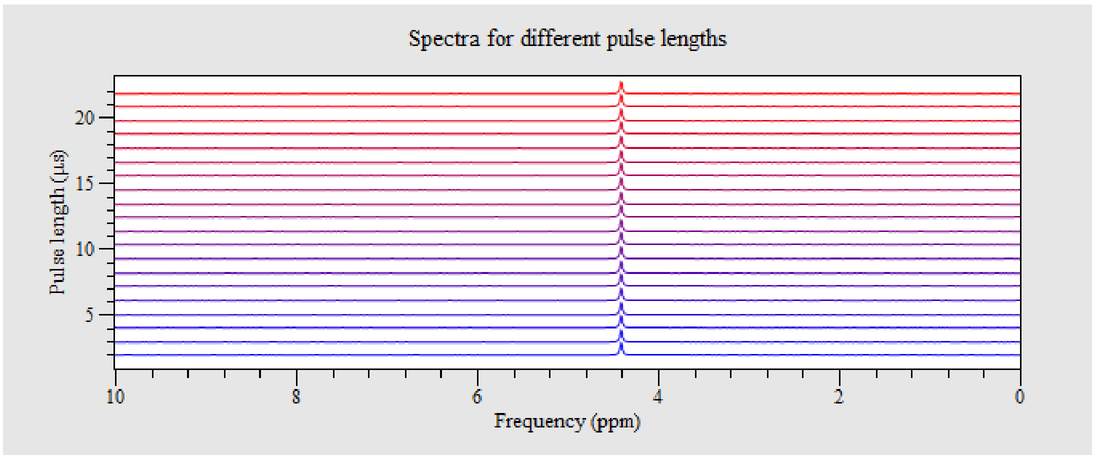
We also need to add a reference to this third plot in the execpp procedure by changing the number of plots from 2 to 3 and returning another plot name `prs`:

```
   (prt,prf,prs) = ucsPlot:getPlotReferences()
```

We can then display the collected data as it is acquired:

```
      next(scan)

     # Save the FID and Spectrum
      fid2d[~,stepNr] = sumData
      spectra[~,stepNr] = spectrum

      StackedPlot(prs,  real(spectra), stepNr, fRange, fAxisDisp, "yes",
      pulseLengths,"linear", "Frequency (ppm)", "Pulse length
      (\G(m)s)","Spectra for different pulse lengths")

   next(stepNr)
```

If the pulse sequence is now recompiled and a sensible pulse length range added (2 to 21 µs in 1 µs steps) we gain the following result

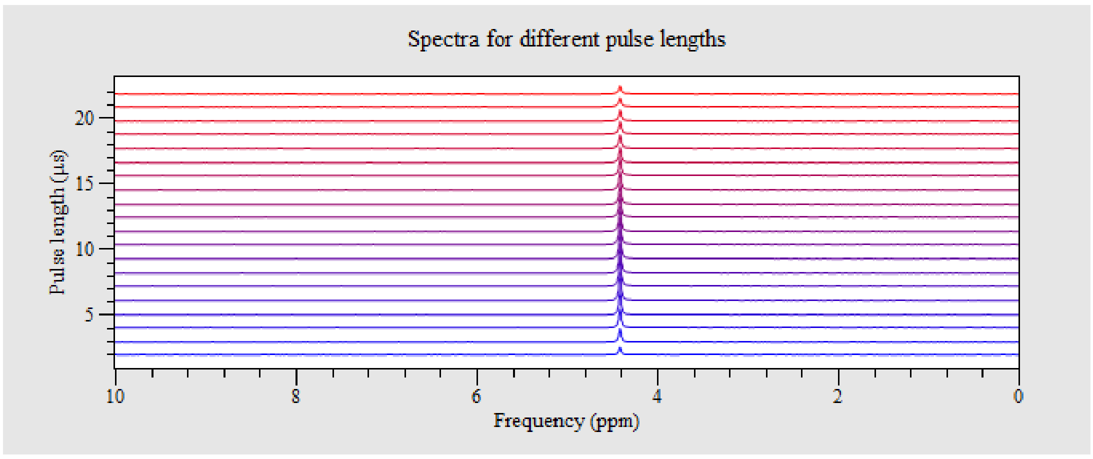


The first thing we notice is that the amplitude of each spectrum is not changing. That is because we have not changed the pulse length being sent to the Spectrometer. To do this we need to modify the ppList before it is sent to the spectrometer. We do this with the utility procedure ucsRun:setPPDelay:

```
# Vary the pulse duration scans
   for(stepNr = 0 to nrLengthSteps-1)

   # Modify ppList with the new pulse length
      ppList = ucsRun:setPPDelay(ppList,varIndex[0],pulseLengths[stepNr])

   # Allocate space for output data
      sumData = cmatrix(totPnts)

   # Accumulate scans
      for(scan = 0 to nrScans-1)
```

If you look in ucsRun you will find a number of other conversion procedures which you need for updating different parameters. (See appendix C for a list of these).

After this change we see the amplitude changing nicely



Just one more step is necessary – we need to save this data. Modify the save commands to save the stacked plot and the raw FID data for MNova analysis:

```
   ucsFiles:savePlot(prs,:getPlotInfo("pt3"),guipar,"noReport")
   ucsFiles:saveMNovaData(fid2d,"",guipar,"simpleReport")
```

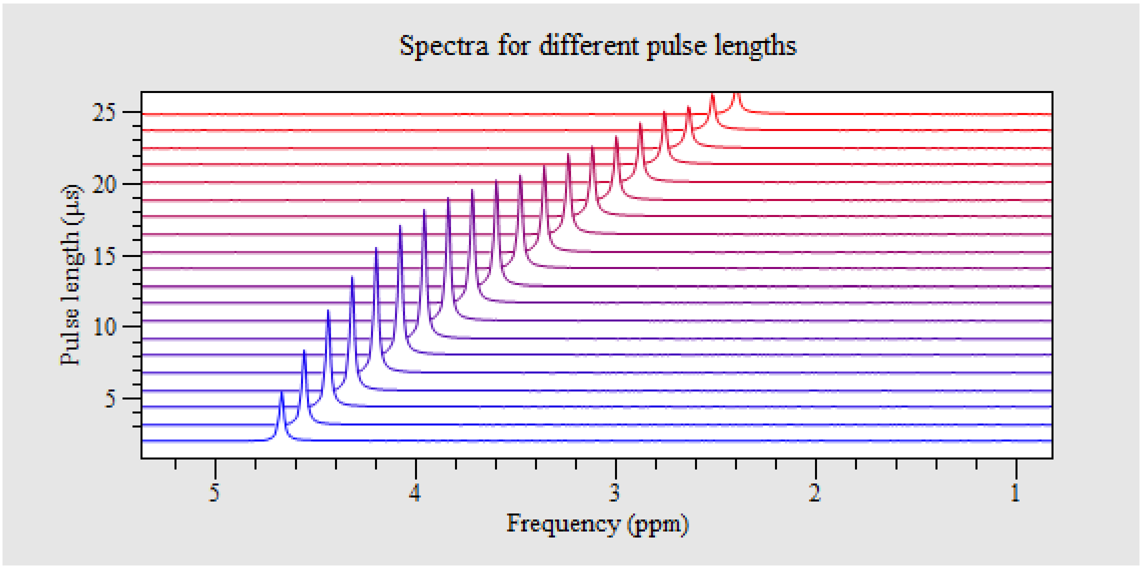
and in the getPlotInfo procedure make sure we save this data:

```
   info = ["pt3","pulseDurationSweep.pt1"]
```

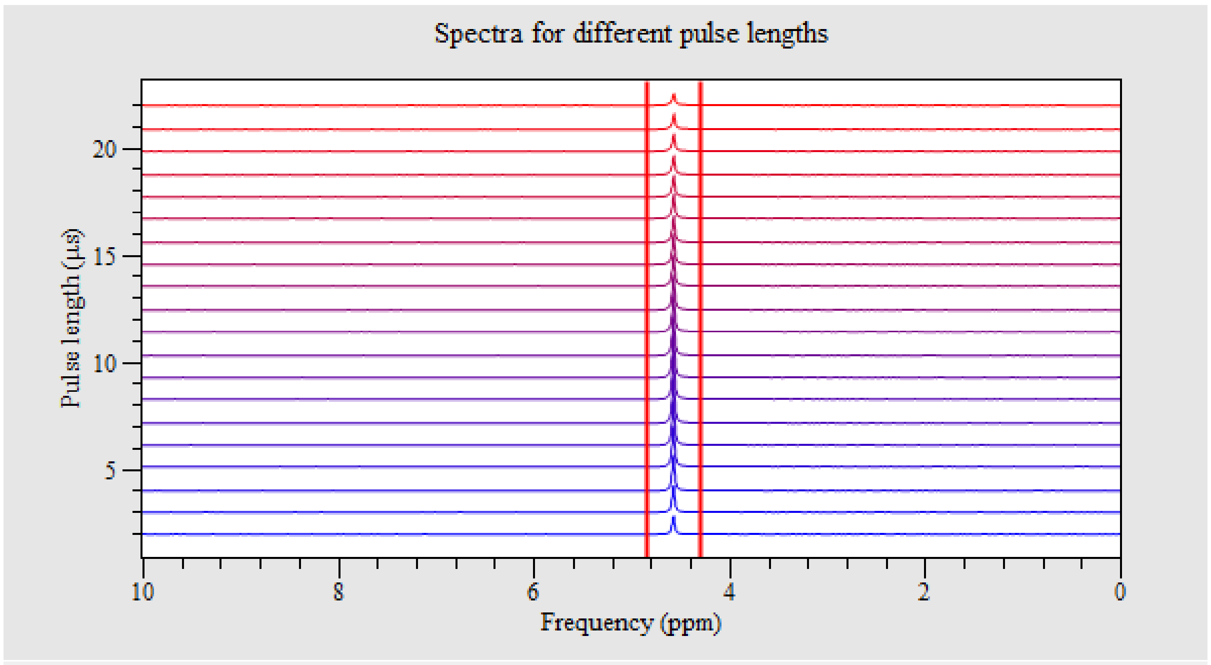
We can also add some post processing tools to the user interface macro to change the stacked plot view , integrate the data and give access to MNova.

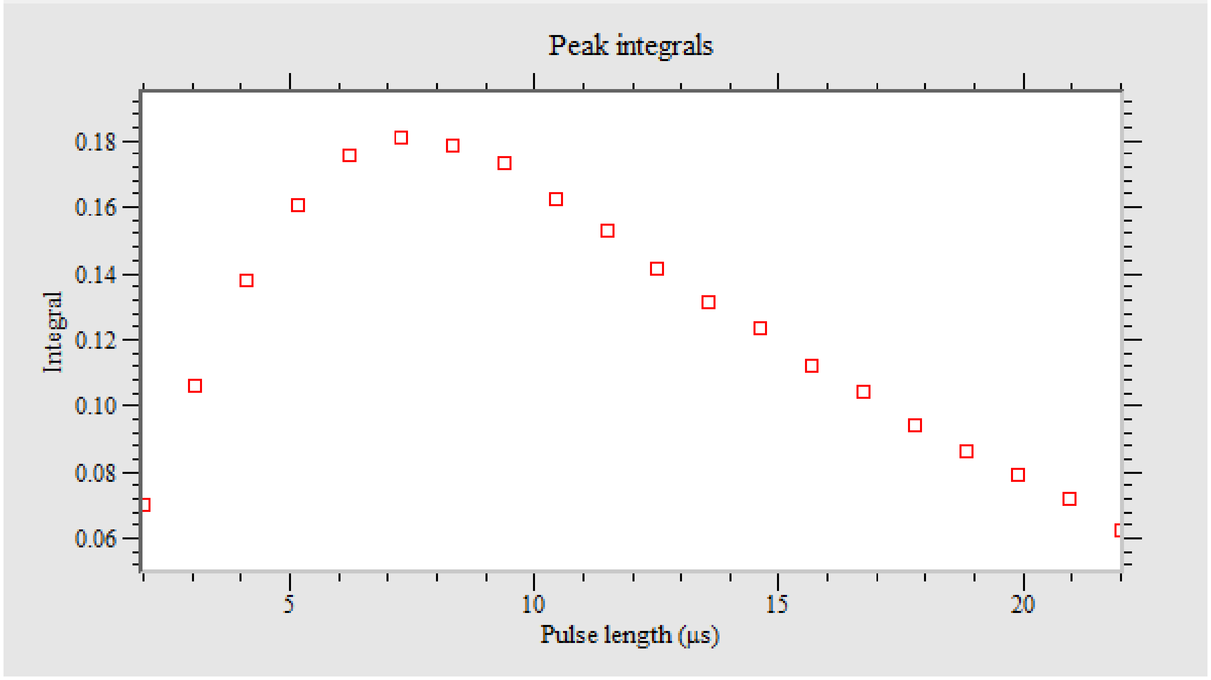
```
procedure(processing_controls)

   layout = ["buttonLabel = \"View\"",       "plotName = \"pt3\"",
"macroToRun = \"StackedPlotSetup()\"";
             "buttonLabel = \"Integ.\"",     "plotName = \"pt3\"",
"macroToRun = \"IntegrateRegions()\"";
             "buttonLabel = \"MNova\"",      "plotName = \"pt3\"",
"macroToRun = \"exportMNova2D(\\\"pt3\\\")\""]

endproc(layout)
```

The View option allows more informative stacked plot displays:



While the Integ option allows analysis of the peak integrals:





An important step after writing and testing a new pulse program is to save the last run set of parameters to the default parameter file. This option can be found in the main Files menu if this has been enabled in the preferences (*Save to default parameter file*). This allows subsequent use of the experiment to have sensible start values. Be careful not to apply this option to the standard experiments!

To give more useful feedback as to the progress of the running experiment the progress bar should be updated correctly. This requires the addition of the expectedDuration procedure:

```
procedure(expectedDuration, guipar)

   assignstruct(guipar)
   totScans = nrLengthSteps*nrScans + useStartDelay
   duration = totScans*repTime/1000

endproc(duration)
```

This just calculates the expected number of scans. useStartDelay is a variable equal to 1 or 0 which controls the addition off a predelay before the experiment starts. This defaults to 0 when in the history mode and 1 when in batch mode.

A completed ProtonDurationSweep experiment is included in the Proton experiment folder for reference. (This version has been modified to allow interactive integration).

Appendix A Pulse program commands

Here is a list and a short description of all pulse program commands grouped according to the function type. More details can be found by typing help and then the command name in the command line interface e.g.

> help pulse

A.1 Acquire NMR data

A.1.1 Blocking acquire

This command acquires data from the Proton or X-channel receiver and stores it in the DSP memory. No other command can be run during this process.

```
acquire( mode, points, [duration])
```

`mode` ... is one of:

`overwrite`	data is written to the beginning of memory.
`append`	data is appended to last data collected.
`sum`	acquired data is summed with the previous data.
`integrate`	acquired data is integrated and returned as single point then appended to previously collected data.
`adc`	acquire data at maximum sample rate (100 MSPS) from the ADC without applying a digital filter and write from the beginning of DSP memory. Note total acquisition time will be memory limited since DW = 0.01 μs.

`points`	the number of complex points collected (maximum 128k). (`nx`)
`duration`	is an optional short delay, (`dx`), (1 μs to 327 ms), which is applied to the append, sum and add modes. This is the total length of time between the start of data acquisition and the next pulse sequence command. This is typically used for all modes except overwrite and adc.

A.1.2 Non-blocking acquire

This command acquires data from the Proton or X-channel receiver in the background, storing it on the transceiver board. It returns immediately so other commands can be run.

`acquireon(points)`

This is equivalent to mode = overwrite. i.e. start acquiring and store at start of memory.

or

`acquireon(mode, points)`

`mode` ... is one of:

`overwrite` ...	data is written to the beginning of memory.
`append`	data is appended to last data collected.
`adc`	acquire data at maximum sample rate (100 MSPS) from the ADC without applying a digital filter and write from the beginning of memory. Note total acquisition time will be memory limited since DW = 0.01 μs)

`points`: the number of complex points collected (maximum 128k). (`nx`)

This command reads in the acquired data obtained with acquireon according to the mode setting, copying it to the DSP board memory.

`acquireoff( mode, points)`

`mode` ... is one of:

`overwrite` ...	stop acquiring and copy to DSP.
`adc`	stop acquiring and copy to DSP.
`pause`	pause acquisition, but don't reset memory address or send to DSP.
`finish`	stop acquiring and append to DSP data.

`points` ... the number of complex points to collect. (`nx`)

For more details see Appendix XX – acquisition explained.

A.2 Clear the DSP data memory

If the acquire command uses the sum mode (i.e. adding to existing memory) you should first run this command to ensure it is zeroed.

`cleardata(number_of_points)`

`number_of_points` .... number of points in DSP memory to clear (`nx`)

A.3 Delay for a specified time

A.3.1 Short delay

`delay(duration)`

`duration` .... is a short delay, (`dx`), (0.25 µs to 327 ms)

A.3.2 Long delay

This delay provides a much larger range of values at the expense of more code in the pulse program.

`wait(duration)`

`duration` .... is a long delay, (`wx`), (2 µs to 167 s)

A.4 Frequency modifications

A.4.1 Set the receiver frequency

`setrxfreq(value)`

`value` ..... frequency to use (MHz) (`fx`).

A.4.2 Set the transmitter frequency

`settxfreq(value)`

`value` ..... frequency to use for channel 1 (MHz) (`fx`).

or

`settxfreq(channel, value)`

`channel` ... which channel frequency to change (`1/2`).
`value` ..... frequency to use (MHz) (`fx`).

or

`settxfreqs(value1, value2)`

`value1 ...`	frequency to use for channel 1 (MHz) (`fx`).
`value2 ...`	frequency to use for channel 2 (MHz) (`fx`).

A.4.3 Increment the receiver frequency

```
incrxfreq(increment)
```

`increment ...` amount frequency should be incremented by (MHz) (`fx`).

A.4.4 Increment the transmitter frequency

```
inctxfreq(increment)
```

`increment ...` amount frequency should be incremented by (MHz) (`fx`).

A.5 Looping

A.5.1 Start a loop

All commands between loop and matching endloop will be repeated loop_number times. loop_number >= 0.

```
loop(loop_name, loop_number)
```

`loop_name ..........`	unique identifier for the loop (`lx`)
`loop_number .........`	number of times to execute the loop (`nx`)

A.5.2 End a loop

```
endloop(loop_name)
```

`loop_name ..........` unique identifier for the loop (`lx`)

Note that loops can be nested.
Also make sure the loopnames are unique, otherwise an error will occur on compilation.

A.5.3 Conditional statements

Execute a block of code is a statement is true. The block is bounded by the iftrue and endiftrue commands.

```
iftrue(block_name, test_value)
```

`block_name`	unique identifier for the block (`sx`)
`test_value`	the value to be tested for non-zero (`nx`)

```
endiftrue(block_name)
```

`block_name`	unique identifier for the block (`sx`)

Note this command was originally called skiponzero or skiponfalse and these variants are still available if the logic makes more sense. This is why the block_name starts with an ‘s’.

A.6 RF pulse production

Several commands can be used to produce an RF output, either as a pulse or a phase, frequency and amplitude modulated waveform.

A.6.1 Simple pulse

Generates a single or dual pulse with specified amplitude, phase, duration and optional frequency.

Single channel version

```
pulse(destination, amplitude, phase, duration, [frequency])
```

`destination`	is either 1 (proton channel) or 2 (X channel)
`amplitude`	the pulse amplitude, (`ax`).
`phase`	pulse phase, (`px`), – set in the phase cycle array.
`duration`	is a short delay, (`dx`), (1 µs to 327 ms).
`frequency`	an optional frequency in MHz (`fx`).

Dual -channel version

```
pulse( 1, amplitude1, phase1, frequency1,
       2, amplitude2, phase2, frequency2, duration)
```

Parameters as above except channels are fixed, frequencies are required, and the duration is common.

A.6.2 Shaped pulse

Generates an RF pulse with modulated amplitude on one channel. Note that for compatibility with the FX3 based spectrometers the shapedrf2 or dualshapedrf2 commands should be used instead.

```
shapedrf(destination, ampTable,
         phaseTable, phase, tableSize, stepTime, [frequency])
```

`destination`	is either 1 (proton channel) or 2 (X channel)
`ampTable`	an amplitude table (magnitude only), (`tx`).
`phaseTable`	a phase table (`tx`).
`phase`	phase offset, (`px`), – set in the phase cycle array.
`tableSize`	the number of entries in each table, (`nx`).
`stepTime`	how long each step will last (µs) (`dx`).
`frequency`	and optional frequency for channel 1 (MHz), (`fx`).

or

```
shapedrf(ampTable, phaseTable, phase1, phase2, tableSize, stepTime,
        frequency1, frequency2)
```

`ampTable`	interleaved amplitude tables (magnitude only), (`tx`).
`phaseTable`	interleaved phase tables (`tx`).
`phase1`	phase offset for channel 1, (`px`), – set in the phase cycle array.
`phase2`	phase offset for channel 1, (`px`), – set in the phase cycle array.
`tableSize`	the number of entries in each table, (`nx`).
`stepTime`	how long each step will last (µs) (`dx`).
`frequency1`	the frequency for channel 1 (MHz), (`fx`).
`Frequency2`	the frequency for channel 2 (MHz), (`fx`).

Generates RF pulses with modulated amplitude and phase on one channel

```
shapedrf2(destination, ampPhaseTable, phase, tableSize, stepTime)
```

`destination`	is either 1 (proton channel) or 2 (X channel)
`ampPhaseTable` .	an interleaved amplitude and phase table (`tx`).
`phase`	pulse phase, (`px`), – set in the phase cycle array.
`tableSize`	the number of entries in each table, (`nx`).
`stepTime`	how long each step will last, (`dx`).

Generates RF pulses with modulated amplitude and phase on both channels

```
dualshapedrf2(ampPhaseTable, phase1, phase2, tableSize, stepTime)
```

`ampPhaseTable` .	2 channel interleaved amplitude and phase tables (`tx`).
`phase1`	phase offset for channel 1, (`px`), – set in the phase cycle array.

`phase2`	phase offset for channel 1, (`px`), – set in the phase cycle array.
`tableSize`	the number of entries in each table, (`nx`).
`stepTime`	how long each step will last, (`dx`).

For more information on using interleaved table see Appendix XX using tables.

A.6.3 Chirped Pulse

Generates an RF pulse with modulated amplitude and frequency on a single channel

```
chirprf(destination, ampTable, freqTable, phase, tableSize,
        stepTime)
```

`destination`	is either 1 (proton channel) or 2 (X channel)
`ampTable`	an amplitude table (magnitude only), (`tx`).
`freqTable`	a frequency table (positive only), (`fx`).
`phase`	pulse phase, (`px`), – set in the phase cycle array.
`tableSize`	the number of entries in each table, (`nx`).
`stepTime`	how long each step will last, (`dx`).

A.6.4 Switch on the RF output

Simply switches on the RF output. Use delays to define the pulse duration

```
txon(destination, amplitude, phase)
```

`destination`	is one of: 1/2/1nb/2nb/w1/w2
`amplitude`	the pulse amplitude, (`ax`).
`phase`	pulse phase, (`px`), – set in the phase cycle array.

See Appendix XX for a discussion of RF pulse destinations.

A.6.5 Switch off the RF output

Switches off the RF output.

```
txoff(destination)
```

`destination`: is one of: 1/2/1nb/2nb/w1/w2

See Appendix XX for a discussion of RF pulse destinations.

A.7 Shim and gradient commands

These commands control the currents in the various shims and where fitted, the single gradient channel.

A.7.1 Shim control

Set a shim value

```
shim16(channel, amplitude)
```

```
channel ..... the shim channel (1-15) (nx)
amplitude ... the shim value (16 bit signed number) (nx/tx)
```

Ramp a shim current linearly

```
shimramp16(channel, start, end, steps, duration)
```

```
channel .... the shim channel (1-15) (nx)
start ...... the start ramp value (16 bit signed number) (nx/tx)
end ........ the end ramp value (16 bit signed number) (nx/tx)
steps ...... number of steps in the ramp (nx)
duration ... how long each step lasts (dx)
```

A.7.2 Gradient control

Switch on the gradient (where fitted)

```
gradon(amplitude)
```

```
amplitude ... the gradient value (a 16 bit signed number) (nx/tx)
```

Switch off the gradient (where fitted)

```
gradoff()
```

Ramp the gradient current linearly

```
gradramp(start, end, steps, duration)
```

```
start ...... the start ramp value (16 bit signed number) (nx/tx)
end ........ the end ramp value (16 bit signed number) (nx/tx)
steps ...... number of steps in the ramp (nx)
duration ... how long each step lasts (dx)
```

A.8 Table commands

These commands control the location of the table index – i.e. which value in a table will be used next.

`decindex(table)`

`table`	...	table index to decrement (`tx`)

`incindex(table)`

`table`	...	table index to increment (`tx`)

`setindex(table, index)`

`table`	...	table index to increment (`tx`)
`index`	...	value for the index (`nx`)

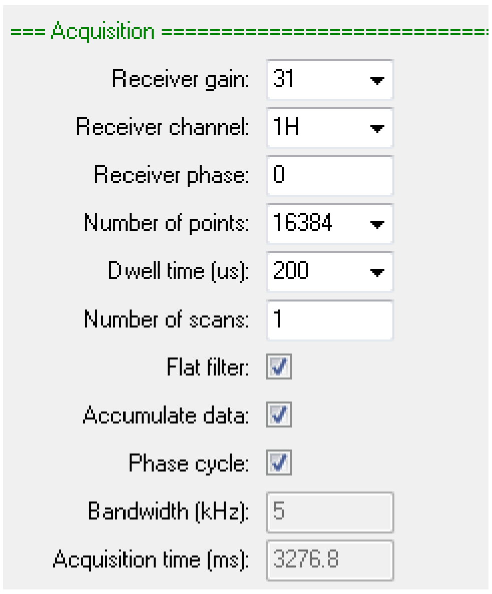
Appendix B Available control groups

Following is a list of the control (tab) groups available in this version of Expert which are used by some of the experiments and the key feature(s) which differentiate them. These may be found in the folder:

<prospa>\Macros\UCS-PP\Tabs\AlternateInterface

B.1 Acquisition

Used when collecting data by specifying a dwelltime (sampling time) and the number of points to collect. This is the natural machine units although not necessarily the most comfortable for the spectroscopist. Note that the bandwidth and acquisition time are automatically updated if the number of points or dwell time is modified.

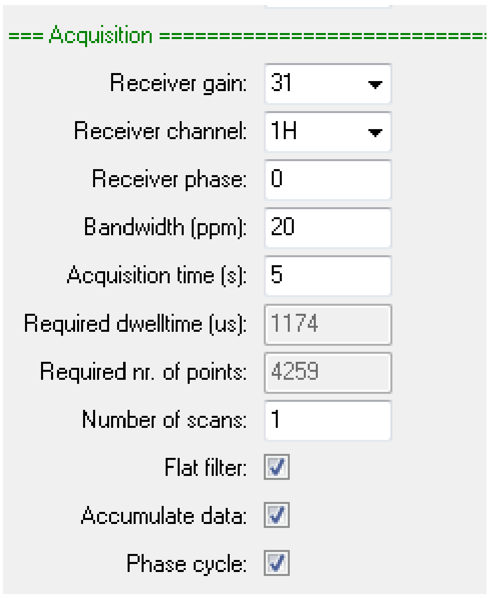


=== Acquisition ===========================

Receiver gain:	31
Receiver channel:	1H
Receiver phase:	0
Number of points:	16384
Dwell time (us):	200
Number of scans:	1
Flat filter:	✓
Accumulate data:	✓
Phase cycle:	✓
Bandwidth (kHz):	5
Acquisition time (ms):	3276.8

B.2 AcquisitionBW

Used when collecting data by specifying a bandwidth in PPM and an acquisition time. Be aware than small bandwidths (or large dwelltimes) may lead to excessive first order phase distortions.



=== Acquisition ===========================

Receiver gain:	31
Receiver channel:	1H
Receiver phase:	0
Bandwidth (ppm):	20
Acquisition time (s):	5
Required dwelltime (us):	1174
Required nr. of points:	4259
Number of scans:	1
Flat filter:	✓
Accumulate data:	✓
Phase cycle:	✓

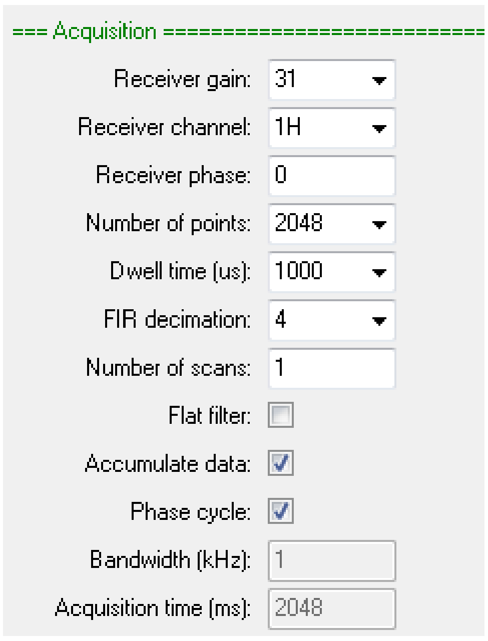
Note that the dwelltime and required number of points are automatically updated if the bandwidth or acquisition time is modified.

B.3 AcquisitionBW2D

A 2D version of AcquisitionBW.

B.4 AcquisitionFIR

The same as the standard acquisition list except it provides the option to specify how much FIR decimation should be applied. This is used with sequences which do not apply the FIR filter in the spectrometer hardware but rather delay this until the data is in the PC. This allows linear prediction to be used to minimise baseline and first order phase correction problems by predicting the first few data points in the FID. However it does require that more data is collected. This is determined by the FIR decimation factor.

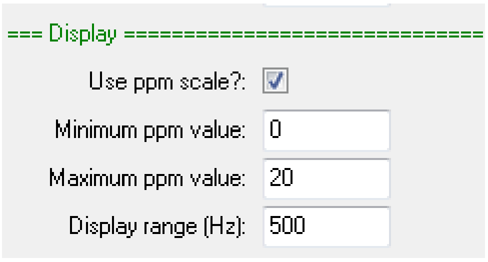


=== Acquisition ===========================

Receiver gain:	31
Receiver channel:	1H
Receiver phase:	0
Number of points:	2048
Dwell time (us):	1000
FIR decimation:	4
Number of scans:	1
Flat filter:	☐
Accumulate data:	✓
Phase cycle:	✓
Bandwidth (kHz):	1
Acquisition time (ms):	2048

B.5 Display_Std

Allows the selection of a ppm or Hz scale. The former allows both scale limits to be selected while the latter just allows a width. Zero in the display range (Hz) field displays all the data if 'Use ppm scale' is not checked.

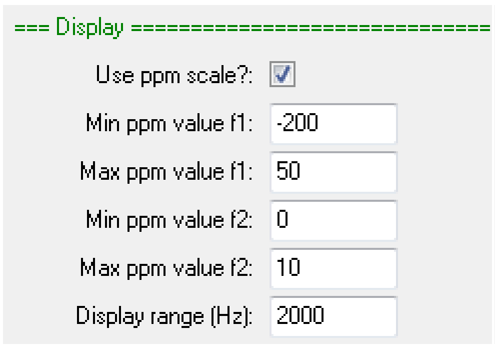


=== Display ==============================

Use ppm scale?:	✓
Minimum ppm value:	0
Maximum ppm value:	20
Display range (Hz):	500

B.6 Display_2D

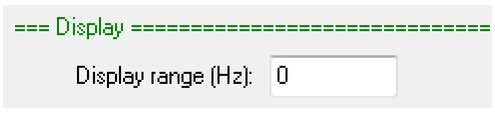
The 2D version of Display_Std which provides control of the 2nd dimension range.


=== Display ==============================

Use ppm scale?:	✓
Min ppm value f1:	-200
Max ppm value f1:	50
Min ppm value f2:	0
Max ppm value f2:	10
Display range (Hz):	2000

B.7 Display_Hz

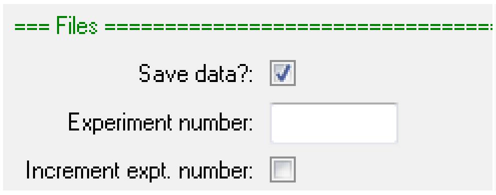
Simply allows the display range to be selected in Hz.


=== Display ==============================

Display range (Hz):	0

B.8 File_Settings

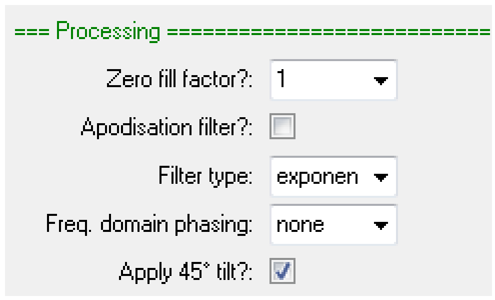
Here whether data should be saved can be selected, whether experiment numbers should be used and if these number should automatically increment.


=== Files ==================================

Save data?:	✓
Experiment number:	
Increment expt. number:	☐

B.9 Jres_processing

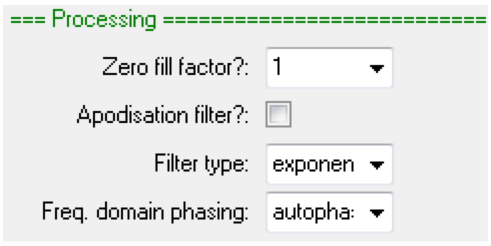
The same as Processing_std except there is an option to tilt the 2D data by 45 degrees.


=== Processing ===========================

Zero fill factor?:	1
Apodisation filter?:	☐
Filter type:	exponen
Freq. domain phasing:	none
Apply 45° tilt?:	✓

B.10 Processing_std

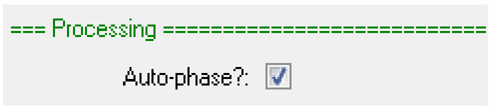
Provides options to zero fill and apodize the time domain data. Different frequency domain phasing options are also provided.


=== Processing ===========================

Zero fill factor?:	1
Apodisation filter?:	☐
Filter type:	exponen
Freq. domain phasing:	autopha

B.11 Processing_autophase

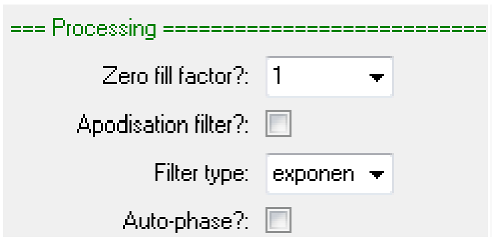
Just provides an autophase option. Useful with the CPMGInt experiments where time domain filtering is not an option.


=== Processing ===========================

Auto-phase?:	✓

B.12 Processing_filter_autophase

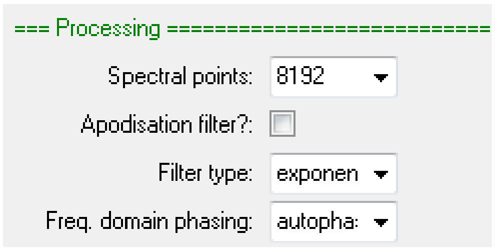
The same as Processing_std except it includes an autophase option as well. Useful when you are wanting to phase a time domain dataset e.g. a T1 or a T2 experiment).


=== Processing ===========================

Zero fill factor?:	1
Apodisation filter?:	☐
Filter type:	exponen
Auto-phase?:	☐

B.13 ProcessingBW

Use this option in combination with AcquireBW to work with acquisition time and bandwidth rather than dwellTime and nrPnts. In this case you define the number of spectral points to be equal to or greater than the number of time domain points collected (which won't necessarily be a factor of 2 which is necessary for the Fourier transform used).


=== Processing ===========================

Spectral points:	8192
Apodisation filter?:	☐
Filter type:	exponen
Freq. domain phasing:	autopha

Appendix C Procedures to change pulse program values

Many pulse program values are presented in a form which is not suitable for the hardware (dB for RF amplitude, frequencies in MHz, delays in microseconds etc.). This conversion is done automatically for most parameters before reaching the execpp procedure. However when modifying a parameter between scans you will need to explicitly convert the user unit into machine units and update the parameter list. Procedures written in the macro ucsRun can be used for this purpose.

All have the same format:

```
pulse_parameter_list  = procedure_name(pulse_parameter_list,
                                       index_of_parameter,
                                       user_unit)
```

In execpp the `pulse_parameter_list` is the variable `ppList`. The parameter index is stored in the array varIndex and `user_unit` you must supply.
Here is the list of helper procedures in ucsRun:

setPPAmplitude	set an RF amplitude (dB) in the pulse sequence parameter array
setPPDelay	set a short delay (µs) in the pulse sequence parameter array
setPPLongDelay	set a long delay (µs) in the pulse sequence parameter array
setPPNumber	set a number in pulse sequence parameter array
setPPPhase	set an RF phase (0-4) in the pulse sequence parameter array
setPPFrequency	set a frequency (MHz) in the pulse sequence parameter array

An example of using one of these commands is shown below

```
      ppList = ucsRun:setPPDelay(ppList, varIndex[0], pulseLengths[stepNr])
```

Note that array index in varIndex. This is zero because this is the first (and only in this case) variable in the variable list found in the pulse program

```
   variables = ["d90Dur"]
```

If there were two variables and you wanted to modify the second then you would use `varIndex[1]`.

Appendix D Acquisition Explained

When data is received by from the NMR probe it first amplified by a factor of few thousand before being digitised by a 16-bit converter at 100 million samples per second. This digital signal is then multiplied by two reference frequencies equal to the NMR receive frequency but differing in phase by 90 degrees. This gives two channels – the in-phase and quadrature signals or in NMR terms the real and imaginary parts. These are then passed through two identical digital filters which reduce the bandwidth by averaging points together. The result of this is a downshifting of the signal from many MHz to kHz, the exact amount depending on the chosen dwell-time or sampling interval. Two filters are available – a fast, so-called, CIC filter which reduces the frequency bandwidth by a large amount but has a poor frequency response and a slower FIR filter which flattens this response, but only reduces the frequency by a factor of 2. The 'Flat filter' option in the Spinsolve parameter list just adds the FIR filter to the CIC filter output. When unchecked only the CIC filter is used. This is typically only used in experiments where we expect all the signal to be close to the centre of the spectrum. A typical case is a CPMG experiment collected with a large bandwidth – say 1 MHz. In this case the CIC only filter will allow shorter echo times and give better time domain SNR because of the narrower frequency response.

Digital filters have a characteristic start-up duration where the amplitude goes from zero to the final value over a several sampling intervals. This results in significant distortion to the FID if not corrected. In the Spinsolve spectrometer we simply throw away these data points and add an equal number to the end. By default, 6 data points are removed.

Because it takes a finite amount of time for the NMR signal to propagate through the digital filter we need to apply a correction offset to start sampling at exactly the right time. This is stored in the variable rxLat (receiver latency). By design this delay is small and constant for the combined FIR+CIC filter but can get quite large and negative in the CIC-only case because here we are throwing away more initial points (6) than the ideal (3). This was done because the initial FID distortion is more pronounced if only 3 points are discarded.

Another time which needs to be considered when collecting data is the overall sampling period. Ideally this would just be the dwell-time multiplied by the number of collected data values. However, there are some additional delays required to get the data through the filter and into memory. These are only important in experiments which collect data between pulses and we need to know how long that will take. In this case there is an internal procedure which calculates this and if you don't leave enough time for the acquisition it will result in an error message. For this reason, almost all acquisition modes have the option to specify a duration so the timing of the pulse sequence can be accurately defined. Of course, this duration should be set to be larger than the expected acquisition time to avoid an error message. This duration can be estimated using the procedure ucsRun:getAcqTime.

## Appendix E	Using Tables

Several pulse-program commands allow the use of waveform tables as arguments. This permits parameters to be modified inside a loop according to a table of values which are stored on the spectrometer. Because these tables can be quite large, they are written to the spectrometer before the experiment begins. A typical table application is to step an amplitude through several values. The following example changes the amplitude of an RF pulse to produce a linear ramp. The output of the transceiver is connected back into the input directly or via an external amplifier or filter. The result is the amplitude response of the combined transceiver and filer/amplifier. Note that the code to save the table to the spectrometer is not part of the pulse program file but happens, nevertheless, behind the scenes.

```
...
   relationships = ["nAmpSteps = ampSteps",
                    "nPnts = nrPnts",
                    "b1Freq = f1",
                    "txMax = ucsRun:convertTxGain(txMaxdB)",
                    "tAmp = linspace(0,txMax,ampSteps)",
                    "totPnts = nAmpSteps",
                    "totTime = acqTime"]

...
```

```
   initpp(dir)                   # Set internal parameter list

      cleardata(nAmpSteps)          # Clear data memory
      setindex(tAmp,0)
      loop("l1",nAmpSteps)          # Measure at nAmpSteps amplitude steps
         txon("1nb",tAmp,p1)        # Switch on tx using new amplitude
         delay(10)                  # Wait for tx to stabilse
         acquire("integrate",nPnts) # Acquire nPnts data points and integrate
         txoff("1nb")               # Switch off tx
         incindex(tAmp,1)           # Increment tx amplitude  by 1
        delay(10)                   # Wait for system to recover
      endloop("l1")                 # Next measurement

   lst = endpp(0)                 # Return parameter list
```

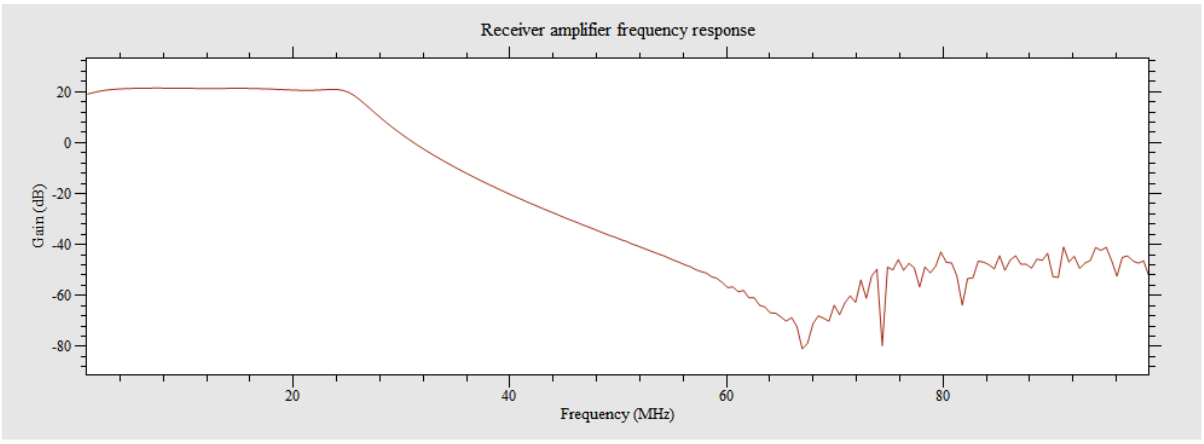
Here the RF amplitude values are stored in the tAmp array (a simple ramp), then between evaluating the relationships list and running the sequence, this array is first saved to the spectrometer, and a table reference is returned with the same name. This has only two entries – the address of the table in memory and the size.

In the sequence the table index is initialised to the start of the table and then each time around the loop the amplitude in the txon command is taken from the current location in the table now stored on the spectrometer. The incindex command then moves to the next element in the table. At each step, the signal that has passed through the external circuit is collected and saved as a single value with the acquire command. Because the transmit and receive frequencies are the same the data will always have an offset frequency of zero.

A more complex case is to sweep the frequency rather than the amplitude. This could be used to determine the frequency response of an amplifier or filter. Similar code is also used to generate the Wobble response in the Spinsolve.

```
   relationships = ["b1Freq = lowFreq",
                    "fSweep = linspace(lowFreq,upFreq,freqSteps)",
                    "tFreq1 = ucsFX3:convertFrequency(fSweep)",
                    "tFreq2 = ucsFX3:convertFrequency(fSweep*10)",
                    "nFreqSteps = freqSteps",
                    "nPnts = nrPnts",
                    "totPnts = freqSteps",
                    "totTime = acqTime"]

...

   initpp(dir)                   # Reset internal parameter list

      cleardata(nFreqSteps)
      setindex(tFreq1,0)
      setindex(tFreq2,0)
      loop("l1",nFreqSteps)           # Measure at nFreqSteps
         setrxfreq(tFreq2)            # Set the receiver frequency
         txon("1nb",aRF,p1,tFreq1)    # Switch on Tx channel 1 using new Tx freq.
         delay(10)                    # Wait for stability
         acquire("integrate",nPnts)   # Acquire nPnts data points and integrate
         txoff("1nb")                 # Switch off the Tx
         incindex(tFreq1,2)           # Increment Tx frequency
         incindex(tFreq2,2)           # Increment Rx frequency
         delay(10)                    # Wait for stability
      endloop("l1")                   # Next measurement
   lst = endpp(0)                    # Return parameter list
```

This works in much the same way as the amplitude sweep except that in this case, we need two tables, because the transmit and receive frequencies differ by a factor of 10 (this is for calculation purposes only, the final internal Tx and Rx frequencies are the same). Also, the frequency values in the table are 32-bits long which is double that of the amplitudes, so we need to increment the table by 2 each time around the loop. Here is an output of the frequency sweep for an amplifier used in the spectrometer receiver.



In addition to explicitly setting a parameter by stepping though a table there are also some dedicated commands such as shapedrfpulse1/2 or chirpedrf which encapsulate this loop in a single command allowing shorter loop times.

Appendix F Programming multi-X spectrometers

If you have a multi-X spectrometer then it has probe coils which can be tuned to different nuclei using internal switching. To see what nuclei your spectrometer supports, (in addition to 1H), run the following command from the CLI:

```
SpinsolveParameterUpdater:getXChannelNames()
```

This will return a list of nuclei.

To ensure that the probe switches are set correctly when you run an experiment, you need to call a special command at the start of the execpp procedure:

```
SpinsolveParameterUpdater:setXChannel(channelName)
```

You only need to do this if the receive channel in the parameter list is not the same as the X nucleus transmit channel. For example, the HMBC experiment detects on the proton channel but transmits on both the proton and carbon channels. If you have a multi-X system then you need to ensure that the probe tuning is optimised for carbon.

A Spinsolve with a tuned Fluorine or Tritium channel is a special case as 19F/3H and 1H share the same receive and transmit channels. In this case you need to use a different command:

```
SpinsolveParameterUpdater:setEnhanced("19F")
```

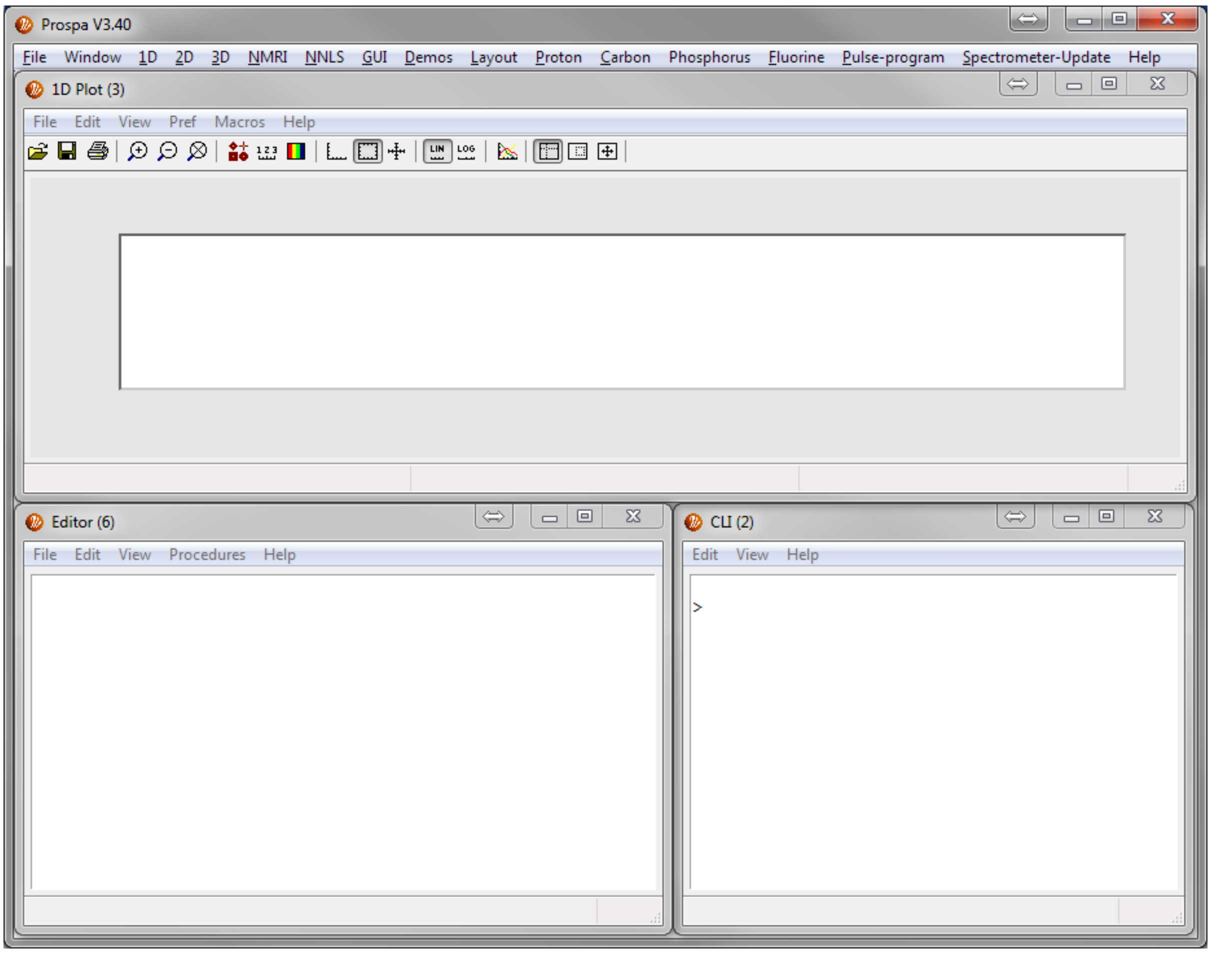
or

```
SpinsolveParameterUpdater:setEnhanced("3H")
```

Appendix G An introduction to Prospa programming

The entire SpinsolveExpert application is built on top of the Prospa programming environment. This is based around an interpreted language which is reminiscent of Matlab, but with some important differences.

When Prospa starts up it runs a script or macro which defines the user interface. If you run Prospa directly from the application folder you will see an interface like this:


Prospa V3.40
File Window 1D 2D 3D NMRI NNLS GUI Demos Layout Proton Carbon Phosphorus Fluorine Pulse-program Spectrometer-Update Help
1D Plot (3)
File Edit View Pref Macros Help
Editor (6)
File Edit View Procedures Help
CLI (2)
Edit View Help
>

This consists of separate windows with simple functionality – a 1D plot, editors and a command line interface. More complex interfaces can be designed (like SpinsolveExpert) which integrate these objects into the one window. (See also the Layout menu for other possibilities).

The scripting langage is quite complex consisting of over 500 built in commands and more than 20 data types.

However for the purposes of using the SpinsolveExpert interface only a fraction of these are needed.

G.1 Data types

The most important data types are:

string	double quoted text based strings which may include escape characters. Example: "A filename"
list	arrays of strings. e.g `["string1","string2","string3"]`
list2d	jagged arrays of lists e.g. `["string11","string12", ... ; "string21","string22", ...]`
float	4 byte floating point numbers (the default number type) e.g. 3.14159
double	8 byte floating point number e.g. 3.14159265358979 31d
structure	a linked list of variables. e.g. `s = struct(a=23, b = [1:10], c = "This is a test")`
class	like a structure but also support functions.
structure_array	an array of structures e.g. `s = struct(a=23, b = [1:10]; c = "This is a test", d= 2+3j; e = "_Another string"_, f = pi)`
vector	1D matrices - arrays of floats or doubles e.g. [1,2,3,4]
matrix	2, 3 or 4 dimensional arrays of floats or doubles e.g. [1,2,3,4;5,6,7,8]
objects	user interface objects such as plots or text boxes. These can be interrogated to extract or set various parameters.

G.2 Variables

Data values are stored in variables. This is done using an assignment statement e.g.

```
myVector = [1,2,3,4]
```

Note that vector name can be any length (although shorter names are interpreted faster), are case sensitive and may include numbers as long as the resultant name is not a valid number. (So 1Pulse is a valid variable name!)

Variables can be collected together in structures. To define a structure use the syntax

```
s = struct()
s->pi = 3.1415926
s->vec = [1,2,3,4,5]
```

or more simply

```
s = struct(pi=3.1415926, vec=[1,2,3,4,5])
```

in a procedure, local variables can be converted to a list using the command:

```
lst = mkparlist()
```

This takes all local variables (see below) and packs them into a string list. Note that you can't include objects or 3D or 4D matrices in list. Other matrices should be small. The alternative it to use a structure.

To convert a string list into local variables use the reverse command

```
assignlist(lst)
```

Similar commands exist for structure.

Variables defined on the command line interface have global scope (i.e. they are accessible everywhere.) Variables defined inside a procedure (see below) have local scope and are only accessible from within the procedure. It is also possible to have variables with window scope which are accessible from all children of the parent window.

G.3 Control statements

Prospa supports the following control statements. Syntax is fairly standard.

```
if-elseif-else-endif

while-exitwhile-endwhile

for-exitfor-next
```

Please refer to the Prospa documentation for details (type F1 when the cursor is on the command).

G.4 Procedures

Prospa scripts are typically broken in procedures (i.e. functions) which are blocks of code surrounded by the statements

```
procedure(name, arguments)
...
```

```
endproc(return_values)
```

Note that multiple values can be returned from a procedure. To exit early from a procedure use the return command.

To call a procedure from within the current file (which should have the extension .mac – for macro or .pex for prospa executable macro), use the following syntax

```
result = :myProc(arg1, arg2)
```

To call a procedure in another file just add the filename

```
result = filename:myProc(arg1,arg2)
```

The extension need not be included as .mac (or .pex) is assumed.

To return multiple values place them in parenthesis

```
(result1,result2) = :procName(arg1,arg2,arg3)
```

All variables defined in a procedure have local scope i.e. they are not visible outside the procedure.

Please refer to the Prospa manual for more details about using procedures.

G.5 Commands

Some 500 predefined commands are included in the Prospa package. A list can be obtained using the `listcom` command. These commands are written in C++ and so are generally much faster than using explicit low level prospa commands. Check the help viewer accessible from the Help menu to see a list of these.

The syntax is exactly as it is for procedures

```
result = command(arg1, arg2, ...)
```

or

```
(result1,result2 ...)  = command(arg1, arg2, ...)
```

These commands include functions to manipulate data types, access the file system and build user interfaces.

G.6 Debugging

You can test the various commands and syntax by typing them into the command line interface. Use the print (pr) command to print results. e.g.

```
> a = 2^4
> pr a

   a = 16
```

G.7 Expressions

Commands, variables and procedures may be combined in expressions where the syntax is valid. e.g.

```
magnitude = sqrt(x^2+y^2)
```

Standard mathematical precedence is followed. Note that if a command with multiple return values appears in the expression only the first return value is used. If you want to extract the nth returned value use the *retvar* command (1 based).

G.8 Matrices and vectors

Both real and complex vectors and matrices can be defined. To define a blank matrix of dimensions 1,2,3,4 use the following synax

```
mat = matrix(dim1, dim2, dim3, dim4)
```

Where dimx is the size of the xth dimension. Note that for 2 and 3D matrices the dimension order is x, y, z ***not*** row, col , depth as it is in Matlab. This syntax is also used when accessing elements

```
result = mat[x,y,z].
```

To define a linear 1D vector use the following syntax

```
result = [start:step:end] (step is optional)
```

or

```
resut = linspace(start,end,number)
```

To access the first element use the index ***0*** while to access last element use the index ***-1*** (the second to last -2 etc). e.g

```
lastValue= vec[-1]
```

To access some elements in the vector use this syntax

subvector = vec[start:step:end]

All elements in one dimension can be signified with a tilde '~' or colon ':' e.g. to extract the yth row from a 2D matrix:

row = mat[~,y]

To get the dimensions of a matrix use the size command

```
(width, height) = size(mat)
```

G.9 User interface objects

User interface ojbects such as buttons, text boxes, plots etc. can be assigned to object variables. They can then be used to access functions and variables embedded in the objects.

Access is done via the arrow operator e.g. for the SpinsolveExpert interface

```
plt1 = pltCtrl1->subplot(1,1)

plt1->plot(x,y)

(x,y) = plt1->getdata()
```

The second command plots the x-y pair in the first region plt1 of plot control pltCltr1 (== pt1). The third command extracts the x-y data from the plot.

To see the commands and variables available for an object just type the object name into the command line interface. Help is also available in the classes topic in the help viewer.

```
> pr plt1

#### 1D plot-region object ####

   PARAMETER           VALUE

   antialiasing1d..... "true"
   autorange ......... "true"
   axes .............. object
   bordercolor ....... [230,230,230]
   bkgcolor .......... [255,255,255]
   clear ............. cmd : clear all trace data
   dim ............... "1d"
   draw .............. "true"
   filename .......... "image_section.pt1"
   filepath .......... "C:\ ... \Example Data\Plots\"
   fileversion ....... 1.4
   getdata ........... cmd : extract (x,y) data
   grid .............. object
   hold .............. "false"
   load .............. cmd : load a pt1 file into plot region
   parent ............ control: 1
   plot ............   cmd : display (x,y) data
   position .......... (1,1)
   border ............ "true"
   save .............. cmd : save plot region contents to a pt1 file
   trace ............. object
   tracelist ......... cmd : return array of trace id numbers
   tracepref ......... cmd : set or get trace drawing preferences
   title ............. object
   xlabel ............ object
   ylabel ............ object
   zoom .............. cmd : set viewing limits
```

G.10 Using the Prospa text editor

If you develop Prospa scripts inside the Prospa editor then you will get some additional features compared with writing it in a simple editor such as Notepad.

1. Syntax coloring. Different types of commands are colored differently.
2. Syntax hints. As you type a command the valid arguments are listed in the editor status box.
3. Help. Pressing F1 when the cursor is on a command will display detailed help and often an example.
4. Procedure searching. Using the editor menus or keyboard shortcuts you can jump to and from Procedure code.
5. Indenting and commenting. The editor menu has commands to rapdily comment and indent block of selected code.

G.11 Additional help

Additional Prospa and pulse programming help can be found in the documents:

Prospa programming manual.pdf
SpinsolveExpert pulse programming manual.pdf
SpinsolveExpert Pulse Program Commands.pdf

These files can be found in the Prospa install directory in the folder "PDF Documentation".

Help can be found in the Viewer found in the main Help menu.

Appendix H System specifications

To be added.